HLPP 2022
15th International Symposium on High-level Parallel Programming and Applications
Porto, Portugal, July 7-8, 2022

Proceedings of the 15th International Symposium on High-Level Parallel Programming and Applications

Porto, Portugal

July 7–8, 2022

Miguel Areias Inês Dutra Jorge Barbosa
(Eds.)

Preface

This volume contains the proceedings of the 15th edition of HLPP, the International Symposium on High-Level Parallel Programming and Applications, which took place in the Department of Computer Science, Faculty of Sciences, University of Porto, Portugal, during July 7–8, 2022.

Since 2001, the HLPP series of workshops/symposia has been a forum for researchers developing state-of-the-art concepts, tools and applications for high-level parallel programming. The general emphasis is on software quality, programming productivity and high-level performance models. Contributions to HLPP are sought in all topics in high-level parallel programming, its tools and applications, including:

- High-level programming and performance models (BSP, CGM, LogP, MPM, etc) and tools
- Declarative parallel programming methodologies
- Algorithmic skeletons and constructive methods
- Declarative parallel programming languages and libraries: semantics and implementation
- Verification of declarative parallel and distributed programs
- Software synthesis, automatic code generation for parallel programming
- Model-driven software engineering with parallel programs
- High-level programming models for heterogeneous/hierarchical platforms
- High-level parallel methods for large structured and semi-structured datasets
- Applications of parallel systems using high-level languages and tools
- Formal models of timing and real-time verification for parallel systems

This year, we received 17 paper submissions. Each paper was reviewed by at least three referees who provided detailed written evaluations. At the end, 10 papers were selected for publication in this volume and presentation at the symposium. The set of selected papers present a variety of contributions and were divided into three sessions for presentation at the symposium. After the symposium, the authors of the selected papers will have the opportunity to revise their papers, taking into account the comments and remarks of the referees, and submit them to the HLPP 2022 Special Issue to be published by Springer in the International Journal of Parallel Programming (IJPP).

We would like to thank our generous sponsors – the Department of Computer Science at Faculty of Sciences, University of Porto (FCUP); the CRACS & INESCTEC research unit; and Huawei – and the EasyChair conference management system for making the life of the Program Chairs easier. We would also like to thank the staff of FCUP (Alexandra Ferreira, Daniel Pereira, Isabel Gonçalves and Paulo Ramos) for making the online event possible.

We want also to express our gratitude to the Steering Committee members, for giving us the opportunity to organize the event, and to the Program Committee members and external reviewers, as the symposium would not have been possible without their knowledge, dedicated time and enthusiastic work. Finally, thanks should go also to the authors of all submitted papers for their contribution and interest in the symposium and to the participants for making the event a meeting point for a fruitful exchange of ideas and feedback on recent developments. Thank you all for your contribution to HLPP 2022.

July 2022,

Miguel Areias
Inês Dutra
Jorge Barbosa

Organization

Steering Committee

Frédéric Dabrowski	Université d'Orléans, France
Marco Danelutto	University of Pisa, Italy
Inês Dutra	University of Porto, Portugal
Arturo Gonzalez-Escribano	Universidad de Valladolid, Spain
Clemens Grelck	University of Amsterdam, Netherlands
Gaétan Hains	Huawei Paris Research Centre, France
Christoph Kessler	Linköping University, Sweden
Herbert Kuchen	University of Münster, Germany
Kiminori Matsuzaki	Kochi University of Technology, Japan
Virginia Niculescu	Babeș-Bolyai University, Romania

Program Chairs

Miguel Areias	University of Porto, Portugal
Inês Dutra	University of Porto, Portugal
Jorge Barbosa	University of Porto, Portugal

Publicity Chair

Carlos Ferreira	Polytechnic Institute of Porto, Portugal

Program Committee

Marco Aldinucci	University of Torino, Italy
Murray Cole	The University of Edinburgh, UK
Iacopo Colonnelli	University of Torino, Italy
Frédéric Dabrowski	LIFO - Université d'Orléans, France
Marco Danelutto	University of Pisa, Italy
João Gama	University of Porto, Portugal
Arturo Gonzalez-Escribano	The University of Valladolid, Spain
Clemens Grelck	University of Amsterdam, Netherlands
Dalvan Griebler	PUCRS/SETREM, Brasil
Gaétan Hains	Huawei Paris Research Center, France
Ali Jannesari	Iowa State University, USA
Christoph Kessler	Linköping University, Sweden
Peter Kilpatrick	Queen's University Belfast, UK
Herbert Kuchen	University of Münster, Germany
Kiminori Matsuzaki	Kochi University of Technology, Japan
Virginia Niculescu	Babeș-Bolyai University, Romania
Aleksandar Prokopec	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Nuno Roma	University of Lisbon, Portugal
Kostis Sagonas	Uppsala University, Sweden
João Sobral	University of Minho, Portugal
Massimo Torquati	University of Pisa, Italy

External Reviewers

Nina Herrmann, Chong Li, Nuno Neves, Wijnand Suijlen,
Thibaut Tachon, João Vieira and Albert Wong

Web Page

`https://hlpp2022.dcc.fc.up.pt`

Sponsors

[dcc] U.PORTO FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO CRACS INESCTEC ASSOCIATE LABORATORY PORTUGAL HUAWEI

Table of Contents

Noname manuscript No.
(will be inserted by the editor)

A Fault-model-relevant Classification of Consensus Mechanisms for MPI and HPC

Grace Nansamba · Amani Altarawneh · Anthony Skjellum

Received: date / Accepted: date

Abstract Large-scale HPC systems experience failures arising from faults in hardware, software, and/or networking. Failure rates continue to grow as systems scale up and out. Crash fault tolerance has up to now been the focus when considering means to augment the Message Passing Interface (MPI) for fault-tolerant operations. This narrow model of faults (usually restricted only to process or node failures) is insufficient. Without a more general model for consensus, gaps in the ability to detect, isolate, mitigate, and recover HPC applications efficiently will arise. Focusing on crash failures is insufficient because a chain of underlying components may lead to system failures in MPI. What is more, clusters and leadership-class machines alike often have Reliability, Availability, and Serviceability Systems (RAS) to convey predictive and real-time fault and error information, which does not map strictly to process and node crashes. A broader study of failures beyond crash failures in MPI will thus be useful in conjunction with consensus mechanism for developers as they continue to design, develop, and implement fault-tolerant HPC systems that reflect observable faults in actual systems. We describe key factors that must be considered during consensus-mechanism design. We illustrate some of the current MPI fault tolerance models based on use case. We offer a novel classification of common consensus mechanisms based on these factors such as the network model, failure types, and based on use cases (e.g., fault detection,

Grace Nansamba
University of Tennessee at Chattanooga
E-mail: jpp751@mocs.utc.edu

Amani Altarawneh
Colorado State University
E-mail: amani.altarawneh@colostate.edu

Anthony Skjellum
University of Tennessee at Chattanooga
E-mail: tony-skjellum@utc.edu

synchronization) of the consensus in the computation process, including crash fault tolerance as one category.

Keywords consensus mechanisms · fault tolerance · replication · synchronization · fault detection · message-passing model

1 Introduction

High performance computing (HPC) requires the aggregation of processing power to solve large science, engineering, and business problems. Processes' interactions in HPC create complexity because of the number of processors used in a given computation and the need for coordination between them (e.g., shared memory and message passing). Interprocess communication may cause faults that lead to system deadlocks, system halts, and unexpected output.

Consensus is a fundamental building block for fault-tolerant HPC systems. Processes in an MPI [Forum(2015)] program need to utilize consensus mechanism to avoid, prevent, and mitigate faults. A consensus model is a mechanism used to make shared decisions. These decisions are mainly for fault tolerance, replication and synchronization[1]. Because of the significant roles of consensus in HPC systems, several mechanisms have been designed to fit application purposes and requirements. Consensus ensures agreement between processes and maintains consistency during interprocess communication in peer models such as MPI provides. Many novel consensus mechanisms have been designed for distributed systems and have been extended for use in HPC systems. However, there is a need to study and classify consensus mechanisms to fill apparent gaps and to design new, reliable consensus mechanisms for HPC systems (and especially for a fault-tolerant MPI) [Hassani et al.(2014)Hassani, Skjellum, and Brightwell, Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror]. Consensus mechanism designers and developers need to be aware of various factors during the mechanisms designing process, and taking to account that the new design must not increase the system overhead or affect the current system scalability [Altarawneh et al.(2020)Altarawneh, Herschberg, Medury, Kandah, and Skjellum].

However, designing a byzantine consensus mechanism is a challenging task because of the complexity of the type of failures that need to be considered during the parallel computation process. There is a need to study the robustness and reliability for such mechanisms. Byzantine failures prevent peer processes from reaching agreement on a given value, which introduces another level of messages integrity requirements [Altarawneh and Skjellum(2020)].

The main contributions of this paper are 1) an explanation of the factors that must be considered during consensus-mechanism design; 2) A description of the types of failures that are encountered in HPC systems beyond crash failures; 3) an illustration of some of the current MPI fault tolerance models

[1] Consensus under fault-free operations is also an inherent property of typical bulk-synchronous parallel programs / data-parallel programs.

based on how they are used; 4) a novel classification of common consensus mechanisms based on these aspects such as the network model, failure types, and based on use cases (e.g., fault detection, synchronization) of the consensus in the computation process.

The remainder of this paper is organized as follows: Importance of this classification 2, background and related work are described in Section 3. Factors that arise during the consensus mechanism design are described in three subsections as follows: network model, usage, and failure source and types in section 4. MPI fault tolerance models are described in 5. Section 6 describes a consensus mechanism classification for HPC. Finally, Section 7 offers conclusions and future work.

2 Why this classification is important

A fault-model-relevant classification of consensus mechanisms provides a useful basis for determining the appropriate consensus algorithms for a variety applications in HPC. The classification presented in this paper is important for the design of consensus algorithm that are fit for purpose and a possibility to design robust generalised consensus models that can be used for different applications. Based on the fact that the classification highlights factors during consensus mechanism design in section 4, a consensus mechanism tailored toward particular usage or network model or failure type can be applied in different HPC applications.

Classifying consensus mechanisms in the HPC domain is a contribution toward achieving fault tolerance for HPC models and applications. The examples of the HPC fault tolerant models and the consensus algorithms they use provides rationale for which other new and old models can be improved. Further, the classification suggest that MPI programs should be re-imagined to have appropriate consensus mechanisms in concert with the faults to be addressed.

We reviewed the consensus mechanisms proposed so-far and we present a classification of the popular consensus mechanisms in HPC. The classification focuses on how useful consensus algorithms are in fault tolerant HPC models such as MPI Stages [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror], ULFM [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra] and FAMPI [Hassani et al.(2014)Hassani, Skjellum, and Brightwell]. The classification will enhance the application and use of consensus algorithms in HPC to design fault tolerant and thus scalable and better performing applications.

3 Background and Related Work

In this section, we describe background and related work. First, RAS (Reliability, Availability, and Serviceability) [Wikipedia contributors(2021c)] Systems provide a rich source of multifarious fault data, including predictive fault information in certain cases. This data source motivates the need for fault models that mirror the complexity and variety of available information at runtime. Second, in complement to fault information, consensus algorithms provide peer processes and threads in an MPI application and/or runtime system with the ability to reach agreement[2] on key state, such as program status and observed faults. Understanding the state of the art is important, as well as potential need for extensions when considering new forms of fault information.

3.1 RAS Systems and Fault Data

We note that RAS systems and mechanisms (Reliability, Availability, and Serviceability) offer rich, timely information[3] about predictive and detected failures in clusters and leadership-class machines, including IPMI [Manage Engine(2021)] and SNP-traps [DNS Stuff(2021)], and Linux/OS event logs, that jointly provide node, network, disk, memory, and other levels of temporary and permanent failure information. Because such data are available, and are usually not exploited in existing MPI Fault Tolerant concepts and systems, there is room for rethinking and extending fault-tolerant MPI approaches. We note that our generalized taxonomy meshes well with this rich, real-time source of fault information [Scargall(2020), Wikipedia contributors(2021c)].

3.2 Consensus Mechanisms

Consensus is a fundamental computing problem in distributed systems; it has been explored over the past 30+ years. It is required for agreement between different actors (e.g., processes, threads, devices, ...) about state changes within distributed systems. Consensus is required to ensure consistency and reliability of a given system. Many researchers have designed, implemented and classified consensus mechanisms for distributed systems that are fit for particular purposes. Most of these classifications, of late, have been in the blockchain technology area because of the recent popularity of this technology for cryptocurrency applications [Bano et al.(2019)Bano, Sonnino, Al-Bassam, Azouvi, McCorry, Meiklejohn, and Danezis].

Blockchain is a well-known cryptocurrency technology that is now extended to other non-cryptocurrency applications in distributed systems [Cachin and Vukolić(2017)]. Consensus is applied in a blockchain protocol to reach agreement on the miner (peer participant in advancing the underlying digital ledger state, a process called mining). Miners cooperate/compute in order to *mine* the next data block and when committing blocks and transactions validation.

[2] Reaching agreement is never guaranteed in theory, but often possible heuristically in practice (cf, FLP [Borowsky and Gafni(1993)]).

[3] There can be security concerns about enabling a parallel program to receive fault information from the exterior of the parallel system. Coping with any possible covert channels through translation and vetting of such information appears tractable in practice.

Consensus algorithms have recently been classified by some of us and others [Altarawneh et al.(2020)Altarawneh, Herschberg, Medury, Kandah, and Skjellum,Bano et al.(2019)Bano, Sonnino, Al-Bassam, Azouvi, McCorry, Meiklejohn, and Danezis,Cachin and Vukolić(2017)] according to different criteria. They have been classified as either leader-based or voting-based. Leader-based algorithms notionally have a leader actors/process that is in charge of broadcasting to other nodes (miners) and committing state changes. The classification was further categorized into *competitive* and *collaborative* consensus. Examples of leader-based algorithms in [Altarawneh et al.(2020)Altarawneh, Herschberg, Medury, Kandah, and Skjellum] include; proof of work (PoW) [Dwork(1993)], proof of stake (PoS) [King and Nadal(2012)], proof of elapsed time (PoET) [Intel(????)] and RAFT [Ongaro and Ousterhout(2014)]. Voting-based consensus means that the processors/actors vote for state change or can be both representative and gossip voting. Some of these are Delegated Proof of Stake (DPoS) [Fan and Chai(2018)], Practical Byzantine Fault Tolerance (PBFT) [Castro and Liskov(1999)], Hotstuff [Yin et al.(2019)Yin, Malkhi, Reiter, Gueta, and Abraham], LibraBFT [Baudet et al.(2019)Baudet, Ching, Chursin, Danezis, Garillot, Li, Malkhi, Naor, Perelman, and Sonnino], hashgraph and the gossip protocol [Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann] , and tangle and weight protocol [Popov(2016)].

Consensus has been used to attain fault tolerance in MPI applications but no classification has been conducted for HPC consensus mechanisms of which we are aware. A classification will aid researchers in the design of fault tolerant consensus mechanisms for HPC system. Consensus in HPC is useful for agreement about the systems state and for making decision on whether to commit or abort state changes, particularly when coping with a transition from a failure state to a new, non-failure state.

Al-Mamun et al. [Al-Mamun et al.(2019)Al-Mamun, Li, Sadoghi, Jiang, Shen, and Zhao] proposed HPChain, an MPI-based blockchain framework designed for HPC systems that employs a new consensus protocol. The protocol requires only 51% of the processes to be correctly running in order for the distributed system performance to continue which implies a level of fault tolerance. HPChain aims to achieve immutability, decentralization, and reliability of HPC data by utilizing blockchains, which boosts data fidelity. HPChain leverages the data provenance that are stored in the blockchain to tolerate failures caused by faulty MPI processes.

In 2020, Al-Mamun and Zhao [Al-Mamun and Zhao(2020)] designed a prototype, BAASH (blockchain-as-a-service framework for HPC). This is a framework of consensus protocols and a fault-tolerant subsystems for MPI to leverage the advantages of blockchain; they overcame the limitations of applying blockchains into HPC (i.e., consensus protocols and serialized I/O subsystems). Earlier, Buntinas [Buntinas(2012)] described a scalable distributed consensus algorithm for fault tolerant features in MPI. It is a ballot-based algorithm where the root broadcasts the ballot, agree message and commit message to other processes using an MPI validate function. Buntinas [Darius(2012)] described a scalable, distributed consensus algorithm that is used to support

MPI fault-tolerance features in MPI-3. Other distributed consensus algorithms are described in [Ranganathan et al.(2001)Ranganathan, George, Todd, and Chidester, Herault et al.(2015)Herault, Bouteiller, Bosilca, Gamell, Teranishi, Parashar, and Dongarra, De Camargo and Duarte(2016)], and elsewhere.

4 Factors during consensus-mechanism design

In this section, we explain the essential aspects or factors that impact consensus-mechanism design. These factors include network model, usage, and failure source and types.

4.1 Network model

A network is a major component in HPC systems because of the presence of many processors that demand use of a network to support communication among peer processes or clusters. In this section, we discuss three network models in HPC: synchronous, asynchronous, and partially synchronous:

– **Synchronous network model**: In the synchronous HPC model, messages from one process to another are expected to be delivered after a finite amount of time . The message passing between processes is in real time and significant delays are reported as process failures. A classic example of synchronous HPC models is the Bulk Synchronous Parallel (BSP) model, which is used in many HPC applications [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror]. In MPI, the synchronous operations block a process till the operation completes and completeness means that the message has been delivered to the receiving process (but not yet necessarily to the receiving buffer).
– **Asynchronous network model**: Messages in asynchronous models in HPC are not time bound. Messages delivered between a pair or a group of processes can be delayed by an unknown and sometimes an infinite amount of time. In MPI, asynchronous message passing is non-blocking and it only initiates the operation and does not need to progress/transfer them immediately. This allows more parallelism in asynchronous operations than in synchronous.
– **Partially synchronous network model**: A partially synchronous model is at the intersection of the two models described above. In this model, messages between processes are delivered in an undefined but finite amount. There is an upper time bound for messages to be delivered from one processor to another that is not known priori [Dwork et al.(1988)Dwork, Lynch, and Stockmeyer]. Fault tolerant protocols use distributed clocks to allow partially synchronous processes to agree on time. Distributed clocks are fault-tolerant variations on the clock as described by Lamport [Lamport(1983)]. This model is more practical in HPC environments as compared to the asynchronous model. This model is a viable solution to the

FLP impossibility result [Borowsky and Gafni(1993)] and thus more applicable and preferred in HPC systems [Moise(2011)].

4.2 Usage

Consensus is useful in HPC environments for synchronization, replication, fault tolerance, decision-making, and optimization. All these operations are applied at different stages in the life cycle of an HPC application. We describe of the use-cases where consensus is applied in HPC environments and particularly in MPI.

4.2.1 Synchronization

In distributed systems, process or node synchronization is fundamental. This is because peer processes must agree on certain system state to ensure reliability of the entire system. Synchronization is important when failures occur in order to make proper recovery decisions. Synchronization is applied when many processes must agree on a single value in order for the system to continue working correctly. Consensus is used to achieve this agreement between the different processes. In nonblocking MPI operations, synchronization is achieved using MPI calls such as `MPI_Wait()` and `MPI_Barrier` or other synchronizing MPI collectives such as `MPI_Allreduce`. In MPI Stages [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror], an agree/commit consensus approach was used such that the live processes synchronize with the relaunched processes during process recovery. In the FA-MPI model [Hassani et al.(2014)Hassani, Skjellum, and Brightwell], synchronization is the third phase in the process of achieving fault tolerance. This is a result of the global state of MPI that requires all peers to remain consistent at the end of every transaction. This phase is also referred to as error synchronization since all the error information should be synchronized among all peers.

4.2.2 Replication

MPI applications can use replication as an alternative to checkpoint/restart in case of failures, as proposed by Ropars et al. using an intra-parallelization technique [Ropars et al.(2015)Ropars, Lefray, Kim, and Schiper]. The intra-parallelization was designed to overcome the resource requirements of full replication through work-sharing between replicas. The algorithm was based on task parallelism where the work load is shared between replicas of a logical process and message passing done with MPI. Process replication was examined and evaluated as a viable solution for fault tolerance and reliability of exascale systems [Ferreira et al.(2011a)Ferreira, Stearley, Laros, Oldfield, Pedretti, Brightwell, Riesen, Bridges, and Arnold]. This research mainly focused on MPI applications and MPI process replication, which require consistency between replicas. They designed a simple MPI library, rMPI, that replicates each MPI process (in `MPI_COMM_WORLD` in an application and in case of failure of original MPI process, the replicas continue. The protocols used for consistency apply the leader-based consensus since they have a leader node for each replicated MPI process and the non-leader processes/nodes hold the replicas. From their results, the state-machine replication approach performed better than classic fault rollback recovery techniques such as checkpoint/restart [Sankaran et al.(2005)Sankaran, Squyres, Barrett, Sahay, Lumsdaine, Duell, Hargrove, and Roman].

Ferreira et al. [Ferreira et al.(2011b)Ferreira, Stearley, Laros, Oldfield, Pedretti, Brightwell, Riesen, Bridges, and Arnold] presented an evaluation for use of state machine replication mechanism as an alternative to checkpoint-restart for fault tolerance in exascale sytems. Woo Son et al. presented a block replication approach where the data is stored redundantly using replication aware derived MPI data types [Woo et al.(2011)Woo, Lang, Latham, Ross, and Thakur]. Transparent replication was achieved within MPI-IO via the profiling interface to MPI (PMPI). This research showed that parallel file system redundancy through block replication creates a more reliable MPI-IO layer, which provides overall better reliability for MPI applications.

The consensus mechanisms that use state machine replication that we described in this paper are mainly for distributed systems (i.e., Paxos [Lamport(1998)], RAFT [Ongaro and Ousterhout(2014)] and PBFT [Castro and Liskov(1999)]).

4.2.3 Faults detection and recovery

There is drastic increase in failure rates arising from increased parallelism in HPC systems. This has motivated the design and implementation of MPI models that are fault tolerant; that is, they can detect failures and support recovery of processes in case of failure. Consensus mechanisms are required in the design of fault-tolerant models, some of which are described in this section.

The ULFM MPI model was designed with capabilities of failure reporting a mechanism for recovery using defined ULFM constructs [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra]. Some of the ULFM constructs include; `MPI_COMM_REVOKE` for resolving non-uniform reporting and `MPI_COMM_SHRINK` for creating a new functional communicator thus recovery. A special construct `MPI_COMM_AGREE` was used for consensus among alive processes to ensure consistent state at completion by returning a boolean value.

FA-MPI (Fault Aware MPI) is a model extension to MPI that adds transactions for failure detection, isolation and recovery [Hassani et al.(2014)Hassani, Skjellum, and Brightwell]. The FA-MPI model uses a `TryBlock` function to detect and propagate failure information in non-blocking transactions. It also includes a timeout mechanism to report when operations successfully or unsuccessfully finish. HPC applications that apply FA-MPI have high chances of running to completion compared to nominal MPI execution that is non-fault aware. FA-MPI uses a fault tolerant Allgatherv/Allreducev protocol for consensus agreement. This creates a list of all failures gathered from all MPI processes (in a given communicator's scope/group) involved in the MPI operation, and they can be reported as error codes to other MPI processes. The list of failures is broadcast to all other processes to create awareness of failures. It further has a query for failure functionality with which the user can retrieve information about failures in the system.

4.2.4 Making decisions

The presence of multiple processes working on the same problem in HPC demands decision-making at all stages of the parallel task. The common decisions made in HPC systems are commit or abort decisions about system state, transaction (e.g., a value to be agreed on) and failure recovery decisions. Consensus mechanisms are essential while every process submits their local decision to the global knowledge of all other peer processes.

4.2.5 Optimization

Designing of algorithms in distributed large-scale system requires optimization. Optimization involves comparisons between different parameters in the algorithms. The optimization problem can be formulated as a consensus problem as showed in Chang et al. and Boyd et al. [Chang et al.(2016)Chang, Hong, Liao, and Wang, Boyd et al.(2011)Boyd, Parikh, Chu, Peleato, and Eckstein]. A given algorithm can be more performant than another depending on the criteria used to optimize consensus. Some of the optimization problems include the global consensus problem where all the local variables should agree.

4.3 Failure source and types

HPC systems are prone to failures that are caused by human errors, hardware failures and software issues in the operating system. Failure in HPC systems lead to substantial performance degradation. Several failure classifications have been proposed for distributed systems. De Angelis [De Angelis(2018)] proposed the Byzantine failures model, a hierarchy showing sub classes of failure in the following ascending order. Fail-stop fault, crash fault, omission fault, timing fault, incorrect computation fault, authenticate Byzantine fault and Byzantine fault (super class).

4.3.1 Crash Failures

Crash failures happen when a compute node or process becomes unresponsive. The process stops abruptly and fails to resume [Wikipedia contributors(2021b)]. General examples of crashes in computing systems include process crash, operating system crashes, device driver crashes, application deadlocks, and hardware failures [Leners et al.(2011)Leners, Wu, Hung, Aguilera, and Walfish]. Crash failures in synchronous systems can be detected due to timeout, but in pure asynchronous systems it is impossible to solve even a single process crash failure [Moses and Raynal(2009)]. This is also called the FLP impossibility result [Altarawneh and Skjellum(2020)]. An example of crash failure in HPC systems is the process failure or fail-stop and crash [Amin(2014)]. Process failure occur when MPI codes on computational nodes in a cluster and the processes involved in an MPI job fail. Process crashes can be a result of hardware, network or software failure, making one or more processes unresponsive. Process failures are also defined as failures which occur even in the presence of a reliable connection channel between processes [Rel(1999)] due to other software or human errors.

4.3.2 Byzantine Failures

Byzantine failure results from erroneous behavior of the processes or nodes. This can be caused by malicious communications from particular nodes. In 1982 Lamport et. al. described the byzantine fault using the byzantine generals problem in 1982 [Lamport et al.(1982)Lamport, Shostak, and Pease]. Due to byzantine faults, byzantine failures occur. This problem states that in a distributed system, some of the processes, imaginary, referred to as 'lieutenants and general' are malicious and cannot be trusted during peer communication. In order to make a final decision about communication a super-majority from processes in the distributed system need to come to a consensus. A super-majority must be greater than $2/3rd$ of the nodes in the system. If a third or greater of nodes are malicious, then the system is susceptible to failures. From Driscol et al. [Driscoll et al.(2004)Driscoll, Hall, Paulitsch, Zumsteg, and Sivencrona] a byzantine fault is a fault presenting different symptoms to different observers whereas a byzantine failure is the loss of a system service due to a byzantine fault in systems that require consensus. Altarawneh et al. affirmed that any failure that prevents nodes or processes from agreeing on a value in a system is a byzantine failure [Altarawneh and Skjellum(2020)].

4.3.3 Hardware failures

Hardware failures [El-Sayed and Schroeder(2013)] occur when a hardware component malfunctions and/or stops working because of hardware errors. Some of these errors undetected by hardware become more frequent as computing systems scale up into exascale and may eventually affect many computations [Snir et al.(2014)Snir, Wisniewski, Abraham, Adve, Bagchi, Balaji, Belak, Bose, Cappello, Carlson, Chien, Coteus, Debardeleben, Diniz, Engelmann, Erez, Fazzari, Geist, Gupta, Johnson, Krishnamoorthy, Leyffer, Liberty, Mitra, Munson, Schreiber, Stearley, and Hensbergen]. Hardware errors are classified into hard errors and soft errors. Hard errors are physical defects that cause malfunctions and the system stops working such as power supply or fan failure.

Silent memory errors are errors that will not be detected yet they corrupt memory while the application continues to operate and eventually wrong results are reported. They are also referred to as soft error which are transient failures that occur in memory and are not correctable by Error Checking and Correcting (ECC) such as bit flips. Chipkill [Wikipedia contributors(2021a)] is an advanced ECC technology that protects the computer memory system from errors that arise from a single memory chip [Wikipedia contributors(2021a)]. Chipkill has been used in HPC environment to design approaches that use prediction to proactively avoid memory errors [Costa et al.(2014)Costa, Park, Rosenburg, Cher, and Ryu, Schroeder and Gibson(2009)].

4.3.4 Timing failures

these failures occur in synchronous HPC systems when the response time of a process exceeds the expected time range. These failures lead to delays in the system since other processes could be waiting for a communication from that particular delaying process. Omission failure occurs when a node's response is infinitely late. The node may fail to send messages or receive massages [1000projects.org(2021)].

4.3.5 Software failures

these failures are associated with the system software and they occur when the software prevents the system from functioning properly [Schroeder and Gibson(2009)].

4.3.6 Network failures

these refer to all incidents that cause network downtime ranging from poorly configured network devices to cable damages. The network requires regular maintenance to prevent failures [Schroeder and Gibson(2009)].

4.4 HPC Models ability to terminate

Termination refers to a state when all processes in a distributed system finish their tasks and become idle. It is a fundamental feature of a consensus algorithm which should be considered in the design of HPC consensus models and algorithms. Termination is one of the properties of a fault tolerant consensus protocol, others including integrity and agreement. Termination detection is a popular problem of study in HPC and distributed systems.

Termination can be achieved and applied to HPC through termination detection algorithms. Huang proposed the original Credit Distribution Algorithm in 1989 [Huang(1989)] which uses messages with weights to determine termination. Bosilca et. al , [Bosilca et al.(2021)Bosilca, Bouteiller, Herault, Le Fèvre, Robert, and Dongarra] defined termination detection algorithms as distributed algorithms that observe that the global state has been reached and then announces it to all processes. Some examples of termination detection algorithms for HPC discusses in [Bosilca et al.(2021)Bosilca, Bouteiller, Herault, Le Fèvre, Robert, and Dongarra] include; Huang's CDA (HCDA) where messages carry credit between processes. The credit that each process has is calculated as its weight, each process starts with an initial credit weight which decreases or increases depending on the number of messages it has sent and received. Termination is detected when the process weight equals its initial total credit. The Four Counters wave algorithm is another algorithm which involves propagation of messages throughout the nodes using counters to record the number of sent and received messages. Each process has an up or down state and when a process is idle, it informs all other processes in the system. Using these messages and counters, the process sends a stop message to the parent or root of the tree and this invokes termination. The Efficient Delay-Optimal Distributed algorithm (EDOD) aims at achieving optimal detection delay for the communication patterns while passing messages. All children processes send stop message to the parent which becomes idle and announce termination. It uses acknowledgement messages to prevent early termination.

5 MPI Fault Tolerance Models

In this section, we provide common MPI fault tolerance models that are used for different uses cases such as synchronization, and detect failures. Many powerful advances and extensions have been added to the MPI protocol, to support the design of fault tolerant MPI. Each of the models such as: ULFM [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra], MPI Stages [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror], and FA-MPI [Amin(2014)] require a consensus mechanism during their life cycle.

5.1 ULFM for failure detection and recovery

The User Level Failure Mitigation (ULFM) [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra] is the proposed fault tolerant model by the MPI Forum which uses error codes to recover from failure. This is a post-failure recovery model with the necessary flexibility for the implementation of fault tolerant MPI applications [Losada et al.(2020)Losada, González, Martín, Bosilca, Bouteiller, and Teranishi]. ULFM's design rationale includes failure detection, communicator revocations and reconfiguration constructs (fault tolerance routines) to restore the communication among the processes and allow continuation of program execution thereafter. Some of the constructs are as follows: `MPIX_COMM_REVOKE` for failure reporting and ensuring that all processes will be notified of processes failure, `MPI_COMM_SHRINK` which creates a new functional communicator excluding the failed processes [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra].

5.2 MPI stages for synchronization and decision-making

MPI stages is a fault tolerant failure recovery approach that significantly reduces the recovery time of Bulk Synchronous applications [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror]. In the implementation of MPI stages, MPI state checkpoint supports transparent replacement of a failed process. The MPI state and application checkpoint are stored and loaded to allow all the processes to start execution from the main computation loop instead of from the main program, thus achieving the goal of MPI stages to reduce recovery time. During failure recovery, failed processes are replaced by new instance of the process. The MPI runtime system is notified by the failure detector to replace the failed process with a new process and relaunch it.

5.3 FA-MPI for synchronization

FA-MPI [Hassani et al.(2014)Hassani, Skjellum, and Brightwell] is a lightweight transactional model that uses fault awareness to decide the level of fault tolerance. FA-MPI supports failure detection, isolation, mitigation, and recovery for non-blocking communication operations. The model includes the *TryBlock*, a fundamental function which is used to try operations and decide on whether to commit when all operations succeed else roll-back or roll-forward when some fail [Hassani et al.(2015)Hassani, Skjellum, Bangalore, and Brightwell]. The *TryBlock* can be applied to three different transaction levels (i.e., local, group-wise, and an in-between mode) which make FA-MPI per transaction fault-aware.

All the above models are designed to fit particular application purposes and we are not able to generalize a single consensus mechanism for all MPI applications. However, with a classification of the common consensus mechanisms based on network model, failure types, and on usage researchers will be able to fill the gaps in the future designs of fault tolerant MPI applications.

6 Consensus classification in HPC

In this section, we provide a classification of common consensus mechanisms based on the usage, the network model, and the type of failures, see Figure 1.

HPC system reliability is a fundamental subject as the systems continue to expand even to Exascale, systems that are complex and more challenging to design. Researchers are focusing on design of fault tolerant systems based on the failure statistics from present HPC systems. The different types of failures as described in Section 4 have different impact on the reliability of HPC systems. Consensus is useful for coordination of processes in HPC systems to attain reliability in the presence of faults or failure. It is at the center of various fault tolerant mechanisms in HPC systems since there is always an agreement required in the presence of many processes or nodes.

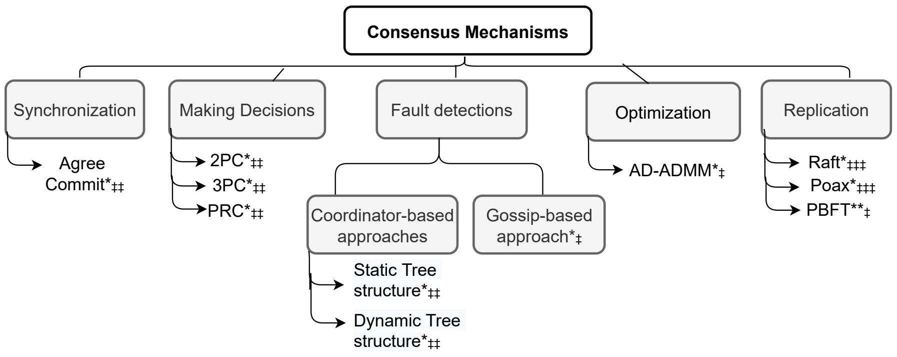


Fig. 1 A Novel HPC Consensus Algorithms Classification based on usage. * – Crash fault tolerant, ** – byzantine fault tolerant, ‡ - Asynchronous, ‡‡ – Synchronous, ‡‡‡ – Partially Synchronous, note that the only asynchronous

The Message Passing Interface (MPI) is a popular communication protocol used in HPC for transmitting messages between peer processes that forms a parallel job execution environment. Consensus is important in the design of fault tolerant MPI, such as MPI Stages [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror]. In MPI Stages, consensus was used to achieve synchronisation. An agree/commit approach of consensus was used to agree on a value called the epoch value for MPI Stages that significantly reduced on the recovery time after failure. Consensus mechanisms have a vital role in fault tolerant applications [Hursey et al.(2011)Hursey, Naughton, Vallee, and Graham] such as Bulk synchronous applications [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror].

State Machine Replication is an approach used in building fault tolerant systems. A state machine stores the state of a systems at any point during process transaction. HPC involves the use decentralization and distributed systems where state machines are required. State machine replication means that the functioning components are replicated through redundancy that supports availability of the system. It supports liveliness and safety in case the replicas are faulty at the same time. Consensus is needed to synchronize the different replicas of the state machine to ensure consistence of the system. Some of the consensus mechanisms such as PBFT and Paxos are classified as state machine replication [Duan(2016)]. Consensus mechanisms are described as follow:

– **agree/commit approach**: This approach was designed and implemented for consensus in MPI Stages, for synchronous applications [Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror]. The value to agree on is the epoch value, which is a key in the MPI Stages implementation. The epoch is a variable, which is used to differentiate between first time processes (epoch value must be zero) and relaunched processes. In this consensus approach, the coordinator gathers all decisions from live processes and sends an agree or disagree message to the head coordinator. The MPI processes compare and to agree on the epoch value, send it to the coordinatorl, which then forwards to all live processes.

- **Partially Reliable consensus (PRC)**: This synchronous mechanism was described and proved by Amin Hassani [Amin(2014)]. This is a centralised approach with one coordinator and many groups. PRC was designed to agree on the correct MPI state in regard to the communicator. It used two reliable operations; reduce and broadcast. In the reduce phase, live process form a list of local errors and forward it to the coordinator with a flag showing the MPI state; that is, failure or no failure. In the reliable broadcast operation, the coordinator broadcasts the list to all love processes. PRC also involves unsolicited voting in contrast to 2PC and 3PC where all live processes send votes to the coordinator. Further in his thesis, Hassani discussed a Better Partially Reliable Consensus (BPRC). This extension overcomes the inconsistencies in PRC where the coordinator fails before broadcasting its final decision. The mechanism uses sub-coordinators, which are basically duplicates of the coordinator and are responsible for sending the final decision to other processes in case of coordinator failure.
- **Two-phase commit protocol (2PC)**: Two-phase commit protocol (2PC) is a specialised synchronous consensus protocol that coordinates all participating processes in a distributed transaction to decide on whether to commit or abort the transaction. In 2PC, one of the nodes is assigned as the coordinator and the rest of the nodes are designated as the participants. The coordinator is responsible for making the final decision after receiving response from the participants. 2PC algorithm has two phases; the commit request or voting phase where the coordinator sends a request to commit message to all participants and waits, the participants respond with an agreement message.
  The second phase is the commit or completion phase where the coordinator sends a commit message to all participants, each participant sends an acknowledgement to the coordinator, which completes the transaction. The shortcoming of 2PC as described in Hassani's dissertation [Amin(2014)] is that if the coordinator fails after making the decision and before broadcasting it, then all the participants should wait until the coordinator recovers making 2PC a blocking protocol. There are variants of the 2PC protocol based on the recovery mechanisms in case of failure and protocol optimizations.
- **Three-phase commit (3PC)**: 3PC is a more failure tolerant protocol that overcomes the limitation of 2PC by introducing a third prepared to commit or ready to commit phase. The algorithm is designed in that before the commit phase the coordinator sends a prepared to commit message to participants and are aware of the decision to commit. In case of coordinator failure before the commit phase, but after the ready to commit phase, other participants commit the transaction on timeout, thus overcoming the blocking of 2PC [Amin(2014)]. 3PC is classified as a synchronous protocol.
- **Coordinator-based Approaches**: the processes in this approach are divided in that one of the processes is a coordinator and others are partic-

ipants. It can also be referred to leader based Katti et. al. classified the coordinator based approach [Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann] by transforming the original 2PC and 3PC to be fault tolerant over a tree structure; that is over a static tree structure and over a dynamic tree structure. The communication in this approach is synchronous.
A) Over a static tree structure [Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann]: this structure applies to the 2PC consensus protocol. The coordinator is at the root of the tree and makes the final decision after receiving all the votes from participants. In the first phase, the coordinator uses a gather operation to gather votes from the participants at the leaves through the intermediate parent. In the second phase, the decision is broadcast from root to leaves. In case of failure of the leaves, the parent recursively adopts its children and if a parent fails, the child queries its grandparent for updates, the child can vote again if the parent fails before broadcasting its vote. If the coordinator fails after broadcasting the decision, a termination algorithm is called and the preceding parent reports success or abort based on the termination status. If the coordinator fails before propagating, then the tree is re-balanced with new alive processes.
B) Over a dynamic tree structure [Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann]: this uses the 3PC consensus protocol combined with reliable broadcast algorithm that constructs broadcast tree dynamically. The three phases are Balloting, broadcasting the agree message and broadcasting the commit message. The second phase, overcomes the challenges that would occur if the coordinator fails before broadcasting the final decision. The algorithm uses several messages such as `REJECT, ACK, ACCEPT` during the balloting phase to ensure proper coordination as the root broadcasts the ballot plus a list of failed processes to the child(ren). If the root fails, the processes with the highest rank among the failed appoints itself as next leader.

– **Gossip-based Approaches**: In the Gossip based consensus mechanism, each processes randomly picks neighbors and shares information in an asynchronous format. There are no leaders or followers but all processes are peers each knowing the information about all other processes. Two gossip based fault tolerant algorithms for HPC, which are intrinsically fault tolerant were proposed in [Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann]. The algorithms are built on the `MPI_COMM_SHRINK()` operation of UFLM [Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra]. Each process detects failure by randomly pinging a process periodically also referred to as stochastic pinging. It involves a gossip cycle in which a process p randomly selects to ping another process q, q must reply before end of the cycle to prove that it is alive else it has failed. Consensus is achieved by maintaining global knowledge at each MPI process using a matrix to store status of all other processes. Another way to achieve consensus is through an efficient heuristic method where each process maintains a list of failed processes of which it is aware.

- **AD-ADMM (Asynchronous Distributed- Alternating Direction Method of Multipliers)**: This is an asynchronous mechanism that uses a star topology to solve consensus problems in parallel mannerS [Chang et al.(2016)Chang, Hong, Liao, and Wang]. In their model, a master node coordinates the rest of the distributed nodes referred to as 'workers' on the topology. The coordinator can make decisions based on a partial set of nodes and does not need to wait for all the nodes on the network. This mechanism overcomes the delays and idle time in synchronous models as there is need to wait for the slowest worker to send information. The key purpose of the AD-ADMM mechanism is optimization of communication performance and how the network may impact the parallel computation.
- **Paxos**: This is a classic algorithm published in 1998 by Leslie Lamport [Lamport(1998)]. The Paxos algorithm guarantees that a set of machines will choose a single proposed value as long as majority of the nodes are available. Paxos is an example of state machine replication, since copies of the same value are shared among the participating processes. This is done in a partially synchronous manner.
  The algorithm uses agents; that is, proposers, 'acceptors' and learners and these are processes or nodes. These processes are involved in two phases per successful round with assumptions that the communication between processes is asynchronous. To ensure safety in the Paxos algorithm validity is enforced since only proposed values are chosen and only one proposed value is selected. The original algorithm solves crash failure [García-Pérez et al.(2018)García-Pérez, Gotsman, Meshman, and Sergey] but improvements have been made enable handling of byzantine failures [Castro and Liskov(1999)].
- **Raft (Reliable, Replicated, Redundant And Fault-Tolerant)**: This algorithm was developed by Diego Ongara during his PhD dissertation with novel features such as: strong leader, leader election, and membership changes. It produces the equivalent results and is as efficient as Paxos [Lamport(1998)] but its structure is more understandable [Ongaro and Ousterhout(2014)]. Raft uses a replicated log, which contains information about the state changes of each node. Raft divides the process into subsections: leader election that happens when the current leader fail or when the leader's term ends. The leader must receive majority of votes from other nodes. The message flow between leader and follower is maintained through a heartbeat mechanism where the leader regularly sends a message about its existence. When the follower stops receiving heartbeat messages, the randomized election timeout ensures that no two or more follower or nodes are being selected to be the new leader [Ongaro and Ousterhout(2014)].
  Secondly, the leader receives and accepts log entries from other nodes. The log entries include the state changes to the system. The leader broadcasts state changes and the followers replicate them and store an individual copy. The follower can reject a message if there are inconsistencies and if they are resolved the leader must go through the previous entry and broadcasts again. Once majority of the followers confirm that replication

of the log entry, the leader applies to its local state machine. The committed leader's state is also adapted by the rest of the follower and applied to their state machine thus log replication in all the nodes. Raft ensures safety through the use of state machine. Safety prevents malicious behavior from happening in the system. Raft is classified into the partially synchronous network model.

- **PBFT (Practical Byzantine Fault Tolerance)**: This is a standard consensus algorithm designed as a solution to Byzantine Fault Tolerance (BFT) problems in 1999 by Lisvok and Castro [Miguel Castro(2002)]. PBFT is an asynchronous mechanism that tolerates faults in a system as long as less than $\frac{1}{3}$ of the nodes are faulty. Its design and implementation is based on the state machine replication approach. There is no leader in PBFT and all nodes participate in voting for state changes in the system. The nodes are grouped into a primary node and replicas where the primary node acts the representative and the replicas have equal right to be representatives [Altarawneh et al.(2020)Altarawneh, Herschberg, Medury, Kandah, and Skjellum]. PBFT is energy efficient since it does not use complex mathematical computations and gives transaction finality as the transactions do not require multiple confirmations.

7 Conclusion and future work

In HPC, various consensus mechanisms that are fit for purpose have been designed. Some of these mechanism are used in fault-tolerant MPI applications mainly to solve crash failures or process failures. Common consensus mechanisms are mostly based on the synchronous network model. Some of the usage of consensus mechanisms in HPC include; failure detection, synchronization, replication, and optimization. However, to the best of our knowledge, these HPC consensus mechanisms are not classified.

In this paper, we explained the essential factors that must be considered during consensus-mechanism design as most of them applied to peer-models like MPI. We described the types of failures that are encountered in HPC systems beyond crash failures because the other failures eventually lead to process failures in HPC. We classified common consensus mechanisms based on various factors such as the network model, which is important when designing fault tolerant HPC applications. We presented a novel consensus classification, see Figure 1. A mechanism such as PBFT [Castro and Liskov(1999)] would be an ideal byzantine consensus mechanisms with an asynchronous underlying model; however, it is not scalable because of its inherently high overhead.

We discussed the current MPI fault tolerance models based on the their intended use. A classification according to use cases is useful in MPI environments to detect failures as soon as reasonably possible as they happen, synchronize the processes on the current system states, and make the processes able to make decisions and take actions in critical conditions. This paper is useful to researchers since it describes the various fault tolerant consensus mechanisms in HPC. Failures in HPC have been explained more broadly than process failures here, which is a common type failure that has been broadly studied in HPC. With the research in this paper, HPC researchers can recognize other types of failures and the factors to consider while designing a consensus mechanism and thus be able to detect, mitigate and recover from failures in a more effective and efficient manner.

The major outcome of this paper is that the availability of RAS data coupled with the fault classification and consensus mechanisms considered here suggests that fault-tolerant models for MPI programs should be re-imagined to use such external data together with generalized faults, not just process and node failure. Then, they should use appropriate consensus mechanisms in concert with the faults to be addressed. This remains as opportunities for future work.

8 Acknowledgement

This work was performed with partial support from the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, CCF-1617690, CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[Rel(1999)] (1999) On classes of problems in asynchronous distributed systems with process crashes. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, IEEE Computer Society, USA, ICDCS ’99, p 470

[1000projects.org(2021)] 1000projectsorg (2021) Types of failures in distributed systems. URL `https://www.dnsstuff.com/snmp-monitoring-tools`, [Online; accessed 23-May-2021]

[Al-Mamun and Zhao(2020)] Al-Mamun A, Zhao D (2020) BAASH: enabling blockchain-as-a-service on high-performance computing systems. CoRR abs/2001.07022, URL `https://arxiv.org/abs/2001.07022`, `2001.07022`

[Al-Mamun et al.(2019)Al-Mamun, Li, Sadoghi, Jiang, Shen, and Zhao] Al-Mamun A, Li T, Sadoghi M, Jiang L, Shen HT, Zhao D (2019) Hpchain: An mpi-based blockchain framework for data fidelity in high-performance computing systems

[Altarawneh and Skjellum(2020)] Altarawneh A, Skjellum A (2020) The security ingredients for correct and byzantine fault-tolerant blockchain consensus algorithms. In: 2020 International Symposium on Networks, Computers and Communications (ISNCC), pp 1–9, DOI 10.1109/ISNCC49221.2020.9297326

[Altarawneh and Skjellum(2020)] Altarawneh A, Skjellum A (2020) The security ingredients for correct and byzantine fault-tolerant blockchain consensus algorithms. In: 2020 International Symposium on Networks, Computers and Communications (ISNCC), pp 1–9, DOI 10.1109/ISNCC49221.2020.9297326

[Altarawneh et al.(2020)Altarawneh, Herschberg, Medury, Kandah, and Skjellum] Altarawneh A, Herschberg T, Medury S, Kandah F, Skjellum A (2020) Buterin’s scalability trilemma viewed through a state-change-based classification for common consensus algorithms. In: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), pp 0727–0736, DOI 10.1109/CCWC47524.2020.9031204

[Amin(2014)] Amin H (2014) Toward a scalable, transactional, fault-tolerant message passing interface for petascale and exascale machines PhD dissertation, The University of Alabama at Birmingham

[Bano et al.(2019)Bano, Sonnino, Al-Bassam, Azouvi, McCorry, Meiklejohn, and Danezis] Bano S, Sonnino A, Al-Bassam M, Azouvi S, McCorry P, Meiklejohn S, Danezis G (2019) Sok: Consensus in the age of blockchains. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies, pp 183–198

[Baudet et al.(2019)Baudet, Ching, Chursin, Danezis, Garillot, Li, Malkhi, Naor, Perelman, and Sonnino] Baudet M, Ching A, Chursin A, Danezis G, Garillot F, Li Z, Malkhi D, Naor O, Perelman D, Sonnino A (2019) State machine replication in the libra blockchain

[Bland et al.(2013)Bland, Bouteiller, Herault, Bosilca, and Dongarra] Bland W, Bouteiller A, Herault T, Bosilca G, Dongarra J (2013) Post-failure recovery of MPI communication capability: Design and rationale. Int J High Perform Comput Appl 27(3):244–254, DOI 10.1177/1094342013488238, URL `https://doi.org/10.1177/1094342013488238`

[Borowsky and Gafni(1993)] Borowsky E, Gafni E (1993) Generalized flp impossibility result for ¡i¿t¡/i¿-resilient asynchronous computations. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, NY, USA, STOC '93, p 91–100, DOI 10.1145/167088.167119, URL `https://doi.org/10.1145/167088.167119`

[Bosilca et al.(2021)Bosilca, Bouteiller, Herault, Le Fèvre, Robert, and Dongarra] Bosilca G, Bouteiller A, Herault T, Le Fèvre V, Robert Y, Dongarra J (2021) Revisiting credit distribution algorithms for distributed termination detection. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 611–620, DOI 10.1109/IPDPSW52791.2021.00095

[Boyd et al.(2011)Boyd, Parikh, Chu, Peleato, and Eckstein] Boyd S, Parikh N, Chu E, Peleato B, Eckstein J (2011) Distributed optimization and statistical learning via the alternating direction method of multipliers. Found Trends Mach Learn 3(1):1–122, DOI 10.1561/2200000016, URL `https://doi.org/10.1561/2200000016`

[Buntinas(2012)] Buntinas D (2012) Scalable distributed consensus to support mpi fault tolerance. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp 1240–1249, DOI 10.1109/IPDPS.2012.113

[Cachin and Vukolić(2017)] Cachin C, Vukolić M (2017) Blockchain consensus protocols in the wild. arXiv preprint arXiv:170701873

[Castro and Liskov(1999)] Castro M, Liskov B (1999) Practical byzantine fault tolerance. In: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, pp 173–186, URL `https://dl.acm.org/citation.cfm?id=296824`

[Chang et al.(2016)Chang, Hong, Liao, and Wang] Chang TH, Hong M, Liao WC, Wang X (2016) Asynchronous distributed admm for large-scale optimization—part i: Algorithm and convergence analysis. IEEE Transactions on Signal Processing 64(12):3118–3130, DOI 10.1109/TSP.2016.2537271

[Costa et al.(2014)Costa, Park, Rosenburg, Cher, and Ryu] Costa CHA, Park Y, Rosenburg BS, Cher CY, Ryu KD (2014) A system software approach to proactive memory-error avoidance. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, SC '14, p 707–718, DOI 10.1109/SC.2014.63, URL `https://doi.org/10.1109/SC.2014.63`

[Darius(2012)] Darius B (2012) Scalable distributed consensus to support mpi fault tolerance. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, pp 1240–1249

[De Angelis(2018)] De Angelis S (2018) Assessing security and performances of consensus algorithms for permissioned blockchains. arXiv preprint arXiv:180503490

[De Camargo and Duarte(2016)] De Camargo ET, Duarte EP (2016) Running resilient MPI applications on a dynamic group of recommended processes. In: 2016 Seventh Latin-American Symposium on Dependable Computing (LADC), pp 15–24, DOI 10.1109/LADC.2016.14

[DNS Stuff(2021)] DNS Stuff (2021) Snmp monitoring tools. URL `https://www.dnsstuff.com/snmp-monitoring-tools`, [Online; accessed 23-May-2021]

[Driscoll et al.(2004)Driscoll, Hall, Paulitsch, Zumsteg, and Sivencrona] Driscoll K, Hall B, Paulitsch M, Zumsteg P, Sivencrona H (2004) The real byzantine generals. In: The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576), vol 2, pp 6.D.4–61, DOI 10.1109/DASC.2004.1390734

[Duan(2016)] Duan S (2016) Building reliable and practical byzantine fault tolerance PhD dissertation, University of California Davis

[Dwork et al.(1988)Dwork, Lynch, and Stockmeyer] Dwork C, Lynch N, Stockmeyer L (1988) Consensus in the presence of partial synchrony. J ACM 35(2):288–323, DOI 10.1145/42282.42283, URL https://doi.org/10.1145/42282.42283

[Dwork(1993)] Dwork M Cynthiaand Naor (1993) Pricing via processing or combatting junk mail. In: Brickell EF (ed) Advances in Cryptology — CRYPTO' 92, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 139–147

[El-Sayed and Schroeder(2013)] El-Sayed N, Schroeder B (2013) Reading between the lines of failure logs: Understanding how hpc systems fail. In: 2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN), IEEE, pp 1–12

[Fan and Chai(2018)] Fan X, Chai Q (2018) Roll-dpos: A randomized delegated proof of stake scheme for scalable blockchain-based internet of things systems. In: Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, ACM, New York, NY, USA, MobiQuitous '18, pp 482–484, DOI 10.1145/3286978.3287023, URL http://doi.acm.org/10.1145/3286978.3287023

[Ferreira et al.(2011a)Ferreira, Stearley, Laros, Oldfield, Pedretti, Brightwell, Riesen, Bridges, and Arnold] Ferreira K, Stearley J, Laros JH, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011a) Evaluating the viability of process replication reliability for exascale systems. In: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12, DOI 10.1145/2063384.2063443

[Ferreira et al.(2011b)Ferreira, Stearley, Laros, Oldfield, Pedretti, Brightwell, Riesen, Bridges, and Arnold] Ferreira K, Stearley J, Laros JH, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011b) Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Association for Computing Machinery, New York, NY, USA, SC '11, DOI 10.1145/2063384.2063443, URL https://doi.org/10.1145/2063384.2063443

[Forum(2015)] Forum MPI (2015) MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015. High-Performance Computing Center Stuttgart, University of Stuttgart, URL https://books.google.com/books?id=Fbv7jwEACAAJ

[García-Pérez et al.(2018)García-Pérez, Gotsman, Meshman, and Sergey] García-Pérez Á, Gotsman A, Meshman Y, Sergey I (2018) Paxos consensus, deconstructed and abstracted. In: Ahmed A (ed) Programming Languages and Systems, Springer International Publishing, Cham, pp 912–939

[Hassani et al.(2014)Hassani, Skjellum, and Brightwell] Hassani A, Skjellum A, Brightwell R (2014) Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp 750–755, DOI 10.1109/DSN.2014.78

[Hassani et al.(2015)Hassani, Skjellum, Bangalore, and Brightwell] Hassani A, Skjellum A, Bangalore PV, Brightwell R (2015) Practical resilient cases for fa-mpi, a transactional fault-tolerant mpi. In: Proceedings of the 3rd Workshop on Exascale MPI, Association for Computing Machinery, New York, NY, USA, ExaMPI '15, DOI 10.1145/2831129.2831130, URL https://doi.org/10.1145/2831129.2831130

[Herault et al.(2015)Herault, Bouteiller, Bosilca, Gamell, Teranishi, Parashar, and Dongarra] Herault T, Bouteiller A, Bosilca G, Gamell M, Teranishi K, Parashar M, Dongarra J (2015) Practical scalable consensus for pseudo-synchronous distributed systems: Formal proof. Tech. Rep. ICL-UT-15-01

[Huang(1989)] Huang ST (1989) Detecting termination of distributed computations by external agents. In: [1989] Proceedings. The 9th International Conference on Distributed Computing Systems, pp 79–84, DOI 10.1109/ICDCS.1989.37933

[Hursey et al.(2011)Hursey, Naughton, Vallee, and Graham] Hursey J, Naughton T, Vallee G, Graham RL (2011) A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In: Cotronis Y, Danalis A, Nikolopoulos DS, Dongarra J (eds) Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 255–263

[Intel(????)] Intel (????) Poet 1.0 specification. https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture/poet.html, [Online, Accessed: Nov/19/2019]

[Katti et al.(2015)Katti, Di Fatta, Naughton, and Engelmann] Katti A, Di Fatta G, Naughton T, Engelmann C (2015) Scalable and fault tolerant failure detection and consensus. In: Proceedings of the 22nd European MPI Users' Group Meeting, Association for Computing Machinery, New York, NY, USA, EuroMPI '15, DOI 10.1145/2802658.2802660, URL https://doi.org/10.1145/2802658.2802660

[King and Nadal(2012)] King S, Nadal S (2012) Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August 19

[Lamport(1983)] Lamport L (1983) The weak byzantine generals problem. J ACM 30(3):668–676, DOI 10.1145/2402.322398, URL https://doi.org/10.1145/2402.322398

[Lamport(1998)] Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169, DOI 10.1145/279227.279229, URL https://doi.org/10.1145/279227.279229

[Lamport et al.(1982)Lamport, Shostak, and Pease] Lamport L, Shostak RE, Pease MC (1982) The byzantine generals problem. ACM Trans Program Lang Syst 4(3):382–401, URL http://dblp.uni-trier.de/db/journals/toplas/toplas4.html#LamportSP82

[Leners et al.(2011)Leners, Wu, Hung, Aguilera, and Walfish] Leners J, Wu H, Hung WL, Aguilera M, Walfish M (2011) Detecting failures in distributed systems with the falcon spy network. pp 279–294, DOI 10.1145/2043556.2043583

[Losada et al.(2020)Losada, González, Martín, Bosilca, Bouteiller, and Teranishi] Losada N, González P, Martín MJ, Bosilca G, Bouteiller A, Teranishi K (2020) Fault tolerance of mpi applications in exascale systems: The ulfm solution. Future Generation Computer Systems 106:467–481, DOI https://doi.org/10.1016/j.future.2020.01.026, URL https://www.sciencedirect.com/science/article/pii/S0167739X1930860X

[Manage Engine(2021)] Manage Engine (2021) Ipmi monitoring. URL https://www.manageengine.com/network-monitoring/ipmi-monitoring.html, [Online; accessed 23-May-2021]

[Miguel Castro(2002)] Miguel Castro BL (2002) Practical byzantine fault tolerance and proactive recovery. ACM Trans Comput Syst 20(4):398–461, DOI 10.1145/571637.571640, URL https://doi.org/10.1145/571637.571640

[Moise(2011)] Moise I (2011) Efficient agreement protocols in asynchronous distributed systems. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, IEEE, pp 2022–2025

[Moses and Raynal(2009)] Moses Y, Raynal M (2009) Revisiting simultaneous consensus with crash failures. J Parallel Distrib Comput 69(4):400–409, DOI 10.1016/j.jpdc.2009.01.001, URL https://doi.org/10.1016/j.jpdc.2009.01.001

[Ongaro and Ousterhout(2014)] Ongaro D, Ousterhout J (2014) In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX Association, USA, USENIX ATC'14, p 305–320

[Popov(2016)] Popov S (2016) The tangle. cit on p 131

[Ranganathan et al.(2001)Ranganathan, George, Todd, and Chidester] Ranganathan S, George A, Todd R, Chidester M (2001) Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. Cluster Computing 4:197–209, DOI 10.1023/A:1011494323443

[Ropars et al.(2015)Ropars, Lefray, Kim, and Schiper] Ropars T, Lefray A, Kim D, Schiper A (2015) Efficient process replication for MPI applications: Sharing work between replicas. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp 645–654, DOI 10.1109/IPDPS.2015.29

[Sankaran et al.(2005)Sankaran, Squyres, Barrett, Sahay, Lumsdaine, Duell, Hargrove, and Roman] Sankaran S, Squyres JM, Barrett B, Sahay V, Lumsdaine A, Duell J, Hargrove P, Roman E (2005) The lam/mpi checkpoint/restart framework: System-initiated checkpointing. The International Journal of High Performance Computing Applications 19(4):479–493, DOI 10.1177/1094342005056139, URL https://doi.org/10.1177/1094342005056139, https://doi.org/10.1177/1094342005056139

[Scargall(2020)] Scargall S (2020) Reliability, availability, and serviceability (ras). In: Programming Persistent Memory, Springer, pp 333–346

[Schroeder and Gibson(2009)] Schroeder B, Gibson GA (2009) A large-scale study of failures in high-performance computing systems. IEEE transactions on Dependable and Secure Computing 7(4):337–350

[Snir et al.(2014)Snir, Wisniewski, Abraham, Adve, Bagchi, Balaji, Belak, Bose, Cappello, Carlson, Chien, Coteus, Debardeleben, Diniz, Eng Snir M, Wisniewski RW, Abraham JA, Adve SV, Bagchi S, Balaji P, Belak J, Bose P, Cappello F, Carlson B, Chien AA, Coteus P, Debardeleben NA, Diniz PC, Engelmann C, Erez M, Fazzari S, Geist A, Gupta R, Johnson F, Krishnamoorthy S, Leyffer S, Liberty D, Mitra S, Munson T, Schreiber R, Stearley J, Hensbergen EV (2014) Addressing failures in exascale computing. Int J High Perform Comput Appl 28(2):129–173

[Sultana et al.(2019)Sultana, Rüfenacht, Skjellum, Laguna, and Mohror] Sultana N, Rüfenacht M, Skjellum A, Laguna I, Mohror K (2019) Failure recovery for bulk synchronous applications with MPI Stages. Parallel Computing 84:1–14, DOI https://doi.org/10.1016/j.parco.2019.02.007, URL `https://www.sciencedirect.com/science/article/pii/S0167819118303260`

[Wikipedia contributors(2021a)] Wikipedia contributors (2021a) Chipkill — Wikipedia, the free encyclopedia. URL `https://en.wikipedia.org/w/index.php?title=Chipkill&oldid=1020462792`, [Online; accessed 18-May-2021]

[Wikipedia contributors(2021b)] Wikipedia contributors (2021b) Consensus (computer science) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Consensus_(computer_science)&oldid=1012661322`, [Online; accessed 18-April-2021]

[Wikipedia contributors(2021c)] Wikipedia contributors (2021c) Reliability, availability and serviceability — Wikipedia, the free encyclopedia. URL `https://en.wikipedia.org/w/index.php?title=Reliability,_availability_and_serviceability&oldid=1015057056`, [Online; accessed 23-May-2021]

[Woo et al.(2011)Woo, Lang, Latham, Ross, and Thakur] Woo S, Lang S, Latham R, Ross R, Thakur R (2011) Reliable MPI-IO through layout-aware replication

[Yin et al.(2019)Yin, Malkhi, Reiter, Gueta, and Abraham] Yin M, Malkhi D, Reiter MK, Gueta GG, Abraham I (2019) Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, ACM, pp 347–356

Noname manuscript No.
(will be inserted by the editor)

Accelerating OCaml programs on FPGA

Loïc Sylvestre · Emmanuel Chailloux · Jocelyn Sérot

Abstract This paper aims to exploit the massive parallelism of Field-Programmable Gate Arrays (FPGAs) by programming them in OCaml, a multi-paradigm, statically-typed language. It presents O2B, an FPGA-based implementation of the OCaml virtual machine using a softcore processor, running the entire OCaml language. It then introduces Macle, a language to express, in ML-style, hardware-accelerated user-defined functions. Macle exposes fine-grained parallelism available at the circuit level and enables to manipulate data structures dynamically allocated by OCaml programs. This hybrid approach, mixing Macle and OCaml codes, allows to easily prototype FPGA applications.

Keywords high-level programming, OCaml, virtual machine, FPGA, parallel computing, hardware acceleration, compiling, finite state machines

Loïc Sylvestre and Emmanuel Chailloux
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
E-mail: Loic.Sylvestre@lip6.fr,Emmanuel.Chailloux@lip6.fr

Jocelyn Sérot
Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
E-mail: jocelyn.serot@uca.fr

1 Introduction

Reconfigurable circuits, like Field-Programmable Gate Arrays (FPGAs), are suited to design custom architectures exploiting the concurrent nature of hardware structures [5]. The configuration of an FPGA is commonly produced by a synthesis toolchain supporting a hardware description language (HDL) such as VHDL or Verilog. Other examples of more expressive HDLs include Chisel [3] which is embedded in Scala, Clash [2] in Haskell, MyHDL [8] in Python and HardCaml[1] in OCaml. Nevertheless, the *Register Transfer Level* (RTL) programming model, on which HDLs are based, is characterized by a very low level of abstraction. Hence, different approaches aim to hardware-accelerate software applications using FPGAs.

– There have been some attempts to compile small applicative languages, such as SHard [19], FLOH [22] and Basic SCI [11], directly to RTL [10]. A representative example is SAFL (*Statically Allocated Parallel Functional Language*) [16], which is a first-order ML-like language limited to tail recursion and static data structures.
– For more complex languages, custom processors or virtual machines can be implemented in RTL to run high-level languages on FPGA. JAIP [23] is a Java Virtual Machine (JVM) written in VHDL, calling a softcore processor[2] to handle dynamic class-loading. JikesRVM [15] is a JVM implemented on a CPU using an FPGA for accelerating automatic managing of dynamic memory (garbage collection / GC).
– High-Level Synthesis (HLS) promotes the use of imperative languages to design hardware [17]. Most of HLS tools, such as Catapult C or Handel-C, support a subset of C annotated with pragmas to optimize the compilation to RTL. LegUp [4] runs C programs on a softcore processor, or Pylog [12] on a hardcore processor, while compiling functions to RTL, (that do not use dynamic allocation and recursion).
– Other HLS tools[3] use OpenCL to express parallel applications and target heterogenous architectures involving Multicores, GPUs and FPGAs. TornadoVM [18], Aparapi [20] and GVM [9] implement the JVM in OpenCL. TAPA [6] is framework for task parallelism targeting OpenCL. These implementations, however, do not sufficiently expose the fine-grained parallelism available on the FPGA as well as their customization possibilities.
– FPGAs allows to implements parallel skeletons [7] and concurrency control constructs [6]. For instance, Lime [1] is a task-based data-flow programming language compiled to OpenCL or Verilog, and interacting with Java bytecode running on a CPU. Kiwi [21] is a subset of C♯ compiled to RTL and offering events, monitors and threads.

[1] https://github.com/janestreet/hardcaml
[2] A Softcore processor is processor implemented in the reconfigurable part of an FPGA.
[3] Such as AMD Vivado HLS and Intel OpenCL SDK.

These approaches highlight several needs:

– runtime systems for high-level programming on FPGA using a softcore processor (like JAIP);
– partitioning between hardware accelerated code and a runtime (like Pylog);
– hardware acceleration of user-defined functions (like SAFL);
– parallel programming constructs (like Kiwi);
– uniformity between a host language and an embedded language used for acceleration (like Lime).

To fulfill these needs, we have ported on a softcore processor the OCaml VM and its runtime (including GC), to support the entire OCaml language. This VM approach is combined with hardware acceleration of functions expressed in an ML-like language extended with parallelism skeletons able to process data structures dynamically allocated by the OCaml runtime. This allows to take full advantage of the fine-grained parallelism of the FPGA, while programming it in a high-level way, in OCaml, allowing quick prototyping, static type-checking, simulation and debugging.

Our contributions are:

– O2B[4] (*OCaml On Board*), a port of the OMicroB [24] implementation of the OCaml Virtual Machine targeting the Nios II softcore processor implemented on an FPGA. O2B enables to call custom hardware accelerators from OCaml programs.
– Macle[5] (*ML accelerator*), a language to program, in ML-style, computation kernels to be accelerated (through a Macle to VHDL compiler). Such computation kernels, called *Macle circuits* thereafter, are used by the OCaml programs executed by O2B on FPGA. The interoperability layer between OCaml and the *Macle functions* is automatically generated. It includes C and OCaml code, VHDL descriptions and scripts to control the synthesis workflow. Macle offers language constructs to manipulate OCaml values, especially data structures (such as lists, arrays and matrices) allocated in the OCaml VM heap. In particular, Macle provides parallelism skeletons over OCaml arrays to expose fine-grained parallelism and optimize memory transfers.

The remainder of this paper is organized as follows. Section 2 introduces the O2B infrastructure to run OCaml programs on FPGA. Section 3 proposes a hybrid approach to accelerate OCaml programs augmented with Macle functions. Section 4 presents the compilation of Macle, using an intermediate language (HSML, *Hierarchical State Machine Language*) to abstract the VHDL target. Section 5 evaluates our approach on different benchmarks to measure the speedup resulting from using hardware-acceleration in Macle. Section 6 describes a mechanism using parallelism skeletons to optimize memory transfers when accessing the OCaml heap. Section 7 discusses the acceleration elements and programming style obtained and then identifies future work.

[4] https://github.com/jserot/O2B

[5] https://github.com/lsylvestre/macle

2 Customizable OCaml programs on FPGAs

O2B (*OCaml On Board*) is a tool to run OCaml programs on FPGAs. It is based on OMicroB [24], an implementation of the OCaml VM dedicated to high-level programming of microcontrollers with scarce resources.

2.1 Compilation flow for OCaml to FPGAs

Figure 1 describes the configuration process used to run OCaml programs on an Intel FPGA[6] via O2B. The OCaml bytecode (generated by the OCaml compiler) is transformed into a static C array, then embedded in the C program implementing the bytecode interpreter and the O2B runtime library (including a GC). The OCaml heap and stack are C static arrays. This program is associated with the functions of the Board Support Package (BSP lib) giving access to the hardware resources of the target board. The resulting application is compiled to binary code executable by the Nios II softcore processor.

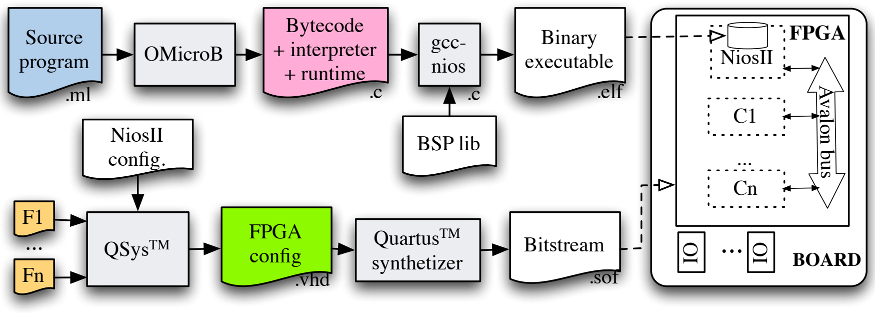


Fig. 1 Compilation flow targeting Intel FPGAs

The complete FPGA configuration includes the exact architecture of the processor used as well as a set of external RTL descriptions `F1` ··· `Fn` to be implemented as *custom components* `C1` ··· `Cn`. Technically, this configuration step is carried out by the QSys tool of the Intel Quartus chain. It generates a set of VHDL files which constitutes the description of the hardware platform. This description includes the components `C1` ··· `Cn` and the Nios II processor to be synthesized through the Quartus chain to reconfigure the FPGA.

The OCaml heap and stack can be stored either in the on-chip memory of the target FPGA (for small programs) or in external DRAM. In both cases, access is provided by means of an interconnection bus[7]. This bus also supports data transfers between the custom components and the binary code executed

[6] This process is general and can be adapted to target other FPGA families.
[7] Avalon bus for Intel platforms.

by the processor. Both the softcore and the custom components can access the physical IOs of the FPGA.

2.2 Calling accelerators from OCaml programs

The OCaml language offers an OCaml/C foreign function interface (FFI) to call C functions from OCaml programs. These C functions, running on the softcore, can in turn invoke custom components implemented on the FPGA. It is thus possible to use custom components from OCaml programs compiled to bytecode executed by O2B. The communication layer between O2B and a custom component is done via a set of dedicated registers associated to the component and manually mapped into the memory of the softcore processor.

Figure 2 shows the source code of an OCaml program designed to run with O2B. It defines three implementations of the `gcd` (*the greatest common divisor*) algorithm. The difference of two calls to `Timer.get_us` (before and after a computation) in the OCaml function `chrono` gives the execution time of the argument function call in microsecond.

OCaml code

```
external gcd_c : int -> int -> int ;;
external gcd_rtl : int -> int -> int ;;

let rec gcd_caml a b =
  if a > b then gcd_caml (a-b) b else
  if a < b then gcd_caml a (b-a) else a ;;

let chrono f a b =
  let t1 = Timer.get_us () in
  let res = f a b in
  let t2 = Timer.get_us () in
  print_int (t2-t1) ;;

let main() =
  Timer.init () ;
  let a = 5000 and b = 7000 in
  chrono gcd_caml a b ;
  chrono gcd_c a b ;
  chrono gcd_rtl a b ;;

main ();;
```

C code

```
value gcd_c(value m, value n){
  int a, b;
  a = Int_val(m);
  b = Int_val(n);
  while ( a != b ) {
    if ( a > b ) a = a-b;
    else b = b-a;
  }
  return Val_int(b);
}

value gcd_rtl(value m, value n){
  int res;
  GCD_ARG(0,Int_val(m));
  GCD_ARG(1,Int_val(n));
  GCD_START();

  while (! GCD_RDY())
    ;
  res = GCD_RESULT();
  return Val_int(res);
}
```

Fig. 2 An OCaml program executable by O2B

The C function `printf`, and by extension, the OCaml functions `print_int` and `print_string` use the Board Support Package of the FPGA target to write on a console[8]. The `gcd_c` and `gcd_rtl` functions are defined as external functions in the OCaml code using the standard FFI mechanism. Calling a

[8] The FPGA board is connected to a host PC via an UART connection for printing and debugging.

custom component from the `gcd_rtl` function involves sending the arguments (resp. retrieving the result) to (resp. from) the corresponding dedicated registers of the custom component. In figure 2, the corresponding operations are abstracted by the macros `GCD_ARG`, `GCD_START`, `GCD_RDY` and `GCD_RESULT`). Moreover, describing the behavior of the component, in synthetizable VHDL, is tedious. For the GCD example, describing this behavior and exchanging the arguments and result respectively requires 50 and 100 lines of VHDL. Finally, this GCD component must be mapped into the global configuration of the system implemented on the FPGA (called the *System on Programmable Chip*, SoPC), either manually (using the QSys tool) or by scripting. With the compilation flow introduced in the next section, RTL descriptions of custom components as well as glue code between OCaml and these components (including OCaml, C and VHDL files) will be automatically generated from a high-level formulation in the Macle language.

3 A hybrid approach for high-level programming of FPGA

The O2B experiment described in the previous section enables to run OCaml programs on FPGA via a softcore processor and call hardware accelerators from them. The difficulty is still to program these accelerators and synthesize them on the same FPGA as the softcore. In this section, we propose to express these accelerators in an ML-like language compiled to RTL. This language, called Macle (*ML Accelerator*), can inter-operate with the OCaml runtime of O2B and therefore can be used to accelerate OCaml *host* programs on FPGA.

3.1 Compilation Flow

Figure 3 shows our compilation flow of OCaml to FPGA. It automatically generates the configuration of an FPGA from an OCaml program extended with hardware-accelerated functions defined in Macle. OCaml code is compiled to bytecode to be executed by O2B targeting a softcore processor implemented on the FPGA.

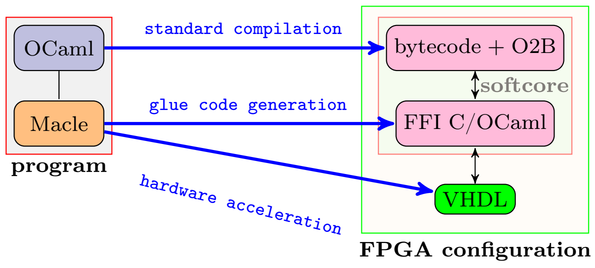


Fig. 3 An hybrid approach to run OCaml programs on FPGA via O2B and Macle

Each Macle circuit is a function compiled to VHDL and then synthesized as a custom hardware component usable from OCaml programs. The glue code is generated from the inferred type of the Macle circuit. The FPGA configuration is automatic and easily programmable without prior knowledge of hardware description languages.

3.2 The Macle language

Macle is a ML-like language which includes:

– a functional-parallel Core language (called Macle Core) compiled to RTL;
– additional language constructs (implemented in RTL) to interact with the OCaml runtime.

Figure 4 defines the syntax of Macle.

The left side of the figure defines Macle Core. This language is independent of OCaml and can be used to program synchronous circuits and compose them in parallel. We denote by $\overrightarrow{o}$ (or $o_1\ \cdots\ o_n$) a non-empty sequence of objects o_i. Macle Core includes variables (taken from a set of name $\mathcal{X}$), constants, application of builtin operators and conditionals. It also offers local mutually tail-recursive functions, function calls and let bindings. A simple let binding **let** $x = e$ **in** e' first computes e, then e'. By extension, a multiple let-binding **let** $x_1 = e_1$ **and** $\cdots\ x_n = e_n$ **in** e' first computes the expressions $e_1\ \cdots e_n$ in parallel and synchronizes before computing "e'". For instance, the hardware implementation of (**let** $x =$ factorial 10 **and** $y =$ factorial 11 **in** $x + y$) instantiates twice the implementation of factorial function in order to enable their parallel execution. Function call uses an implicit parallel let-binding to compute the arguments passed to the function. Non-recursive functions can take functions as arguments[9].

Macle Core		**Interaction with OCaml**	
circuit	$ci ::= \textbf{circuit}\ f\ \overrightarrow{x} = e$	exception	$exn ::= \texttt{Failure}\ \langle string \rangle$
constant	$c ::= \textbf{true} \mid \textbf{false} \mid \langle integer \rangle \mid ()$	pattern	$p ::= C \mid C(x_1, \cdots\ x_n)$
variable	$x, y, f \in \mathcal{X}$	expression	$e ::= \cdot\cdot$
operator$_1$	$\ominus ::= - \mid \textbf{not} \mid \cdot\cdot$		$\mid \textbf{raise}\ exn$
operator$_2$	$\oplus ::= + \mid\ <\ \mid \cdot\cdot$		$\mid \textbf{match}\ e\ \textbf{with}\ \overrightarrow{p \to e'}$
expression	$e ::= x \mid c \mid \ominus e \mid e_1 \oplus e_2$		$\mid\ !\ e$
	$\mid \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$		$\mid e := e'$
	$\mid \textbf{let}\ x_1 = e_1\ \textbf{and}\ \cdots\ x_n = e_n\ \textbf{in}\ e'$		$\mid e.(e')$
	$\mid \textbf{let}\ \overrightarrow{f\ \overrightarrow{x} = e}\ \textbf{in}\ e$		$\mid e.(e') \leftarrow e''$
	$\mid \textbf{let rec}\ \overrightarrow{f\ \overrightarrow{x} = e}\ \textbf{in}\ e'$		$\mid \textbf{array\_length}\ e$
	$\mid f\ \overrightarrow{e}$		$\mid e\ ;\ e'$
	$\mid \cdot\cdot$		$\mid \textbf{for}\ x = e\ \textbf{to}\ e'\ \textbf{do}\ e''\ \textbf{done}$

Fig. 4 Syntax of the Macle language

[9] Each call of these functions are specialized and inlined at-compile time.

The right side of Figure 4 presents the Macle constructs used to interact with the OCaml runtime :

– `!`e for accessing to the content of the reference e;
– $e := e'$ for setting the content of the reference e to the value of e';
– e`.(`e'`)` for accessing to the index e' of the array e;
– e`.(`e'`)` $\leftarrow e''$ for setting the value of e'' at the index e' of the array e.
– for raising a built-in exception parametrized by literal strings,
– for surface pattern matching on algebraic datatypes (ADT),

Note that Macle circuits cannot allocate data structures; they can only manipulate values previously allocated in the OCaml heap by the VM.

Finally, the sequence e ; e' is a syntactic sugar for **let** $x = e$ **in** e' where x is a fresh name. *For-loops* are encoded with *let-rec*.

To preserve the semantics and the safety of the Macle code, multiple let-bindings are sequentialized when they contain memory accesses or raise an exception. General recursion is supported via a program transformation producing code containing only tail-recursive calls and using an explicit stack.

Figure 5 shows three Macle circuits and an OCaml program calling a Macle circuit. The circuit `gcd_rtl` expresses the *Gcd* algorithm in Macle Core. The circuit `rev` reverses the order of the elements of an OCaml array. The circuit `collatz` computes the *stopping time* of a Collatz [13] sequence (also called *Syracuse*) starting from a given integer.

Computations in Macle

```
circuit gcd_rtl m n =
  let rec gcd a b =
    if a > b then gcd (a-b) b else
    if a < b then gcd a (b-a) else a
  in gcd m n ;;

circuit collatz n =
  let rec next len u =
    if u <= 1 then len else
    if u mod 2 == 0
    then next (len+1) (u/2)
    else next (len+1) (3*u+1)
  in next 0 n ;;

circuit rev a =
  let n = array_length a in
  for i = 0 to (n-1) / 2 do
    let t = a.(i) in
    a.(i) <- a.(n-1-i);
    a.(n-1-i) <- t
  done ;;
```

Mixing OCaml and Macle codes

```
type exp =
  | Int of int
  | Var of int
  | Add of exp * exp ;;

 circuit eval_exp env e =
   let rec eval e =
     match e with
     | Int(n) -> n
     | Var(k) -> env.(k)
     | Add(e1,e2) ->
         eval e1 + eval e2
   in eval e ;;

let main() =
    let env = [|100|] in
    let e = Add(Int(1),Var(0)) in
    try  print_int (eval_exp env e)
    with Failure s -> print_string s ;;

main();;
```

Fig. 5 Examples of Macle circuits and call from OCaml program

The circuit `eval_exp` evaluates an abstract syntax tree allocated in the OCaml heap. It safely accesses the OCaml heap since the exception `Failure` is (implicitly) raised in case of an out of bounds index or a non-exhaustive pattern matching. This exception can then be caught in OCaml by the **try** $\cdots$ **with** construct. This program evaluates the expression `Add(Int(1),Var(0))` recursively and prints the result. Evaluate `Var(0)` fetches the value at the index 0 of the array `env = [|100|]`. Recursion in Macle uses an explicit call stack, as described in section 5. Tail-recursion does not require a stack.

4 Compiling Macle

The global compilation flow from Macle to VHDL is depicted Figure 6. It involves four passes. The first pass consists in normalizing the source code:

– renaming all bindings in the source code with unique names;
– rewriting the code in so-called *Administrative Normal Form* [14] (introducing let-bindings for each step of computation);
– inlining functions by recursively duplicating their body at each call site (except recursive ones);
– transforming recursive functions which are not tail-recursive into tail-recursive ones using an explicit stack.

The second pass compiles Macle into an intermediate language, called HSML (*Hierarchical State Machine Language*), allowing to express parallel composition of hierarchical finite state machines. The third pass flattens the hierarchical structure of HSML. The fourth pass translates a flat HSML description into VHDL.

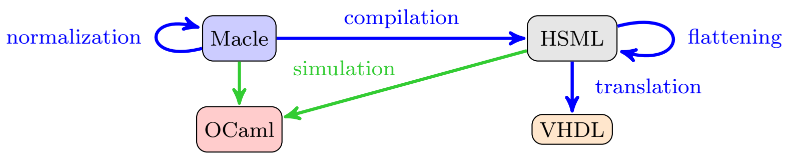


Fig. 6 Compilation flow of Macle to VHDL

At each point of the compilation flow, an OCaml backend is provided for simulation and debugging on a PC.

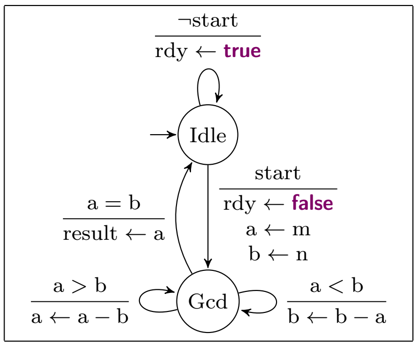
Due to space limitations, the rest of this section only describes the compilation of Macle Core to HSML.

4.1 Targeting the register transfer level

Synchronous finite state machines (FSM) are commonly used to describe computations at the register transfer level (RTL). A FSM is classically defined by a set of states (names) and a set of transitions. Each transition connects a source state to a destination state and can be associated to a set of guards and a set of actions. Guards define when the transition is enabled. They can depend on inputs and local variables. Actions are performed when the transition is enabled and can write outputs and local variables. Transitions are only taken at the rising edge of a global clock. At each clock edge, if a transition starting from the current state has all its guards validated, it is enabled, the associated actions are performed (instantaneously) and the destination state becomes the current state.

FSMs are classically encoded in VHDL as synchronous processes with asynchronous reset. Inputs, outputs and local variables are implemented as VHDL signals with a dedicated signal representing the current state. At each rising edge of the input clock, depending on the value of the current state and some conditions involving inputs and local variables, the next state value is selected and the value of outputs and local variables is updated. The FSM is re-initialized, asynchronously, whenever the reset input signal becomes true.

Figure 7 gives a graphical representation of a FSM describing the computation of a `gcd` function and its encoding in VHDL.



```
entity gcd_rtl is
  port( signal clk, reset : in std_logic;
        signal start : in std_logic;
        signal rdy : out std_logic;
        signal m, n : in signed(30 downto 0);
        signal result : out signed(30 downto 0));
end entity;

architecture rtl of gcd_rtl is
  type t_state is (Idle, Gcd);
  signal STATE : t_state;
  signal a, b : signed(30 downto 0);
```

```
begin process(reset,clk) begin
  if reset = '1' then
    STATE <= Idle;
  elsif rising_edge(clk) then
    case STATE is
      when Idle =>
        if start then
          rdy <= false;
          a <= m;
          b <= n;
          STATE <= Gcd;
        else
          rdy <= true;
          STATE <= Idle;
        end if;
      when Gcd =>
        if a > b then
          a <= a - b;
          STATE <= Gcd;
        elsif a < b then
          b <= b - a;
          STATE <= Gcd;
        else
          result <= a;
          STATE <= Idle;
        end if;
    end case;
  end if
 end process;
end architecture;
```

Fig. 7 FSM and VHDL implementation of the *Gcd* algorithm (given in Macle Figure 5)

The `start` input and `rdy` output are used respectively to start and signal the end of the computation. In the VHDL code, modifications of the state variable `STATE` as well as the outputs and local variables are denoted using signal assignments (`<signal_name> <= <expression>`). Assignments performed at the same clock edge are performed concurrently, *i.e.* the expressions denoted by the right hand sides (RHSs) are all evaluated in parallel and then, and only then, the signals designated by the left hand sides (LHSs) are updated simultaneously. Note that in the code given Figure 7, arguments and result are encoded as 31-bit signed integers. This is to have the same representation of OCaml value than in the O2B runtime, in order to call this circuit from OCaml programs.

By declaring separate processes, each encoding a given FSM, within the same entity/architecture, it is easy to implement synchronous parallel composition of FSMs. Each FSM is triggered by the same global clock and has access to the signals declared in the architecture. However, these signals can only be shared for reading as a signal written by a process cannot be written by another process.

4.2 HSML : a FSM-based intermediate language

We do not compile Macle circuits directly to VHDL. Instead, we use an intermediate language, HSML (*Hierarchical State Machine Language*) for describing the behavior of FSMs and expressing their composition, and which can be easily translated to VHDL.

Figure 8 defines the syntax of HSML A circuit is a parallel composition of FSMs ($A_1 \parallel \cdots A_n$) depending on inputs, modifying outputs and using local variables. A FSM is a set of mutually recursive transitions in the scope of a body used to initialize it. A transition is a thunk $f\,() = A$ associating a name f to a FSM A. HSML offers a notion of hierarchy. For instance, a FSM **let rec** t_1 **and** $\cdots$ t_m **in** (**let rec** t'_1 **and** $\cdots$ t'_n **in** $f()$) is a hierarchical formulation of the FSM **let rec** t_1 **and** $\cdots$ t_m **and** t'_1 **and** $\cdots$ t'_n **in** $f()$.

circuit	$\phi ::=$ **circuit** $f\ \overrightarrow{x_{in}}$ **returns** $\overrightarrow{x_{out}} =$ **var** $\overrightarrow{x}$ **in** P
parallel composition	$P ::= A_1 \parallel \cdots A_n$
FSM	$A ::=$ **let rec** ts **in** A_{init}
	$\mid$ **if** e **then** A_1 **else** A_2
	$\mid$ **do** $x_1 \leftarrow e_1$ **and** $\cdots$ $x_n \leftarrow e_n$ **then** A
	$\mid f()$
	$\mid P$ **in** A
transitions	$ts ::= \epsilon \mid f_1() = A_1$ **and** $\cdots$ $f_n() = A_n$
expression	$e ::= x \mid c \mid \ominus\ e \mid e_1 \oplus e_2$
operator$_1$	$\ominus ::= \cdot\cdot$
operator$_2$	$\oplus ::= \cdot\cdot \mid \wedge \mid \vee$

Fig. 8 Syntax of HSML

A HSML expression e is a variable, a constant or the application of a built-in operator. The construct (**do** $x_1 \leftarrow e_1$ **and** $\cdots$ $x_n \leftarrow e_n$ **in** A) evaluates the expressions e_1, $\cdots$ e_n, then assigns the results to the variables x_1 $\cdots$ x_n and finally computes A.

Figure 9 shows an HSML circuit corresponding to the VHDL code given Figure 7. This circuit was automatically generated from the Macle circuit `gcd_rtl` defined Figure 5.

```
circuit gcd_rtl (start,m,n) returns (rdy,result) = var a, b in
  let rec idle() =
    if start then
       (do rdy ← false and a ← m and b ← n then gcd())
    else
       (do rdy ← true then idle())
  and gcd() =
    if a > b then
      (do a ← (a-b) then gcd())
    else if a < b then
      (do b ← (b-a) then gcd())
    else
      (do result ← a then idle())
  in (do rdy ← true then idle())
```

Fig. 9 HSML circuit implementing the *Gcd* algorithm

HSML exposes the semantics of the RT level (described informally on the VHDL code of Figure 7) while offering a notion of hierarchy which makes it close to an expression language. In particular, some HSML constructs (like *let rec* and conditional) are common with Macle. Thus, HSML constitutes a useful intermediate language for compiling Macle to VHDL.

4.3 Compiling Macle Core

The compilation $\mathcal{C}[\![\textbf{circuit}\ f\ \overrightarrow{x} = e]\!]$ of a Macle Core circuit is defined as the compilation of the body e of the circuit, from which the inputs, outputs and local variables are inferred.

$$\mathcal{C}_{\mathsf{ci}}[\![\textbf{circuit}\ f\ \overrightarrow{x} = e]\!] = \textbf{circuit}\ f\ \overrightarrow{x_{in}}\ \textbf{returns}\ \overrightarrow{x_{out}} = \textbf{var}\ \overrightarrow{x_{local}}\ \textbf{in}\ \overbrace{\mathcal{C}[\![e]\!]^{start,rdy,result}}^{s}$$
$$\text{where} \begin{cases} \overrightarrow{x_{in}}, \overrightarrow{x_{out}} \text{ and } \overrightarrow{x_{local}} \text{ are inputs, outputs and local} \\ \text{variables declarations inferred from } s \\ start, rdy, result \text{ are fresh names} \end{cases}$$

The compilation $\mathcal{C}[\![e]\!]^{start,rdy,result}$ of a Macle Core expression e is a hierarchical FSM initialized in a special state *idle*. It waits for the input *start* to be set to the value **true** to start the computation. This computation assigns a value to the output *result*. The output *rdy* notifies when the computation is done. The auxiliary function $\mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{result,idle}$ is defined next. The compilation environment ρ maps functions names to the list of their formal arguments.

The compilation $\mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{result,idle}$ of a subexpression is inductively defined on the syntax of the expressions. The compilation of a subexpression e which do not contain control structures is defined as an affectation of e to a variable *result* continuing with a tail-call to a destination.

$$\mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{r,idle} = \textbf{do}\ r \leftarrow e\ \textbf{in}\ idle()$$
$$\text{if } e \text{ is a variable, a constant or an application of operator}$$

The compilation of a Macle conditional is a HSML conditional, subexpressions being inductively compiled.

$$\mathcal{C}_{\mathsf{e}}[\![\textsf{if } x \textsf{ then } e_1 \textsf{ else } e_2]\!]_\rho^{r,idle} = \textsf{if } x \textsf{ then } \mathcal{C}_{\mathsf{e}}[\![e_1]\!]_\rho^{r,idle} \textsf{ else } \mathcal{C}_{\mathsf{e}}[\![e_2]\!]_\rho^{r,idle}$$

Compiling a *let rec* globalizes function parameters. To achieve this, each function name introduced by a *let rec* is bound to the list of its formal parameters within the compilation environment ρ. The extension of ρ with a function name f bound to its parameters $x_1 \ \cdots \ x_n$ is denoted by $\rho[f/(x_1, \ \cdots \ x_n)]$, assuming that f is not in the domain of ρ. Alternatively, the compilation of a function call $(f \ x_1 \ \cdots \ x_n)$ is an assignment of the values $x_1 \ \cdots \ x_n$ to the formal parameters $y_1 \ \cdots \ y_n$ given by $f(\rho)$, continuing with a call to $f()$.

$$\begin{array}{ll}
\mathcal{C}_{\mathsf{e}}\left[\!\!\left[\begin{array}{l}\textsf{let rec } f_1 \ \overrightarrow{x_1} = e_1 \\ \textsf{and } \cdots \ f_n \ \overrightarrow{x_n} = e_n \textsf{ in } e\end{array}\right]\!\!\right]_\rho^{r,idle} & = \begin{array}{l}\textsf{let rec } f_1 \ () = \mathcal{C}_{\mathsf{e}}[\![e_1]\!]_{\rho'}^{r,idle} \\ \textsf{and } \cdots \ f_n \ () = \mathcal{C}_{\mathsf{e}}[\![e_n]\!]_{\rho'}^{r,idle} \textsf{ in } \mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho'}^{r,idle}\end{array} \\
& \quad \text{where } \rho' = \rho[f_1/\overrightarrow{x_1}] \cdots \ [f_n/\overrightarrow{x_n}] \\
\mathcal{C}_{\mathsf{e}}[\![f \ x_1 \ \cdots \ x_n]\!]_\rho^{r,idle} & = \textsf{do } y_1 \leftarrow x_1 \textsf{ and } \cdots \ y_n \leftarrow x_n \textsf{ then } f() \\
& \quad \text{if } \rho(f) = (y_1, \ \cdots \ y_n)
\end{array}$$

The compilation $\mathcal{C}_{\mathsf{e}}[\![\textsf{let } x = e \textsf{ in } e']\!]_\rho^{r,idle}$ of a *let* with a single binding is defined as the compilation of the subexpression e into the variable x continuing with the compilation of the body e'.

$$\begin{array}{l}
\mathcal{C}_{\mathsf{e}}[\![\textsf{let } x = e \textsf{ in } e']\!]_\rho^{r,idle} = \textsf{let rec } f() = \mathcal{C}_{\mathsf{e}}[\![e']\!]_\rho^{r,idle} \textsf{ in } \mathcal{C}_{\mathsf{e}}[\![e']\!]_\rho^{x,f} \\
\qquad\qquad\qquad\qquad\qquad \text{where} f \text{ is a fresh name}
\end{array}$$

The compilation of a *let* with more than one binding is defined as a parallel composition of FSMs followed by a synchronization barrier activating the execution of the compiled body of the *let*.

$$\begin{array}{ll}
\mathcal{C}_{\mathsf{e}}\left[\!\!\left[\begin{array}{l}\textsf{let } x_1 = e_1 \\ \textsf{and } \cdots \ x_n = e_n \textsf{ in } e \\ \qquad\qquad (\text{if } n > 1)\end{array}\right]\!\!\right]_\rho^{r,idle} & = \left(\begin{array}{l}\textsf{let rec } f() = \\ \quad \textsf{do } start_1 \leftarrow \textsf{false and } \cdots \ start_n \leftarrow \textsf{false then} \\ \quad (\mathcal{C}[\![e_1]\!]^{start_1,rdy_1,x_1} \,\|\, \cdots \ \mathcal{C}[\![e_n]\!]^{start_n,rdy_n,x_n}) \textsf{ in} \\ \quad \textsf{if } rdy_1 \wedge \cdots \ rdy_n \textsf{ then } \mathcal{C}_{\mathsf{e}}[\![e]\!]_\rho^{r,idle} \textsf{ else } f() \\ \textsf{in do } start_1 \leftarrow \textsf{true and } \cdots \ start_n \leftarrow \textsf{true then } f()\end{array}\right) \\
& \text{where } \begin{cases} i \in \{1, \cdots \ n\} \\ f, start_i, rdy_i \text{ are fresh names}\end{cases}
\end{array}$$

Since they expose parallelism, let-bindings provide the main possibilities of acceleration of OCaml programs on FPGA as shown in the next section.

5 Examples and benchmarks

We now evaluate the speedup that can be achieved by running OCaml programs on FPGA via O2B and Macle, following our hybrid approach. These programs are compared by taking as reference equivalent C code running on the same softcore processor. We first consider programs written in Macle Core (as described on the left side of figure 4), and then Macle circuits interacting with the OCaml runtime (right side of figure 4).

5.1 Methodology

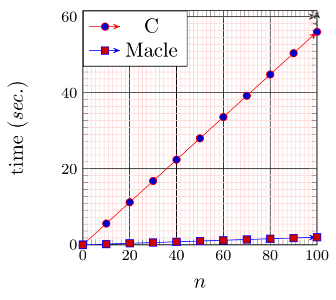
Experimental setup We use a Max10 Intel FPGA embedded on a Terasic DE10-Lite board. This FPGA has limited resources: 50K logic elements (LEs); 1,638 Kb of on-chip memory; a clock frequency of 50 MHz[10]. From a given OCaml source program, O2B creates a C program containing the bytecode generated by the OCaml compiler, the VM, its runtime library (including a GC) and additional C code. The bytecode as well as the OCaml stack and heap are both implemented with C static arrays, both stored in the on-chip memory. The whole is compiled via the Nios II backend of `gcc` with optimizations enabled (-Os). All data structures manipulated by OCaml, C and Macle code using the OCaml heap and the OCaml arrays bounds are dynamically checked at each access.

Measuring elapsed time on a FPGA Macle circuits are called from a C block running on the softcore. Indeed, as described in section 2.2, is necessary to write arguments in the dedicated registers of the custom component implementing the circuit, start the circuit and wait for the end of the computation to read the result (again in the dedicated registers of the custom component). We measure the execution time of each Macle circuit from the beginning to the end of the corresponding C block.

5.2 Macle Core

We here assess the efficiency gains obtained both by rewriting a C function as a Macle circuit and, possibly, replicating this circuit to parallelize the corresponding computations.

Pure Computations Figure 10 shows the execution time of the `gcd_rtl` Macle circuit (given Figure 5) and the `gcd_c` C function (given Figure 2) called by an OCaml program. The observed Macle *vs* C speedup factor is 30.



```
(* test program *)
let main () =
   let n = 40 in
   let nb_it = n * (1000 * 1000) in

   print_int (gcd_c nb_it 1);
   print_int (gcd_rtl nb_it 1) ;;

main () ;;
```

Fig. 10 Execution time of a simple computation (gcd) in Macle and C

[10] The DE10-Lite is also equiped with a 64 Mb external SDRAM but it is not used in this series of experiments.

A similar experiment with the Macle circuit `collatz` (given Figure 5) leads to a $\times 60$ speedup. The hardware implementation of `gcd_rtl` and `collatz` both use approximately 360 logic elements (LEs), *i.e.* 0.75% of the total available on the target FPGA used here.

Parallel computations Figure 11 gives a circuit `sum_gcd2` calling twice a function `gcd_rtl` and combining results. The **let** $\cdots$ **and** $\cdots$ **in** $\cdots$ constructs is implemented by a synchronization barrier involving a parallel composition of two instances of the FSMs given Figure 7.

```
circuit sum_gcd2 a1 ··· an y =
  let rec gcd n m =
    if n > m then gcd (n-m) m else
    if n < m then gcd n (m-n)
             else n
  in
  let x1 = gcd a1 y and ···
  and xn = gcd an y in
  (x1 + ··· xn)
```

$\texttt{sum\_gcd}_n$	**size (LEs)**
$\texttt{sum\_gcd}_2$	753
$\texttt{sum\_gcd}_4$	1,413
$\texttt{sum\_gcd}_8$	2,828
$\texttt{sum\_gcd}_{16}$	5,135
$\texttt{sum\_gcd}_{32}$	9,823

Fig. 11 Parallelization of a computation and impact on the size of the generated hardware

The global execution time of the barrier is the max of the execution times of the expressions (`gcd` $\texttt{a}_i$ `y`), to which is added the execution time of the rest of the computation (here instantaneous). For instance, calling the circuit $\texttt{sum\_gcd}_2$ with equal arguments a_1 and a_2 doubles the previous $\times 30$ speedup observed in Macle *vs* C (Figure 10). Generalizing this example to circuits $\texttt{sum\_gcd}_n$ (computing n times `gcd_rtl` and summing results) gives a speedup of $30 \times n$ in Macle *vs* C (e.g., $\texttt{sum\_gcd}_{32}$ is 960 times faster in Macle than in C). This gain is only possible because the `gcd` local function is inlined n times, the generated hardware using more LEs as shown on the right side of Figure 11.

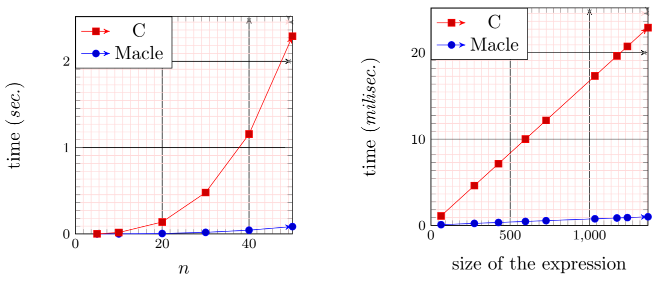
5.3 Interacting with the OCaml runtime

Macle enables hardware acceleration for the functional-imperative fragment of OCaml, accessing directly the OCaml heap (a shared memory allocated in the RAM) using a bus.

Left side of Figure 12 shows the execution time of a Macle circuit `product` multiplying two integer matrices of size $n \times n$, *vs* a C version. The Macle version is 27 times faster than the C one. The generated hardware uses 1602 LEs.

Right side of Figure 12 shows the execution time of the Macle circuit `eval_exp` (given Figure 5) *vs* an OCaml version, recursively evaluating trees of arithmetic expressions of various sizes (in number of constants and variables). Note that the realization of this Macle circuit uses 17,566 LEs because it requires an explicit stack which is (here) implemented using LEs instead of on-chip memory blocks. The resulting speedup is encouraging: the Macle circuit (using a recursive formulation) is 23 times faster than the C formulation.

From a programmer's point of view, this speedup is simply obtained by replacing a **let** keyword in the original OCaml formulation by "**circuit**".



(a) product (b) eval_exp

Fig. 12 Execution time of Macle circuits using imperative features (a) and recursion (b)

This preliminary evaluation shows that reformulating side-effect-free C functions as Macle circuits can bring substantial speedups (eg., up to 30 for the `gcd_rtl` of Figure 5). Replicating the hardware corresponding to these circuits, intrinsically resulting in their parallel execution, allows to further boosts these speedups (e.g., up to 960 for the $\texttt{sum\_gcd}_{32}$ example given Figure 10).

Macle also offers computations on data structures dynamically allocated in the VM heap and accessed in an imperative manner. But for large data structures, such as arrays, the cost of accessing the corresponding memory can quickly create a bottleneck, as discussed in the next section.

6 Optimised tranfers and parallelism skeletons

Allowing Macle circuits to manipulate values stored in the OCaml heap has a cost. Because this heap is implemented in shared memory[11], each access requires a bus transaction. When manipulating large data structures, like arrays, the corresponding overhead can quickly become prohitive. To overcome this problem, Macle provides some dedicated constructs, called *parallelism skeletons* aiming at minimizing this overhead and offering higher-level parallelism. These skeletons are listed Figure 13.

array_map$\langle n \rangle : (\alpha \to \beta) \to \alpha$ `array` $\to \beta$ `array` $\to$ `unit`
array_reduce$\langle n \rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta$ `array` $\to \alpha$
array_scan$\langle n \rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta$ `array` $\to \alpha$ `array` $\to$ `unit`

Fig. 13 Simple parallelism skeletons available in Macle

[11] On-chip memory in our experimental platform, but the problem would be worst if the heap was allocated in external DRAM.

Each skeleton is parameterized by an integer n, which statically specifies the size of a buffer used internally to transfer slices of the source and destination arrays between the OCaml heap and the Macle circuits.

For instance, the expression (**array_map**$\langle 64\rangle$ f `src` `dst`) copies the 64 first elements of the OCaml array `src` into a VHDL array, computes the function f *in parallel* on each element of this array and writes back the 64 resulting values in the OCaml array `dst`. Processing the whole OCaml array is carried out by iterating this transfer-execution-transfer sequence.

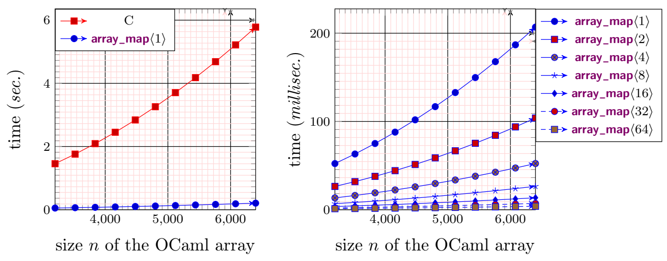
Figure 14 is a simple OCaml program mixing imperative features, computations and a parallelism skeleton **array_map**$\langle k\rangle$ within a Macle circuit `filter_mul`$_k$. It implements the Eratosthene sieve: determining all the prime numbers less than a natural number n, by filtering an OCaml array of size n containing integer from 1 to n. The circuit `filter_mul`$_k$ removes array elements that are multiple of a given integer y using the `gcd` algorithm. This computation is performed in parallel by group of k elements of the array, encoding the removed elements by the integer zero. The current prime number used to filter the rest of the array is determined by a loop traversing the array, element by element, skipping zeros (i.e., elements already removed).

Macle code	OCaml code

```
circuit filter_mul_k y a =
  let rec gcd n m =
    if n > m then gcd (n-m) m else
    if n < m then gcd n (m-n)
             else n
  in
  let remove x =
    if x <= 1 then 0 else
    if x == y then x else
    if gcd x y == 1 then x else 0
  in

  if y <= 1 then () else

  array_map⟨k⟩ remove a a ;;
```
	```
	let interval n =
	  Array.init n (fun x -> x + 1) ;;

	let print_if_not_zero x =
	  if x != 0 then print_int x ;;

	let eratostene_k a =
	  for i = 1 to Array.length a - 1 do
	    filter_mul_k src.(i) a
	  done ;;

	let main() =
	  let n = (32*100) in
	  let a = interval n in
	  eratostene_k a;
	  Array.iter print_if_not_zero a ;;

	main();;
	```

Fig. 14 A Macle circuit with a parallelism skeleton computing the Eratosthene sieve

Figure 15 shows, according to k, the size (in LEs) of the `filter_mul`$_k$ circuit and the execution time of `filter_mul`$_k$ with argument y being 2 and a being `(interval` n`)`. Results are compared to a sequential C version. Doubling the degree of parallelism k almost doubles both the size of the circuit and the speedup (taking into account the transfer time). For instance, `filter_mul`$_{64}$ is 53 times faster than `filter_mul`$_1$. Moreover, `filter_mul`$_1$ is 28 times faster than the C version, resulting in a cumulated speedup of $53 \times 28 = 1{,}484$.



k	**size (LEs)**
1	1240
2	1655
4	2487
8	4182
16	7521
32	15107
64	29739

Fig. 15 space/time trade-off of the Macle circuits `filter_mul`$_k$ and comparison with C

7 Conclusion

In this paper, we proposed an hybrid approach for programming FPGAs using the OCaml language This approach consists in:

– running OCaml programs by embedding their bytecode and the OCaml VM in a C program running on a softcore processor;
– calling hardware accelerated functions, user-defined in the Macle language, from OCaml.

Macle is a functional-imperative subset of OCaml supporting:

– parallel and sequential compositions of computations;
– mixing computations with sequential accesses to the OCaml heap (within the dynamic memory of the softcore processor);
– use of parallelism skeletons on dynamic data structures with optimization of memory transfers.

Macle, as well as the intermediate language HSML used by the Macle compiler, are statically typed and this feature provides much stronger guarantees on the safety of the generated circuits than using classical HDLs.

We described an implementation of this approach based on the O2B platform and a complete compilation flow from Macle circuit descriptions to VHDL. This compilation flow is fully automatized and easy to use. Moreover, it includes a simulation mode generating OCaml code from different points of the compiler to test the applications on PC before loading them on FPGA.

Preliminary results, obtained on small benchmarks are very encouraging. They show in particular that important speedups (up to the three orders of magnitude, compared to C code running on the hosted softcore) can be obtained by combining the ability to compile a function to hardware and the possibility to replicate the corresponding hardware in order to use data parallelism. Parametrizable parallelism skeletons both offer a way to tackle the bottleneck occurring when exchanging data between the OCaml program and the accelerated functions and also a very practical way to explore the time *vs.* space trade-off, a classical issue when programming FPGAs (reducing computing time by increasing the number of used logic elements).

The work described here offers many interesting paths for future work.

First of all, scaling up for larger applications, both symbolic and numerical, is an important point to convince the OCaml community to use FPGAs, but also the FPGA community to use high level languages. For this, a technical but critical issue is the ability to use larger, external memory chips, with optimized transfers (using DMA facilities for example) to store large dynamically allocated data structures. The ability to implement local stacks used by circuits to realize non-tail recursion (such as evoked in section 5.3) in on-chip memory (instead of LEs) is another key point to allow large and complex symbolic computations to be implemented on moderately-sized FPGAs. From a programmer's point of view, the definition and implementation of new parallelism skeletons, including, possibly, domain-specific skeletons, could also help.

Concerning the tool chain itself, we plan to switch to fully open source design and synthesis tools, with the idea that using such tools would facilitate the static analysis of the Macle circuits and the prediction of the space and time characteristics of the generated hardware (LE usage and execution time). These information could be used, for example, to decide which circuit should be duplicated, and also to provide guarantees on applications interacting with the outside world, including critical applications using synchronous programming models (close to synchronous FSMs).

In a longer term, we could also explore other ways to accelerate both the runtime (memory and exception management) and the VM interpreter by partially implementing them as circuits, or even try to create applications using different levels of parallelism by using multiple VMs sharing Macle circuits. The latter could provide an interesting approach to exploit heterogeneous platforms including multi-cores, GPUs and FPGAs for example.

References

[1] J. Auerbach, D. F. Bacon, P. Cheng, *et al.*, "Lime: A java-compatible and synthesizable language for heterogeneous architectures", in *ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 89–108.

[2] C. Baaij, M. Kooijman, J. Kuper, *et al.*, "Clash: Structural descriptions of synchronous hardware using haskell", in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, IEEE, 2010, pp. 714–721.

[3] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: constructing hardware in a Scala embedded language", in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[4] A. Canis, J. Choi, M. Aldham, *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems", in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011, pp. 33–36.

[5] J. M. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey", *ACM Computing Surveys (CSUR)*, vol. 42, no. 4, pp. 1–65, 2010.

[6] Y. Chi, L. Guo, J. Lau, *et al.*, "Extending high-level synthesis for task-parallel programs", in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2021, pp. 204–213.

[7] M. Danelutto, G. Mencagli, M. Torquati, *et al.*, "Algorithmic skeletons and parallel design patterns in mainstream parallel programming", *Int. J. Parallel Program.*, vol. 49, pp. 177–198, 2021.

[8] J. Decaluwe, “MyHDL: a Python-Based Hardware Description Language”, *Linux journal*, pp. 84–87, 2004.

[9] J. Fumero, A. Stratikopoulos, and C. Kotselidis, “Running parallel bytecode interpreters on heterogeneous hardware”, in *4th International Conference on Art, Science, and Engineering of Programming*, 2020, pp. 31–35.

[10] P. Gammie, “Synchronous digital circuits as functional programs”, *ACM Computing Surveys (CSUR)*, vol. 46, no. 2, pp. 1–27, 2013.

[11] D. R. Ghica, A. Smith, and S. Singh, “Geometry of synthesis iv: Compiling affine recursion into static hardware”, in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011, pp. 221–233.

[12] S. Huang, K. Wu, H. Jeong, *et al.*, “Pylog: An algorithm-centric python-based FPGA programming and synthesis flow”, *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[13] Y. Ito and K. Nakano, “A hardware-software cooperative approach for the exhaustive verification of the Collatz conjecture”, in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE, 2009, pp. 63–70.

[14] A. Kennedy, “Compiling with continuations, continued”, in *12th ACM SIGPLAN International Conference on Functional programming*, 2007, pp. 177–190.

[15] M. Maas, K. Asanović, and J. Kubiatowicz, “A hardware accelerator for tracing garbage collection”, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 138–151.

[16] A. Mycroft and R. Sharp, “A Statically Allocated Parallel Functional Language”, in *International Colloquium on Automata, Languages, and Programming*, Springer, 2000, pp. 37–48.

[17] R. Nane, V.-M. Sima, C. Pilato, *et al.*, “A survey and evaluation of fpga high-level synthesis tools”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[18] M. Papadimitriou, J. Fumero, A. Stratikopoulos, *et al.*, “Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs”, *The Art, Science, and Engineering of Programming*, vol. 5, no. 2, pp. 8–1, 2020.

[19] X. Saint-Mleux, M. Feeley, and J.-P. David, “SHard: a Scheme to Hardware Compiler”, in *Workshop on Scheme and Functional Programming*, 2006.

[20] O. Segal, M. Margala, S. R. Chalamalasetti, *et al.*, “High level programming framework for FPGAs in the data center”, in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–4.

[21] S. Singh and D. J. Greaves, “Kiwi: Synthesis of fpga circuits from parallel programs”, in *16th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2008, pp. 3–12.

[22] R. Townsend, M. A. Kim, and S. A. Edwards, “From Functional Programs to Pipelined Dataflow Circuits”, in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 76–86.

[23] C.-J. Tsai, H.-W. Kuo, Z. Lin, *et al.*, “A Java processor IP design for embedded SoC”, *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 2, pp. 1–25, 2015.

[24] S. Varoumas, B. Vaugon, and E. Chailloux, “A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project”, in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Jan. 2018.

Noname manuscript No.
(will be inserted by the editor)

Assessing Application Efficiency and Performance Portability in Single-Source Programming for Heterogeneous Parallel Systems

August Ernstsson · Dalvan Griebler · Christoph Kessler

Abstract We analyze the performance portability of the skeleton-based, single-source multi-backend high-level programming framework SkePU across multiple different CPU-GPU heterogeneous systems. Thereby, we provide a systematic application efficiency characterization of SkePU-generated code in comparison to equivalent hand-written code in more low-level parallel programming models such as OpenMP and CUDA. For this purpose, we contribute ports of the STREAM benchmark suite and of a part of the NAS Parallel Benchmark suite to SkePU. We show that for STREAM and the EP benchmark, SkePU regularly scores efficiency values above 80% and in particular for CPU systems, SkePU can outperform hand-written code.

Keywords algorithmic skeletons · parallel efficiency · performance portability · heterogeneous parallel computing · high-level parallel programming

1 Introduction

High-level parallel programming aims to simplify programming of systems with parallel (and possibly heterogeneous) hardware architectures. A high-level parallel programming model typically achieves this by abstracting away properties such as load balancing, synchronization, data movement, and other practical considerations, e.g., languages, compilers, and underlying APIs. Typically, the goal is also to provide *portability* across a large number of different platforms

August Ernstsson · Christoph Kessler
PELAB, Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
E-mail: <firstname>.<lastname>@liu.se

Dalvan Griebler
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
E-mail: <firstname>.<lastname>@pucrs.br

and even types of platforms (sometimes called ”backends”). However, the efficiency of the resulting platform-specific code may vary considerably across the different platforms. This can be a particular challenge for single-source multi-backend high-level parallel programming models that need to generate, from the same high-level source, code for architecturally very diverse target platforms.

For high-level programming models, and in particular their concrete implementations in languages, libraries, and frameworks, it is therefore of interest to measure *performance portability* in addition to absolute performance. Performance portability is a property of a program and a set of specific target platforms; it quantifies the program's ability to run correctly and at decent efficiency across the given set of platforms without requiring (significant) modification of the source code. In order to achieve a good understanding of the overall performance portability, it is therefore important to find a representative platform set to gather experimental results from. The program(s) should also be representative of the target application area of the high-level parallel programming model, or alternatively, a general set of programs that can be used for comparison between a range of high-level parallel programming models or implementations.

A promising class of single-source high-level programming models are multi-backend *skeleton programming* frameworks such as SkePU [19], MueSLi [18], FastFlow [1], GrPPI [15] or SPar [21]. These frameworks provide a set of composable generic programming constructs (known as algorithmic skeletons) that implement certain parallelizable computation patterns, that can be parameterized in sequential, problem-specific code, and for which different platform-specific implementations (backends) are available.

This work investigates the performance portability of the skeleton-based, single-source multi-backend high-level data-parallel programming framework SkePU [19] across multiple different CPU-GPU heterogeneous systems. We provide a systematic application efficiency characterization of SkePU-generated code in comparison to equivalent hand-written code in more low-level parallel programming models such as OpenMP and CUDA. For this purpose, we contribute new SkePU ports of the STREAM benchmark suite and of a part of the NAS Parallel Benchmark suite.[1]

We show that for STREAM and the EP benchmark, SkePU regularly scores efficiency values above 80% and in particular for CPU systems, SkePU can outperform hand-written code.

The remainder of this paper is organized as follows: Section 2 presents background about high-level parallel programming, SkePU, performance portability and relevant benchmarks, esp. STREAM and NAS benchmarks. Related work is discussed in Section 3. Section 4 presents details about our experimental method. Results are listed and discussed in Section 5 and 6, respectively. Section 7 presents conclusions and Section 8 proposes future work.

[1] The SkePU implementations will be published as open source.

2 Background

2.1 High-level parallel programming and SkePU

Parallel programming is widely considered being more difficult and error-prone than sequential programming, because the parallel execution dimension introduces new challenges, bug risks and potential performance issues that do not exist in the sequential computing world, such as load imbalance, race conditions, deadlocks and overheads for parallelism management, communication and synchronization.

Simple low-level extensions of sequential programming models by multithreading, accelerator control or message passing constructs, such as Pthreads, OpenCL or MPI respectively, leave the programmer alone with this additional complexity exposed. High-level parallel programming models promise to reduce complexity by providing structured programming constructs that manage parallelism, synchronization and communication for certain patterns of parallel computation. In particular, *skeleton programming* [11,10] has been intensively researched during the last three decades, and improvements in programmability have been experimentally demonstrated [13,6,2,3]. The approach is based on expressing computations in terms of pre-defined high-level constructs called *(algorithmic) skeletons* such as *map*, *reduce*, *scan* or *stencil*, which capture a specific, parallelizable computation pattern as a higher-order function that can be parameterized in user-provided code to instantiate executable code. All details of managing parallelism, communication and synchronization are encapsulated in the skeleton implementation. In this way, skeleton programs are, conceptually, no harder to write, read and maintain than well-structured sequential code for the same problem.

The reduced programming effort is usually paid for with some efficiency overheads compared to expert-written explicitly parallel code, and the skeleton approach is not applicable to computations that do not match any of the supported computation patterns. Nevertheless, the approach has been successfully demonstrated in research projects such as FastFlow [1], SkePU [19], SPar [21], and also been adopted in many modern parallel programming interfaces, such as Intel TBB, Nvidia Thrust, Hadoop Mapreduce or Apache Spark.

Skeleton programming is particularly promising as a means to provide better code portability through a high-level abstraction which can more easily map to different types of target architectures (e.g., multicore CPU, GPU or cluster) in today's heterogeneous parallel computer systems. Even performance portability (see below) can benefit, as we shall see in this paper, as the programming system (compiler, runtime library) can build its own internal performance models for skeleton-based computations and is, in general, free to automatically select the expected fastest backend for each skeleton call.

In this work, we consider *SkePU*[2] [19] as a case study. SkePU is a domain-specific skeleton programming language embedded into modern C++. It pro-

[2] `https://skepu.github.io/`

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

Fig. 1: Formula for computing the performance portability (PP) metric [26]

vides currently 7 data-parallel variadic skeletons as well as STL-like generic abstractions for multidimensional operand data in memory and abstractions for different data access patterns. For each skeleton, implementations (*backends*) are provided for single-threaded and multi-threaded (OpenMP) CPU execution, single- and multi-GPU execution in CUDA and OpenCL, hybrid CPU-GPU execution, and cluster execution. The data abstractions for 1D, 2D, 3D and 4D generic array-based operands wrap array based data structures in main memory; they are referred to as "*smart*" *data-containers* as they transparently perform run-time optimizations such as coherent software caching and lazy device memory allocation and copying [12] and data locality optimizations [20]. SkePU is implemented by a light-weight source-to-source precompiler and an include-only runtime library for the interfaces and implementations of skeletons and data abstractions which makes intensive use of template metaprogramming in C++. SkePU is available as open-source with a permissive modified BSD license. Recent examples for the use of SkePU in HPC applications are described in [25].

2.2 Performance portability

Performance portability is commonly understood as the ability of an application codebase, together with tools or layers of the hardware-software stack, to automatically achieve decent performance across different target architectures without significant changes. This is an intuitive quantity which is harder to define formally. In this work, we define performance portability in terms of *application efficiency* across platforms, which means the measured performance as a fraction of the best observed performance on the specific platform (by some other, ideally well optimized and tuned, code). This is contrasted with *architectural efficiency*, which instead compares the measured performance to the *peak hardware throughput* of the target system. Architectural efficiency is by definition lower than application efficiency.

Pennycook et al. [26] recently proposed a concrete metric for performance portability (PP) in terms of efficiency measurements on a given specific set H of platforms, see Fig. 1. It is determined as the harmonic mean of the (application or architectural) efficiencies across all platforms in H, or 0 if any of these platforms does not support the computation (i.e., the program crashes or produces incorrect[3] results).

Notably, this metric is dependent on the considered platform set H and not an inherent property of an application. It makes sense to consider PP for architecturally related subsets of H, e.g., for the subset of all GPUs of interest, of all CPUs, as well as for all platforms together, because this will show common performance problems with certain architectural properties, as well as the sensitivity to product family variations (e.g., cache sizes). Taking the harmonic mean has a similar effect as minimization over the efficiencies for the different platforms, and also reflects the intuitive notion that adding new platforms to H will generally reduce the PP score, while algorithmic specialization (i.e., adding special code paths for some platforms) will generally increase the PP score.

We use this PP metric in this work, and time will tell if it achieves broad adoption in the scientific community.

2.3 Benchmarks

There is an ongoing effort to create SkePU implementations, and subsequently evaluations, of many benchmark workloads across several benchmark suites. Such suites include Rodinia [9], PARSEC [8] and its parallel derivate P3ARSEC [13], PolyBench [30], and NAS Parallel Benchmarks [7,23,4]. The complexity and effort required for benchmarking parallel programming models, interfaces, and frameworks is well-known [28] and examples of ongoing efforts to simplify and standardize parallel benchmark suites are many, including P3ARSEC and Task Bench. These efforts are seemingly conducted mostly in parallel to the work toward a widely accepted performance portability metric, and it remains one of the scientific goals of high-level parallel programming to merge these efforts into a methodology to evaluate programming models and frameworks across both application domains and platforms in a holistic process.

2.3.1 STREAM benchmark suite

The STREAM benchmark suite by McCalpin [24] of University of Virginia is primarily intended for measuring and comparing memory bandwidth of high-performance computing architectures. Versions of STREAM for distributed memory systems also exist, e.g. using MPI, but in this work we are working with single-node systems. However, heterogeneous architectures equipped with accelerators with separate memory spaces are also considered here.

[3] The notion of "correct" behavior is not always obvious: especially when using accelerators or for more efficient parallelization, one might want to tolerate small differences in the result values within some limits, e.g. with respect to round-off errors of floating-point computations or the behavior of parallel pseudorandom number generation.

2.3.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) was created and made available by the NASA Advanced Supercomputing division for benchmarking parallel hardware and software in the Computational Fluid Dynamics (CFD) application domain [7]. The benchmark suite is composed of five kernels (named Embarrassingly Parallel - EP, Multi Grid - MG, Conjugate Gradient - CG, Discrete 3D Fast Fourier Transform - FT, and Integer Sort - IS) and three pseudo-applications (named Block Tri-diagonal solver - BT, Scalar Penta-diagonal solver - SP, and Lower-Upper Gauss–Seidel solver - LU). They are well-known in the research community and represent recurrent linear algebra computations. The user can execute these programs with predefined workloads (named classes S, W, A, B, C, D, E, and F) that vary the computational problem's size. The original version was written in Fortran and the parallel implementations were in OpenMP and MPI. In recent years, an effort was made to provide parallel versions for C/C++ parallel programming frameworks on multi-core systems [22,23] as well as heterogeneous parallel programming on GPUs [5,4,16].

3 Related Work

Deakin et al. [14] provide ports of the STREAM benchmark set to several single-source parallel programming models: Kokkos, RAJA, OpenMP 4.x, OpenACC, SYCL, OpenCL and CUDA. (The "modern" version of SkePU that we use in our work did not yet exist at that time.) They evaluate performance portability for these programming models on a variety of GPU and CPU types from different vendors, including Intel Xeon Phi (Knights Landing).

A number of papers such as [17] present and evaluate multi-platform implementations of the NPB, including GPU implementations in OpenCL and CUDA. A recent review and comparison of previous work on NPB parallelizations is given by Löff et al. [23]. In the interest of brevity, we focus here on work based on single-source high-level programming models. Xu et al. [29] study the efficiency of the NAS Parallel Benchmarks rewritten in the directive-based single-source programming model OpenACC on GPUs and identify performance-critical GPU-specific optimizations such as array privatization that need be addressed by an OpenACC compiler. Performance is compared to hand-written OpenCL code for the NPB. A fundamental difference from our (NPB) implementations in SkePU is that SkePU is based on the more high-level skeleton concept rather than annotated sequential loop-based code as in OpenACC, so that SkePU's skeleton backend implementations are not constrained by the sequential code structure.

4 Method

We compare an implementation of the STREAM benchmark suite in SkePU to the reference implementation across a set of 10 target *platforms*. Application *efficiencies* are calculated from the performance data, measured in terms of throughput data-rate, and used as a basis for calculating the performance portability metric. The four STREAM workloads are evaluated on single and double precision floating-point data resulting in eight applications in total across ten platforms. We also document the programming effort required to implement the STREAM benchmarks in SkePU.

This work was conducted in three main steps: benchmark selection, platform selection, and performance evaluation.

4.1 Benchmark selection and implementation

The first step of this work was to select a set of benchmarks used to evaluate performance portability of SkePU. As the metric used takes efficiency data as input, a baseline requirement was to find data-parallel benchmark applications with independent reference-implementations available for all target platforms used in the evaluation. The choice fell on the STREAM benchmark suite as it is simple and well-known, facilitating the above requirement also with consideration of the limited scope of this project. SkePU has also not been evaluated on STREAM before, and a secondary aim is to further grow the set of benchmarks targeted by SkePU as part of the project.

In addition to the lightweight STREAM workloads, we also considered NPB as a means to select additional possible evaluation points, since these are different computations and are considered standard benchmarks for HPC evaluation. Like STREAM, NPB has not been subject to SkePU parallelization before, and given the recent work on NPB implementations in both C++ parallel CPU frameworks and CUDA for Nvidia GPUs [23,4], we have good reference points for efficiency comparison against SkePU implementations. However, SkePUizing the entirety of NPB will be a future work, because the experience from this initial effort will indicate the viability of such a project. We therefore select two NPB kernels: *EP* and *CG*. EP shares similar properties to the STREAM kernels, such as being memory-bound, while the computational pattern modeled is not only a single Map, like STREAM, but rather a MapReduce, with a global reduction. One aspect that is also shared between STREAM and EP is that, during the entire program runtime, synchronization is only necessary at the start and end phases. In typical SkePU usage, we expect also to be able to handle multiple skeleton invocations in sequence in an efficient manner. To evaluate this, CG provides an iterative workload with each iteration also containing several global synchronization points. I.e., a SkePU implementation will have to consist of a substantial sequence of skeleton calls.

While the STREAM reference implementation in C could be applied mostly unchanged, manual implementation work was required for SkePU versions of the workloads. Reference STREAM results for GPUs was based on open-source third-party implementations, but those required more tweaks to provide compatibility and a fair evaluation methodology. Table 1 lists the four workloads that make up STREAM: `copy`, `scale`, `add`, and `triad`.

Table 1: Benchmark workloads in STREAM.

Benchmark	**Memory reads**	**Memory writes**	**Total mem accesses**	**Unique mem accesses**	**FLOPS**
`copy`	1	1	2	2	0
`scale`	1	1	2	2	1
`add`	2	1	3	3	1
`triad`	2	1	3	3	2

As NPB base-line to compare with, we use the CUDA implementations of EP and CG described in [23] as these have been experimentally shown [23] to perform on-par or better than other state-of-the art EP and CG implementations such as [17], see also Section 3.

4.2 Platform selection

The platform selection is primarily guided by availability, but with the goal of having at least two physically distinct systems represented and also several different types of computational units. In this work, we are using the term **platform** in the sense of a *compilation target* + a *physical host system*, e.g., ”sequential C++ processing on a server” or ”laptop GPU with OpenCL runtime”.

The platforms are derived from three physical systems: laptop computer, the local server *Excess*, and the supercomputer cluster *Tetralith/Sigma*[4].

The laptop is equipped with a single 2 GHz quad-core *Intel Core i5* with *Intel Iris Plus* graphics and 16 GiB main memory and runs Mac OS. This GPU notably does not support double-precision compute kernels and cannot run CUDA programs.

Excess is a 12-core server (two six-core *Intel Xeon E5-2630L* CPUs with two-way hardware multi-threading, thus 24 logical cores) with one *Nvidia K20c* GPU and 64 GiB main memory. This system runs Ubuntu.

Tetralith and Sigma are large clusters with thousands and hundreds of nodes, respectively. We only use one node at a time in this work, of various configurations. Each Tetralith/Sigma node contains two Intel Xeon Gold 6130 CPUs for a total of 32 cores, with no hardware multi-threading. The minimum amount of node memory is 96 GiB, with more available on GPU nodes. Tetralith GPU nodes are equipped with one Nvidia Tesla T4 GPU with 16 GiB of GPU memory, and the Sigma GPU nodes contain 4 Nvidia Tesla V100 SXM2 with 32 GiB of memory each.

From these systems we derive and define 14 target platforms, using either different computational units or backend interfaces. Sequential platforms are targeted with C++ (OpenMP extensions disabled). For multi-core platforms

[4] Tetralith and Sigma are sister clusters and share most of their hardware and software. Each clusters offer special nodes equipped, for example, with different GPU accelerators. We have used nodes from both clusters in this work.

Table 2: Platforms used in the performance portability evaluation.

Platform name	**System**	**Progr. model**	**Note**
`excess-seq`	Excess	C/C++	
`excess-omp-12`	Excess	OpenMP	All physical cores
`excess-omp-24`	Excess	OpenMP	All logical cores
`excess-cl`	Excess	OpenCL	
`excess-cuda`	Excess	CUDA	
`laptop-seq`	Laptop	C/C++	
`laptop-omp-4`	Laptop	OpenMP	All physical cores
`laptop-omp-8`	Laptop	OpenMP	All logical cores
`laptop-cl-flush`	Laptop	OpenCL	
`laptop-cl-noflush`	Laptop	OpenCL	
`cluster`	Tetralith/Sigma	C/C++	
`cluster-omp-32`	Tetralith/Sigma	OpenMP	All physical cores
`cluster-v100-cuda`	Sigma	CUDA	
`cluster-t4-cuda`	Tetralith	CUDA	

we chose to run OpenMP with as many threads as there are physical and logical cores, respectively, but no thread pinning or similar was utilized.

On Excess, the GPU is targeted with both OpenCL and CUDA as separate platforms. The laptop GPU has a platform only for OpenCL, but for the sake of investigation, we define one ”platform” as requiring a flush of GPU memory buffers between each measurement sample run, and one that only requires synchronizing with the GPU. This is purely a software differentiator but allows some insights into GPU bandwidth bottlenecks.

4.3 Evaluation method

The STREAM and NPB benchmarks are compiled with GCC 10 and 11 in C++ mode (even for the reference program) with -O3 optimization level and no further optimization passes explicitly turned on. Evaluation is done using the default STREAM parameters: array sizes of 10 million elements and 10 runs per data point; and the default NPB problem size classes, except the D and E classes, as the time and memory requirements are impractically large.

Evaluation on STREAM benchmarks is repeated on single precision floating-point numbers and double-precision floating-point numbers, resulting in a memory load of 0.1 GB for the former and 0.2 GB for the latter.

5 Results

Results are reported in three evaluated categories: programming effort, performance, and performance portability.

Listing 1: Smart data-container model SkePU-STREAM.

```
skepu::Vector<STREAM_T> ske_a(&a[0], STREAM_ARRAY_SIZE+OFFSET, false);
skepu::Vector<STREAM_T> ske_b(&b[0], STREAM_ARRAY_SIZE+OFFSET, false);
skepu::Vector<STREAM_T> ske_c(&c[0], STREAM_ARRAY_SIZE+OFFSET, false);
```

5.1 Parallel implementation in STREAM

The baseline code-churn of adapting the reference STREAM benchmarks to SkePU is very low. SkePU first needs one line minimum for its header inclusion: `#include <skepu>`.

5.1.1 Data model

Next, the three arrays used in the workloads needs to be wrapped in SkePU smart data-containers: SkePU cannot use raw arrays.

The smart data-container definitions in Listing 1 utilize a SkePU smart data-container feature that claims ”ownership” of raw memory regions and uses them internally for the lifetime of the data-container. This pointer is used as the memory buffer for sequential CPU or multi-threaded OpenMP backends; for GPUs additional device memory is allocated. The `false` argument passed here indicates that SkePU shall not deallocate the pointer as the lifetime of the container ends.[5]

Finally, the skeleton instances need to be actually invoked during the benchmarking part of the program. Normally, we could replace the existing for-loops which forms the reference implementations of each workload, but the STREAM codebase provides

5.1.2 Benchmarking bookkeeping

SkePU skeleton invocations are not guaranteed to be synchronous. There are explicit optimizations in the framework relying on the opposite: that invocations are lazily evaluated only when strictly necessary. In fact, since STREAM is embarrassingly parallel, the iterated invocations are perfect targets for the tiling optimization on such lazily-built invocation chains. Even when this feature is explicitly disabled, e.g. the GPU backend implementations are also partly asynchronous, relying on internal GPU driver scheduling queues for synchronization of computation and requests for memory transfers. SkePU's interface exposes a `flush` directive (and related ”skepu::external” constructs) which guarantees a full synchronization to a completed skeleton invocation, but it is arguably too strong, as this operation will also trigger deallocation of

[5] In fact, STREAM allocates these arrays on the stack, so freeing the pointers is always undefined behavior in C++.

Listing 2: STREAM kernels as SkePU skeletons.

```
auto skel_copy  = skepu::Map([](STREAM_T a)
{
  return a;
});
auto skel_add   = skepu::Map([](STREAM_T a, STREAM_T b)
{
  return a + b;
});
auto skel_scale = skepu::Map<1>([](STREAM_T a, STREAM_T s)
{
  return a * s;
});
auto skel_triad = skepu::Map<2>([](STREAM_T b, STREAM_T c, STREAM_T s)
{
  return b + s * c;
});
```

GPU memory and possible memory transfers. The benchmarking code therefore uses an internal SkePU synchronization directive between each measurement run to ensure as accurate results as possible.

Similarly, the result verification component of STREAM needs a full flush directive before it can run its checking algorithm.

5.1.3 Computations

Next, the skeletons representing the workload need to be defined. Each benchmark workload is straightforward to model with a single `Map` skeleton instance, and with a lambda expression this is done with a single statement/line-of-code per benchmark. For readability reasons, especially when presented in this report, it can be desirable to format the skeleton instances across multiple lines.

In Listing 2, `STREAM_T` is a preprocessor macro symbol from the STREAM reference code defined to be either `float` or `double` at compile-time. While SkePU sometimes struggles with macro-heavy code, this usage here is handled properly by the precompiler.

Note that the element-wise arity declared within chevrons is optional for `copy` and `add`, but mandatory for `scale` and `triad` as SkePU cannot otherwise distinguish the scalar `s` as not being an element-wise parameter.

5.2 Parallel implementation of NPB-EP

The EP kernel requires relatively little effort to adapt for SkePU. A single `MapReduce` models the entire kernel, but the reduction step is more involved than the typical ”dot-product” `MapReduce` archetype which reduces only a single value. EP computes global sums of `sx`, `sy`, and `q` values (see Listing 3), where the latter is a static array. SkePU can model this multi-way reduction

Listing 3: Excerpt from the EP reference implementation in OpenMP.

```
#pragma omp parallel
{
    double t1, t2, t3, t4, x1, x2;
    int kk, i, ik, l;
    double qq[NQ];    /* private copy of q[0:NQ-1] */
    double x[NK_PLUS];

    for (i = 0; i < NQ; i++) qq[i] = 0.0;

    #pragma omp for reduction(+:sx,sy)
    for(k=1; k<=np; k++){
      int thread_id = omp_get_thread_num();
      /* ... business logic code omitted  */
    }

    #pragma omp critical
    {
        for (i = 0; i <= NQ - 1; i++) q[i] += qq[i];
    }
} /* end of parallel region */
```

pattern either by declaring a custom type and reducing on said type, or by using variadic tuple-return syntax introduced in SkePU 3 [19]. Notably, SkePU's internal reduction implementation for the OpenMP backend performs the final summation of thread-partial sums sequentially, while the reference OpenMP code uses a mix of OpenMP pragma directives and a shared critical section, as shown in Listing 3.

5.3 Parallel implementation of NPB-CG

NPB's CG kernel is considerably more complex than the other benchmark workloads considered in this paper. This is exemplified by the fact that CG cannot be implemented with just one SkePU skeleton call; rather twelve distinct skeleton instances are invoked in sequence for each CG iteration, totaling hundreds of skeleton calls during an entire CG execution. SkePU conceptually induces a global synchronization point between skeleton invocations, and while this can result in performance bottlenecks especially for iterative workloads, it is in most cases required by the CG algorithm as it contains global reductions and other non-local dependency structures. The skeleton structure of the SkePU implementation is closely following the CUDA kernels of the reference CUDA code[6], which makes the GPU efficiency results of particular interest in this work.

In the process of porting the CUDA CG kernel to SkePU, we noted two limitations of the SkePU interface. Firstly, the current SkePU version does

[6] This is, in our experience, a good approach for SkePUizing applications with GPU implementations already available.

Listing 4: Sparse matrix-vector multiplication in SkePU.

```
auto skepu_skel_three = skepu::Map([](
    skepu::Index1D     index,
    skepu::Vec<int>    colidx,
    skepu::Vec<int>    rowstr,
    skepu::Vec<double> a,
    skepu::Vec<double> p) -> double {
  int begin = rowstr(index.i);
  int end   = rowstr(index.i + 1);
  double sum = 0.0;
  for (int k = begin; k < end; ++k) {
    sum += a(k) * p(colidx(k));
  }
  return sum;
});

skepu_skel_three(q, colidx, rowstr, a, p);
```

not include a smart data-container abstraction for sparse matrices. A sparse matrix can be modeled by a series of vector containers, but with additional programmer effort. Secondly, we observe a limitation in matrix-vector multiplication patterns as modeled by `Map` and SkePU's random-access container interface (”container proxies”). The typical way of encoding such computations in SkePU is shown in Listing 4.

Compared to the reference CUDA kernel in Listing 5, the SkePU variant parallelizes strictly on the elements in the output vector. The CUDA kernel allocates one block of threads for each output element, and can share the row calculations among the threads in the block, which in the SkePU case has to be managed by one thread. The reason for the discrepancy is that SkePU user functions are completely independent with no observable side effects, and as such the user functions executed within the same GPU block can neither communicate, synchronize, or even access the block size.

With the exception of this parallelization scheme, all of the CUDA kernels are straightforward to adapt into SkePU skeletons while simplifying and reducing code size.

5.4 Performance evaluation

The benchmark performance results for STREAM are presented in Figures 2 and 3, visualizing the relative efficiency of the SkePU programs as compared to reference STREAM results. Note that the single-precision results contain two additional platforms: the laptop GPU is evaluated with and without explicit memory flushes between test runs.

Similarly, the performance results for NPB, EP in Figure 4 and CG in Figure 5, are presented as collected on the 12 platforms which can run double-precision calculations.

Listing 5: Sparse matrix-vector multiplication in CUDA.

```
__global__ void gpu_kernel_three(int colidx[],
    int rowstr[],
    double a[],
    double p[],
    double q[]) {
  double* share_data = (double*)extern_share_data;
  int j = (int) ((blockIdx.x*blockDim.x+threadIdx.x) / blockDim.x);
  int local_id = threadIdx.x;
  int begin = rowstr[j];
  int end   = rowstr[j+1];
  double sum = 0.0;
  for (int k=begin+local_id; k<end; k+=blockDim.x) {
    sum = sum + a[k]*p[colidx[k]];
  }
  share_data[local_id] = sum;

  __syncthreads();
  for (int i=blockDim.x/2; i>0; i>>=1) {
    if (local_id<i) { share_data[local_id]+=share_data[local_id+i]; }
    __syncthreads();
  }
  if (local_id==0){q[j]=share_data[0];}
}

gpu_kernel_three<<<blocks_per_grid_on_kernel_three,
    threads_per_block_on_kernel_three,
    size_shared_data_on_kernel_three>>>(
        colidx_device, rowstr_device,
        a_device, p_device, q_device);
```

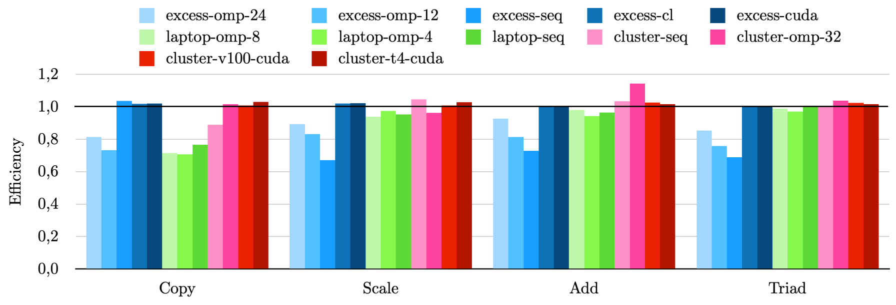


Fig. 2: Efficiency per platform on double-precision STREAM workloads.

5.5 Performance portability

We use *efficiency cascade plots* in Figure 6 as proposed in [27] to visualize performance portability of the workloads when considering successively smaller platform sets. The rightmost data point in each line shows the PP metric for the entire set of platforms, and in each successive data point to the left, the least efficient platform is removed from the set and the PP metric is

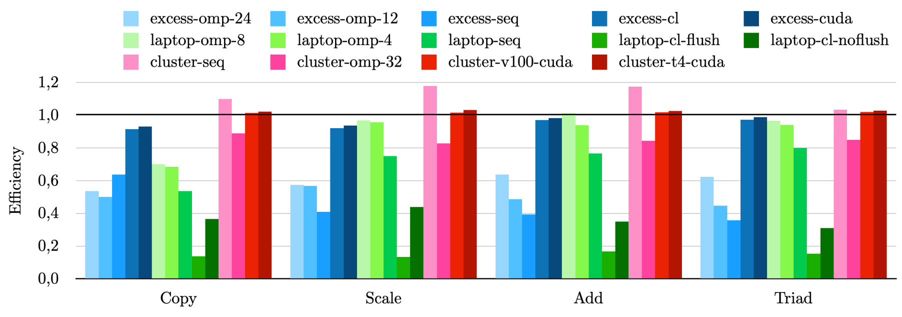


Fig. 3: Efficiency per platform on single-precision STREAM workloads.

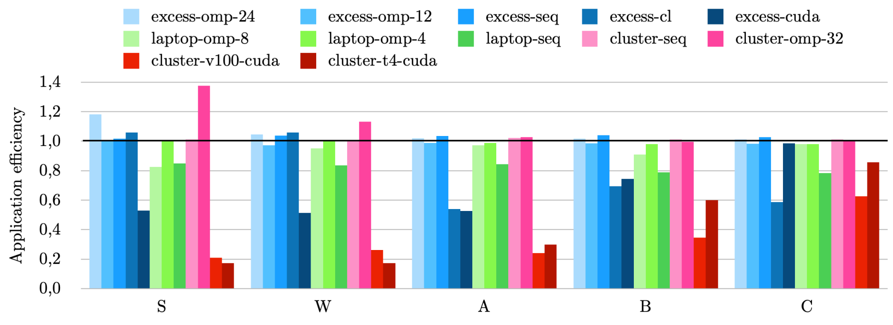


Fig. 4: Efficiency per platform and problem size class on NPB-EP.

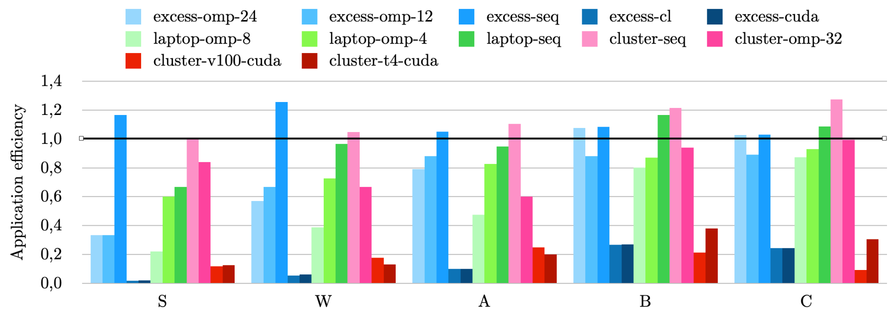


Fig. 5: Efficiency per platform and problem size class on NPB-CG.

re-evaluated on the new subset. We use this method of visualization as the singular PP number is heavily influenced by the least efficient platform in the set. By computing the PP metric for the most interesting subsets of platforms, more detailed information can be conveyed.

Note that the order of platform removals may be different across workloads, so the different lines in each graph are not directly comparable.

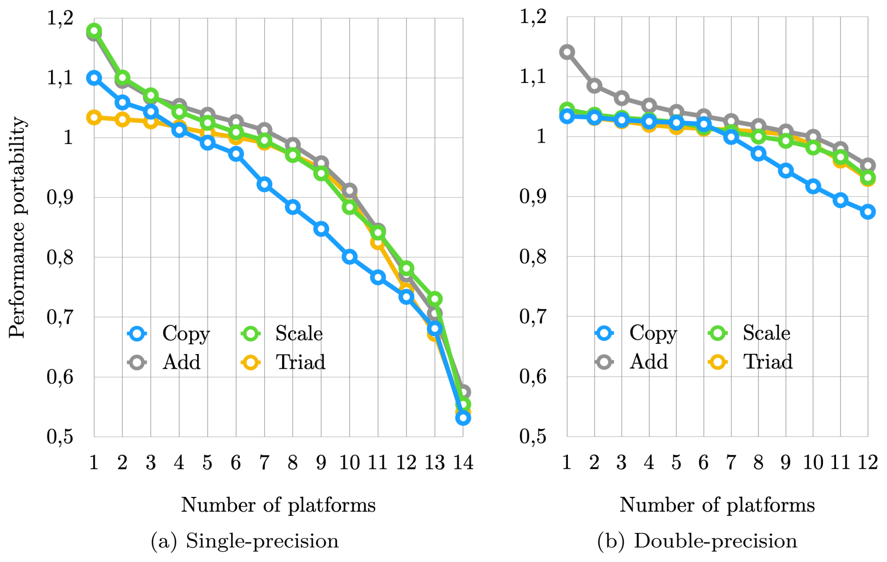


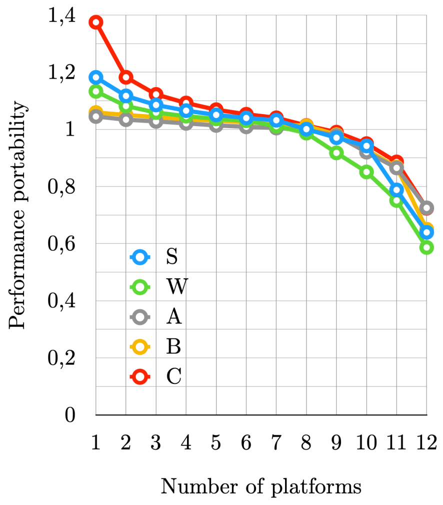
Fig. 6: Efficiency cascade plots for STREAM kernels by kernel and precision.

This approach of visualization makes the different trade-offs between single- and double-precision workloads apparent. We see that the single-precision PP metrics are overall significantly lower, but the programs which depends on double-precision suffers from incompatibility on two platforms and as such the PP metric drops to zero on the rightmost platform subsets here.

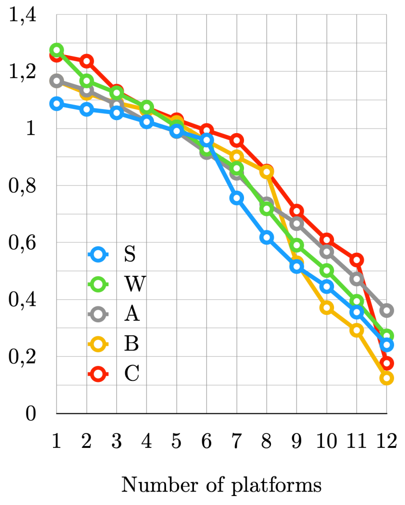
The efficiency cascade plots clearly show that the SkePU implementation of the `copy` workload exhibits worse performance portability than the other three workloads, which cluster tightly together in the plot. This fact is not as easy to discern in the traditional bar graphs.

6 Discussion

From the STREAM results, we observe that the SkePU programs with single-precision workloads are less efficient than the corresponding double-precision ones. The STREAM reference implementation is a set of microbenchmarks in a tightly controlled environment: program parameters are all compile-time configuration options, including array size. The arrays are allocated on the program stack, so all program addresses are statically known, allowing for far-reaching optimization opportunities by the compiler.In comparison, SkePU skeleton code is generated behind layers of abstractions. SkePU smart data-containers operate on pointers that are normally dynamic allocations of arbitrary size, and the backend selection is entirely a run-time decision. This means that even though SkePU is embedded into the STREAM reference codebase



(a) EP



(b) CG

Fig. 7: Efficiency cascade plots for NPB kernels by problem size class.

with only minor alterations, the static information is unlikely to propagate all the way down to the skeleton internals unless the backend compiler is extremely aggressive with global static analysis. Examples of optimization stages that can be affected include inlining of SkePU user function calls, loop unrolling, loop fusion, auto-vectorization, and pointer de-aliasing.

The STREAM workloads are all memory-bound, so the hyper-threaded platforms do not scale linearly in throughput from the baseline platforms with thread counts matching the number of physical cores. However, the laptop still benefits quite a bit from hyper-threading, while Excess is largely flat.

The application efficiency numbers presented in the paper could be made more accurate if per-system-tuned reference implementations were used. Performance efficiency properties differ a lot between the evaluated systems, even for the same types of backend targets, and several times the efficiency is recorded as over 100%. This indicates that the task of finding optimal reference implementations requires implementations with built-in platform-tuning capabilities. In particular, the EP kernel frequently results in cases where the SkePU version outperforms the reference code. For OpenMP, it is likely that the synchronization approach by SkePU contributes to this, as discussed in Section 5.2.

For the CG kernel, the efficiency results on GPU platforms are very low. For smaller problem sizes, the skeleton invocation overhead is contributing to these inefficiencies, but more importantly SkePU programs induce initialization delays due to environment set-up and lazy memory allocations on device, and so on. These delays may still occur in hand-written implementations, but the abstractions in SkePU make it impossible to assure that initialization delays happen outside of the critical timing regions, without considerable code instrumentation. For larger problem sizes, such overhead is a smaller quota of the total execution time. For large CG sizes, instead, the issue discussed in Section 5.3 becomes important. The matrix-vector multiplication phase becomes dominating at large problem sizes, and the parallelization inefficiency in the SkePU version becomes a performance bottleneck.

7 Conclusions

Of the three benchmarks evaluated in this work, we have shown that STREAM benchmarks, specifically in their original double-precision form, and the EP kernel in NPB result in high efficiency and performance portability when implemented using the SkePU skeletons and smart data-containers. For the CG benchmark, the results are not as consistent and SkePU's efficiency is highly platform-dependent. We have identified specific performance bottlenecks in SkePU, in particular for GPU backends.

The results demonstrate that SkePU-parallelized programs can achieve good application efficiency and even outperform hand-written parallel code in some cases, even though SkePU code is single-source and often shorter than any of the individual platform-specific programming models. Note also that this work does not even take the auto-tuning functionality of SkePU into account, which does provide an advantage for SkePU compared to distinct and unrelated parallel implementations.

8 Future work

For a future revision of the paper, we plan to extend the performance portability evaluation to also consider multi-node platform configurations on the Tetralith/Sigma cluster. If possible, we would also like to evaluate multi-node GPU platforms or hybrid CPU-GPU variants.

In the future, we intend to extend the pattern set of SkePU with a multi-backend sorting skeleton. Such a construct would be useful for a convenient and efficient implementation of the IS kernel in NPB. In the long term, we aim for complete SkePU coverage of the NPB kernels. Further extending the available SkePU implementations of benchmarks and other representative applications remains a topic of interest.

A further direction for future work is to extend our performance portability analysis of SkePU by including other single-source high-level parallel programming models, such as directive based models like OpenACC as well as other single-source skeleton programming frameworks.

Acknowledgments

This work has been partly funded by ELLIIT, project GPAI. We thank SNIC and NSC for providing access to HPC resources Tetralith (SNIC 2021/22-971) and Sigma (LiU-gpu-2021-1) that were used for this work.

References

1. Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons, Ltd, 2017.
2. Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, Marco Danelutto, and Luiz G. Fernandes. Assessing coding metrics for parallel programming of stream processing programs on multi-cores. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 291–295, 2021.
3. Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, and Luiz G. Fernandes. A parallel programming assessment for stream processing applications on multi-core systems. *Computer Standards & Interfaces*, 2022.
4. Gabriell Araujo, Dalvan Griebler, Dinei A. Rockenbach, Marco Danelutto, and Luiz G. Fernandes. NAS parallel benchmarks with cuda and beyond. *Software: Practice and Experience*, n/a(n/a), 2021.
5. Gabriell Alves de Araujo, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo Fernandes. Efficient NAS parallel benchmark kernels with CUDA. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 9–16, 2020.
6. M. Arvanitou, A. Ampatzoglou, N. Nikolaidis, A. Tzintzira, A. Ampatzoglou, and A. Chatzigeorgiou. Investigating trade offs between portability, performance and maintainability in exascale systems. In *46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 59–63, 2020.
7. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 158–165, New York, NY, USA, 1991. Association for Computing Machinery.
8. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.
9. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
10. Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
11. Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation.* Pitman and MIT Press, Cambridge, Mass., 1989.
12. Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.
13. Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing parallel patterns out of the corner: The p3arsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 14(4), October 2017.
14. Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across

diverse parallel programming models. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, pages 489–507, Cham, 2016. Springer International Publishing.
15. David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, 2017.
16. Y. Do, H. Kim, P. Oh, D. Park, and J. Lee. SNU-NPB 2019: Parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 93–105. IEEE, 2019.
17. Youngdong Do, Hyungmo Kim, Pyeongseok Oh, Daeyoung Park, and Jaejin Lee. Snu-npb 2019: Parallelizing and optimizing npb in opencl and cuda for modern gpus. In *Int. Symposium on Workload Characterization (IISWC)*, pages 93–105, 2019.
18. Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, apr 2012.
19. August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49:846–866, 2021.
20. August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019.
21. Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, March 2017.
22. Dalvan Griebler, Junior Löff, Gabriele Mencagli, Marco Danelutto, and Luis Gustavo Fernandes. Efficient NAS benchmark kernels with C++ parallel programming. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740, 2018.
23. Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.
24. J. D. McCalpin. STREAM benchmark, 1995.
25. Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I. Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, Samuel Thibault, Athanasios Salamanis, Theodoros Ioakimidis, and Dionysios Kehagias. Exa2pro: A framework for high development productivity on heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):792–804, 2022.
26. S.J. Pennycook, J.D. Sewall, and V.W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.
27. Jason Sewall, S. John Pennycook, Douglas Jacobsen, Tom Deakin, and Simon McIntosh-Smith. Interpreting and visualizing performance portability metrics. In *IEEE/ACM Int. Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–24, 2020.
28. Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichlerk, Patrick McCormick, and Alex Aiken. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.
29. Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yuonghong Yuan, and Barbara Chapman. NAS parallel benchmarks for GPGPUs using a directive-based programming model. In *Proc. LCPC 2014, LNCS 8967*, pages 67–81. Springer, 2015.
30. Tomofumi Yuki and Louis-Noël Pouchet. Polybench 4.0, 2015.

Noname manuscript No.
(will be inserted by the editor)

Declarative Data Flow in a Graph-Based Distributed Memory Runtime System

Fabian Knorr · Peter Thoman · Thomas Fahringer

Received: date / Accepted: date

Abstract Runtime systems can significantly reduce the cognitive complexity of scientific applications, narrowing the gap between systems engineering and domain science in HPC. One of the most important angles in this is automating data distribution in a cluster. Traditional approaches require the application developer to model communication explicitly, for example through MPI primitives. Celerity, a runtime system for accelerator clusters heavily inspired by the SYCL programming model, instead provides a purely declarative approach focused around access patterns. In addition to eliminating the need for explicit data transfer operations, it provides a basis for efficient and dynamic scheduling at runtime. However, it is currently only suitable for accessing array-like data from runtime-controlled tasks, while real programs often need to interact with opaque data local to each host, such as handles or database connections, and also need a defined way of transporting data into and out of the virtualised buffers of the runtime. In this paper, we introduce a graph-based model and a declarative API to express side-effect dependencies between tasks and move data from the runtime context to the application space.

Keywords Runtime System · DAG · Accelerator · Data Flow · API

1 Introduction

Modern scientific and High Performance Computing (HPC) is a challenging environment for software engineering. In order to increase compute throughput despite the ever tighter constraints on power efficiency, modern supercomputer hardware embraces heterogeneous processor architectures, deep memory hierarchies with non-uniform access characteristics and specialized network topologies. Most of the increasing complexity is directly passed onto the application

Fabian Knorr, Peter Thoman, Thomas Fahringer
University of Innsbruck, Austria
E-mail: {fabian,petert,tf}@dps.uibk.ac.at

developer in the form of intricate APIs—and in some cases entirely disjoint programming models—allowing optimal utilization of the available technologies in every use case. While the resulting increase in up-front development cost can be acceptable for large-scale applications such as general-purpose simulation toolkits, specialized single-use codes for novel discovery will have not the development budget required to test a research hypothesis that might turn out to be a dead-end.

Distributed Memory Runtime Systems are an established concept for easing select aspects of the complexity in these heterogeneous systems, such as performance portability, optimizing execution schedules with unbalanced loads or automatic data migration between computation steps. They typically incur a trade-off between expressiveness, correctness guarantees, and the level of permitted user control.

The mission statement of Celerity[15], a task-based distributed memory runtime system for accelerator clusters, is to make programming heterogeneous HPC systems more accessible and time-efficient by facilitating low-effort porting of single-node SYCL[1] accelerator programs. The Celerity model decomposes a problem into compute tasks and their data dependencies, using subdivision of the computational index spaces to transparently distribute work onto a cluster. Celerity exposes a declarative data flow API operating on virtualized buffers to infer dependencies and necessary data transfers in the distributed program, relieving the programmer of manual scheduling decisions and data relocation.

Celerity's APIs allow it to statically guard against unmanaged buffer accesses and race conditions between tasks, greatly reducing the potential for programming errors. The runtime implementation benefits from an information-dense API that supports the generation of efficient execution schedules, while the user is assured of their code's correctness by an expressive programming paradigm, allowing them to focus on core algorithm development instead.

A notable use of Celerity is the Cluster-accelerated magnetohydrodynamics simulation CRONOS [10], which demonstrates the viability of the Celerity model for real-world applications. It is also sufficiently generic to serve as the basis for further abstractions like the Celerity High-level API [16], a programming model exposing data transformations using composable functional operator pipelines similar to the C++20 *ranges* library.

While domain-specific problems can be fully described by compute tasks and data dependencies between them, real codes need additional features to perform I/O operations with side effects. Incremental porting from single-node SYCL applications, an important development goal of Celerity, further requires data movement between the legacy host application and runtime-controlled virtual buffers.

In this paper, we present an approach to augmenting the Celerity execution model with declarative mechanisms for tracking I/O side effects and safely moving data out of the managed context on pre-existing synchronization points.

```
using mat = buffer<float, 2>;
const range<2> size{256, 256};

void diag(handler& cgh, mat& M, float d) {
    accessor m{M, cgh, access::one_to_one{}, write_only, no_init};
    cgh.parallel_for(size, [=](item<2> i) {
        m[i] = i[0] == i[1] ? d : 0;
    });
}

void mul(handler& cgh, mat& A, mat& B, mat& C) {
    accessor a{A, cgh, access::slice<2>{1}, read_only};
    accessor b{B, cgh, access::slice<2>{0}, read_only};
    accessor c{C, cgh, access::one_to_one{}, write_only, no_init};
    cgh.parallel_for(size, [=](item<2> i) {
        c[i] = 0;
        for(size_t k = 0; k < i.get_range(0); ++k) {
            c[i] += a[i[0]][k] * b[k][i[1]];
        }
    });
}

void is_diag(handler& cgh, mat& C, float d, buffer<bool>& ok_buf) {
    accessor c{C, cgh, access::one_to_one{}, read_only};
    auto ok_r = reduction(ok_buf, cgh, sycl::logical_and<bool>{},
            property::reduction::initialize_to_identity{});
    cgh.parallel_for(size, ok_r, [=](item<2> i, auto &ok) {
        ok.combine(c[i] == (i[0] == i[1] ? d : 0));
    });
}

int main() {
    distr_queue q;
    mat A{size}, B{size}, C{size};
    q.submit([=](handler& cgh) { diag(cgh, A, 2); });
    q.submit([=](handler& cgh) { diag(cgh, B, 3); });
    q.submit([=](handler& cgh) { mul(cgh, A, B, C); });
    buffer<bool> ok{1};
    q.submit([=](handler& cgh) { is_diag(cgh, C, 6, ok); });
    return /* ok[0] is true */ ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 1: Simple Celerity program computing the product of two diagonal matrices and verifying the result.

2 The Celerity Runtime System

Celerity is a high-level C++ API and runtime system bringing the SYCL [1] accelerator programming model to distributed-memory clusters. Using an enhanced declarative description of data requirements, it transparently distributes compute kernels onto the nodes of a cluster while maintaining an API very close to its single-node ancestor. Celerity has evolved significantly beyond what has previously been published [14][15], so we give a broad overview of the interface and execution model.

Listing 1 exemplifies the source code of a typical Celerity application. The `main` function allocates three two-dimensional buffers for square matrices and instantiates a *distributed queue*. It then launches a sequence of kernels that initialize A and B as diagonal matrices (`diag` function) and compute the naïve matrix product $C := A \cdot B$ (`mul` function). Finally, the result is verified by launching a fourth kernel that computes the expected value of each c_{ij} and combines the results using a distributed reduction over the `&&` operator.

Work is submitted to the asynchronous distributed queue in the form of *command group functions*, which are implemented as lambdas receiving a `command group handler` called `cgh` in the example. A command group declares a set of buffer requirements and specifies the work to be executed.

Buffer access is guarded by *accessors*, which bind buffers to the command group handler and inform Celerity of the mode of access and the access ranges through *range mappers* (here `one_to_one` and `slice`). Captured inside the kernel function that is passed on to `parallel_for`, these accessors facilitate reading and writing of the actual buffer contents.

All submissions to the distributed queue happen asynchronously and instruct Celerity to build an internal representation of data requirements and execution ranges. The actual scheduling, distribution and execution of the submitted kernels within the cluster is transparently managed by the runtime. The completion of all submitted command groups is finally awaited implicitly by the `~distr_queue()` destructor.

As indicated by the comment in the last line of `main`, Celerity does not have a designated mechanism for transporting data managed by the runtime back to the host application. Closing this gap is non-trivial and a core contribution of this work, for which workarounds need to be inserted currently.

2.1 Celerity's Graph-Based Execution Model

Execution of a Celerity program is distributed unto *nodes*, where a designated *master node* creates the execution schedule for the entire cluster and determines how data and computational load is distributed. This centralized approach has the potential to incorporate dynamic scheduling decisions such as load balancing at runtime without requiring costly synchronization between equal nodes in a distributed scheduling setting. By relying on fully asynchronous work assignment, Celerity is able to avoid the scalability problems that a more traditional lock-step implementation of centralized scheduling would be certain to encounter.

As command groups are submitted from the application thread of a Celerity program, a coarse-grained, directed acyclic graph (DAG) called the *task graph* is constructed. Each command group creates a corresponding task node, and data dependencies between command groups manifest as true- or anti-dependencies as if the entire program was executed on a single node.

On the master node, the scheduler then constructs a fine-grained *command graph* that models the distributed executions and all data transfers that arise

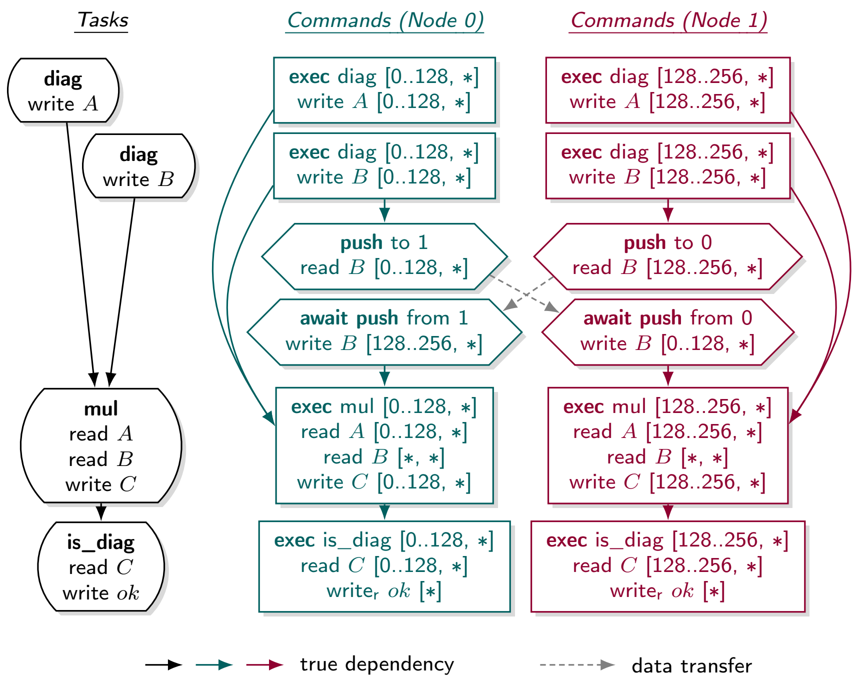


Fig. 1: Task graph (left) and command graph (right) arising from Listing 1 for two nodes in stable Celerity. Kernel execution commands show the 2-dimensional iteration sub-range and the resulting data requirements as assigned by the scheduler. In each dimension, the interval a..b includes a but excludes b, and ∗ denotes the entire range. The necessary inter-node data exchange generates auxiliary *push* / *await push* command pairs.

with it. Commands are always bound to a particular node, but the precise projection of tasks onto commands varies with the task type. For example, device execution tasks, which are generated from command groups invoking `handler::parallel_for`, may be split such that each worker node receives one part of the total execution range.

Figure 1 shows possible task and command graphs for the program in Listing 1. While the task graph reflects the high-level dependency structure visible in the source code, the command graph contains only dependencies induced by the execution subranges on each node.

Within task and command graph, dependencies are assigned based on the access modes of buffer accesses and the submission order. For example, a command group with write access followed by a command group with read access to the same buffer region will generate a true dependency, while the inverse order will generate an anti-dependency.

A unique concept in Celerity, and one of the major points where its API differs from SYCL, are *range mappers*. These projections, required on each accessor, inform the runtime which portions of each buffer an arbitrary subdivision of the execution space will access.

The stream of serialized commands is forwarded to the respective worker nodes, which place them into their *executor queue*. The executor of each worker node will then make its own local scheduling decisions to best allocate its resources to the pending commands. While all nodes construct identical task graphs in parallel, the command graph structure only exists on the master node in its full form. Pure worker nodes only reconstruct the relevant dependency graph locally from the serialized commands.

3 Modeling Node-Local Side Effects

SYCL and Celerity share the concept of *host tasks* that asynchronously schedule the execution of arbitrary code on the host, avoiding host-device synchronization and scheduler stalls. Similar to the common device tasks, host tasks can read and write buffers through the accessor mechanism. Additionally, they are able interact with operating system APIs such as file I/O and reference objects allocated in the main thread, since they operate in the same address space. Once more than one host task references the resource, the resulting synchronization or ordering constraints need to be enforced during execution.

The only synchronization primitive offered by Celerity are cluster-wide barriers that can be inserted between command groups through the aptly-named `distr_queue::slow_full_sync()` API. These barriers additionally serialize the execution on each node and synchronize between the main and executor threads of the runtime.

In order to avoid race conditions around node-local state, the application developer must currently insert a barrier in any place where an invisible node-local dependency exists between them. This "sledgehammer synchronization'; is not only error-prone, but also detrimental to application performance due to the barrier operation and subsequent stalling of any further work submission.

In the following, we want to explore how to establish ordering on node-local state while conserving as much scheduling freedom as possible through an in-graph mechanism.

3.1 SYCL Precedent: Explicit Control-Flow Dependencies

SYCL supports specifying explicit control-flow dependencies between command groups through the `depends_on()` API, as shown in Listing 2. This interface is primarily motivated by *Unified Shared Memory* (USM), an alternative memory management API introduced with SYCL 2020 operating on raw pointers, which lacks the implicit tracking of buffers and accessors.

Since Celerity relies on precise knowledge of buffer access patterns through access modes and range mappers to create efficient schedules, the USM model relying on hardware support for implicit data transfers cannot be implemented with reasonable efficiency. Invisible dependencies thus only arise between host tasks, so by designing an interface more closely geared towards side effects, we can retain Celerity's strong focus on declarative data flow.

```
int main() {
    sycl::queue q;
    auto event = q.submit([](sycl::handler& cgh) {
        cgh.host_task(...);
    });
    q.submit([&](sycl::handler& cgh) {
        cgh.depends_on(event);
        cgh.host_task(...);
    });
}
```

Listing 2: SYCL allows introducing explicit command-group dependencies

3.2 Dataflow-Centric: Host Objects and Side Effects

As a novel data-flow centric API, we introduce the concept of *host objects* and *side effects* as shown in Listing 3. Similar to how buffers and accessors manage distributed data, they provide an expressive and safe interface for creating data-flow dependencies between command groups.

A **host object** is a wrapper to a reference or value type with semantics that are entirely user-defined, but for which access is guarded by the runtime. Any host object is guaranteed to outlive its last observing host task, so no dangling reference problems arise from deferred kernel execution.

A **side effect**, when defined in a command group, grants the host task access to a host object and communicates the resulting local ordering constraints to the runtime. The host object–side effect duality is deliberately similar to the one between buffers and accessors, both in SYCL and Celerity.

The example in Listing 4 shows how a file handle is wrapped in a host object to capture it in a host task. Thereafter, accessing the handle itself is only possible by constructing a side effect. This statically guarantees that the object state can only be observed inside host tasks and resulting ordering constraints are always known to the runtime.

To guard against the accidental observation of non-managed state, we assert at compile time that a command group function does not capture by reference[1] unless it is passed with the `allow_by_ref` tag. Since buffers and host objects have shared-pointer semantics internally, by-value captures are almost always sufficient.

3.3 Accurate Scheduling Constraints Through Side Effect Orders

By default, side effects as proposed above will always serialize execution between host tasks observing the same object. Since host objects are opaque and the precise semantics of interactions within the host task cannot be further

[1] In C++, references and types transitively containing references are not considered *standard layout types*, so this property can be conservatively verified using `std::is_standard_layout_v<>`.

```
template <typename T>
class host_object {
    using object_type = T;
    host_object(T&& obj);
};

template <typename T>
class host_object<T&> {
    using object_type = T;
    host_object(std::reference_wrapper<T> obj);
};

template <>
class host_object<void> {
    using object_type = void;
    host_object();
};

enum class side_effect_order { relaxed, exclusive, sequential };

template<side_effect_order> struct /* exposition only */ order_tag {};
inline constexpr order_tag<side_effect_order::relaxed> relaxed_order;
inline constexpr order_tag<side_effect_order::exclusive> exclusive_order;
inline constexpr order_tag<side_effect_order::sequential> sequential_order;

template <typename T, side_effect_order Order = sequential>
class side_effect {
    using object_type = typename host_object<T>::object_type;
    side_effect(const host_object<T>& object, handler& cgh,
        order_tag<Order> = {} /* for class template argument deduction */);
    object_type& operator*() const; // when T is not void
    object_type* operator->() const; // when T is not void
};
```

Listing 3: Host Object and Side Effect API

```
int main() {
    distr_queue q;
    host_object<std::ofstream> ofs("file.txt");
    q.submit([=](handler& cgh) {
        side_effect e{ofs, cgh, /* sequential by default */};
        cgh.host_task(on_master_node, [=] { *e << "Hello "; });
    });
    q.submit([=](handler& cgh) {
        side_effect e{ofs, cgh, sequential_order /* deduction tag */};
        cgh.host_task(on_master_node, [=] { *e << "world!"; });
    });
}
```

Listing 4: Using side effects to serialize writes to a shared file handle

inspected by the runtime, this can be overly restrictive. For example, incrementing an atomic counter from multiple host tasks does not need to introduce any scheduling or synchronization constraints, but the user should still be able to rely on the runtime for the liveness guarantees on the host object.

Choosing between different scheduling guarantees for side effects is reminiscent of access modes on buffer access. However, the read–write dichotomy itself is not a good fit for this new use case: First of all, whether two "writing" side effects can be scheduled concurrently or not depends on the level of synchronization employed by the object itself, which is outside of Celerity's control. Also, for buffers, the access modes are instructive of implicit data movement by the runtime, which does not apply to host objects either.

We therefore propose three distinct *side effect orders* that can optionally be specified when a side effect is declared:

– *sequential order*: The task cannot be re-ordered against or executed concurrently with any other task affecting the same host object.
– *exclusive order*: The task may be re-ordered, but not executed concurrently with any other task affecting the same host object.
– *relaxed order*: The task may be executed concurrently with and freely re-ordered against other tasks affecting the same host object.

Relaxed-order side effects are sufficient if the contained object provides synchronization internally, or if the task only performs inherently thread-safe non-mutating accesses while any mutating operations in other tasks occur in the context of a sequential-order side effect.

An exclusive-order side effect is indicated when execution order is irrelevant, but concurrent accesses would violate synchronization requirements. This is superior to a relaxed-order side effect combined with manual locking if the lock would have to be held for any significant amount of time. Instead of stalling executor threads, each worker node is able to generate efficient local schedules around the resulting constraints ahead of time.

A sequential-order side effect must be used when re-ordering would change the semantics of the node-local state in a way that invalidates results, or concurrency on execution would violate synchronization requirements. This is the strongest guarantee and also the default behavior.

Note that between a pair of tasks affecting the same host object, the more restrictive side effect order decides the level of freedom with respect to re-ordering and concurrency. As a consequence, relaxed side effects give a stronger guarantee than an unmanaged reference-capture of the raw object would, since they are guaranteed to not be re-ordered against sequential effects.

To implement re-ordering constraints, we augment the task and command graph structures to track undirected *conflict edges* between tasks in addition to the existing directed dependency edges. Conflict edges indicate mutual exclusion between tasks, a strictly weaker requirement than the serializing dependencies impose. Task and command graphs thus become mixed graphs as seen in Figure 2. Algorithm 1 shows how dependencies and conflicts are derived from side effects.

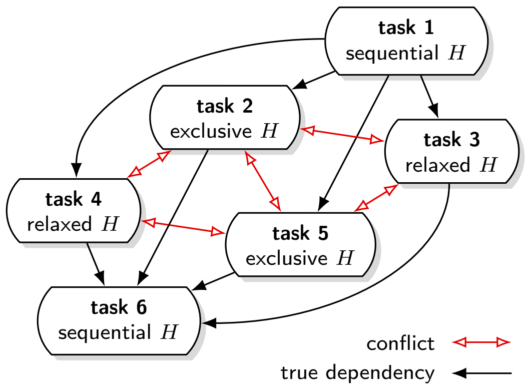


Fig. 2: Mixed task graph originating from side effects with different orders on a single host object H. Sequential-order side effects serialize against other tasks using temporal dependencies, whereas exclusive-order side effects introduce conflict edges to otherwise concurrent tasks. No edge arises between the two relaxed tasks 3 and 4, so this pair remains concurrent. The associated command graph (not shown here) will have an equivalent structure.

```
**procedure** AddSideEffect($t, h$)
    **if** $s_h$ exists $\wedge$ $(r(t,h) \neq \text{sequential} \vee A_h = \emptyset)$ **then**
        $D \leftarrow D \cup \{(t \rightarrow s_h)\}$
    **end if**
    **if** $r(t,h) = \text{sequential}$ **then**
        $D \leftarrow D \cup \{(t \rightarrow t') \mid t' \in A_h\}$
        $A_h \leftarrow \emptyset$
        $s_h \leftarrow t$
    **else**
        $C \leftarrow C \cup \{(t \leftrightarrow t') \mid t' \in A_h$
              $: r(t,h) = \text{exclusive} \vee r(t',h) = \text{exclusive}\}$
        $A_h \leftarrow A_h \cup \{t\}$
    **end if**
**end procedure**
```

Legend

t	task
h	host object
s_h	last task with sequential side effect on h
$r(t,h)$	side effect order of task t on host object h
A_h	active conflict set of tasks on h
D	set of dependencies (directed edges)
C	set of conflicts (undirected edges)

Algorithm 1: Generating dependency and conflict edges for side effects on the task graph. This algorithm also applies to the command graph, where states (D, C, A, s) are tracked separately per worker instead.

3.4 Opportunistic Scheduling of Mixed Command Graphs

The output of the existing Celerity scheduler is a stream of commands per node consisting of kernel execution ranges, metadata, and an list of prior command identifiers that it depends on. These commands are serialized to worker nodes in a topological order of the directed dependency graph. Executors do not need to reconstruct the command graph from this stream, but can instead maintain a set of *eligible commands* which contains all those that have no remaining unmet dependencies. The executor can then perform local scheduling on the eligible set to dynamically optimize resource utilization.

With the addition of conflict edges to the command graph, we extend the local scheduler to handle mutual exclusions between commands. The theory behind efficient scheduling around conflict graphs has been studied in the context of scheduling tasks with known completion times on a fixed number of general-purpose processors [4]. For certain classes of graphs, optimal solutions can be found efficiently [7].

Because Celerity has no *a priori* knowledge of kernel execution times and aims to minimize latencies by intentionally leaving low-level allocation of resources like GPU cycles to the operating system scheduler, the scheduling target is to maximize the number of active concurrent tasks.

A correct but sub-optimal implementation would execute all eligible conflicting commands in receiving order without allowing concurrency This however misses potential concurrency between tasks, and to properly harness the increased scheduler freedom, we instead find the largest conflict-free set of eligible commands.

As a classic NP-hard graph theory problem, the Maximum Independent Set can be found in exponential time through backtracking [9], although other, more efficient algorithms exist [13][17]. Since we expect the eligible set to be rather small most of the time, we implement a simple backtracking solution that will yield sufficient performance in the common case. Independent of the algorithm, the exponential growth of run time can thwart potential efficiency gains of the scheduler, so we stop backtracking early after rejecting 100 candidate solutions to limit evaluation time to a constant on degenerate graphs.

This method is opportunistic as the full set of eligible commands may not be known at the time a scheduling decision is made. Commands should begin execution as soon as they arrive to maximize throughput, so waiting for a certain filling degree is infeasible. However, since we expect most commands to have an execution time that greatly exceeds that of command generation, executors will have a well-filled command queue—and thus the full set of eligible commands for one earlier time step—most of the time.

4 Data Extraction from Runtime-Managed Structures

Although the Celerity runtime mostly concerns itself with distributing work while keeping actively managed buffer data coherent between nodes, real-world applications must be able to convert existing in-memory data into Celerity data structures on startup and extract buffer contents and host object state once execution has completed.

The former is already available in Celerity today: like in SYCL, buffers can be initialized from a pointer to host memory on construction, assuming that all nodes pass identical initialization data. In the same fashion, host objects can be constructed from arbitrary values.

There is however no native way for the application thread to observe buffer data or host object state in the application thread after construction. Instead, host tasks must be used to export data through the file system.

SYCL solves this issue using the `host_accessor` guard type that instructs necessary data transfers and locks buffer contents for synchronous access by the host application until the end of its scope. Constructing such a guard interrupts the asynchronous flow of command group submissions like an explicit barrier. This will negatively impact performance by stalling the SYCL sched-

```
int main() {
    bool host_ok;
    {
        distr_queue q;
        // ...
        buffer<bool> ok{1};
        q.submit([=](handler& cgh) { is_diag(cgh, C, ok); });

        q.submit(allow_by_ref, [=, &host_ok](handler& cgh) {
            accessor passed_acc{ok, cgh, access::all{}, read_only_host_task};
            cgh.host_task(on_master_node, [=, &host_ok] {
                host_ok = passed_acc[0];
            });
        });
    } // await implicit synchronization shutdown from ~distr_queue()
    return host_ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 5: Reference-capture workaround for retrieving buffer data. Necessary data transfers are requested through a host task accessor and awaited in the queue destructor.

uler and the compute device by not submitting any new work as long as the `host_accessor` is live.

Such stalls induce much greater latencies in a distributed setting than it does in single-node applications (see Section 5). However, Celerity already has explicit synchronization points where this performance impact is anticipated: The non-recurring implicit shutdown on queue destruction, where each node awaits all currently pending commands, and explicit barriers issued through `distr_queue::slow_full_sync()`.

Both of these synchronization points currently serve as a workaround to manually extract managed data using a host task. Listing 5 shows how the verification result from Listing 1 can be observed from the application thread by reference-capturing a result value and relying on the implicit shutdown as a synchronization point.

While functionally correct, this method is non-obvious, requires significant boilerplate, and can easily lead to undefined behavior if the application developer does not ensure that the reference-captured object outlives the synchronization point. In the following, we present a programming model allowing the extraction of arbitrary managed data data by-value and without the aforementioned hazards using existing synchronization points.

4.1 Attaching Data Requirements to Synchronization Points with Epochs

In stable Celerity, barrier synchronization and convergence on runtime shutdown and is orchestrated using ad-hoc *control commands* which are sent to workers like regular commands, but are not part of the command graph.

While this enables a less involved implementation, it is not compatible with Celerity's graph-based mechanisms of orchestrating and tracking the necessary

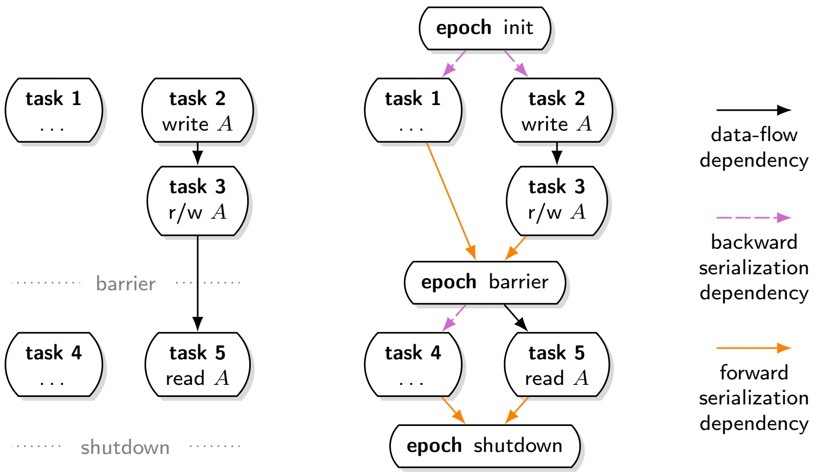


Fig. 3: Ad-hoc synchronization with broadcast commands (left, implied) and in-graph synchronization with epoch tasks (right). The barrier epoch becomes the effective producer of A, so task 5 receives a data-flow dependency on it. Serialization dependencies are inserted whenever no other transitive dependencies exist to the preceding or succeeding epoch to enforce correct temporal ordering.

data migrations ahead of any synchronization point that wants to extract buffer data. The first step is therefore to integrate these synchronization points into the task and command graphs.

To that end, we introduce the concept of *epoch* tasks and commands that fully serialize execution on each node by placing appropriate dependencies in the graphs. In this model, each task or command (except for the first epoch) has exactly one preceding epoch, and no task or command can ever depend on an ancestor of its preceding epoch.

Figure 3 illustrates the approach. We begin by inserting an epoch task in to the task graph, from which the scheduler generates exactly one epoch command per node. To ensure correct temporal ordering, each epoch graph node receives a *forward serialization* true-dependency on the entire previous execution front, and all nodes without other true-dependencies (pure producers) receive a *backward serialization* true-dependency on the preceding epoch.

On each worker node, all synchronizing API calls block the application thread until the local executor reaches the epoch command.

Since dependency information from before an epoch is irrelevant for generating future command dependencies, as an optimization, all commands preceding an epoch can be eliminated from the graph once the epoch command has been issued to executors and the epoch can be regarded as the producer of any value currently available on that node.

```
template <typename T, int Dims>
class buffer_data {
    decltype(auto) operator[](size_t idx);
};

template <typename T, int Dims>
class capture<buffer<T, Dims>> {
    using value_type = buffer_data<T, Dims>;
    explicit capture(buffer<T, Dims> buf);
};

template <typename T>
class capture<host_object<T>> {
    using value_type = T;
    explicit capture(host_object<T> ho);
};

class distr_queue {
    template <typename T> typename capture<T>::value_type
        slow_full_sync(const capture<T>& cap);
    template <typename... Ts> std::tuple<typename capture<Ts>::value_type...>
        slow_full_sync(const std::tuple<capture<Ts>...>& caps);

    template <typename T> typename capture<T>::value_type
        drain(const capture<T>& cap);
    template <typename... Ts> std::tuple<typename capture<Ts>::value_type...>
        drain(const std::tuple<capture<Ts>...>& caps);
};
```

Listing 6: Capture API around `celerity::distr_queue` (excerpt)

4.2 Extracting Buffer Data and Host Object State with the Captures API

With epoch-based synchronization in place, the runtime can attach data dependencies onto synchronization commands and thus automatically generate data migrations for reading up-to-date buffer contents on every node.

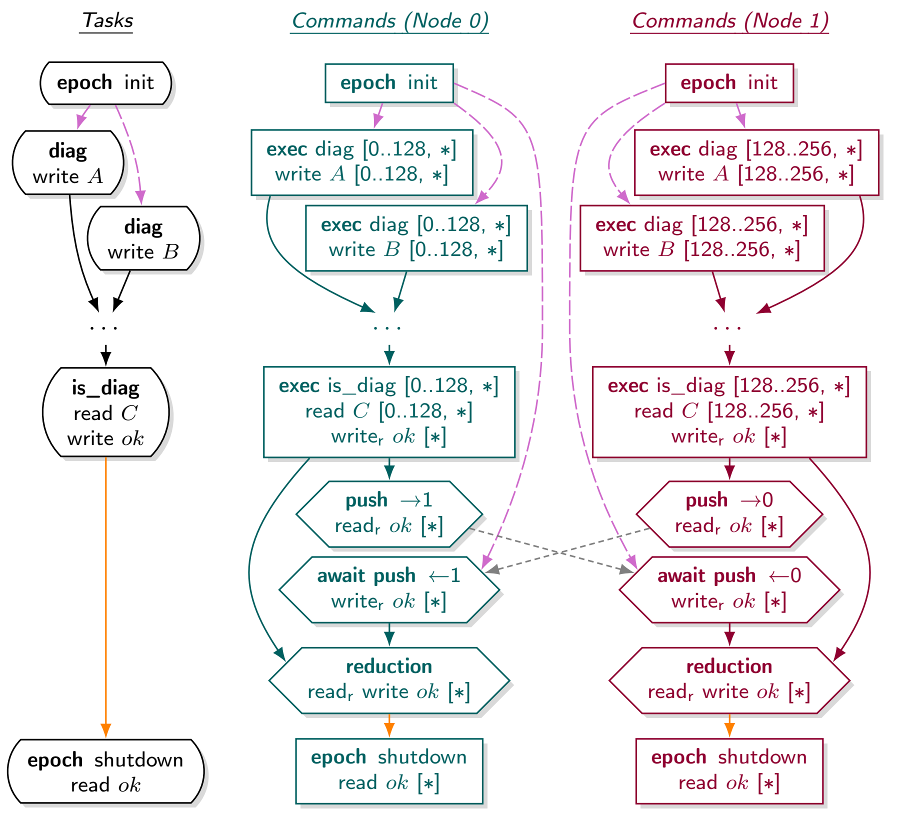
To safely inspect buffer contents and host objects without introducing unnecessary additional submission stalls, we propose *captures*, a declarative API for attaching data requirements to shutdown and barrier epochs, which will be returned to the caller as snapshots by value.

Listing 6 shows how the `distr_queue` class is extended to allow data extraction at existing synchronization points. The existing `slow_full_sync()` barrier primitive gains additional optional parameters, and shutdown convergence can be triggered explicitly using the `drain()` function. Both functions either accept a single *capture* or a tuple of captures and returns a single value or tuple of values as a result.

Each capture adds the necessary dependencies and data transfers to the generated epoch nodes and creates a snapshot of the data once the epoch has executed. As Celerity requires all MPI processes to perform the same sequence of API calls in order to allow centralized scheduling without worker-to-master communication, all nodes must currently request identical captures.

```
int main() {
    // ...
    buffer<bool> ok{1};
    q.submit([=](handler& cgh) { is_diag(cgh, C, ok); });
    return q.drain(capture{ok})[0] ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 7: Data retrieval through the high-level `capture` construct. Data transfers are generated and awaited inside the `drain()` function.



Tasks
Commands (Node 0)
Commands (Node 1)
epoch init
diag write A
diag write B
...
is_diag read C write ok
epoch shutdown read ok
epoch init
exec diag [0..128, ∗] write A [0..128, ∗]
exec diag [0..128, ∗] write B [0..128, ∗]
...
exec is_diag [0..128, ∗] read C [0..128, ∗] write_r ok [∗]
push →1 read_r ok [∗]
await push ←1 write_r ok [∗]
reduction read_r write ok [∗]
epoch shutdown read ok [∗]
epoch init
exec diag [128..256, ∗] write A [128..256, ∗]
exec diag [128..256, ∗] write B [128..256, ∗]
...
exec is_diag [128..256, ∗] read C [128..256, ∗] write_r ok [∗]
push →0 read_r ok [∗]
await push ←0 write_r ok [∗]
reduction read_r write ok [∗]
epoch shutdown read ok [∗]

Fig. 4: The updated task and command graph, first seen in Fig. 1, after the introduction of epochs and capture-based data extraction following Listing 7. The reduction operation in `verify()` places the *ok* buffer in *pending reduction state* indicated by the subscript in read$_r$ and write$_r$. A *reduction* command is generated as the result of the data requirement in the shutdown epoch and reverts the buffer back to *distributed state*.

Listing 7 shows how the verification result from Listing 1 can be inspected in the application thread on the shutdown convergence explicitly triggered by `distr_queue::drain()`.

Figure 4 shows the DAGs resulting from the capture-augmented Listing 7. With the switch to epoch-based synchronization, the task and command graphs first shown in Figure 1 will now explicitly include the data requirement on the verification result buffer *ok*.

5 Evaluation

While work focuses primarily on API expressiveness and programmability, the introduction of declarative side effects promises a performance improvement. Conversely, the introduction of epoch-based synchronization increases internal complexity, so the proposed changes demand further assessment.

We evaluated Celerity's performance on the Marconi 100 supercomputer in Bologna, Italy, which holds rank 18 of the TOP500 list as of November 2021[2]. Each node is powered by dual-socket IBM POWER9 AC922s and 256 GB of RAM, while inter-node communication is handled by dual-channel Infiniband EDR with a unidirectional bandwidth of 12.5 Gbit/s.

Although this system is GPU-accelerated and Celerity is built around accelerator computation, no device kernels are executed as part of the benchmarks. Celerity unconditionally depends on a SYCL implementation for type definitions such as `sycl::range`, but results are expected to be independent of the backend choice. For the following evaluation, we compiled against the most recent development version of hipSYCL[3] on with Clang 12.0.1 as the host compiler and IBM Spectrum MPI 10.4.0 as recommended on Marconi 100.

For all multi-process benchmarks, we allocated 4 Celerity processes per cluster node through SLURM except for the 1- and 2-process case, where all processes were mapped to a single node. Since Celerity currently requires one process per compute device, this matches the typical configuration on a system with 4 GPUs per node. Each measurement was repeated 10 times.

Figure 5 compares the latency of Celerity's `slow_full_sync` synchronization primitive against a synchronous `MPI_Barrier`. The latency of the Celerity implementation is elevated compared to the explicit MPI call as the broadcast-synchronization command or epoch command has to be sent to each worker before they can initiate their own `MPI_barriers`,. The epoch-based version is additionally delayed by graph generation overhead by a polynomial factor.

Figure 6 compares the overhead of serializing host tasks through barrier synchronization (the necessary workaround in stable Celerity) to the novel, local method using side effects. The benchmark measures a chain of 10 empty host tasks, serialized either through calls to `slow_full_sync` or side effects on a common host object. The local method, which only requires the introduction of scheduling dependencies, has much lower latency than the global barrier method, which introduces unnecessary synchronization between nodes.

Figure 7 shows the performance implications of introducing shutdown epochs on graph generation in the master node. We measured the time required to construct task and command graphs for 4 dependency topologies: *chain*, an artificial chain of command groups that require all-to-all communication between worker nodes; *soup*, an artificial, loose collection of disconnected tasks; *jacobi*, the task chain resulting from a 2D Jacobian solver; and `wave_sim`, the graph of a wave propagation stencil.

[2] `https://www.top500.org/lists/top500/list/2021/11`

[3] `https://github.com/illuhad/hipSYCL/commit/1046a787`

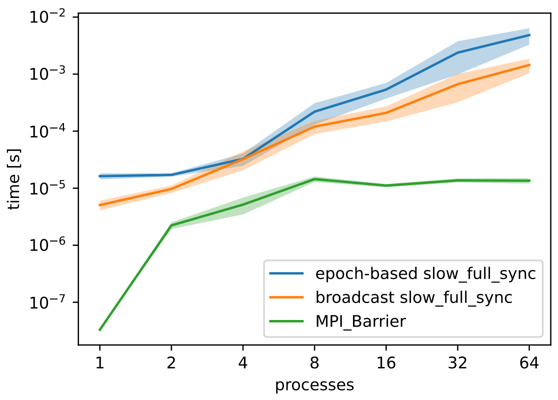


Fig. 5: Latency of barrier synchronization primitives (95% confidence intervals). `slow_full_sync` (blue and orange curves) has additional communication cost compared to the MPI baseline (green curve). Epoch-based synchronization (blue curve) further adds a constant overhead for graph generation that is amortized for higher node counts.

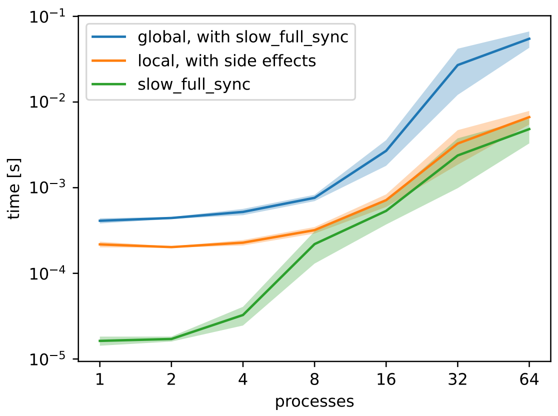


Fig. 6: Efficiency gains from replacing global barrier synchronization (blue curve) with side-effect dependencies (orange curve) to serialize a chain of 10 host tasks (95% confidence intervals). The local method does not require communication between worker nodes. Timings are measured using a single `slow_full_sync` barrier per run, which is included as a baseline (green curve).

While adding the extra work of generating a shutdown epoch will increase runtime unconditionally, this is especially pronounced for graphs with a large execution front, such as the artificial and degenerate *soup* topology. As expected, generating a forward serialization dependency from each task in the execution front and subsequently updating tracking structures has a measurable overhead. Graphs that more closely resemble real-world applications, which typically manifest as a chain of time steps, have a much smaller execution front and are therefore affected to a much smaller degree. As the number of nodes increases, scheduling is dominated by satisfying data dependencies instead. For adverse patterns such as the all-to-all communication required by the *chain* topology, this increase can be superlinear.

To summarize, the introduction of declarative side effects has a net-positive performance impact, which will help overall system performance as we expect their use to arise repeatedly during application life cycle. As data extraction from runtime-managed structures is usually only relevant on shutdown, we argue that the demonstrated increase in synchronization latency has minimal impact on overall runtime and is justified by the improved programmability.

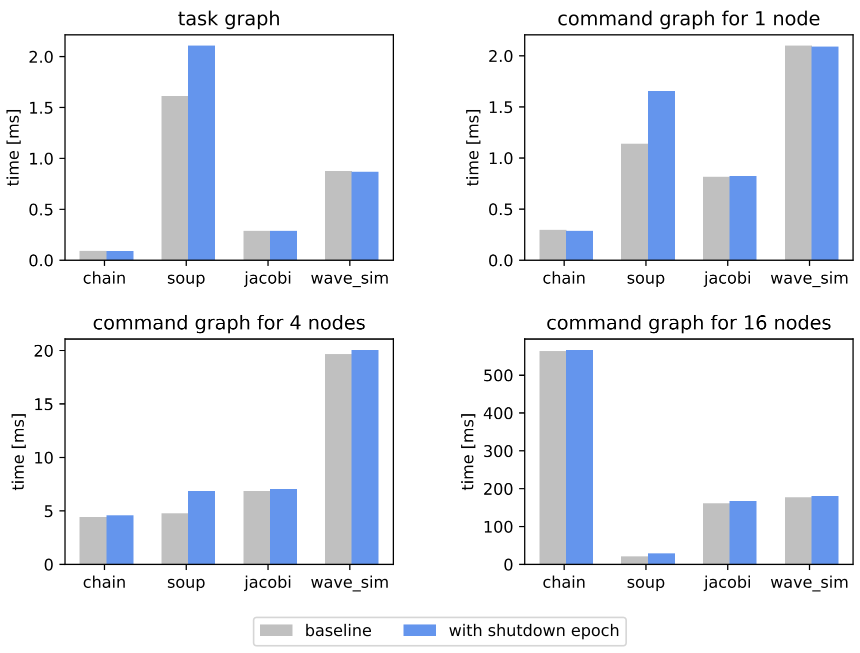


Fig. 7: Isolated time measurements for task and command graph generation on the master node (95% confidence intervals). Introducing a shutdown epoch requires forward serialization dependencies which cause measurable overhead if the execution front is large. This is pronounced for the artificial and degenerate *soup* topology of a set of disconnected tasks.

6 Related Work

SYCL[1] is an industry standard for a single-source programming model for parallel software targeting hardware accelerators. A multitude of implementations exist, with backends for GPUs [2], multi-core CPUs and application-specific FPGAs [12]. SYCL's 2020 revision introduces HPC-relevant features such as reductions and (per-work-group) parallel scans which the widely-implemented earlier 1.2.1 revision lacks. SYCL is the primary influence on the API of Celerity, which aims to ease porting from single-node SYCL programs to distributed-memory applications.

StarPU [3] is a well-established runtime system and research platform for heterogeneous computing. Its task-based execution model is supported by semi-automatic DAG generation, where dependencies can both be inferred from data accesses on buffers and also specified manually. Unlike in Celerity, task partitioning and data migration between cluster nodes also remains the user's responsibility. The latter can be handled through StarPU's MPI Support library, which adds the explicit operations of the classic MPI model to the StarPU API. Focusing explicitly on heterogeneity, the user is required user to specify kernel implementations for the targeted backends manually, whereas Celerity and SYCL will generate them from inline C++ code.

Legion [6] is a runtime system for distributed heterogeneous architectures. It dynamically extracts parallelism from programs consisting of hierarchical tasks, achieving high efficiency on irregular loads through a work-stealing scheduler. This model requires the user to specify *mappers* assigning work to target hardware, and manual decomposition of tasks. Compared to Celerity, this has the potential to improve system performance and target a wider range of applications at the expense of increased development cost and a high barrier to entry for non-experts users. Dependencies on cluster-global state such as files in a parallel file system can be embedded into Legion through the Iris [11] programming model.

Legate [5] is a high-level application of the Legion concept and a numerical toolkit acting as drop-in replacement for the popular NumPy library for array-based computation. Instead of user-defined kernels targeted by bare Legion and also Celerity, it delegates the efficient execution of NumPy array operations on arbitrary-scale hardware to the runtime system.

PaRSEC [8] is a task-based, dataflow-driven runtime system utilizing dynamic scheduling to maximize resource utilization on heterogeneous systems. It features different programming models, where the user either specifies tasks and data flow explicitly though a domain-specific language, or this information is automatically extracted from canonical sequential code by a specialized compiler frontend. This is in contrast to the Celerity approach, where data flow and kernels of coarse-granular tasks are expressed in a single-source format and transparent scheduling decisions automatically assign work to cluster nodes.

7 Conclusion

In this work, we have investigated how a graph-based distributed-memory runtime system can be extended with safe, declarative APIs to track dependencies on opaque node-local objects and transfer runtime-managed data back to the application thread to ease porting of legacy applications.

Specifically, we add the concept of *host objects* and *side effects* to the Celerity runtime system, a declarative mechanism for guarding access to and generating scheduling constraints around arbitrary node-local objects.

We further introduce the *captures* mechanism that allows observing snapshots of Celerity-managed data from to the application thread without introducing unnecessary stalls in the asynchronous execution flow. In order to model the required data movements, existing synchronization points are instead fully integrated into the task and command graphs as *epochs*, which allow the expression of captured ranges as ordinary dependencies.

Experimentally, we confirmed that declarative node-local side effects are much more efficient than the previously necessary workaround employing barrier synchronization. While the epoch-based execution model required for data extraction can incur measurable overhead for command generation, this time is quickly amortized in a highly parallel setting.

Acknowledgement

This research is supported by the European High-Performance Computing Joint Undertaking (JU) project LIGATE under grant agreement No 956137.

References

1. SYCL™ 2020 Specification (revision 4). `https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html` (2021)
2. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In: International Workshop on OpenCL, pp. 1–1 (2020)
3. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-aware task scheduling on multi-accelerator based platforms. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems, pp. 291–298. IEEE (2010)
4. Baker, B.S., Coffman Jr, E.G.: Mutual exclusion scheduling. Theoretical Computer Science **162**(2), 225–243 (1996)
5. Bauer, M., Garland, M.: Legate numpy: Accelerated and distributed array computing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3295500.3356175. URL `https://doi.org/10.1145/3295500.3356175`
6. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE (2012)
7. Bodlaender, H.L., Jansen, K.: On the complexity of scheduling incompatible jobs with unit-times. In: International Symposium on Mathematical Foundations of Computer Science, pp. 291–300. Springer (1993)
8. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. Computing in Science and Engineering **15**, 36–45 (2013)
9. Golomb, S.W., Baumert, L.D.: Backtrack programming. Journal of the ACM (JACM) **12**(4), 516–524 (1965)
10. Gschwandtner, P., Kissmann, R., Huber, D., Salzmann, P., Knorr, F., Thoman, P., Fahringer, T.: Porting Real-World Applications to GPU Clusters: A Celerity and Cronos Case Study. In: 2021 IEEE 17th International Conference on eScience (eScience), pp. 90–98. IEEE (2021)
11. Jia, Z., Treichler, S., Shipman, G., Bauer, M., Watkins, N., Maltzahn, C., McCormick, P., Aiken, A.: Integrating external resources with a task-based programming model. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), pp. 307–316 (2017). DOI 10.1109/HiPC.2017.00043
12. Keryell, R., Yu, L.Y.: Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended abstract of technical presentation. In: Proceedings of the International Workshop on OpenCL, pp. 1–8 (2018)
13. Robson, J.M.: Algorithms for maximum independent sets. Journal of Algorithms **7**(3), 425–440 (1986)
14. Thoman, P., Jordan, H., Gschwandtner, P., Fahringer, T., Cosenza, B., Juurlink, B.: CELERITY: Towards an Effective Programming Interface for GPU Clusters. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 18–28 (2018)
15. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-Level C++ for Accelerator Clusters. In: European Conference on Parallel Processing, pp. 291–303. Springer (2019)
16. Thoman, P., Tischler, F., Salzmann, P., Fahringer, T.: The Celerity High-level API: C++20 for Accelerator Clusters. International Journal of Parallel Programming (accepted, to appear in 2022)
17. Xiao, M., Nagamochi, H.: Exact algorithms for maximum independent set. Information and Computation **255**, 126–146 (2017)

Noname manuscript No.
(will be inserted by the editor)

Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments

Nina Herrmann · Herbert Kuchen

Received: date / Accepted: date

Abstract Programmers without expertise in parallel programming are often limited in their ability to execute their programs on clusters efficiently. Especially for intense calculation programs such as in natural sciences, this can be a limiting factor, as calculations are either programmed by the researcher with poor parallelization or additional packages are used, which only speed up subsets of the calculations, e. g. packages for Matrix Algebra might miss reduce steps. Moreover, those packages are often provided for languages which are easier to program but slower in their execution time, mainly Python. Lastly, the frameworks and libraries often leave programmers with choices that significantly impact the performance of the program, e. g. the number of threads. Therefore, it is essential to procure a framework where the user might have the additional effort to learn another programming language but, in contrast, can be guaranteed to exploit the hardware available on all levels, namely multiple nodes, CPUs, GPUs, and other accelerators. For using multiple nodes, each having multiple cores and accelerators, skills in combining frameworks such as MPI, OpenMP, and CUDA are required. One way to abstract from details of parallel programming is to use algorithmic skeletons. This work evaluates the implementation of multi-node, multi-CPU and multi-GPU implementation of Map, Reduce and Zip. The main contribution of this paper is a discussion of the efficiency of using multiple parallelization levels and the consideration of which fine-tune settings should be offered to the user.

Keywords Parallel Programming · Skeleton Programming · Heterogeneous Computing Environments · High-Level Frameworks · Usability

N. Herrmann, H. Kuchen,
University of Münster, Leonardo-Campus 3, 48149 Münster, Germany
Tel.: +49 251 83-38216
E-mail: {nina.herrmann,kuchen}@uni-muenster.de

1 Introduction

The field of High Performance Computing (HPC) is growing, using multiple nodes, central processing units (CPUs) and graphics processing units (GPUs) to speed up computations. This field becomes increasingly important for natural sciences and companies to evaluate the extensive data collected. Programmers have to deal with multiple low-level frameworks to exploit those levels of hardware where expertise for the single frameworks and the combination of those frameworks is required. Examples for those frameworks are Message Passing Interface (MPI) (9), OpenMP (12), and CUDA (4).

Constructing a program with those frameworks without compilation errors is already a time-consuming task, as e. g. out of memory errors, and invalid memory accesses are troublesome to identify. Even if a functioning program was constructed, the lack of knowledge leads to poor design decisions such as not using different memory spaces, the distribution of workload to computational units, or the number of threads. Inferential, most programmers have no other option than relying on high-level concepts or requiring massive computation time. Most high-level concepts have multiple benefits as portable code for different hardware architectures and requiring less maintenance for the end-user as the framework is updated.

Cole introduced algorithmic skeletons as one of the major high-level approaches to abstract from low-level details (3). Algorithmic skeletons enclose reoccurring parallel and distributed computing patterns, such as Reduce. This concept is wide-spread and beside others implemented as libraries (2; 6; 7), Domain-specific languages (DSLs) (16), and general frameworks (1; 8). Those approaches rarely support all three levels of parallelization, namely multiple nodes and simultaneously executing code on CPUs and accelerators.

In this work, we will firstly discuss related high-level frameworks contributing to the parallelization on multiple hard-ware levels 2. Section 3 describes the design of our chosen high-level approach, Muenster Skeleton Library (Muesli) and section 4 describes the implementation of the added features. The runtimes of exemplary programs are presented and discussed in Section 5. Finally, possible extensions and the conclusion of the work are described in section 6 and 7.

2 Related Work

Most importantly for skeleton approaches, Ernstsson et al. propose SkePU3 in combination with StarPU to calculate on heterogeneous clusters. However, their work misses an evaluation of distributing the calculation on all possible levels. Either the program is executed on one node with multiple CPUs and GPUs (11) or the programs are executed on multiple nodes with single backends (either GPU (OpenCL) or CPU (OpenMP)) (8). SkePU support one, two, three, and four-dimensional data stuctures. Other skeleton frameworks which are in continuous development do not consider all three layers of parallelization (e.g. FastFlow (1), SkelCL (15), Musket (13)). Hybrid execution of programs on CPUs and accelerators has been topic in multiple publications such as SkePU (11), Marrow (14) and Qilin (10). SkePU and Marrow distribute the load statically between the CPU threads and the GPUs, while Qilin dynamically distributes the working packages. Findings regarding the optimal partition are often hardware and problem-dependent and rarely comparable. Noteworthy, skeletal programming is also used for commercial products such as Intel TBB for multicore CPU parallelism.

This work enriches all previous approaches by discussing the distribution of calculation on multiple nodes, CPUs and GPUs. It aims to set the starting point to automatically distribute workload between different computational units relieving the programmer from estimating suitable partition ratios. To the best of our knowledge, other approaches either distributed the workload dynamically, which produces communication overhead, or left the choice to the programmer who might not have the expertise to decide on a reasonable split.

3 Muenster Skeleton Library

Originally, skeletal parallel programming was mainly implemented in functional languages since it derives from functional programming (3). Today, the majority of skeleton frameworks are based on C/C++ (e.g. 1; 2; 5; 8), since the language is known for good performance and interoperability with low-level parallel frameworks such as CUDA, OpenMP, and MPI. Although Python has become popular in many natural science applications, as packages can be easily written and integrated, this does not apply to calculation intense applications as the language entails a major slowdown. Therefore, especially in the HPC context, C/C++ is still the first choice.

The C++ library used for this work is called Muesli (7). Muesli provides task- and data-parallel skeletons such as Fold, multiple versions of Map, Gather, and multiple versions of Zip. Those operations can be used to write programs for clusters of multiple nodes, multicore processors, and GPUs based on MPI, OpenMP, and CUDA. Other low-level frameworks such as OpenACC and OpenCL have been considered. Muesli relieves the programmer from tasks which require expertise in parallel programming, such as the number of threads started and copying data to the correct memory spaces and helps to avoid common errors in parallel programming such as concurrent access to data structures. Although the additional abstraction causes some extra steps, it does not increase the execution time significantly. The contrast to previous versions Muesli now supports not only distributed arrays (DA) and matrices (DM) but also distributed cubes (DC) as data structures. Especially in the scientific context, e.g. computational fluid dynamics cubes are essential to model 3D objects. A distinctive feature of Muesli is that for Map and Zip, there are in-place variants and variants where the index is used for calculations. Additionally, the MapStencil skeleton allows using all data structure elements in a defined pattern.

Listing 1 shows a simple program for calculating the scalar product for the distributed array `a` and `b` (in a slightly simplified syntax).

```
class Sum : public Functor2<int, int, int>{
  public: MSL_USERFUNC int operator() (int x, int y)
                              const {return x+y;}};
Sum sum;
auto product = [] (int i, int j) {return i*j;};
DA<int> a(3,2);                     // delivers: {2,2,2}
DA<int> b = a.mapIndex(sum);        // delivers: {2,3,4}
a.zipInPlace(b,product);            // delivers: {4,6,8}
int scalarproduct = a.fold(sum);    // delivers: 18
```

Listing 1: Scalar product in Muesli.

4 Data distribution and data structures in heterogeneous computing environments

Previous work in Muesli discussing Stencil computations already provided the foundation for distributing matrices between computational nodes. This approach is not also used for Map, Zip, Fold and variants of those. This section introduces the data distribution mechanism and the metrics which are used to determine the workload allocated to the computational nodes.

4.1 Distributed cubes

The added data structure type cube is similarly designed to previous data structures in muesli which makes the syntax easy for programmers. For constructing a distributed cube, at least three arguments have to be passed to define the cube's dimensions. Optionally, a default value can be passed to be filled in all elements of the cube. A user function which should be applied to the cube can be either a C++ function or a C++ functor. They use the concept of currying, where arguments can be supplied one by one rather than all simultaneously. When, e.g. a MapIndex skeleton is used, the library completes the index parameters. Listing 2 creates two distributed cubes `a` and `b`, one without default value one with 2 as default value. The other one uses a `mapIndexInPlace` Skeleton, which adds for each element the row-index, column-index, and the index of the third dimension. Afterwards, the two cubes' values are added together for each element.

```
class Sum : public Functor2<int, int, int>{
  public: MSL_USERFUNC int operator() (int x, int y)
              const {return x+y;}};
class Sum4 : public Functor4<int, int, int, int, int>{
  public: MSL_USERFUNC int operator() (int i, int j, int x, int y)
              const {return i+j+x+y;}};
Sum sum;
Sum4 sum4;
DC<int> a(3,3,3);
DC<int> b(3,3,3,2);
a.mapIndexInPlace(sum4);
a.zipInPlace(b,sum);
```

Listing 2: Exemplary cube computation in Muesli.

4.2 Segmentation of data structures

A simplified version of the approach chosen for the mapStencil Skeleton can be seen in Figure 1. Each node is responsible for multiple rows of the data structure, and within each node, the data structure is again split between the available CPUs and GPUs in a row-wise manner.

In the context of stencil calculations, it was reasonable to distribute complete rows or rectangles of data to minimize the required data transfers for communicating border values. Therefore, always complete rows were distributed. Skeletons where the calculation does not depend on other data structure values, such as Map and Zip, are easier to distribute as incomplete rows do not decelerate execution time. Data transfers are rarely needed; therefore, it is assumed that distributing complete rows is less important than equally splitting the workload. Figure 2 and 3 demonstrate how incomplete rows are distributed and how the concept is transferred to a cube. This presentation also portions the amount of work (elements calculated) unequally for the two

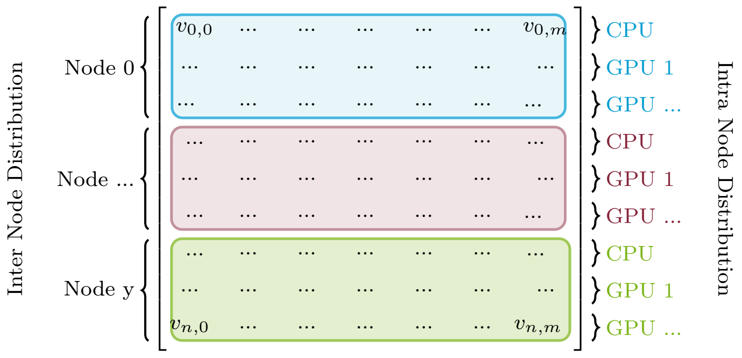


Fig. 1: Data Distribution

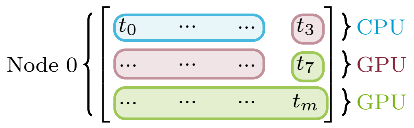


Fig. 2: Intra-node distribution using multiple accelerators.

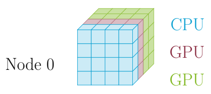


Fig. 3: Intra-node distribution of a Cube

GPUs. Prospectively, Muesli automatically calculates suitable workload splits to relieve the programmer from the fine-tune parameter of splitting work.

4.3 Work Load Partitioning

The current implementation uses the number of cores of the used GPU to allocate more elements to GPUs, which can start more threads concurrently. This is extremely important to relieve the user from low-level details and exploit the available hardware. More precisely, CUDA provides `DeviceProperties` which, amongst others, states the number of multiprocessors available. To calculate the number of cores the function `_ConvertSMVer2Cores(props.major, props.minor) * props.multiProcessorCount;` has to be used as the number of multiprocessors is dependent on the version of the GPU. However, a good approximation of the maximum possible parallelism can be calculated with this reference number. In the future, this number might also be dependent on the version of the GPU to prefer newer GPUs. Besides splitting the workload between multiple GPUs the fraction which is calculated by the CPU has to be automatically chosen by the library. The experimental results section evaluates which partition is reasonable for different skeletons, determining good default values for different calculation patterns.

5 Experimental Results

We have tested varying distribution possibilities with the distributed cubes for the skeletons Map, MapInPlace, MapIndex, MapIndexInPlace, Fold, Zip, ZipIndex, ZipInPlace and ZipIndexInPlace. The distributions include multi-node multi-GPU set-ups and different fractions of calculations which are outsourced to the CPU. The runtimes of all experiments are the result of calling skeletons multiple times. For the experiments, the HPC machine Palma II[1] and a local computer were used. With Palma we used the GeForce RTX 2080 Ti GPUs partition equipped with 2 nodes with each 4 GPUs and an Zen3 (EPYC 7513) CPU. Each node has 24 CPU cores. To provide generalizable results, each skeleton was tested on a stand-alone basis. For this purpose, we used multiple sizes and CPU-fractions. For the sequential version of the

[1] https://confluence.uni-muenster.de/display/HPC/GPU+Nodes

HPC the broadwell partition was used equipped with a Broadwell (E5-2683 v4) CPU. We let each skeleton run 25 times (without data transfers between them) to produce meaningful run times for the calculations. The local computer was used to have a comparison for the discussion of a suitable CPU-fraction. GPUs with fewer streaming multiprocessors can start fewer threads concurrently, making the use of the CPU more reasonable. The local computer is equipped with one Quadro K620, one GeForce GTX 750 Ti, and eight Intel(R) Core(TM) i7-4790 CPUs with 3.60GHz. The sequential version only used one of the available CPUs.

5.1 CPU Usage on a Local Computer

Allocating a fraction of the work to the CPU did not speed up the Map Stencil skeleton as Map Stencil has communication overhead for transferring the padding between different computational units after each skeleton call. Hence it is reasonable to test CPU-fractions for skeletons which require less communication. Map and Zip are suitable examples as calculations only depend on the current element. Figure 4 displays a subsection of the results of running Map and Zip locally. As can be seen with increasing data size, all skeletons are optimal at a CPU-fraction of 2%. CPU-fractions greater than 16% are not displayed as their runtime is increasing as expected. Interestingly, at a data size of 50^3, all runtimes are nearly equivalent and, from that point on, show clearly a difference. In Table 2 exemplary speedups for the mixed usage of the CPUs and the GPU are listed. Our results aim to automatically identify those changing points for the end-user to adjust the generated code to the system. In this context, it is especially noteworthy that the number of CPUs available on the laptop is relatively high. Still, the fraction allocated to the CPU is small. Hence hardware with less CPU should, by default, not use the CPU for Map and Zip.

In contrast to Map and Zip, creating a new data structure and the Fold skeleton are less calculation intense. Hence it was expected that CPU variants would be faster. Figure 5 displays both. Creating a data structure does not require a lot of time. It can be seen that for smaller sizes, the sequential and only CPU version are faster as no GPU memory needs to be allocated. However, for growing data sizes, they are similar. All CPU and GPU mixed programs show no significant difference in their runtimes. As they are in milliseconds done, this aspect of the program will not determine the runtime and is therefore not of primary importance. In contrast, the Fold skeleton requires a lot of time (around 12-17 s for parallel programs). In contrast to the previous skeletons, Fold performs best for 20% CPU-fraction. For 80^3 elements, 20% achieves a speedup of 1,2 in contrast to the only GPU version.

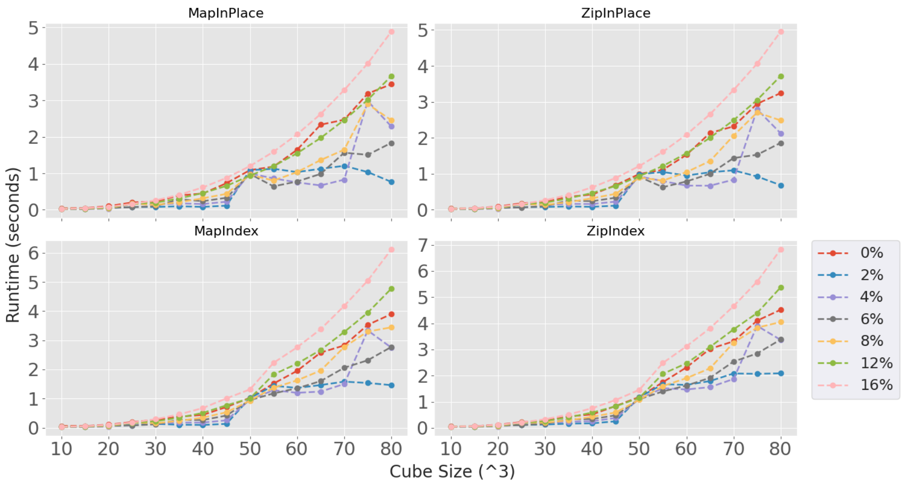


Fig. 4: Runtimes (in s) for different CPU-fractions calculated by the CPU for Map- and Zipvariants on a local computer.

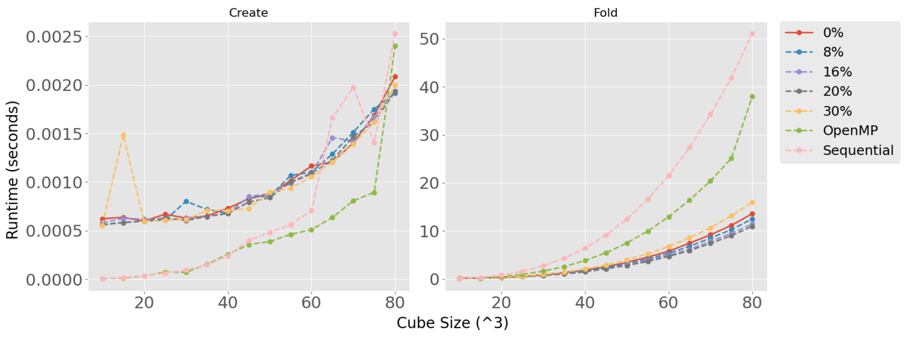


Fig. 5: Run-times (in s) for different CPU-fractions calculated by the CPU.

	Runtime					Speedup		
size^3	Seq.	OpenMP	GPU	Opt. Mix	Opt. Mix	Seq.	OpenMP	GPU
50	13,09	7,47	1,09	0,94	0,12	13,98	7,98	1,16
60	22,6	12,87	1,64	0,75	0,04	30,12	17,15	2,19
70	35,88	20,38	2,47	0,83	0,04	43,42	24,66	2,99
80	53,65	30,71	3,44	0,76	0,02	70,14	40,15	4,5

Table 1: Run-times (in s) and speedups on a single node using different CPU-fractions for MapInPlace on a local computer.

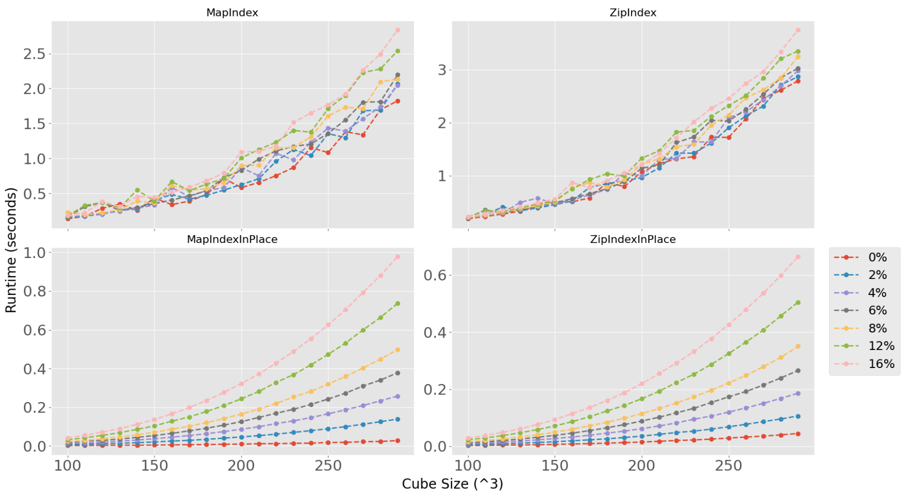


Fig. 6: Run-times (in s) for different CPU-fractions calculated by the CPU for Map- and Zipvariants on the HPC Palma.

5.2 CPU Usage on a HPC

In contrast to the local computer, Palma has a relatively strong GPUs and weak CPUs. A GeForce RTX 2080 Ti can start up to 69632 threads in parallel, while the aforementioned GPUs can start 6144 and 10240 threads in parallel. Hence, using the CPU is expected to be less beneficial. In contrast to the local computer skeletons were called up to 10.000 times as otherwise the runtime would have been to short. Figure 9 depicts the runtimes for the MapIndex, ZipIndex, MapIndexInPlace, and ZipIndexInPlace skeleton. Two major observation can be made. Firstly, for InPlace variants, no speedup is achieved when using the CPU. This underlines that with extremely powerful GPUs the CPU should not be used for calculations. Secondly, Index variants show a rather mixed behaviour, having only a slight improvement for different CPU-fractions. In contrast to the previous experiment, the size of the cube was increased to make use of all threads which can be started (max 290^3=24.389.000 elements). Index Skeletons require to create a new data structure where the results are stored. As the skeletons beside using one or two data structures use the same user-function the effect must be produced by creating and writing to a different data structure. The effect of having longer runtimes for Index skeletons could also be observed for the local variant which required double the amount of time for running Index variants instead of InPlace variants.

For the Fold skeleton the ideal CPU fraction is hard to determine. Figure 7 shows that all GPU programs perform only as good as the OpenMP program. Conclusively outsourcing calculation to the CPU produces similar runtimes. However, allocating 40-50% of the calculation to the CPU is the best fit in most cases.

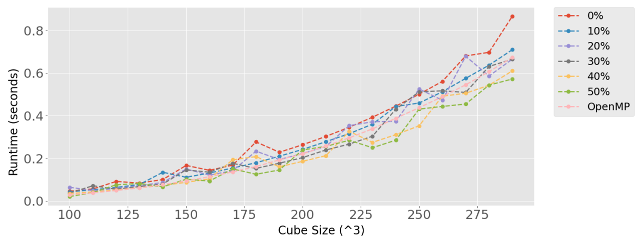


Fig. 7: Run-times (in s) for different CPU-fractions calculated by the CPU for the Fold skeleton on the HPC Palma.

5.3 Multi-Node and Multi-GPU on a local computer

As the local computer has eight CPUs and only two GPUs measuring the speedup of the program was limited in the available hardware. For this purpose we measure the performance of starting one MPI process with one GPU, one MPI process with two GPUs, and two MPI processes with each one GPUs. For the depiction, the optimal CPU fraction has been taken, which varied between 0.02% and 0.04%. Minor runtime decreases are caused by data sizes closer to a multiple of the maximum number of threads that can run in parallel. In this case, fewer threads idle. Again at the breaking point of 50^3 elements, it is beneficial to use multiple GPUs or multiple nodes. Using two MPI processes with each one GPU is better than using two GPU with one process. Again it can be seen that Index skeletons require more time caused by the additional creation of a data structure. This work showed that in specific hardware settings, using the CPU can speed up the program. This is especially relevant for Map and Zip Skeletons as they do not require communication between CPU and GPU in contrast to the Fold Skeleton and the creation of data structures. This work also showed that for strong GPUs, it is often more efficient to let the GPU do all calculations.

For the Fold skeleton, only a minor speedup could be achieved on the local computer. At 25^3 the multi-node and multi-GPU variants become faster than the single GPU variant. However, both variants are only slightly faster than the single GPU program. Creating distributed cubes is faster for one GPU. Multi-node and multi GPU programs have the disadvantage of having multiple calls to allocate memory, which creates some overhead.

			1 Node		2 Nodes		
size^3	Seq.	OpenMP	1 Node 1 GPU	2 GPUs	2 GPUs	Opt.	Speedup
45	9,24	5,5	0,11	0,12	0,1	0,1	93,94
55	16,88	10,02	0,63	0,36	0,33	0,33	51,89
65	27,96	16,55	0,67	0,67	0,34	0,34	82,75

Table 2: Run-times (in s) and speedups on for multiple nodes and GPUs for ZipInPlace on a local computer.

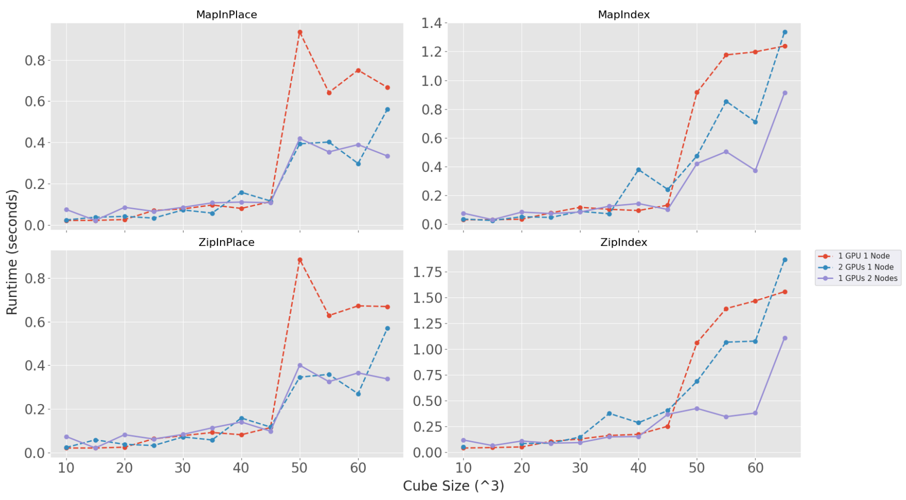


Fig. 8: Run-times (in s) for multiple nodes and GPUs for Map- and Zipvariants on a local computer.

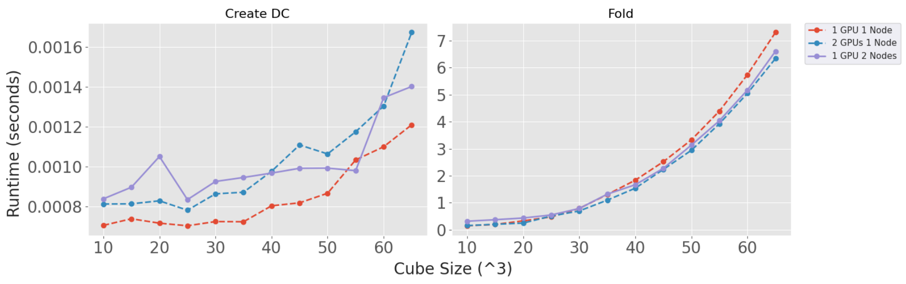


Fig. 9: Run-times (in s) for multiple nodes and GPUs for Map- and Zipvariants on a local computer.

5.4 Multi-Node and Multi-GPU on a HPC

On Palma setups with up to four GPUs per node can be tested. Although initializing additional GPUs produced overhead calculations can be distributed on more computational power. Results for different skeletons can be seen in Figure 10. For Index variants, there is nearly no visible speedup between the different GPU versions. The creation of new data structures is dependent on the CPU; hence using multiple GPUs on one node does not speed up the runtime for skeletons which require the creation of new data structures. In contrast for InPlace skeletons the four GPU variant shows a significant advantage to the one GPU variant. However, we can not reason why the two GPU variant is not faster than the program using one GPU. Although calling a skeleton produces some overhead, this should not outweigh the calculation time. Therefore, more investigation in this area is required.

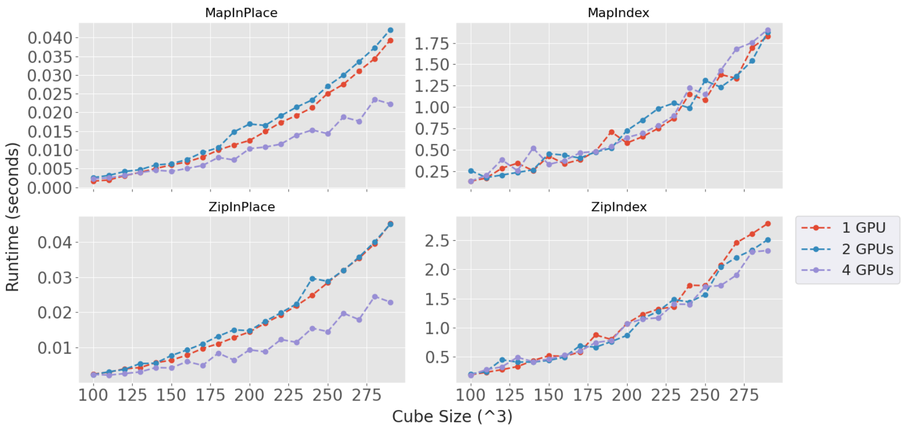


Fig. 10: Run-times (in s) for multiple GPUs for Map- and Zipvariants on the HPC Palma.

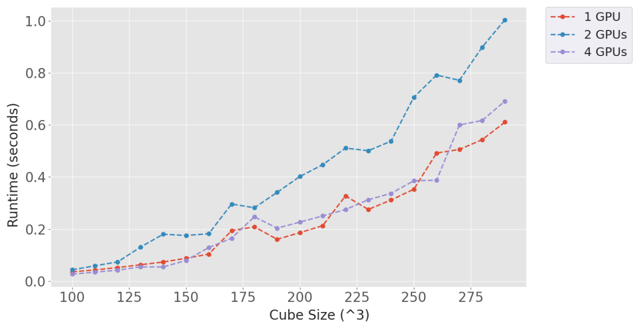


Fig. 11: Run-times (in s) for multiple GPUs for the Fold skeleton on the HPC Palma.

Interestingly, for the Fold skeleton the single GPU program and the four GPU program have approximately the same runtime. This findings can be used to be default fold only on one GPU with the best found CPU-fraction.

6 Discussion and Outlook

As could be seen, different hardware requires different distribution of data and calculations. Therefore, we aim to fine-tune Muesli to the specific hardware. Besides using macros and hardware details available at the runtime, this could include a precompiler. SkePU is using a precompiler, and they mentioned the usefulness of using static code analysis from the precompiler to autotune a program (11). However, this has not been implemented yet to the best of our knowledge. Especially interesting would be the workload of user functions as communication intense functions work well for CPUs and small calculations are suitable for GPUs. Extending the work on autotuning automatic detection of skeletons which could use InPlace variants instead of creating a new data structure, would facilitate the usage of muesli for inexperienced programmers.

7 Conclusions

This work showed that in specific hardware settings, using the CPU can speed up the program. This is especially relevant for Map and Zip Skeletons as they do not require communication between CPU and GPU in contrast to the Fold Skeleton and the creation of data structures. This work also showed that for strong GPUs, it is often more efficient to let the GPU do all calculations.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2017)
2. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: European Conference on Parallel Processing, pp. 761–770. Springer (2005)
3. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman London (1989)
4. Corporation, N.: Cuda. `https://developer.nvidia.com/cuda-zone` (2021). Accessed: 10.05.2021
5. Emoto, K., Fischer, S., Hu, Z.: Generate, test, and aggregate. In: H. Seidl (ed.) Programming Languages and Systems, pp. 254–273. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
6. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-gpu systems and clusters. International Journal of High Performance Computing and Networking **7**(2), 129–138 (2012)
7. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. International Journal of Parallel Programming **45**(2), 283–299 (2017)
8. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. International Journal of Parallel Programming **49**(6), 846–866 (2021)
9. Forum, M.: Mpi standard. `https://www.mpi-forum.org/docs/` (2021). Accessed: 10.05.2021
10. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55. IEEE (2009)
11. Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid cpu–gpu execution support in the skeleton programming framework skepu. The Journal of Supercomputing **76**(7), 5038–5056 (2020)

12. OpenMP: Openmp the openmp api specification for parallel programming. `https://www.openmp.org/` (2021). Accessed: 10.05.2021
13. Rieger, C., Wrede, F., Kuchen, H.: Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1534–1543 (2019)
14. Soldado, F., Alexandre, F., Paulino, H.: Towards the transparent execution of compound opencl computations in multi-cpu/multi-gpu environments. In: European Conference on Parallel Processing, pp. 177–188. Springer (2014)
15. Steuwer, M., Gorlatch, S.: Skelcl: a high-level extension of opencl for multi-gpu systems. The Journal of Supercomputing **69**(1), 25–33 (2014)
16. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. The Journal of Supercomputing **76**(7), 5098–5116 (2020)

Noname manuscript No.
(will be inserted by the editor)

Distributed-memory FastFlow Building Blocks

Nicolò Tonci · Massimo Torquati ·
Gabriele Mencagli · Marco Danelutto

Received: date / Accepted: date

Abstract We present the new distributed-memory run-time system (RTS) of the C++-based open-source structured parallel programming library *FastFlow*. The new RTS enables the execution of *FastFlow* shared-memory applications written using its *Building Blocks* (`BBs`) on distributed systems with minimal changes to the original program. The changes required are all high-level and deal with introducing *distributed groups*, i.e., logical partitions of the BBs composing the application streaming graph. A distributed group, which in turn is implemented using *FastFlow*'s `BBs`, can be deployed and executed on a remote machine and communicate with other groups according to the original shared-memory *FastFlow* streaming graph semantics. We present how to define the distributed groups and how we faced the problem of data serialization and communication performance tuning through transparent messages' batching and their scheduling. Finally, we present a preliminary study of the overhead introduced by the distributed groups when executing a small set of *FastFlow* shared-memory benchmarks on a sixteen-node cluster.

Keywords High-Level Parallel Programming · Distributed Programming · Parallel Building Blocks · Parallel Patterns · Skeleton Programming

1 Introduction

High-end computing servers show a clear trend toward using multiple hardware accelerators to provide application programmers with thousands of computing cores. However, many challenging applications demand more resources than those offered by a single yet powerful computing node. In these cases, application developers have to deal with different nested levels and kinds of parallelism to squeeze the full potential of the platform at hand.

N. Tonci, M. Torquati, G. Mencagli, M. Danelutto
Computer Science Department, University of Pisa, Largo B. Pontecorvo, 56122, Italy
E-mail: nicolo.tonci@phd.unipi.it E-mail: {torquati,mencagli,marcod}@di.unipi.it

In this scenario, the C++-based *FastFlow* parallel programming library [1], initially targeting multi/many-core architectures, aspires to define a *single programming model* for shared- and distributed-memory systems leveraging a streaming data-flow programming approach and a reduced set of structured parallel components called `Building Blocks` (BBs). *FastFlow*'s `BBs` provide the programmer with efficient and reusable implementations of essential parallel components that can be assembled following a LEGO-style model to build and orchestrate more complex parallel structures (including well-known algorithmic skeletons and parallel patterns) [2]. With `BBs`, the structured parallel programming methodology percolates to a lower-lever of abstraction [3].

In this paper, we present the new distributed-memory run-time system (RTS) introduced in the `BBs` software layer of the *FastFlow* library aiming to target both scale-up and scale-out platforms preserving the programming model. It enables the execution of *FastFlow* applications written using `BBs` on distributed systems. Already written applications require minimal modifications to the original shared-memory program. New *FastFlow* applications can be first developed and debugged on a single node, then straightforwardly ported to multiple nodes. The motivations that have led us to work at the `BB` level of the *FastFlow* library are twofold: a) provide the programmer with a quick and easy porting methodology of already written *FastFlow* data-streaming applications to distributed systems by hiding all low-level pitfalls related to distributed communications; b) prepare a set of mechanisms (e.g., specialized RTS `BBs`, class wrappers, serialization features, message batching) that can be used as the basis for implementing high-level ready-to-use parallel and distributed exploitation patterns (e.g., Map-Reduce, D&C).

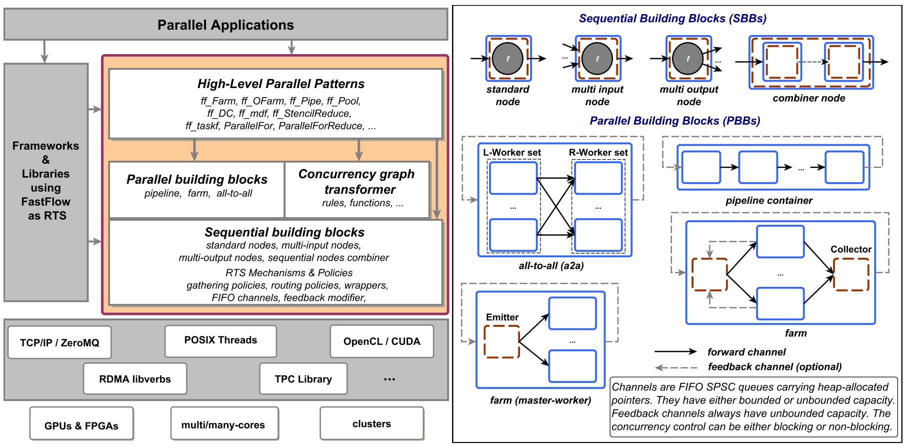
We present the idea of *FastFlow*'s *distributed groups* and its associated API as well as some experimental results that validate functional correctness and provide preliminary performance assessments of our work.

The outline of the paper is as follows. Section 2 presents an overview of the *FastFlow* library and its `BBs`. Section 3 introduces the *distributed group* concept and semantics. Section 4 presents the experimental evaluation conducted. Section 5 provides a discussion of related works and Section 6 draws the conclusions of this paper.

2 *FastFlow* Overview and Background

The *FastFlow* library is the result of a research effort started in 2010 with the aim of providing application designers with key features for parallel programming via suitable parallel programming abstractions (e.g., ordered farm, pipeline, divide&conquer, parallel-for, macro data-flow, map+reduce, etc.) and a carefully designed RTS [1]. The structured parallel programming methodology [4] was the fertile ground that has allowed the development of the initial idea and then guided the *FastFlow* library implementation.

The latest *FastFlow* version (v. 3.x) has been released in 2019 where the lower-level software layers have been redesigned, and the concept of *Building-*


Parallel Applications
Frameworks & Libraries using FastFlow as RTS
High-Level Parallel Patterns
ff_Farm, ff_OFarm, ff_Pipe, ff_Pool, ff_DC, ff_mdf, ff_StencilReduce, ff_taskf, ParallelFor, ParallelForReduce, ...
Parallel building blocks
pipeline, farm, all-to-all
Concurrency graph transformer
rules, functions, ...
Sequential building blocks
standard nodes, multi-input nodes, multi-output nodes, sequential nodes combiner
RTS Mechanisms & Policies
gathering policies, routing policies, wrappers, FIFO channels, feedback modifier,
TCP/IP / ZeroMQ
POSIX Threads
OpenCL / CUDA
RDMA libverbs
TPC Library
...
GPUs & FPGAs
multi/many-cores
clusters
Sequential Building Blocks (SBBs)
standard node
multi input node
multi output node
combiner node
Parallel Building Blocks (PBBs)
L-Worker set
R-Worker set
all-to-all (a2a)
pipeline container
Collector
farm
Emitter
farm (master-worker)
forward channel
feedback channel (optional)

Channels are FIFO SPSC queues carrying heap-allocated pointers. They have either bounded or unbounded capacity. Feedback channels always have unbounded capacity. The concurrency control can be either blocking or non-blocking.

Fig. 1 `Left)`:*FastFlow* software stack. `Right)`:The set of *FastFlow*'s Building Blocks.

Block introduced to support the development of new patterns and domain-specific libraries [2]. In addition to the *farm* and *pipeline* core components present in the previous releases, two new `BBs` have been added, namely *all-to-all* and *node combiner*. Furthermore, a new software layer called *Concurrency graph transformer* is now part of the *FastFlow* software stack. Such a layer is in charge of providing functions for the concurrency graph refactoring to introduce optimizations (e.g., fusing parallel `BBs`) and enhancing the performance portability of the applications. *FastFlow*'s software layers are sketched in the left-hand side of Fig. 1.

Building Blocks

Building Blocks (`BBs`) [2,3] are recurrent data-flow compositions of concurrent activities working in a streaming fashion, which are used as the basic abstraction layer for building *FastFlow* parallel patterns and, more generally *FastFlow* streaming topologies. The Data-flow streaming model and the `BBs` are the two fundamental ingredients of the *FastFlow* library. Following the same principles of the structured parallel programming methodology, a parallel application (or one of its components) is conceived by selecting and adequately assembling a small set of well-defined `BBs` modeling both data and control flows. Differently from "pure" algorithmic skeleton-based approaches, where highly specialized, reusable, and efficient monolithic implementations of each skeleton are defined for a given architecture, the `BB`-based approach provides the programmer with efficient and reusable implementations of lower-level basic parallel components that can be assembled following a LEGO-style methodology to build and orchestrate more complex parallel structures [2]. They can be combined and nested in different ways forming either acyclic or cyclic concurrency graphs, where graph nodes are *FastFlow* concurrent entities and edges are communication channels. A communication channel is implemented as a lock-free

Single-Producer Single-Consumer (SPSC) FIFO queue carrying pointers to heap-allocated data [5]. Collective communications involving multiple producers and/or consumers are realized through broker nodes employing multiple SPSC queues. More specifically, we consider *Sequantial Building Blocks* (`SBBs`) and *Parallel Building Blocks* (`PBBs`). `SBBs` are: the *sequenatial node* (in three versions) and the *nodes combiner*. `PBBs` are: the *pipeline*, the *farm* (in two versions), and the *all-to-all*. The right-hand side of Fig. 1 shows the graphical notation of all `BBs`. A description of each BB follows.

$\boxed{\textit{\textbf{node}}}$. It defines the unit of sequential execution in the *FastFlow* library. A node encapsulates either user's code (i.e. business logic) or RTS code. Based on the number of input/output channels it is possible to distinguish three different kinds of sequential nodes: *standard node* with one input and one output channel, *multi-input node* with many inputs and one output channel, and *multi-output node* with one input and many outputs. A node performs a loop that: i) gets a data item (through a memory reference) from one of its input queues; ii) executes a functional code working on the input data item and possibly on a state maintained by the node itself by calling its service method (`svc()`); iii) puts a memory reference to the resulting item(s) into one or multiple output queues selected according to a predefined (i.e., on-demand, round-robin) or user-defined policy (e.g., by-key, random, broadcast, etc.).

$\boxed{\textbf{node combiner}}$. It enables to combine two `SBBs` into one single sequential node. Conceptually, the combining operation is similar to the composition of two functions. In this case, the functions are the service functions of the two nodes (i.e., the *svc()* methods). This `SBB` promotes code reuse through fusion of already implemented nodes and it is also used to automatically reduce the number of threads implementing the concurrent graph when possible.

$\boxed{\textbf{pipeline}}$. The pipeline is the topology builder. It connects `BBs` in a linear chain (or in a toroidal way if the last stage is connected to the first one with a *feedback channel*). Also, it is used as a *container* of `BBs` for grouping them in a single parallel component. At execution time, the *pipeline* models the data-flow execution of its `BBs` on data elements flowing in streaming.

$\boxed{\textbf{farm}}$. It models functional replication of `BBs` coordinated by a sequential master `BB` called *Emitter*. The default skeleton is composed of two computing entities executed in parallel (this version is called *master-worker*): a multi-output Emitter, and a pool `BBs` called *Workers*. The *Emitter* node schedules data elements received in input to the *Workers* using either a default policy (i.e., round-robin or on-demand) or according to the algorithm implemented in the business code defined in its service method. Optional *feedback channels* connect *Workers* back at the *Emitter*. A second version of the *farm*, comprises also a multi-input *BB* called *Collector* in charge of gathering results coming from *Workers* (the results can be gathered either respecting *farm* input ordering or without any specific ordering). Also in this version, optional *feedback channels* may connect both *Workers* as well as *Collector* back to the *Emitter*.

$\boxed{\textbf{all-to-all}}$. The all-to-all (briefly a2a) defines two distinct sets of *Workers* connected according to the *shuffle communication pattern*. Therefore, each

Worker in the first set (called L-Worker set) is connected to all Workers in the second set (called R-Worker set). The user may implement any custom distribution policy in the L-Worker set (e.g., sending each data item to a specific Worker of the R-Worker set, broadcasting data elements, executing a by-key routing, etc). The default distribution policy is round-robin. Optional *feedback channels* may connect R-Worker with L-Worker sets, thus implementing an *all-to-all communication pattern.*

`BBs` can be composed and nested, like LEGO bricks, to build concurrent streaming networks of nodes executed according to the data-flow model. The rules for connecting `BBs` and generating valid topologies are as follows:

1. Two `SBBs` can be connected into a pipeline container regardless of their number of input/output channels.
2. A `PBBs` can be connected to `SBBs` (and vice versa) into a pipeline container by using *multi-input* (*multi-output*) sequential nodes;
3. Two `PBBs` can be connected into a pipeline container either if they have the same number of nodes, or through multi-input multi-output sequential nodes if they have different number of nodes at the edges.

In several cases, to help the developer when possible, the RTS automatically enforces the above rules transforming the edge nodes of two connecting `BBs` by using proper node wrappers or adding helper nodes via the *combiner* `BB`. For example, in the *farm* `BB`, the sequential node implementing the *Emitter* is automatically transformed in a multi-output node. Additionally, if an *all-to-all* `BB` is connected to a *farm*, then the *Emitter* is automatically transformed in a *combiner node* where the left-hand side node and the right-hand side node are multi-input and multi-output, respectively. The *Concurrency graph transformer* software layer (see Fig. 1) provides a set of functions to aid the expert programmer statically change (parts of) the *FastFlow* data-flow graphs by refactoring and fusing `BBs` to optimize the shape of the concurrency graph.

All high-level parallel patterns provided by the *FastFlow* upper layer (e.g., *ParallelFor*, *ParalleForReduce*, *Ordered Farm*, *Macro Data-Flow*, etc.) were implemented with the sequential and parallel *BBs* presented [2].

`BBs` usage example

A simple usage example of a subset of *FastFlow* `BBs` is presented in Fig. 2. In the top-left part of the figure, we defined three sequential nodes: `Reader`, `Worker`, and `Writer`. The `Reader` node takes in input a comma-separated list of directory names and produces in output a stream of *file_t* data elements each associated to a file contained in one of the input directories (the `ff_send_out` call at line 7 is used to produce multiple outputs for a single activation of the service method `svc()`); in the end, the `Reader` produces the special value *End-Of-Stream* (`EOS` at line 8) to start the pipeline termination of the next `BBs`. The `Worker` node executes a given `search` function on each input file, and then it produces in output only non-empty matching (the special value `GO_ON` at line 15 is not inserted into the output channel and is meant to keep the node

```
#include <ff/ff.h>
using namespace ff;
struct Reader:ff_node_t<file_t> {
  Reader(const char dir[]):dir(dir){}
  file_t *svc(file_t*) {
    for(auto file: dir_iterator(dir))
      ff_send_out(new file_t(file));
    return EOS; /* end-of-stream */
  }
  const char[] dir;
};
struct Worker:ff_node_t<file_t,
     result_t> {
  result_t *svc(file_t *in){
    auto r=search(in,"Hello");
    return (r ? r : GO_ON);
  }
};
struct Writer:ff_minode_t<result_t> {
  result_t *svc(result_t*in){
    R.push_back(in);
    return GO_ON; /* keep-going */
  }
  void svc_end() {print_result(R);}
  std::vector<result_t*> R;
};
```

```
// ----- (Version 1) -------------
int main() {
  ff_pipeline pipe;
  pipe.add_stage(new Reader(
        "dir1,dir2,dir3,dir4"));
  pipe.add_stage(new Worker());
  pipe.add_stage(new Writer());
  return pipe.run_and_wait_end();
}
```

```
// ----- (Version 2) -------------
int main() {
  ff_a2a a2a;
  a2a.add_firstset<Reader>(
      {new Reader("dir1,dir2"),
      new Reader("dir3,dir4")});
  a2a.add_secondset<Worker>(
      {new Worker(),
      new Worker(),
      new Worker()});
  ff_pipeline pipe;
  pipe.add_stage(&a2a);
  pipe.add_stage(new Writer());
  return pipe.run_and_wait_end();
}
```

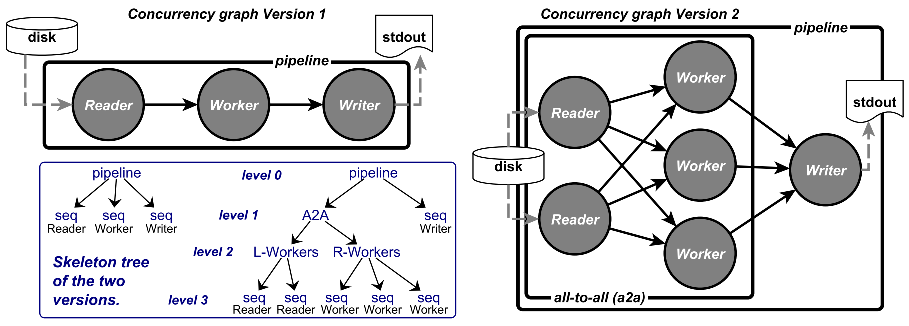


Fig. 2 Simple usage example of the *FastFlow* BBs implementing two different parallel skeletons of the same program. Top): (*Version 1*) three-stage pipeline of SBBs, and (*Version 2*) two-stage pipeline where the first stage replicates two times the Reader node and three times the Writer node by using the *all-to-all* PBB. Bottom): Graphical schema of the two versions and their skeleton trees showing the nesting levels of BBs.

alive and ready to receive the next input). Finally, the Writer node collects all results, one at a time, and then writes the final result on the standard output by using the function print_result in the svc_end() method. Such method is called once by the *FastFlow* RTS when the node receives (all) the EOS(s) from the input channel(s) and before terminating.

In the top-right part of Fig. 2 the three sequential nodes defined are instantiated and combined in two different concurrent streaming networks: the

Version 1) is a standard 3-stage pipeline; the *Version 2)* is a 3-stage pipeline in which the first stage is an `a2a BB` replicating two and three times the `Reader` and `Worker` nodes, respectively. They will be automatically transformed into multi-output and multi-input nodes.

Finally, at the bottom of the figure are sketched the schemas of the two versions and their skeleton trees showing the levels and nesting of `BBs`. The leaves of the tree are implemented as POSIX threads in the *FastFlow* RTS.

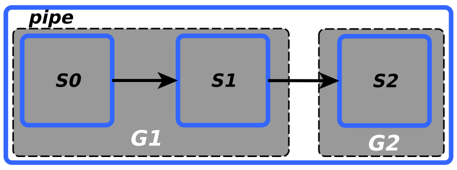
Previous FastFlow distributed RTS

The first versions of the *FastFlow* library (before v. 3.x) provided the programmer with the possibility to execute *FastFlow* programs on a distributed system [6]. Based on the ZeroMQ communication library, the distributed RTS was developed in 2012 by Massimo Torquati. Later a tiny message-passing layer atop InfiniBand RDMA was also implemented as a ZeroMQ alternative [7]. To support inter-process communications, the old *FastFlow* `node` was extended with an additional "external channel" (either in input or in output). The extended node was called `dnode`. Edge nodes of the *FastFlow* data-flow graph, once transformed into `dnodes`, could communicate with `dnodes` of others *FastFlow* applications running on different machines, through a pre-defined set of communication collectives (i.e., *unicast*, *onDemand*, *Scatter*, *Broadcast*, *fromAll*, *fromAny*). The programmer had to annotate each `dnode` with the proper collective endpoint to make them exchange messages according to the selected communication pattern.

There are many differences between the previous (old) version and the new one presented in this paper. We report here only the most relevant points wholly redesigned in the new version. In the old version, the programmer should explicitly modify the edge nodes of a *FastFlow* program to add the `dnode` wrapper with the selected communication pattern. It also had to define two non-trivial auxiliary methods for data serialization. Moreover, the old version exposed two distinct programming models to the programmer, one for the local node (i.e., streaming data-flow) and one for the distributed version (i.e., Multiple-Programs, Multiple-Data with collectives). Finally, the old version did not provide the *FastFlow*'s system programmer with any basic distributed mechanisms to define new high-level distributed patterns.

3 From shared-memory to distributed-memory *FastFlow* applications

This section presents the *FastFlow* library extensions to execute applications in a distributed-memory environment. By introducing a small number of modifications to programs already written using *FastFlow*'s `BBs`, the programmer may port its shared-memory parallel application to a hybrid implementation (shared-memory plus distributed-memory) in which parts of the concurrency graph will be executed in parallel on different machines according to the well-known SPMD model. The resulting distributed application will adhere to the



```
1 auto G1 = pipe.createGroup("G1");
2 auto G2 = pipe.createGroup("G2");
3 G1 << S0 << S1;
4 G2 << S2;
```

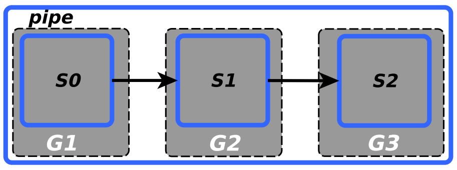
Fig. 3 Pipe example 1: creating two *dgroups* from a three-stage *FastFlow* pipeline.

same data-flow streaming semantics of the original shared-memory implementation. The modifications consist of identifying disjoint *groups* of `BBs`, called *distributed groups* (or simply *dgroups*), according to a small set of rules described in the following. Then mapping such distributed groups to the available machines through a JSON-format configuration file.

Each *dgroup* represents a logical partition of *FastFlow*'s `BBs` implementing a portion of the *FastFlow* streaming concurrency graph. It is implemented as a process that runs alone or together with other *dgroups* on a remote node. Furthermore, to exploit the full potential of a single node, a *dgroup* is internally realized as a shared-memory *FastFlow* application properly enriched with customized sequential `BBs` and node wrappers to transparently realize the communications among *dgroups* according to the original data-flow graph. The API to define the distributed groups comprises two functions: the *dgroup* creator, `createGroup` function and the *dgroup* inclusion implemented through the C++ operator '`<<`'. A *dgroup* can be created from any level 0 or level 1 `BB` in the *FastFlow* skeleton tree. The `createGroup` function takes as an argument a unique string that uniquely identifies the distributed group. The inclusion operator is helpful when the programmer wants to create multiple *dgroups* from a single `BB`, and only a subset of its nested `BBs` need to be included in a given *dgroup*.

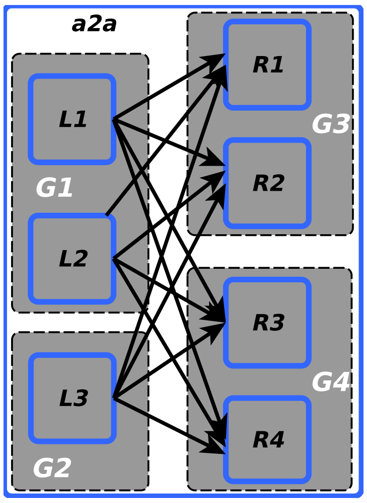
In Fig. 3, a generic application structured as a pipeline of three, possibly parallel, `BBs` is organized in two distributed groups. The first *dgroup*, *G1*, is composed of the first two stages while the second *dgroup* contains the last stage. In this example `S0`, `S1` and `S2` can be any valid nesting of the available library `BBs`. As can be seen from the code in the same figure, the two groups are created from the same pipeline (`pipe`) at line 1 and 2, and then the direct children `BBs` (i.e., those at level 1 of the skeleton tree) are included in the correct *dgroup* (lines 3 and 4). A *dgroup* created from a pipeline must have all included `BBs` contiguous to respect the pipeline order. For instance, it is not possible to place `S0` and `S2` in *dgroup* "G0", without including also `S1`.

The `BBs` that can be added to a distributed group by using the '`<<`' operator are the direct children of the `BB` from which the group has been created. This constraint has two reasons: 1) to keep the implementation simple and manageable; 2) to not reduce too much the granularity of the single groups in order to have a coarse enough concurrency graph to be efficiently executed in a single multi-core node and thus capable of exploiting the available local parallelism. However, such constraint might be relaxed in future releases of the *FastFlow* library.



```
S0.createGroup("G1");
S1.createGroup("G2");
S2.createGroup("G3");
```

Fig. 4 Pipe example 2: creating *dgroups* from each stage of a pipeline.



```
auto G1 = a2a.createGroup("G1");
auto G2 = a2a.createGroup("G2");
auto G3 = a2a.createGroup("G3");
auto G4 = a2a.createGroup("G4");
G1 << L1 << L2;
G2 << L3;
G3 << R1 << R2;
G4 << R3 << R4;
```

```
{ "groups" : [{
    "name" : "G1",
    "endpoint": "node1:8080"
 },{"name" : "G2",
    "endpoint": "node2:8080"
 },{"name" : "G3",
    "endpoint": "node3:8080"
 },{"name" : "G4",
    "endpoint": "node4:8080"
 }]}
```

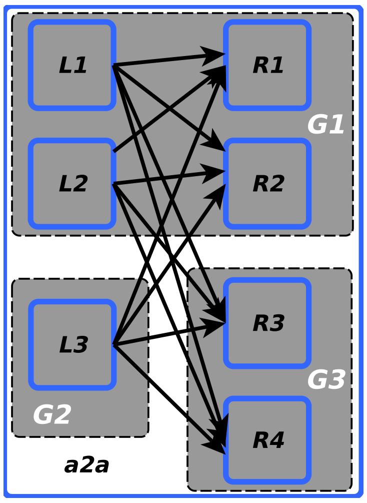
Fig. 5 A2A example 1: creating four *dgroups* from a *FastFlow* `a2a` by aggregating some L-Worker and some R-Worker `BBs` without intersections between the two sets. In the bottom-right part, the JSON configuration file specifies the *dgroups*' mapping to nodes.

From a sequential `node` and a `farm` `BBs` it is possible to create only one *dgroup* [1], whereas from the `pipeline` and `all-to-all` `BBs` it is possible to create multiple *dgroups*. Finally, a `BB` can be included only in a single *dgroup*.

Another possibility to create n distributed groups from an n-stage pipeline is to create them directly from the stages, as shown in Fig. 4. Indeed, when we create a *dgroup* from a `BB` at level 1 of the skeleton tree, we might be implicitly expressing the willingness to put the whole `BB` inside the *dgroup*. If this is the case, then there is no need to include all the nested `BBs` manually as the RTS automatically includes them all.

Distributed groups from a single `a2a` `BB` can be derived in different ways, either by cutting the `a2a` graph horizontally or vertically or in both directions (i.e., oblique cuts). Vertical and non-inter-sets horizontal cuts produce only distributed communications between L-Worker and R-Worker `BBs`. Differently, in horizontal inter-sets cuts, some *dgroups* contain both L-Worker and R-Worker `BBs` of the `a2a`. Therefore, some communications will happen in the

[1] For the `farm`, this is a limitation of the current release that will be relaxed in the future releases of the *FastFlow* library.



```cpp
auto G1 = a2a.createGroup("G1");
auto G2 = a2a.createGroup("G2");
auto G3 = a2a.createGroup("G3");
G1 << L1 << L2 << R1 << R2;
G2 << L3;
G3 << R3 << R4;
```

```json
{ "groups" : [{
    "name" : "G1",
    "endpoint": "node1:8080"
 },{
    "name" : "G2",
    "endpoint": "node2:8080"
 },{
    "name" : "G3",
    "endpoint": "node3:8080"
 }]}
```

Fig. 6 A2A example 2: creating three *dgroups* from a *FastFlow* `a2a` performing one inter-sets horizontal cut, and one vertical cut. In the bottom-right part, the JSON configuration file specifies the *dgroups*' mapping to hosts.

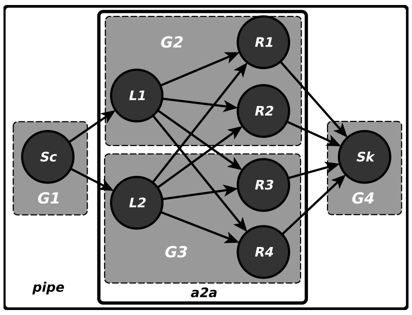
shared-memory domain, while others will happen in the distributed-memory domain (i.e., inter-node communications). When applicable, for example, for the on-demand and round-robin data distribution policies, the distributed RTS will privilege local communications vs. distributed communications.

An example of four groups produced as a result of a vertical and two non-inter-sets horizontal cuts for an `a2a` `BB` is sketched in Fig. 5. In this example, some L-Worker `BBs` are grouped together in a single group (as a result of non-inter-set cuts). The same for some R-Worker `BBs`.

In Fig. 6, the same `a2a` `BB` presented in Fig. 5 is split into three *dgroups* by making two cuts, one horizontal producing the group *G1* that aggregates `BBs` coming from the two Worker sets (as a result of the horizontal inter-sets cut), and one vertical cut producing two distinct *dgroups*, *G2* and *G3*. With such a division, communications between *L1*, *L2* and *R1*, *R2* `BBs` are local through SPSC shared-memory channels. In contrast, all the other pairs of communications among L-Worker and R-Worker sets are inter-nodes. The different kinds of communications are handled transparently by the *FastFlow* RTS. Notably, for remote communications, data-types must be serialized. In Fig. 5 and Fig. 6 we also show the JSON configuration files containing the mapping $< group - host : port >$ (for the default TCP/IP message transport protocol). It also contains addition attributes that can be specified either for the entire application or just for a single group. We will discuss some of these attributes in the following.

So far, we have introduced the basic grouping rules through simple generic examples showing the small extra coding needed to introduce distributed groups in a *FastFlow* program. In the next example, we give a complete overview of a still straightforward but complete distributed *FastFlow* `BB`-based

```
 1 #include <ff/dff.hpp>
 2 #include <nodesDef.hpp>
 3 using namespace ff;
 4 int main(int argc, char *argv[]){
 5   // distributed RTS init ------
 6   DFF_Init(argc,argv);
 7
 8   // ---- FastFlow graph -------
 9   SourceNode Sc;
10   LeftNode   L1, L2;
11   RightNode  R1,R2,R3,R4;
12   SinkNode   Sk;
13   ff_pipeline pipe;
14   ff_a2a a2a;
15   a2a.add_firstset<LeftNode>(
16             {&L1, &L2});
17   a2a.add_secondset<RightNode>(
18          {&R1,&R2,&R3,&R4});
19   pipe.add_stage(&Sc);
20   pipe.add_stage(&a2a);
21   pipe.add_stage(&Sk);
22
23   // --- distributed groups ----
24   auto G1 = Sc.createGroup("G1");
25   auto G2 = a2a.createGroup("G2");
26   G2 << &L1 << &R1 << &R2;
27   auto G3 = a2a.createGroup("G3");
28   G3 << &L2 << &R3 << &R4;
29   auto G4 = Sk.createGroup("G4");
30   // -- distributed execution --
31   return pipe.run_and_wait_end();
32 }
```



```
{"protocol" : "MPI",
 "groups" : [{
   "batchSize": 10,
   "name" : "G1",
   "messageOTF": 64,
   "endpoint":"node1"
  },{
   "name" : "G2",
   "batchSize": 20,
   "threadMapping" : "0,2,4,6,8",
   "endpoint":"node2"
  }, {
   "batchSize": 20,
   "name" : "G3",
   "endpoint":"node3"
  },{
   "name" : "G4",
   "endpoint":"node4"
}]}
```

Fig. 7 A complete application example composed of a 3-stage pipeline: `multi-output node`, `a2a`, `multi-input node`. The distributed version comprises four `dgroups`, two coming from the `a2a` `BB`. The definition of Sk,Sc,L_i, and R_i nodes, is not shown.

application. This application, sketched in Fig 7, is made of a 3-stage pipeline in which the first and last stages are sequential `BBs`, and the middle one is a 2×4 `a2a` `BB`. The distributed version comprises four groups: "G1" containing the source node, "G2" containing the top half of the `a2a`, "G3" containing the bottom half part of the `a2a`, and "G4" containing the sink node. In this example, all the `dgroups` are created directly from the level 1 `BB`, whereas the inclusion operator is used to assign the `a2a`'s Worker `BBs` to the desired group. The definition of the `dgroups` is listed from line 24 to line 29 in the code listing on the left-hand side of Fig 7. The changes compared to the shared-memory version are as follows: 1) at line 1 the include file `dff.hpp` enabling a set of distributed RTS features; 2) the `DFF_Init` function at line 6 needed to identify the *dgroup* name to execute and to collect all needed information provided by the launcher (e.g., location of the JSON configuration file); 3) the previously discussed annotations needed to create *dgroups* (from line 24 to 29). The configuration file, on the right-hand side of Fig. 7, is richer

```
1: function CREATEGROUP($bb$, $groupName$)
2:     if $\exists G \mid name(G) = groupName$ then error()
3:     if $type(bb) \in \{seq, comb, farm\} \wedge \exists G \mid parent(G) = bb$ then error()
4:     if $level(bb) > 1$ then error()
5:     if $level(bb) = 0 \wedge \exists G \mid level(parent(G)) = 1$ then error()
6:     $Groups \leftarrow Groups \cup \{G_{groupName}\}$
7: function ADDTOGROUP($bb$, $g$)
8:     if $bb \notin parent(g)$ then error()
9:     if $\exists g' \mid bb \in g'$ then error()
10:    $g \leftarrow g \cup \{bb\}$
11: function RUNGROUP($g$)
12:    if $type(parent(g)) = pipe$ then $checkPipe(g, prent(g))$
13:    $parseJSON()$
14:    ffg $\leftarrow buildFFnetwork(g)$
15:    $run$(ffg)
```

name(g)	returns the unique name of the group g
parent(g)	returns the `BB` from which the group g has been created
type(bb)	returns the type $\in \{seq, comb, farm, pipe, a2a\}$ of the `BB` bb
checkPipe(g, p)	checks whether the subset of stages of the pipe p in g are contiguous
level(bb)	returns the nesting level of the `BB` bb in the skeleton tree
parseJSON()	parses the configuration file
buildFFnetwork(g)	generates the *FastFlow* concurrent graph for the group g
run(g)	executes the *FastFlow* graph implementing the group g

Fig. 8 Pseudo-code of the rules used by the *FastFlow* RTS for creating valid distributed groups (*creatGroup* and *addToGroup*). The *runGroup* reports the pseudo-code to build the distributed group and run it as *FastFlow* application.

than the one in previous examples. In line 1 we specify to use MPI instead of the default TPC/IP transport protocol. In line 3 we specify that the *dgroup* "G1" will send out messages in batches of 10 per destination, and in lines 9 and 13, in the same way, we specify that the *dgroups* "G2" and "G3" will send results to "G4" in batches of 20 messages. The batching of messages is *completely transparent* to the application and it is particularly helpful for small-size messages to optimize the network link bandwidth. In addition, the JSON configuration file may contain other non-functional attributes to regulate some low-level knobs of the *FastFlow* RTS, for example the thread-to-core affinity mapping for each *dgroup* (`threadMapping`), or to set the size of the logical output queues representing the distributed channels through the attributes `messageOTF/internalMessageOTF`, which set the maximum number of "on-the-fly" messages for a channel. All these attributes, have default values.

For the sake of completeness, the pseudo-code of Fig. 8 gives a complete overview of the semantics checks of the two functions provided by the distributed-memory *FastFlow* RTS, namely *creatGroup* and *addToGroup*, the latter mapped onto the inclusion operator '`<<`'. We also included the *runGroup* function, which the programmer does not directly call since it is automatically invoked by RTS passing the proper group name. It builds the *FastFlow* graph implementing the given *dgroup* and runs it.

```
struct data_t {
  std::string key;
  uint64_t id, ts;

  template<class Archive>
  void serialize(Archive & ar){
      ar(key,id,ts);
  }
};
```

```
struct data_t {
  char key[MAXWORD];
  uint64_t id, ts;
};

template<class Pair> // <ptr,size>
void serialize(Pair &p, data_t *d){
  p={(char*)d, sizeof(rdata_t)};
}
template<class Pair> // <ptr,size>
void deserialize(const Pair &p,data_t *&d){
  d = new (p.ptr) data_t;
}
```

Fig. 9 Data serialization methods in the *FastFlow* library. `Left)`: *Cereal*-based serialization function of the `data_t` type. `Right)`: custom serialization of a memory-contiguous version of the `data_t` type that enables zero-copy transfers.

Data serialization

Data serialization/deserialization (briefly data serialization from now on) is a fundamental feature of any distributed RTS. It is the process of transforming a possibly non-contiguous memory data structure into a format suitable to be transmitted over a network and later reconstructed, possibly in a completely different computing environment, preserving the original data. In the distributed *FastFlow* RTS, data serialization can be carried out in two different ways. The programmer may select the best approach, between the two, for each data type flowing into the inter-group channels (i.e., the data types produced/received by the edge nodes of a *dgroup*).

The first approach employs the *Cereal* serialization library [8]. It can automatically serialize base C++ types as well as compositions of C++ standard-library types; for instance, a `std::pair` containing a `std::string` and a `std::array<int>` objects can be serialized without writing any extra line of code. *Cereal* requests a serialization function only for user-defined data types. A user-defined data type containing an explicit (yet straightforward) serialization function is sketched on the left-hand side of Fig. 9.

The second approach allows the user to fully specify its serialization and deserialization function pair. This might be useful, when feasible, to avoid any extra copies needed by the serialization process itself. This method is beneficial when the data types are contiguous in memory (i.e., trivial types in C++), thus a zero-copy sending protocol can be employed. An example of this custom approach is shown on the right-hand side of Fig. 9.

Distributed group implementation and program launching

A *dgroup* is implemented through the *FastFlow*'s `farm` `BB`. The Emitter is the *Receiver*, and the Collector is the *Sender*. The `farm`'s Workers are the `BBs` of the original application graph included in that particular *dgroup* (either implicitly or explicitly). The `BBs` that communicate with the Sender and/or

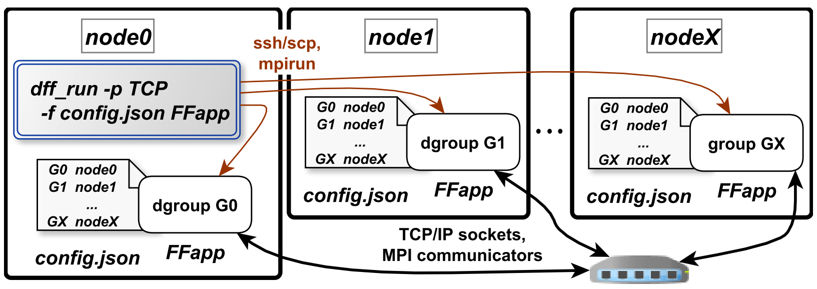


Fig. 10 Logical schema of the launching of a *FastFlow* distributed application with `dff_run` using the TCP/IP protocol.

Receiver via shared-memory *FastFlow* channels, are automatically wrapped by the RTS with class wrappers that transparently perform serialization activities on the input/output data-types of the `BBs`. Such activities happen in parallel with data communications. Horizontal inter-sets cuts in an `a2a` are implemented using customized `BBs` in the L-Worker and R-Worker sets.

Concerning *FastFlow* program launching, we have designed a software module called `dff_run`. It takes care of launching the application processes, each one with the appropriate parameters (e.g., *dgroup* name), following the mapping host-group described in the JSON configuration file. For applications using the MPI library as a communication layer, the `dff_run` is just a wrapper of the well-known *mpi_run* launcher. It produces a suitable `hostfile` which will be passed to the *mpi_run* command. A simplified overview of the launching phase when using the TCP/IP protocol as a communication layer is sketched in Fig. 10. For an application composed of X groups where one group is executed locally on `node0`, the `dff_run` launcher creates X-*1* `ssh` connections towards each remote node and launch the executable with the correct parameters. If the user wishes to receive the standard output of each process; in that case, the `dff_run` gathers each output stream from each ssh connection and combines them into a single annotated stream where the annotation is the group name from which the output comes. The current version of the `dff_run` launcher does not deploy the *FastFlow* executable and the configuration file on the remote hosts. This limitation will be addressed in the next releases.

4 Experimental Evaluation

Experiments was conducted on the *openhpc2* cluster hosted by the Green Data Center of the University of Pisa. It is composed by 16 nodes interconnected at 1Gbit/s. Each node has two Intel Silver Xeon 4114 CPUs running at 3.0GHz for a total of 20 physical cores (and 40 hardware threads) and 128GB of RAM.

The first test evaluates the throughput attainable using different message sizes as well as the impact of varying the `batchSize` attribute in the JSON configuration file without modifying the program. This test considers two nodes

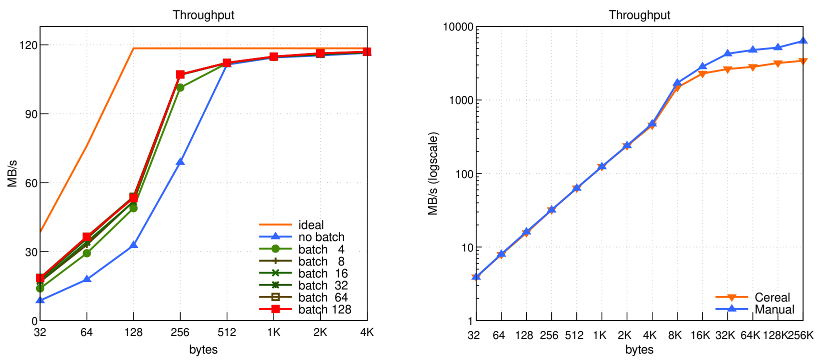
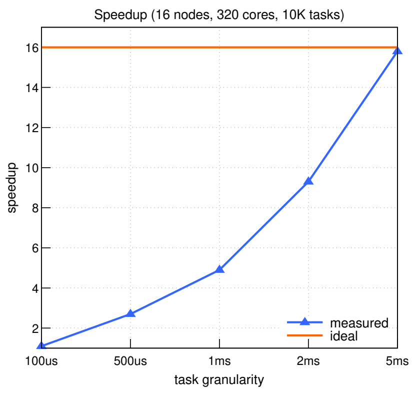


Fig. 11 Point-to-point throughput as a function of the message size between twp *dgroups*. `Left)`: Measured throughput between two cluster nodes. `Right)`: Measured throughput on a single cluster node by using Cereal-based vs. Manual serialization.

of the cluster. The results are shown in Fig. 11 (left-hand side). The ideal throughput has been measured using the `netcat` network utility program. As expected, the transparent batching feature, is particularly useful for small messages and becomes less relevant for messages bigger than 512B. A batch size of 8-32 messages is enough to reach the maximum throughput attainable.

The second test evaluates the cost of the automatic serialization using the Cereal library. We compared Cereal-based serialization (the default) to manual serialization of a memory-contiguous data type that allows the RTS to perform a zero-copy message transfer. In this case, to avoid potential bottlenecks introduced by the network, both sender and receiver *dgroups* are executed on the same node on different CPUs through the *threadMapping* configuration file attribute. The right-hand side of Fig. 11 shows the results. The two serialization approaches behave the same for messages smaller than 8KB, while above that threshold, manual serialization has less overhead, as expected. However, significant variances in performance are located above 1.2GB/s. Therefore, we could expect almost no differences for applications running on clusters whose interconnection is up to 10 Gbit/s.

The third test mimics a Master-Worker parallel pattern implementation (i.e., a *farm* skeleton without the Collector) using *FastFlow* `BBs`. The starting point is a *FastFlow* shared-memory micro-benchmark using a `a2a` `BB`, in which a single *multi-output* sequential `BB` in the L-Worker set implements the Master, and a set of sequential `multi-input BBs` in the R-Worker set implement the Workers. The Master generates 100K tasks at full speed. Each task is a message whose payload is 128B. For each input task, the Workers execute a controlled synthetic CPU-bound computation corresponding to a predefined time (we considered values in the range $0.1-5$ milliseconds). The task scheduling policy between L- and R-Worker sets is *on-demand*. The distributed version is derived from the shared-memory benchmark by cutting the *a2a BB* graph both horizontally and vertically. The horizontal inter-set *dgroup* aggregates the Master and 20 Workers of the R-Worker set. The vertical *dgroups* aggregate



Task Grain (ms)	Ideal time (S) 1 node 20 Workers	Exec Time (S) 16 nodes 320 Workers	Speedup
0.1	0.5	0.464	1.08
0.5	2.5	0.915	2.73
1	5	1.02	4.90
2	10	1.075	9.30
5	25	1.583	15.79

Task	N. of tasks assigned to Workers		
Grain	in Master	in other dgroups	
(ms)	dgroup	Average	StdDev
0.1	89011	732.1	10.1
0.5	36048	4263.5	12.9
1	20339	5309.2	8.2
2	10680	5952.9	3.5
5	6239	6250.1	15.1

Fig. 12 Master-Worker experiment. `Left)`: Speedup varying the task's computation granularity. `Right-Top)`: Execution time (in seconds) and speedup for each task grain (in milliseconds). `Right-Bottom)`: tasks computed by the Master *dgroup* and by all others *dgroups*.

the remaining Workers of the R-Worker set (20 Workers for each *dgroup* to fill in all physical cores of a node). Distinct *dgroups* are deployed to different cluster nodes. The logical schema is that of Fig. 6 with $1L_i$ and $20R_i$ *BBs* for each *dgroup*. The tables on the right-hand side of Fig. 12 summarize the results obtained using all physical cores of the *openhpc2* cluster (i.e., 320 cores in total). All tests have been executed using a transparent batching of 32 messages and 1 as the maximum number of messages on-the-fly. The baseline is the ideal time on a single node considering the task granularity (e.g., for tasks of 100us, the ideal execution time is 500ms). We also measured the number of tasks received by the local Workers in the Master *dgroup* and by all remote Workers in the other *dgroups* (see the right-bottom table in Fig. 12). The speedup increases with computational granularity, and the number of tasks computed by local Workers in the Master group is inversely proportional to the task granularity. This is what we can expect from the *on-demand* task scheduling policy that privileges local *dgroup* Workers: the coarser the grain of tasks, the higher the number of tasks sent to remote Workers. As a final note for this test, since the completion time is quite short, an initial barrier has been artificially introduced in the *DFF_Init* to synchronize all *dgroups* and obtain a more accurate measurement. The reported times are thus the maximum time observed among all *dgroups* without considering the spawning time introduced by the `dff_run` launcher (which accounts for about 200ms).

The last experiment is *WordCount*, a well-known I/O-bound streaming benchmark. Its logical data-flow schema is sketched on the top left-hand side of Fig. 13. There is a source stage (*Sc*) that reads text lines from a file or a socket; a line tokenizer or splitter (*Sp*) that extracts words from the input line and sends all words with the same hash value (called key) to the same destination; a counter (C) that counts the number of instances of each word received; and a sink (*Sk*) that collects words and prints all statistics (e.g., unique words, current number of words, etc.). Our test considers Twitter's

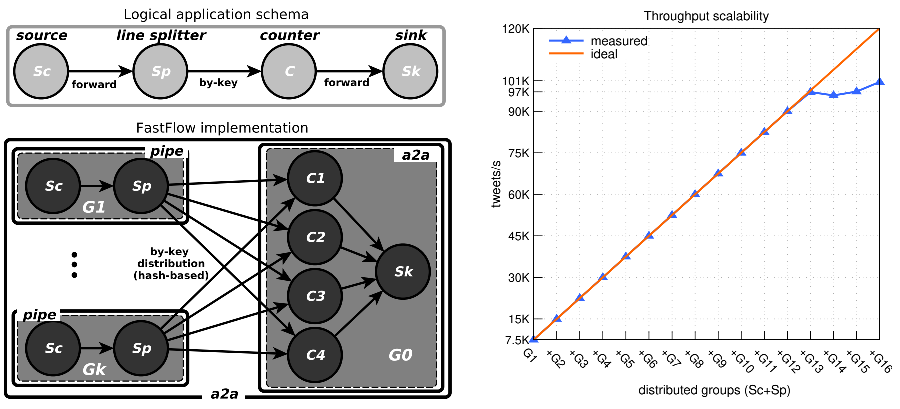


Fig. 13 WordCount experiment. `Left)`: Logical schema and the *FastFlow* implementation. `Right)`: Throughput scalability (tweets/s) increasing the number of source-splitter *dgroups*.

tweets as text lines (max 280 characters including spaces), multiple replicas of the pair source-splitter stages (to emulate tweet streams coming from various sources), four replicas of the counter, and one sink stage. The *FastFlow* data-flow graph implementing the test is shown on the bottom left-hand side of Fig. 13. The grey rectangles identify the *dgroups*: each source-splitter replica is part of a *dgroup* (*G1...Gk* in the figure), whereas the counter replicas and the sink stage own to a single *dgroup* (*G0* in the figure). By running a single replica of the source-splitter *dgroup* and the counter-sink *dgroup* on the same cluster node on different CPU cores with `batchSize=32`, we found a maximum attainable throughput of about 120K tweets/s. Consequently, we configured each source stage to constantly produce $120/16 = 7.5$K tweets/s towards the splitter stage. Then, to stress-test the shuffle communication pattern with the by-key distribution, we replicated the source-splitter *dgroup* multiple times (up to 16 replicas). The results obtained in these tests are plotted on the left-hand side of Fig.13. The scalability is linear up to 13 replicas for an aggregated throughput of about 97K tweets/s, then it flattens until we reach 16 *droup* replicas where the last replica is executed on the same node of the counter-sink *dgroup* (on different CPU cores) thus reaching a maximum throughput of about 101K tweets/s (about 84% of the maximum).

Discussion

The experiments conducted have shown: a) the *FastFlow* shared-memory streaming model is transparently preserved when porting to distributed-memory domains, applications that use nontrivial communication patterns (e.g., hash-based message scheduling in WordCount), and with both horizontal as well as vertical cuts of the concurrent graph; b) the designed distributed RTS can achieve close to nominal bandwidth on 1Gbit/s networks using the TCP/IP protocol; c) the transparent batching feature, which can be enabled from the

JSON configuration file, is helpful to optimize communications for small messages without modifying the application code; d) the RTS can efficiently balance tasks workload among multiple distributed groups yet privileging local communications (i.e., towards group local Workers) to minimize communication overheads; e) streaming computations with computational tasks of a few milliseconds can benefit from the distributed RTS to reduce the execution time.
NOTE TO REVIEWERS: In the final paper version we will include an assessment of the MPI transport protocol. At the time of writing, the transparent batching feature is not complete for MPI.

5 Related Work

High-level parallel programming frameworks abstract the low-level threading, communication, and synchronization details necessary for effectively utilizing parallelism and liberate the programmer from writing error-prone concurrent procedures. Such programming environments also increase the portability of applications by taking care of non-functional parameters tuning (e.g., parallelism degree, task's granularity) as a function of the target platform. The de facto standard for shared-memory parallel programming is the *OpenMP* programming model, whereas, in the HPC context, the most broadly used model is "MPI + X", where X is usually either OpenMP or CUDA [9] [10]. However, several higher-level parallel programming libraries or domain-specific languages (DSLs) have been proposed in the context of structured parallel programming [4]. Several of them are implemented in C/C++ (SkePU [11], SkeTo [12], SkelCL [13], GrPPI [14], Muesli [15]), some others are DSL-based such as Musket [16] and SPar [17]. They provide the user with a restricted set of pre-defined and optimized parallel components targeting heterogeneous multi/many-cores and, in some cases, distributed-memory systems (e.g., GrPPI, Muesli, Musket). Recently some of the concepts coming from the algorithmic skeletons and parallel design patterns research communities have also fertilized some commercial/industrial programming environments such as Intel TBB [18] for multi-core parallelism, Khronos SYCL [19] for heterogeneous multi/many-core equipped with GPUs and Apache Spark[20] and Apache Flink [21] for cluster-level data-stream processing. In recent years there has been a proliferation of frameworks aiming to ease the communication in distributed systems [22]. For example, in the HPC context, Mercury [23] leverages multiple HPC fabrics protocols to implement efficient remote procedure calls. Instead, in big data analytics and cloud environments, there are ActiveMQ and ZeroMQ [2] among the most used Message Queueing systems.

FastFlow [1, 2] has been developed in the context of structured parallel programming methodology, and it mainly targets streaming applications. What mainly characterizes *FastFlow* compared to other notable approaches in the

[2] ActiveMQ: `https://activemq.apache.org/`. ZeroMQ:`https://zeromq.org/`.

field is its ambition to offer different yet structured software layers to the system as well as application programmers. At the bottom level of the abstraction, a reduced set of flexible, efficient, and composable `BBs` are provided for building new domain-specific frameworks such as WindFlow [24], and highly-distributed streaming networks. *BBs* mainly target parallel-expert programmers. At the higher level of the abstraction, *FastFlow* provides some well-known parallel exploitation patterns (e.g., ParallelFor and D&C) mainly targeting application developers. Currently, all patters can be used only inside a single *dgroup*. With its new distributed RTS, *FastFlow* also aspires to offer a *single programming model* for both shared- and distributed-memory systems.

6 Conclusions and Future Work

We extended the *FastFlow*'s `BBs` layer with a new RTS enabling the execution of `BBs`-based *FastFlow* applications on distributed platforms. Changes to the code-base required to port the applications to the hybrid shared/distributed-memory environments are minimal and straightforward to introduce. First experiments conducted on a 16-node cluster demonstrate that: i) the new distributed RTS preserves the *FastFlow* programming model and does not introduce unexpected overheads; ii) the transparent batching of messages is a useful feature for tuning the distributed application throughput. Future extensions will consider: a) adding support for the `farm BB` and bearing cyclic *FastFlow* networks; b) introducing heuristics for automatically defining *dgroups* to relieve the programmer from this decision; c) augmenting the number of transport protocols provided to the user, and enabling the coexistence of multiple protocols on different zones of the *FastFlow*'s nodes graph; d) developing some high-level parallel patterns using the distributed RTS; e) expanding the functionalities of the *dff_run* launcher to improve the deployment phase.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2017). DOI 10.1002/9781119332015.ch13
2. Torquati, M.: Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns. Ph.D. thesis, University of Pisa (2019)
3. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. International Journal of Parallel Programming **42**(6), 1012–1031 (2014). DOI 10.1007/s10766-013-0273-6
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing **30**(3), 389–406 (2004). DOI 10.1016/j.parco.2003.12.002
5. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Euro-Par 2012 Parallel Processing, pp. 662–673. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-32820-6_65
6. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in fastflow. In: Proceedings of the 18th International Conference on

Parallel Processing Workshops, Euro-Par'12, p. 47–56. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-36949-0_7
7. Secco, A., Uddin, I., Pezzi, G.P., Torquati, M.: Message passing on infiniband rdma for parallel run-time supports. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 130–137 (2014). DOI 10.1109/PDP.2014.23
8. Grant, W.S., Voorhies, R.: cereal–a c++ 11 library for serialization. URL https://github. com/USCiLab/cereal (2013)
9. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro international conference on parallel, distributed and network-based processing, pp. 427–436. IEEE (2009). DOI 10.1109/PDP.2009.43
10. Smith, L., Bull, M.: Development of mixed mode mpi/openmp applications. Scientific Programming **9**(2, 3), 83–98 (2001). DOI 10.1155/2001/450503
11. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. International Journal of Parallel Programming **49**(6), 846–866 (2021). DOI 10.1007/s10766-021-00704-3
12. Tanno, H., Iwasaki, H.: Parallel skeletons for variable-length lists in sketo skeleton library. In: European Conference on Parallel Processing, pp. 666–677. Springer (2009). DOI 10.1007/978-3-642-03869-3_63
13. Steuwer, M., Kegel, P., Gorlatch, S.: Skelcl-a portable skeleton library for high-level gpu programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 1176–1182. IEEE (2011). DOI 10.1109/IPDPS.2011.269
14. López-Gómez, J., Muñoz, J.F., del Rio Astorga, D., Dolz, M.F., Garcia, J.D.: Exploring stream parallel patterns in distributed mpi environments. Parallel Computing **84**, 24–36 (2019). DOI 10.1016/j.parco.2019.03.004
15. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster Skeleton Library Muesli: A comprehensive overview. Ercis working papers, University of Münster, European Research Center for Information Systems (ERCIS) (2009)
16. Rieger, C., Wrede, F., Kuchen, H.: Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, p. 1534–1543. ACM, New York, NY, USA (2019). DOI 10.1145/3297280.3297434
17. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: Spar: A dsl for high-level and productive stream parallelism. Parallel Processing Letters **27**(01), 1740005 (2017). DOI 10.1142/S0129626417400059
18. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technology Journal **11**(4) (2007). DOI 10.1535/itj.1104.05
19. Reyes, R., Lomüller, V.: Sycl: Single-source c++ accelerator programming. In: Parallel Computing: On the Road to Exascale, pp. 673–682. IOS Press (2016). DOI 10.3233/978-1-61499-621-7-673
20. Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z.: Big data analytics on apache spark. International Journal of Data Science and Analytics **1**(3), 145–164 (2016). DOI 10.1007/s41060-016-0027-9
21. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)
22. Ramon-Cortes, C., Alvarez, P., Lordan, F., Alvarez, J., Ejarque, J., Badia, R.M.: A survey on the distributed computing stack. Computer Science Review **42**, 100422 (2021). DOI 10.1016/j.cosrev.2021.100422
23. Soumagne, J., Kimpe, D., Zounmevo, J.A., Chaarawi, M., Koziol, Q., Afsahi, A., Ross, R.B.: Mercury: Enabling remote procedure call for high-performance computing. In: CLUSTER, pp. 1–8 (2013). DOI 10.1109/CLUSTER.2013.6702617
24. Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., Danelutto, M.: Windflow: High-speed continuous stream processing with parallel building blocks. IEEE Transactions on Parallel and Distributed Systems **32**(11), 2748–2763 (2021). DOI 10.1109/TPDS.2021.3073970

Noname manuscript No.
(will be inserted by the editor)

Efficient High-Level Programming in plain Java

Rui S. Silva · João L. Sobral

Received: date / Accepted: date

Abstract This paper introduces a framework that supports the development of high-level parallel programs in Java, delegating the performance tuning to later stages of development. The framework supports a complete set of locality optimisations that are essential to build efficient multicore applications. The optimisation are introduced with Java annotations keeping the original code platform independent. Performance results show that data layout improvements can provide a 2.5 speedup, and when combined with the exploitation of parallelism can deliver a 50x speedup when compared with an unoptimised sequential base application on a 24-core machine.

Keywords High-Performance · Locality · Java · Model Driven

1 Introduction

The Java language is one of the most widely used programming languages. Java provides several features to support safe and portable applications. This is accomplished by running applications on a managed environment: applications are executed on a virtual machine using a Just-in-Time (JIT) compiler to generate native code. Moreover, at run-time, it checks all object references (type-checking, null pointer and range checking). This provides safer applications but can introduce additional overheads, if the JIT compiler is not able to remove these additional run-time checks in cases where safety can be ensured.

The Java object model is also responsible for additional run-time overheads, due to the support of those safety features and to the support flexible object collections. Java object implementations in modern JVMs (e.g., OpenJDK 17)

João L. Sobral
Centro Algoritmi, Universidade do Minho, Braga, Portugal
E-mail: jls@di.uminho.pt

Rui S. Silva
Centro Algoritmi

include an additional 8-byte header in the object data footprint to store meta-object information (e.g., object type and array size) and collections of objects (including multidimensional arrays of primitive values) are stored using an array of pointers to objects. Unfortunately, the Java language is not popular among the parallel programming community, due to these inefficient memory layouts and due to the additional run-time costs of safety features.

On the other hand, high level languages, such as Java, have the potential to better encapsulate optimisations required to build efficient parallel applications. Common optimisations include the implementation of efficient data layouts and loop tiling, among many others. Typically, implementing these optimisations leads to "premature optimisation", where domain abstractions are obfuscated by the implementation of these optimisations. Moreover, these optimisations make the code less abstract and less portable (e.g., the code includes low-level implementation details and/or platform dependent optimisations).

This paper shows how the Gaspar framework enables the development of high performance parallel applications. The framework introduces a new two-step development process: 1) write the domain code using high-level abstractions; 2) optimise the implementation by introducing optimisations for the specific target platform.

2 Common Optimisations

Today's multicore systems suffer from the lack of enough memory bandwidth to keep all processing units busy. The design of scalable parallel applications for those systems requires a careful optimisation of parallel applications in order to overcome that memory hall. Unfortunately, optimising the code has a negative impact on the code readability. Even worse, it negatively impacts the usage of domain abstractions in the code and those optimisations become an integral part of the code. This complicates future evolutions of the parallel applications. In some sense, optimisations "bind" the code to a specific computing platform with a set of features. This defeats the overall idea of developing high-level and portable applications.

2.1 Data Locality

This section gives an overview of the most frequent optimisations to improve data locality (e.g., that ameliorate the impact of the memory hall) and discuss their impact on the code. Basically, those optimisations fall into optimisations that maximise memory bandwidth usage (i.e., spatial locality) and optimisations that maximise data reuse (temporal locality). Table 1 lists the most common locality optimisations, as well as the coding impact when implementing each of those optimisations.

The most used optimisation and more challenging to implement is to improve the data layout in memory, specially in object oriented systems, like

Table 1 Most common data locality optimisations

Optimisation	Type of locality	Implementation places
Data layout transformation	Spatial	Many
Data sorting	Spatial	Single
Padding and Alignment	Spatial	Single
Packing	Spatial	Single/Many
Loop tiling	Temporal	Single
Loop fusion	Temporal	Single
Loop reorder	Spatial and Temporal	Single

Table 2 Example of AoP to SoA data layout transformation

`AoP layout`	`SoA layout`
`Particle particles[];`	`double positionX[];`
	`double positionY[];`
	`...`
`particles = new Particle[size];`	`positionX = new double[size];`
	`positionY = new double[size];`
	`...`
`particle.force(particles,...);`	`force(particles, i,particles, ...);`
`this.positionX...`	`particles.positionX[i]...`

Java. The most common data layouts for a collection of objects (or data structures) are: Array of Pointers (AoP), Array of Structures (AoS) and Structure of Arrays (SoA). AoP and AoS support coding styles where the code is closer to real-world entities (e.g., developers can work with data structures instead of array indexes) [1]. The AoP layout, adopted in Java, is popular due to its support for abstract data types. The collection is an array of pointers to the concrete data type. The other layouts improve spatial locality by storing data in contiguous memory addresses. In AoS, the entities are stored in contiguous memory addresses, as in SoA, which stores fields into separate arrays. The AoP requires additional space to hold the array of pointers, when compared to SoA, but provides more flexibility to manage the data storage.

The data layout can have a significant impact on performance, and the choice of the best might depend on the platform and algorithm [2,3]. Moreover, the change from one layout to another might require considerable code refactoring. Table 2 illustrates the data layout transformation required in Java to use a SoA layout instead of the default AoP layout: 1) the array of instances of class particle must be replaced by several arrays of particle coordinates; 2) the creation of a new collection of particles is performed by creating the corresponding raw arrays of coordinates; 3) the computation involving particles is now performed using array indexes instead of particle fields. The most important drawback is the effort required to perform the transformation. Moreover, after the transformation there is no "particle" entities in the code, only arrays of raw data.

Table 3 Example of loop tiling

original	loop tiling

`for (int i = 0; i < mdsize; i++) {`
`    one[i].force(...);`
`}`
	`for (int jj = 0; jj < nblocks; jj++) {`
	`  for (int i = 0; i < mdsize; i++) {`
	`    one[i].force(...); // new parameter jj`
	`  }`
	`}`

The data sorting and padding/alignment are optimisations that are easy to deal with since their impact on coding is more local. Data sorting reorders the data storage in memory according to the way it is accessed. Padding and alignment introduce additional fields in data structures for more efficient data accesses (e.g., recent machines are more efficient when transferring data aligned at 32 or 64 bytes boundaries). Packing (or compaction) copies the data into a new storage area in order to remove unnecessary data. It can have a wider coding impact if the new data structure is different from the original one.

The loop tiling optimisation changes the order of processing the data in order to reuse the data in cache as much as possible. In terms of coding effort this optimisation generally involves introducing one or more loops in the code (see Table 3). Loop tiling can become more complex when multiple levels of tiling are used. Loop tiling is frequently used with the packing optimisation for a more compact storage of the data reused across inner iterations.

Other frequent optimisations include loop fusion and loop reorder. Both optimisations have a relative small impact on the code.

The development of efficient parallel applications usually involves the implementation of several of these optimisations. For instance, it is quite common to compose an efficient data layout (e.g., SoA) with tiling and packing, however, packing implementation depends on the layout (i.e., it is implemented in a different way for an AoS or a SoA layout). Developing a framework supporting a wide range of optimisations is a complex task, specially if the goal is to keep the code at high-level, platform independent. The Gaspar framework address this challenge by relying on a model driven approach (i.e., the specification of a domain model) as the base of code generation, and on Java annotations to further generate optimised code.

2.2 Java collections

A key part of Java is the Java Collections Framework (JCF) that supports a set of data containers that manage the memory without developer intervention and allow hiding the collection layout (e.g., ArrayList vs LinkedList). All containers available in the JCF support generics. JCF collections use the AoP layout, making the default implementations not suited for HPC.

The ArrayList is the most used data structure of the JCF, approximately 47% in the study by Costa [4]. The OpenJDK implementation uses an array of pointers. Additionally, the ArrayList adds a mechanism that allows the developer to add and remove elements without overloading the developer with the array dimension management.

The Java collections support generic types, although, in many cases, the developer only needs a collection of a single data type. In Java, the developer can use Java collections or arrays of objects, but both have a negative impact on performance. These are collections of objects, where the array has the pointers to objects. This representation requires one extra instruction for each access and spends more space (object headers and the pointers). Moreover, in Java, it is not guaranteed that the objects are allocated in contiguous memory, thus, the spatial locality is low. On the other hand, primitive arrays are allocated in contiguous positions in memory. So, there is only one header for the array, and one data item can be accessed with a single instruction.

JCF does not provide containers of primitive data types, but it can use collections of objects that represent the primitive types. For this, Java provides a mechanism for converting primitive type variables into objects of the same type [5]. The solution creates an overhead [6], due to primitive type conversion into an object, and the AoP layout. There are several approaches to use primitive data types arrays backed by arrays of primitive data [7–9]. Thus, these approaches improve the performance by removing the load instructions to access the object and reducing the memory footprint (remove the object header). However, these approaches do not support structured data types and remove the domain abstractions from the code.

Java provides iterators to process the entire container. Iterators allow hiding the container implementation to the developer, so it is possible to process multiple containers types using the same source code. Iterators are typically a safe approach for accessing containers, as they limit access to the elements in the container. Iterators, being more abstract than using indexes, introduce more instructions. The JIT compilation can remove these additional instructions in most cases.

Java iterators use the natural order to process the collection (e.g., the next iterator method) and it possible to modify the iterator implementation (e.g., provide Java-compatible iterators). Java uses internal iterators to process the collection elements with foreach and in the streams interface.

3 Gaspar framework

In traditional parallel programming frameworks, the developer accesses data directly, frequently using indexes in arrays of raw data (e.g., see the SoA layout example in Table 2). This makes it hard to implement optimisations involving data layout transformations since the data representation is exposed in the source code. The Gaspar framework was developed to address this issue by promoting an abstract (e.g., object oriented) and efficient way for accessing data: iterators and higher-order functions. Java iterators hide the implemen-

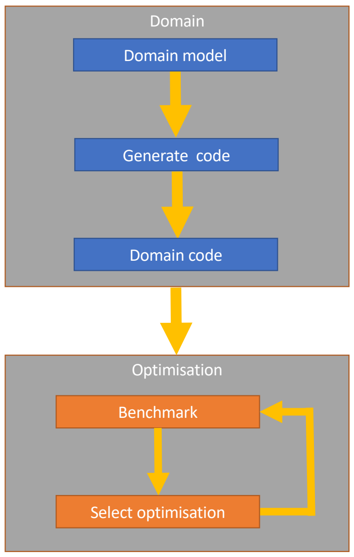


Fig. 1 Approach overview

tation of a collection of objects. However, to support efficient data layouts, the approach needs to hide both the collection and objects implementation.

The approach supports the Java collections API (e.g., the List and Iterator interfaces) but requires the usage of getter and setter methods for accessing the object data fields, which is a common practice in object-oriented programming. The combination of iterators and getter/setter methods allows hiding the layout representation and providing efficient data access, namely by transparently using a SoA collection implementation. The approach supports two generic types of locality optimisations for HPC in Java applications:

1. Change of data layout - transparently supports different layouts for object collections (e.g., AoP and SoA) with the same programming interface. The developer creates the domain code without depending on the internal data representation of collections of objects. More efficient data layouts (e.g., the SoA layout) can be used without removing the object abstraction from the code (e.g., see Table 2: Particle objects and methods call on these objects).
2. Change of the execution flow - provides a high-level constructor that allows applying optimisations that change the execution flow in the final development step. Thus, the optimisation step is independent of the domain code development.

The proposed approach involves two generic steps (figure 1): Domain and Optimisation. In a first step, the developer specifies the domain code without being concerned with optimisations. For this, the approach provides a high-level programming interface compatible with the Java collections. The interface provides collections, iterators and higher-order functions for processing collections of objects. In a second step, the developer specifies the layout for those collections and other optimisations. Those optimisations are specified by annotations that make the optimisation code pluggable (it is possible to enable or disable those optimisations), as such, this step does not require any rewrite of the domain code developed in the first step.

The Domain specification involves three sub-steps (blue boxes in figure 1). First, the developer designs the domain model in the Unified Modelling Language (UML). The developer represents the domain concepts and the relationship between them. Second, the framework uses the domain model to generate a library containing collections implementations. The compositions relationships of 1 to N are converted into the corresponding collections. Third, the developer writes the domain code using Java interfaces without knowing the data layout implementation.

At the end of the Domain step, the program already implements all the functional requirements. The program performance is not relevant at this first step: the emphasis is on program abstraction and correctness. In the Optimisation step, the performance is analysed and improved. The developer analyses the program execution and identifies the code parts that should be optimised using profiling tools (Benchmark step). At this step the developer can also introduce parallelism. Additionally, the framework includes a tool that enables access to the processor counters in Java to help the developer on this step. In the Select optimisation, the developer can optimise the program using two distinct mechanisms: layout and domain decomposition (gSplitMapJoin mechanism that will be explained later).

3.1 Domain specification

The domain specification starts with the creation of the domain model, a standard task on object-oriented designs, where the system is decomposed "according key abstractions in the problem domain" [10]. The domain model in the Gaspar approach is a UML class diagram, where the developer specifies the entities from the domain, their attributes and the relationships among entities. The framework comprises a tool, implemented as an eclipse plugin, to specify that model. The framework generates collections to support the different collections layouts for each entity from the model. The entities specified should have a composition relationship from 1 to N or 1 to 1. The current tool does not allow polymorphic entities (i.e., all entities in collection must belong to the same concrete class), which enables high performance implementations. Figure 2 shows a simplified example for a Particle object from a molecular dynamics (MD) case study: a particle is composed by three 3D vectors, representing the particle position, velocity and the force acting on that particle in an 3D space. Note that particle is a structured data type, composed of three internal objects (3D vectors).

After generating the required classes from the domain model the developer can write the domain code using the generated interfaces. The framework support the basic Java interface, using the List and Iterator interfaces, which

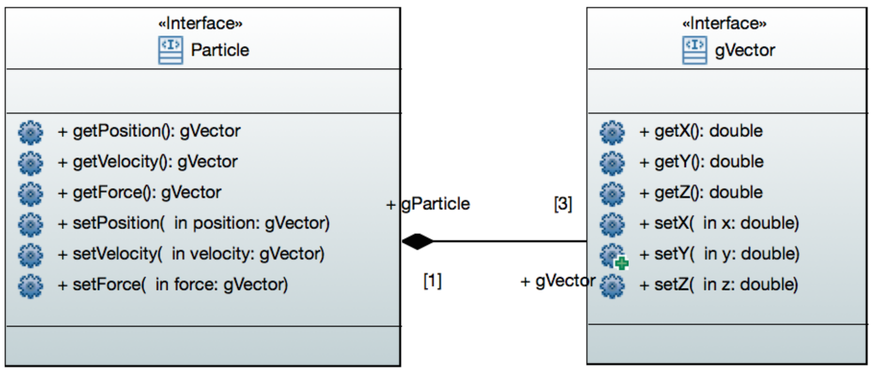


Fig. 2 Domain model comprising a Particle entity, composed of three 3D vectors

```
// the same method for all data layouts
void forceParticle(Particle p1, gList<Particle> particleSet) {
   // get coordinates of particle p1
   xi = p1.getPosition().getX();
   yi = p1.getPosition().getY();
   zi = p1.getPosition().getZ();

   // iterate over particleSet
   gIterator<Particle> iterator = particleSet.iterator();
   while(iterator.hasNext()){
      Particle p2 = iterator.next();
      // compute distance
      xx = xi - p2.getPosition().getX();
      yy = yi - p2.getPosition().getY();
      zz = zi - p2.getPosition().getZ();
      (...)
  }
}
```

Fig. 3 Example of the framework usage

is specially useful to adapt code already developed to use those generated collections.

Figure 3 illustrates the program to compute the force between a Particle p1 and the other particles in a collection. To obtain the Position vector, the developer uses getPosition, and then, accesses to the vector fields through the getX, getY and getZ getter methods. To access to particleSet, the developer uses a Java iterator and a while loop to process all elements. Note that Particle, gList<Particle> and gIterator<Particle> are interfaces, that will be implemented by concrete classes according to the data layout (this code is layout independent). The Gaspar gList and gIterator interfaces provide a more advanced API that is compatible with the Java List and Iterator interfaces (actually, the code in figure 3 is plain Java code with these small name differences).

```
@gSplitMapJoin(name = "Tiling", map = "Sequential", \\
     split = {none ,"Virtual"}, reduce = {" default", "default"})}
void computeForces(gList<Particle> c1, gList<Particle> c2){
    gIterator<Particle> it1 = c1.iterator();
    while(it1.hasNext()) {
       gIterator<Particle> it2 = c2.iterator();
       while(it2.hasNext()) {
         ...
```

Fig. 4 Example of the specification of tiling using the gSplitMapJoin annotation

3.2 Optimisation specification

The approach makes it possible to develop a program where the code has the domain entities (e.g., Particle interface in the example of Figure 3), but the layout details are hidden (the developer writes the code using an API closer to the AoP layout). The layout can be selected subsequently in the execution step, making it possible to test layouts easily. The developed tool generates two layouts by default: AoP and SoA. It is also possible to change the layout on a specific part of the program. For this, the developer uses the packing optimisation that will be presented later.

In addition to the specification of efficient data layouts, the framework supports several other optimisations by using a domain decomposition approach that is described next. Generically the approach works as follows: the problem domain is divided into sub-problems, the original function is applied to each sub-domain and, after processing, processed sub-domains are joined together.

In OOP codes data collections are accessed using iterators. Accessing several collections with several iterators is equivalent to nested "indexed fors" in traditional non object-oriented codes. In those cases, the developer can insert one annotation in the original method (see Figure 4) to specify how to split the domain, process sub-domains and join processed sub-domains. The annotation syntax is defined as follows:

– @SplitMapJoin - identifies the annotation
– name - defines the internal name to use (for more advanced uses)
– map - kind of map to use (sequential, parallel, etc ...)
– split - defines how collections are split (one for each method parameter)
– join - define how to join processed sub-collections.

The approach works by changing the method parameters by dividing the collections, thereby the problem is processed as several subproblems. For each collection used as method parameter, the developer defines the option for domain decomposition (the split annotation field). On the other hand, there are several options to aggregate the subcollections after processing (the join annotation field). A new method is internally generated (defined by the name field), that calls the original method with each subdomain. These calls can be performed in sequential or in parallel, according to the option specified in the map field.

In practice, the example in Figure 4 applies the tiling optimisation to the loops inside of the computeForces method. The annotation processing tool creates a new method (Tiling.computeForces(...)) that calls the original method multiple times (each time with one of the subdomains). In this case, it decomposes the collection c2 into several collections.

Current processors have multiple cache levels, so, sometimes, it is more efficient to partition the domain several times. The approach supports tiling with multiple levels since the same mechanism can be applied again by annotating the generated method (Tiling.computeForces(...)). In those cases, the annotation "name" field helps to generate a method with a meaningful name.

The following subsections provide more details about how the annotation supports several types of optimisations

3.2.1 Packing

The annotation supports the specification of the packing optimisation by selecting a specific kind of split. In the figure 4 the collection c2 is divided using a Virtual split. This kind of split creates views over the original c2 collection. Alternatively, the developer can choose to divide the collection in a way where the data of each subdomain is copied into a new collection (i.e., to perform packing). In this case there are two packing alternatives: Packing and PackingOnDemand. Packing creates all subcollections before processing and writes the results, into the original collection, after processing all subdomains. PackingOnDemand creates a subcollection and copies the data into the subcollection only when necessary. After processing each subproblem, the data is rewritten into the original collection. If the processing is performed in parallel, there is one subcollection for each thread. The packing allows changing the layout of a collection or subcollection. For this purpose, the developer can use PackingSoA or PackingOnDemandSoA in the split specification.

3.2.2 Parallelism

The current approach implementation has several options to implement maps including sequential and parallel versions. The sequential version processes the elements by the natural order. The developer has several schedulers available for parallel processing, namely: ParallelBlock, ParallelCycle and ParallelDynamic. The ParallelBlock aggregates elements into blocks. The block size is computed as the number of elements divided by the number of threads. If the number of elements is not divisible by the number of threads, the last block is larger. In the Cycle scheduling, the thread TT processes the element%number of threads. If the number of elements is less or equal to the number of threads, this distribution is the same as of Block distribution. ParalllelDynamic scheduler creates one task per element in the collection and a thread pool distributes the tasks by the threads in a dynamic way.

```
public static void reduceMethod(Particle c, Particle ret){
  ret.getForce().setX(c.getForce().getX() + ret.getForce().getX());
  ret.getForce().setY(c.getForce().getY() + ret.getForce().getY());
  ret.getForce().setZ(c.getForce().getZ() + ret.getForce().getZ());
}
```

Fig. 5 Example of reduce method that sums all private values

```
@gSplitMapJoin(name = "PForce", map = "ParallelBlock", \\
   split = {"Virtual", "PrivateForce"}, reduce = {"default","md::reduceMethod"})
```

Fig. 6 Annotation example using a private collection in the second method argument

3.2.3 Privatisation

The map annotation field can be used to specify parallel execution, but it may originate data races due to concurrent data accesses. The developer can use the Java mechanisms to avoid data races in a per-object base (e.g., locks), however, these mechanisms generally introduce an unacceptable overhead, since they serialise the execution. Thread local data (aka privatisation) is an important optimisation to improve the scalability of parallel programs in such cases. The framework supports the specification of private fields in the domain model. For this purpose, the developer adds private attributes into an Annotated Element, where it also specifies the split method name to generate. Based on this element the UML tool generates a new split that returns a collection where the private field is local to each thread and shared fields can be accessed by all threads. Note that, when introducing thread-private data fields, the domain model and the domain code remains unchanged, only specific fields in the domain model are annotated and a rebuild of the generated classes is required. Thus, the developer uses the original domain code, but the data access methods hide if the field is private or shared (e.g., one of the Particle getter methods in Figure 3 can return a thread local value).

The privatisation mechanism might require an explicit specification of how to reduce the thread local objects. In the developed framework the problem is more complex since thread local objects may be implemented using different data layouts (e.g., AoP or SoA). Thus, a data layout-independent specification is necessary to support the privatisation mechanism. Figure 5 shows an example of a user specified reduce method for the MD case study. This method receives two parameters: the object pointing to the private collection, and the object to access the original collection. The implementation just needs to follow the same coding rules of the domain code (e.g., Figure 3). In this case, the collection's values are reduced simply using the sum operation.

In the MD example a more efficient implementation uses a private force vector. Figure 6 shows a gSplitMapJoin that uses a private Force vector on each thread. The developer uses the split which name is equal to the comment defined in the UML tool ("PrivateForce"). Additionally, the developer uses the

Table 4 Illustrative code of the Sum versions tested

Java Index (AoP layout)	Java Interator (AoP)	Gaspar (AoP and SoA layout)

int i;
i=0;
for (; i<c.size(); i++) {
result += c.get(i); }
	Iterator<Double> it;
	it = c.iterator();
	while(it.hasNext()) {
	result += it.next(); }
		Iterator<gDouble> it
		it = c.iterator()
		while(it.hasNext()) {
		result += it.next().getValue(); }

md::reduceMethod to reduce the private fields, created by the "PrivateForce" split.

3.2.4 Data sorting

Data sorting can be applied to the AoP collections. The consecutive collection elements are placed in successive memory positions, to improve spatial locality. The developer just calls the collection's sort method.

4 Evaluation

The performance is evaluated with five applications. The first algorithm sums all the elements in a collection. The second algorithm is DAXPY, which calculates the y = alpha * x + y operation for each element in the y collection, where alpha is a constant and x is the element in another collection. This case study allows auto-vectorisation, so the goal is to analyse the iterators impact on vectorisation. The third case study evaluates the approach in the context of an open-source Java framework for scientific applications whose code was already developed. JECoLi [11] is a Java framework for evolutionary computing that uses Object Oriented programming in an effective manner. The fourth case-study uses a more complex data structure. The base code comes from the JGF moldyn benchmark [12]. The last case study is a highly tuned matrix multiplication (MM) kernel developed in Java, whose base version uses a MM kernel developed in plain Java, that was converted from a C-like code and requires the use of tiling to attain a competitive base performance.

The results presented were obtained on a Linux compute node that has two Intel Xeon E5-2695v2 processors with the Ivy Bridge architecture. Each processor has 12 cores and supports 24-threads. The processor supports Intel Turbo Boost technology, but in this evaluation, the frequency is fixed at 2.4GHz, allowing the results to suffer less variation. The programs execution use the JVM from the OpenJDK 1.8.0 20 package. Additionally, the evaluation uses several JVM tuning parameters: -Xmx32G, -XX:LoopUnrollLimit=100, -XX:+UseCompressedOops and -XX:+UseNUMA.

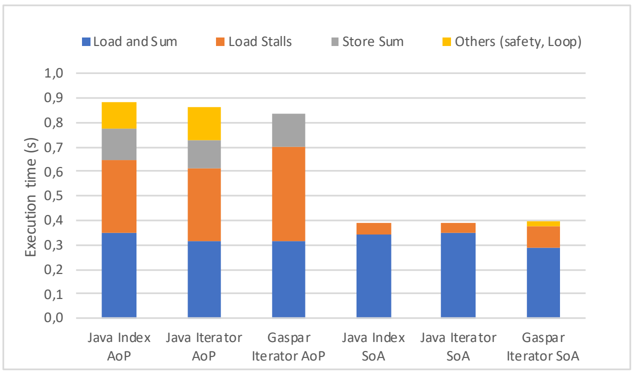


Fig. 7 Performance of sum versions tested

4.1 Iterator overheads

This section evaluates the overhead of different kinds of accesses to data collections. For this specific purpose the sum benchmark is used. The goal is to compare the performance of using different data structures and ways of iterating over those data structures. Table 4 illustrates the different codes tested in this subsection. Performance results are presented in Figure 7, for a sum of 16M of doubles. The overall execution time was divided into sub-types (taken using the linux perf tool) that are explained next.

The sum benchmark loads a double value from memory and performs a sum into an accumulated value. This is the base "Load and Sum" time in the figure. However, there might be additional overheads in this process, the most important being the time waiting for the memory to deliver the required value. This time is the "load stalls" time in the figure and it is measured using a specific hardware event count. The results presented show that there is no performance difference between index-based and iterator-based when using a SoA layout (e.g., raw data storage). Moreover the Gaspar framework provides the same performance as the base Java (note that the Gaspar uses a higher-level API that hides the data layout).

The AoP layout imposes two significant overheads: 1) additional memory stalls due to less locality of an AoP layout; 2) Less optimized code due to the inability to keep the accumulated sum in a register ("store sum" overhead in the figure). In this case the AoP layout takes 32 bytes per double value, due to the object header and 16-byte boundary alignment, plus an additional 4-byte for the object pointer. This results in more than 4x used space in L1 cache and in other memory levels, almost doubling the time required to perform the sum. Java introduces additional small overheads that are avoided in the Gaspar framework in this simple case study (range check, type check, etc.).

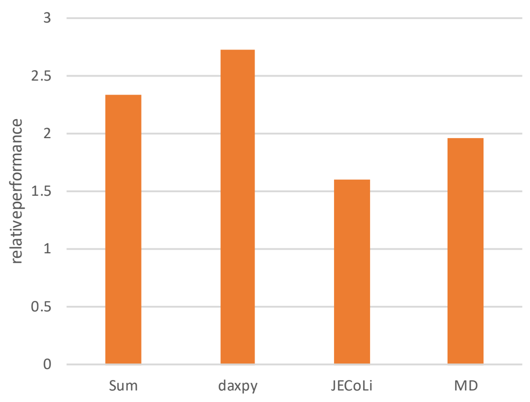


Fig. 8 Data-layout impact: speed-up when moving from an AoP to a SoA data layout

4.2 Gaspar and Data-layout impact

In simple case studies, such as the Sum and DAXPY the Gaspar framework delivers the same performance as low level Java (e.g., index based in Figure 7). The base performance of the Gaspar framework in the MD case study ranges from 0.82x with an AoP layout to 0.95x with a SoA layout (an JGF version converted to this layout).

Figure 8 shows the gains delivered by the Gaspar framework due to layout improvement, computed as the speed-up in execution time over these base sequential Gaspar AoP implementations. For Sum the SoA layout improves the performance in almost 2.5 times. In DAXPY, the layout provides a gain greater than 2.5x. Part of this gain is due to the vectorisation enabled by the SoA data layout. This explains why this is the case study with the best speed-up. In the MD case, an 2x speed-up is observed with the SoA layout. The JECoLi framework provides more moderate improvements, however, the given number is an average of three cases studies (CountOnesCAGATest, CountOnesEATest,CountOnesSPEA2Test).

4.3 Parallelism

Figure 9 shows the improvement in performance when combining the exploitation of parallelism with the data layout improvements. It was possible to reduce the execution time by almost 20 times, for all case studies that support parallelism. In the MD case the execution time was reduced by 50 times. In JECoLi, this improvement is less since the parallel execution is not usable in the examples available in the framework repository. The improvement is just due to a more efficient layout.

Note that in these case studies several other mechanism are required in order to enable parallelism, namely, thread-private fields are required in the Sum (i.e., the partial sum), in the MD (i.e., partial forces sum) and in the

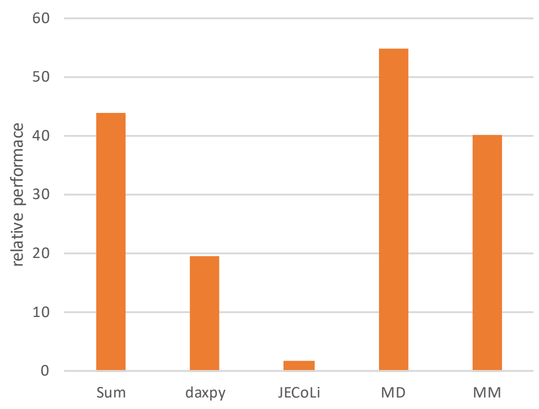


Fig. 9 Combined performance of data-layout improvement and parallelism

```
protected int countOnes(ILinearRepresentation<Boolean> genomeRepresentation){
  int countOneValues = 0;
  for(int i = 0;i < genomeRepresentation.getNumberOfElements();i++)
    if(genomeRepresentation.getElementAt(i))  countOneValues++;
  return countOneValues;
}
```

a) Original version

```
protected int countOnes(ILinearRepresentation<gBoolean> genomeRepresentation) {
  int countOneValues = 0;

  for(int i = 0;i < genomeRepresentation.getNumberOfElements();i++)
    if(genomeRepresentation.getElementAt(i).getValue())  countOneValues++;
  return countOneValues;
}
```

b) GasPar Version

Fig. 10 Illustration of JECoLi required code changes

MM (i.e., C sub-matrix local to each thread). Additionally, in the MM case study packing and tilling are mandatory the have an efficient sequential implementation. In the MD case study the tiling was tested but the SoA layout, by itself, delivers the best performance (note that these kind of tests are quite easy to perform after converting the code to the Gaspar framework). Similarly, the best MM parallel implementation is a combination of different packing approaches for each matrix: the best implementation uses packing for matrices A and B and packing on demand for matrix C. The basic virtual split option (e.g., without packing) does not provide an acceptable performance.

4.4 Programming effort

Evaluating the programming effort is a complex task. In this subsection we, instead, present illustrative examples of the overall programming effort in terms of the amount of code changes and/or additional code required.

Table 5 JGF AoP MD code changes

Original MD (AoP layout)	Gaspar
Particle one[];	gList<Particle> one;
	...
one = new Particle[size];	one= factory.creategList(mdsize);
	...
positionx = one[i].xposition;	positionx = one.get(i).getPositionX();
one[i].xposition = positionx;	one.get(i).setPositionX(positionx);
	...
for (i = 0; i < mdsize; i++)	for (gIterator it : one)
one[i].force(..., i, one);	force(..., it.get(), one);

The JECoLi framework only required a small amount of code changes (one day to convert all the code examples in the framework repository). This small amount of work is due to the object-oriented nature of the framework, that already provided the required getter methods. The JECoLi uses a List of generic objects. In practice, the examples provided in the framework repository use a List of Double, Integer or Boolean. The programming effort was basically to convert these List to use Gaspar collections, using the necessary getter methods. Figure 10 illustrates this change.

The MD case study required more extensive code changes in the base JGF implementation, however the programming effort is clearly less than the effort required to implement an SoA layout (see Table 2). Table 5 illustrates these changes.

The changes are of four types: 1) arrays are replaced by Gaspar collections; 2) collections are created using a factory method; 3) particle coordinates are accessed using setter and getter methods; 4) for loops are converted into iterator-based syntax. Overall the usage of getter and setter methods is the one that requires the most extensive changes. However, it might be possible to perform a simple search and replace approach in this case.

5 Discussion and related work

Table 6 shows a comparison of related approaches. The proposed framework provides the most complete set of optimisations, including support for parallel execution. The next paragraphs provide a discussion of alternatives to support each of proposed optimisation, with an emphasis on High-Level programming.

5.1 Data layout

The choice of the data layout, in traditional approaches, is made at the beginning of the code development. Thus, changing it after the initial coding step can imply wide range code changes. On the other hand, the most efficient layouts typically do not use domain concepts, as they are closer to the execution

Table 6 Support for common data locality optimisations

	Layout	Sorting	Packing	Tilling	Parallelism	Priv.
Sharma, SDLT, Wimmer	X					
Hirzel		X				
OpenMP					X	X
OpenACC			X	X	X	
MInt				X	X	
Pluto, Polly				X	X	
Proposed Approach	X	X	X	X	X	X

platform (e.g. the SoA layout). There are several options to allow the choice of the layout in the final development step. The most common options are data encapsulation, code transformation, use of proxies and use of the JVM to manipulate the data accesses.

The data encapsulation technique hides the layout over an API by creating temporary objects (e.g., using the adapter pattern). The overhead of creating this object is removed in most cases, in recent versions of the JVM. However, in older versions of the JVM, it introduces a significant performance overhead [6]. The proposed approach behaves closer to Java collections, since it returns an iterator/proxy that enables direct access to the object in the collection. This approach reduces the overhead by using the same object several times (e.g., using the same iterator/proxy to iterate over all the elements in a collection). Actually, a data encapsulation strategy could be created to simulate the proposed framework, however, the programming overhead would be very high, specially if other types of optimisations are supported.

In the code transformation approach, the source code is processed and changed to implement the desired layout. Sharma [3] presents a C++ framework that can change the AoS layouts to SoA or hybrid layouts. The tool processes the code changing the layout (source to source approach) specified by metadata. It can also create hybrid layouts (between AoS and SoA) automatically, according to the way that fields are accessed.

Wende [13] suggest one approach based on proxy objects. The approach uses macro-based C++ to generate the code referring to the proxy that allows using both the AoS layout and the SoA layout. This approach is based on a strategy similar to the proposed approach.

Intel suggests using the AoS layout for design and using the SoA layout for performance. SIMD Data Layout Templates (SDLT) is a template library for C++ language that enables abstract code (use of an AoS- based API) and uses the layout SoA in memory. The template library creates an implementation in the pre-compiler step, where the layout is also selected. SDLT uses the C++ operator overloading to enable the traditional access API (e.g., a[k].field1). However, the use of this API introduces new copies of the elements and consequently introduces overhead. To reduce the overhead, SDLT provides an alternative API that implies accessing data using methods (e.g., a[k].field1()). This strategy is similar to the proposed API and implies the same code rewrite (get and set for each field). However, in SDLT the data structure can only have primitive types, although the proposed approach supports complex structures.

The Java language provides an implementation alternative at the JVM level since it can modify the data layout in memory. This approach allows a more efficient layout, but it is only possible to turn AoP into an AoS. Wimmer et. al. [14] propose an improvement to the JVM to automatically in-line object fields by placing the parent and children objects in consecutive memory places and by replacing memory accesses by address arithmetic. The authors point out that using arrays as in-lining parents is complex since the Java byte-codes for accessing array elements have no static type information. They claim that an automatic AoP to AoS transformation at JVM level is impossible without a global data flow analysis.

5.2 Data sorting

Hirzel [15] modified a JVM Garbage Collector to sort objects into memory according to their temporal affinity. Objects that used at the same time are placed in nearby memory zones. The JVM sorts the objects during the garbage copying. This technique still maintains the AoP layout and thus cannot avoid the overhead of pointer indirection. This approach is transparent to the developer, but this requires a modified JVM. In our approach, we sort objects according to their position in the collection (AoP) for a similar effect.

5.3 Tiling and packing

The tiling can use two approaches: loop rewrite or decompose the domain into multiple subdomains. The loop rewrite implies adding new loop(s) in the code and redefine the internal loop(s) limits. It can be implemented manually, by annotating the code or by a specific compiler. In these two last strategies, the tiling optimisation is applied by a code analysis and transformation tool. The loop rewrite has no meaning at the domain level. However, the domain decomposition technique is abstract (closer to the domain). Additionally, it makes it easy to apply packing optimisation.

OpenACC and Mint [16] are two programming frameworks that provide OpenMP-like directives to support the loop tiling through a specific loop clause. Both approaches apply tiling through primitives which simplifies code development by reducing development errors.

The Polyhedral model allows the analysis of dependencies within nested loops. It can identify the tiling optimisations. Pluto [17] and Polly [18] are tools that uses the Polyhedral model to apply the tiling optimisation.

The packing is typically associated with the tiling. The tile is copied into consecutive memory positions. OpenACC provides the cache directive. So, the compiler uses this directive to explore data access optimisations (data in registers, software-managed cache, or read-only cache) [19].

Our approach uses annotations that are similar to the directives. However, in our case, annotation partitions the collections in order to redefine the limits of the loops. On the other hand, our approach generates a new method that allows us to create more levels of tiling. The proposed approach uses domain decomposition to apply tiling optimisation. Additionally, the approach allows the developer to use packing.

The approaches presented for tiling are based on loops. Our approach uses domain decomposition to decompose the domain into smaller parts, bringing the optimisation technique closer to the domain. However, this option has a little cost since it requires new structures to support the subdomains. On the other hand, the approach enables the packing optimisation for each subdomain. In short, our approach increases the code abstraction and enables packing optimisation with a small overhead.

5.4 Parallelism and privatisation

OpenMP uses annotations to introduce parallel execution in (sequential) base codes. OpenMP provides a fork-join execution model. The parallel for is one of the most commonly used annotations since it runs loop iterations across multiple threads. OpenACC uses primitives similar to OpenMP for developing parallel code to run on accelerators (e.g., GPUs). Our approach partitions the domain to support parallel execution. OpenMP and similar approaches partitions loop having no direct meaning in the domain. Therefore, our approach inserts parallelism through domain concepts.

6 Conclusion

Developing high-level and abstract code requires the implementation of optimisation mechanisms in order to effectively use the current multicore systems. The Gaspar framework provides an high-level programming interface that is compatible with Java collections. Simultaneously, the framework implements a development cycle where the abstract code can be optimised in later development stages. Thus, programmer can start by building a correct implementation, and later focus on performance issues in a pluggable manner (e.g., no source code rewriting is necessary).

The framework delivers the most complete set of locality optimisations which are essential in modern high-level programming frameworks. Performance results show that the sequential execution time can decrease to more than half (e.g., 2.5 speed-up). This speed-up, combined with the parallelism exploitation can provide more than 50x speed-up over a non-optimised base sequential implementation. The evaluation also showed the feasibility of applying the proposed approach to improve "legacy" Java code in a low effort manner.

Future work includes the support for more platform mappings, namely, by supporting mappings for GPUs. In the long term, the framework can contribute to a more automatic performance tuning for multicore execution.

References

1. James Jeffers, James Reinders, and Avinash Sodani. Intel Xeon Phi Processor High Perfor- mance Programming: Knights Landing Edition. Morgan Kaufmann, 2016.
2. Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. Compiler- driven data layout transformation for heterogeneous platforms. In European Conference on Parallel Processing, pages 188–197. Springer, 2013
3. Kamal Sharma, Ian Karlin, Jeff Keasler, James R McGraw, and Vivek Sarkar. User-specified and automatic data layout selection for portable performance. Rice University, Houston, Texas, USA, Tech. Rep. TR13-03, 2013.
4. Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and per- formance of java collections. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, pages 389–400. ACM, 2017.
5. Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In Proceedings of the 38th International Conference on Software Engineering, pages 225–236. ACM, 2016.
6. Nuno Faria, Rui Silva, and Joao L Sobral. Impact of data structure layout on performance. In 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pages 116–120. IEEE, 2013.
7. S Osinski and D Weiss. Hppc: High performance primitive collections for java, 2015.
8. GNU Trove. High performance collections for java.
9. Sebastiano Vigna. fastutil: Fast and compact type-specific collections for java, 2016.
10. Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. 2007. Object-oriented analysis and design with applications, third edition (Third. ed.). Addison-Wesley Professional.
11. P. Evangelista, P. Maia, and M. Rocha. Implementing metaheuristic optimization algorithms with jecoli. In 2009 Ninth International Conference on Intelligent Systems Design and Applica- tions, pages 505–510, Nov 2009.
12. M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande appli- cations. In Proceedings of the Second International Conference on The Practical Applications of Java, Manchester, UK, pages 63–73, 2000.
13. Florian Wende. C++ data layout abstractions through proxy types. In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 758–767. IEEE, 2019.
14. Christian Wimmer and Hanspeter Mossenbock. Automatic array inlining in java virtual ma- chines. In Proceedings of the 6th annual IEEE/ACM international symposium on Code gener- ation and optimization, CGO '08, pages 14–23, New York, NY, USA, 2008. ACM.
15. Martin Hirzel. Data layouts for object-oriented programs. In Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, SIGMETRICS '07, pages 265–276, New York, NY, USA, 2007. ACM.
16. Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In Proceedings of the international conference on Supercomputing, pages 214–224. ACM, 2011.
17. Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A practical and fully automatic polyhedral program optimization system. In ACM SIGPLAN PLDI, volume 10, 2008.
18. Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simburger, Armin Großlinger, and Louis-Noel Pouchet. Polly-polyhedral optimization in llvm. In Proceedings of the First Inter- national Workshop on Polyhedral Compilation Techniques (IMPACT), volume 2011, page 1, 2011.
19. Ahmad Lashgar and Amirali Baniasadi. Openacc cache directive: Opportunities and opti- mizations. In 2016 Third Workshop on Accelerator Programming Using Directives (WACCPD), pages 46–56. IEEE, 2016.

Noname manuscript No.
(will be inserted by the editor)

Generic Exact Combinatorial Search at HPC Scale

Ruairidh MacGregor · Blair Archibald · Phil Trinder

Received: date / Accepted: date

Abstract Exact combinatorial search is essential to a wide range of important applications, and there are many large problems that need to be solved quickly. Searches are extremely challenging to parallelise due to a combination of factors, e.g. searches are non-deterministic, dynamic pruning changes the workload, and search tasks have very different runtimes. YewPar is a new C++/HPX framework that generalises parallel search by providing a range of sophisticated search skeletons.

This paper demonstrates *generic* high performance combinatorial search, i.e. that a variety of exact combinatorial searches can be easily parallelised for HPC using YewPar. We present a new mechanism for profiling key aspects of YewPar parallel combinatorial search, and demonstrate its value. We exhibit, for the first time, generic exact combinatorial searches at HPC scale. We baseline YewPar against state-of-the-art sequential C++ and C++/OpenMP implementations. We demonstrate that deploying YewPar on an HPC can dramatically reduce the runtime of large problems, e.g. from days to just 100 seconds. The maximum relative speedups we achieve for an enumeration search are near-linear up to 195(6825) compute-nodes(workers), super-linear for an optimisation search on up to 128(4480) (pruning reduces the workload), and sub-linear for decision searches on up to 64(2240) compute-nodes(workers).

Keywords Combinatorial search · Parallel skeletons · Constraint programming · High performance computing

1 Introduction

Exact combinatorial search is essential to a wide range of important applications including constraint programming, graph matching, and planning. A classic

School of Computing Science, University of Glasgow, Glasgow, UK
E-mail: {blair.archibald, phil.trinder}@glasgow.ac.uk

example would be to allocate parcels to vans, and to plan delivery routes for the vans. Combinatorial problems are solved by systematically exploring a space of (partial) solutions, and doing so is computationally hard both in theory and in practice, encouraging the use of approximate algorithms that quickly provide answers yet with no guarantee of optimality. Alternatively, *exact* search exhaustively explores the search space and delivers provably optimal answers. Conceptually exact combinatorial search proceeds by generating and traversing a (huge) tree representing alternative options. Backtracking branch and bound search is a well known example. Although searches can be time consuming, combining parallelism, on-demand tree generation, search heuristics, and pruning can effectively reduce execution time. Thus a search can be completed to meet time constraints, for example to load and dispatch a fleet of delivery vans each morning.

Exact combinatorial search is very different to classical HPC applications. The problems are NP-hard rather than polynomial. There is almost no use of vectors or matrices: the primary data structure is a huge, dynamically generated, irregular search tree. Almost all values are discrete, e.g. integers or booleans. In place of nested loops there are elaborate recursive control structures. The parallelism is highly *irregular*, that is the number and runtimes of tasks are determined by the search instance, and vary hugely, i.e. by several orders of magnitude[21].

There are three main *search types*: *enumeration*, which searches for all solutions matching some property, e.g. all maximal cliques in a graph[1]; *decision*, which looks for a specific solution, e.g. a clique of size k; and *optimisation*, which looks for a solution that minimises/maximises an objective function, e.g. finding a maximum clique. There are standard *instances* for many important *search applications*, e.g. the DIMACS instances [10].

YewPar is a new C++ parallel search framework [5,2]. YewPar generalises search by abstracting search tree generation, and by providing algorithmic skeletons that support the three search types. The skeletons use sophisticated search coordinations that control the parallel search including when new tasks are generated. These are inspired by the literature, and are currently: Sequential, Depth-Bounded, Stack-Stealing and Budget. It also provides low-level search specific schedulers and utilities to deal with the irregularity of search and knowledge exchange between workers. YewPar uses the HPX library for distributed task-parallelism [11], allowing search on multi-cores, clusters, HPC systems etc.

This paper makes the following contributions.

Only a small number of specific exact combinatorial searches have been hand-crafted for HPC scale (around 1000 cores), e.g. [9,4].

We present **a new mechanism for profiling key aspects of generic parallel combinatorial search** in YewPar. While the extreme irregularity of parallel combinatorial search is well known, it is seldom measured, and then

[1] A clique in a graph is a set of vertices C such that all vertices in C are pairwise adjacent. Maximal cliques cannot be extended by including one more adjacent vertex and Maximum cliques are the largest cliques in the graph.

only for specific search applications, e.g. [21]. Key novelties are (1) to provide profiles that quantify the irregularity of various search applications in the generic YewPar framework, e.g. to report median task runtime, maximum task runtime, etc. (2) to make irregularity characteristics visible to the developer to aid performance tuning (Section 5).

We demonstrate for the first time generic exact combinatorial searches at HPC scale. Generic YewPar searches have previously only been demonstrated on relatively small-scale clusters: 17 compute nodes and 270 cores [5]. We use a combination of techniques like repeated measurements and using multiple search instances to address the challenges of parallel search, e.g. non-determinism, non-fixed workloads, irregular parallelism, and the nature of NP-hard problems. Baselining against state-of-the-art sequential C++ and C++/OpenMP implementations on 9 standard (DIMACS) search instances shows that the generality of YewPar incurs a mean sequential slowdown of 9.6%, and a mean parallel slowdown of 27.6% on a single 18-core compute node. Guided by the profiling we effectively parallelise seven standard instances of the three searches, and systematically measure runtime and relative speedups at scale. We demonstrate that deploying YewPar on an HPC can dramatically reduce the runtime of large problems, e.g. from days to just 100 seconds. The maximum relative speedups we achieve for the Numerical Semigroups enumeration search are near-linear up to 192(6825) compute-nodes(workers), super-linear for a Maximum Clique optimisation search on up to 128(4480) (pruning reduces the workload), and sub-linear for a k-clique decision search on up to 64(2240) compute-nodes(workers) (Section 6).

2 Background

2.1 Exact Combinatorial Search as an HPC Domain

An exact combinatorial search generates and explores a massive tree of possible solutions. The search trees are generated on-demand to limit memory requirements, and provide ample opportunities to exploit parallelism. Space-splitting approaches explore subtrees in parallel, speculatively for optimisation and decision searches. Portfolio approaches run multiple searches, with varying search orders/heuristics, and share knowledge between searches.

We focus on space-splitting approaches here. As the search trees are so large (potentially exponential in the input size) we can easily generate millions of parallel tasks: far more than commodity hardware can handle, but ideal for modern HPC clusters.

However there are many aspects of search that make effective parallelisation extremely challenging in practice. Searches differ from standard parallel workloads due to their heavy use of symbolic/integer data and methods as opposed to floating point, and widespread use of conditionals meaning that neither vectorisation nor GPUs are beneficial.

Here we outline some specific challenges raised by combinatorial search at HPC scale (100+ compute nodes, 5000+ cores), and show that despite these issues *exact combinatorial search is a fruitful HPC domain.*

Task Irregularity. Parallel search tasks are highly *irregular* that is (1) some tasks have short runtimes, e.g. several milliseconds, while others take orders of magnitude longer, *e.g.* many minutes (2) tasks are generated dynamically, and the number of tasks varies depending on the search instance, e.g. as determined by the number of children of a search tree nodes. Many classical HPC workloads, *e.g.* computing over a homogeneous mesh, are regular with tasks having similar runtimes, and the number of tasks can be statically predicted.

Search tasks explore subtrees, and the sizes of these often varies by orders of magnitude. The problem is compounded as the shape of the search tree changes at runtime as new knowledge, like improved bounds, is learned. In practice improved knowledge can make a significant proportion of existing tasks redundant. We give a detailed analysis of search task irregularity in Section 5.

Speculation. Tree searches are commonly parallelised by *speculatively* exploring subtrees in parallel. Most searches are compute, rather than memory or communication bound, and to achieve substantial speedups large amounts of speculation are used.

Speculatively exploring subtrees earlier than a sequential algorithm can dramatically improve performance: some parallel task may find a solution or strong bound long before the sequential algorithm would. So superlinear speedups are not uncommon. Conversely, speculation may result in the parallel search performing far more work than the sequential search, and may lead to slowdowns as more cores are used. These superlinear speedups and slowdowns due to increased workload are known as *performance anomalies.*

Preserving Heuristics. State of the art algorithms make essential use of *search heuristics* that minimise search tree size. Parallel searches must preserve the heuristic search ordering as far as possible, so standard random work stealing isn't appropriate. Rather, scheduling is often carefully designed to preserve search heuristics [19,3].

Global Knowledge Exchange. Search tasks discover information that must be shared with other search tasks, e.g. a better bound in a branch and bound search. Sharing global state must be managed carefully at HPC scale to avoid excessive communication and synchronisation overheads. It is likely that some search algorithms that share significant amounts of data, such as clause learning SAT solvers, will struggle to scale onto HPC. However many searches only share small amounts of data globally. Moreover as the global data primarily provides opportunities to optimise (e.g. prune the search tree), there is no strong synchronisation constraint: remote search tasks are neither stalled, nor producing incorrect data prior to receiving the new knowledge.

Programming Challenges. State of the art search implementations are intricate, and it is unusual to see large scale parallelisations. Moreover few of the parallel search implementations fully utilise modern architectures, e.g. provide two level (thread + process) parallelism. Even when parallel searches share common paradigms, they are typically individually parallelised and there is little code reuse. One approach to minimise development effort is to parallelise existing sequential solvers [20]. Alternatively high level frameworks provide developers generic libraries to compose searches [1, 24, 8].

This paper focuses on YewPar, the latest such framework, and designed to scale to HPC.

2.2 YewPar

YewPar[2] is carefully designed to manage the challenges of parallel search. Lazy Node Generators produce the search trees, and skeletons are provided to efficiently search them in parallel. To support distributed memory parallelism, YewPar builds on the HPX [11] task parallelism library and runtime system. HPX is routinely deployed on HPC and Cloud systems, and YewPar can readily exploit this portability at scale. Complete descriptions of the YewPar design and implementation are available in [3, 2]; it has the following key components.

Lazy Node Generators Search trees are too large to realise in memory, and searches proceed depth-first lazily generating only the subtrees being searched. The Lazy Node Generator for a specific search application is a data structure that takes a parent search tree node and enumerates its children in traversal (*i.e.* heuristic) order. While node generators create the children of a node, *how* and *when* the search tree is constructed is determined by the skeletons.

Search Coordinations. To minimise search time it is critical to choose heuristically a *good* node to search next. We follow prior work e.g. [23], and use both application heuristics (as encoded in the Lazy Node Generator), and select large subtrees (to minimise communication and scheduling overheads) that we expect to find close to the root of the search tree. In addition to **sequential** depth first search, YewPar currently provides three parallel search coordinations. **Depth-Bounded** search converts all nodes below a cut-off depth d_{cutoff} into tasks. **Stack-Stealing** search dynamically generates work by splitting the search tree on receipt of a work-stealing request. In **Budget** search workers search subtrees until either the task completes or the task has backtracked as many times as specified in a user-defined *budget*. At that point new search tasks are spawned for the top-level nodes of the current sub-tree.

YewPar Skeletons compose a search coordination with one of the three search types, for example `BudgetDecision`, or `DepthBoundedOptimisation`. There are currently four search coordinations, and hence 12 (3×4) skeletons. The skeletons are implemented for runtime efficiency, e.g. C++ templates are used to specialise the skeletons at compile-time. The skeleton APIs expose parameters like depth *cutoff* or backtracking *budget* that control the parallel search.

Search Specific Schedulers. YewPar layers the search coordination methods as custom schedulers on top of the existing HPX scheduler. That is, the

[2] https://github.com/BlairArchibald/YewPar

HPX scheduler manages several lightweight YewPar scheduler threads that perform the search. In addition to search worker threads, each compute-node has a *manager* HPX thread that handles aspects like messages and termination. The schedulers seek to preserve search order heuristics, e.g. by using a bespoke order-preserving workpool [2,3].

Knowledge Management. The sharing of solutions and bounds relies on HPX's partitioned global address space (PGAS). To minimise distributed queries, bounds are broadcast to compute-nodes that keep track of the last received bound. The local bound does not need to be up-to-date to maintain correctness, hence YewPar can tolerate communication delays at the cost of missing pruning opportunities.

3 Ease of Use

YewPar is designed to be easy to use by combinatorial searchers who lack expertise in parallel programming. For example, although each of the three searches we describe in Section 4 uses a published state-of-the-art algorithm, they require only around 500 lines of code[3]. Most of the search application is parameterised generic code. Notable exceptions are the node generator that produces the search tree, and the bounding function (pruning predicate), that are search-specific and must be specified.

Crucially a YewPar user only composes extremely high level parallel contructs: they select and parameterise a search skeleton. For example the search specified in the first 6 lines of Listing 1 returns the maximal node in some space. The user specifies the search coordination, in this case `StackStealing`, provides the application-specific Lazy Node Generator `Gen` to generate the search tree, and chooses the type of search, here `Optimisation`. The listing also illustrates how YewPar implements pruning. The user provides an application-specific `BoundFunction` that is called on each search tree node and prunes if the bound cannot beat the current objective.

Providing such high level abstractions of the parallel search makes it easy to experiment with alternate parallel searches and search parameters. As an example, the last 8 lines of Listing 1 specify a parallel budget version of the maximal node search, where each search task has a budget of 50000 backtracks. In contrast, hand-written parallel search applications like[19,7] usually add parallelism constructs directly to the main search algorithm, obfuscating the algorithm and making it very difficult to experiment with alternate parallelisations without major refactoring.

A more complete description of how to specify parallel searches in YewPar is provided in [5], together with a docker image artefact containing 6 example search implementations.

The search coordinations provided by YewPar are significantly more advanced than those commonly used in hand-written parallel searches. For example, in Section 6 we baseline our YewPar implementation against a simple

[3] https://github.com/BlairArchibald/YewPar/apps

Listing 1: Two YewPar Parallel Search Skeletons

```
Node maximal_solution_ws =
  YewPar::Skeletons::StackStealing< // search coord
    Gen,                            // lazy node gen
    Optimisation,                   // search type
    BoundFunction<upperBound>       // bound for pruning
  >::search(space, root);


Params params;
searchParameters.backtrackBudget = 50000;
Node maximal_solution_budget =
  YewPar::Skeletons::Budget <       // search coord
    Gen,                            // lazy node gen
    Optimisation,                   // search type
    BoundFunction<upperBound>       // bound for pruning
  >::search(space, root, params);
```

OpenMP version that adds pragmas to the main search loop. The programming effort required to produce the OpenMP and YewPar versions is very similar, but the OpenMP implementation is limited to shared memory, and to a simple depth 1 bounded search. Implementing more complex search coordinations, e.g. a heuristic-preserving depth 2 bounded search is far more intricate. Search coordinations such as StackStealing require access to the scheduler. Moreover conventional parallel schedulers often disrupt search heuristics [19], and we show how the OpenMP scheduler disrupts the baselining search instances in Section 6. This highlights the need for custom schedulers as in YewPar.

While the vast majority of parallel searches are handwritten, there are some generic frameworks like TASKWORK [12], and Muesli [24]. These provide a similar level of programming abstraction to YewPar, but are designed for branch and bound optimisation and, unlike YewPar, do not currently support decision or enumeration searches and tend to only support one search coordination.

4 Generic Exact Combinatorial Search

Due to the challenges of engineering performant parallel implementations of exact combinatorial search, only a small number of specific exact combinatorial searches have been hand-crafted for HPC scale, e.g. [9,4]. YewPar is designed to minimise the effort required to engineer performant searches by providing a library of re-usable skeletons and search coordinations. While this genericity has previously been demonstrated by parallelising seven searches at cluster scale (100s of cores) [5], it has never been demonstrated at HPC scale (1000s of cores).

We further demonstrate the ease of construction (Section 3) by exhibiting parallel searches covering the three search types.

Numerical Semigroups The first application comes from group theory, and tackles the problem of counting the number of numerical semigroups of a particular genus, which is useful for areas such as algebraic geometry[6]. For mathematical searches such as this, *exactness* is essential: an approximate answer has no value.

A numerical semigroup can be defined as “Let $\mathbb{N}_0$ be the set of non-negative integers. A numerical semigroup is a subset Λ of $\mathbb{N}_0$ which contains 0, is closed under addition and has finite complement, $N_0 \setminus \Lambda$. The elements in $N_0 \setminus \Lambda$ are the gaps of Λ, and the number $g = g(\Lambda)$ of gaps is the genus of Λ” [7]. A numerical semigroup can be viewed as taking a finite set of non-negative integers X such that $\mathbb{N}_0 \setminus X$ remains closed under addition.

A numerical semigroups search enumerates the number of semigroups with genus g, e.g. of genus 46. The YewPar node generator uses Hivert’s algorithm [7] and exploits the relation between subsequent numerical semigroups to generate search tree nodes. So the search counts the number of search tree nodes at the specified depth, e.g. at depth 46.

Maximum Clique The maximum clique optimisation problem seeks to find the largest clique in graph G. A clique $C \subseteq V$ such that $\forall u, v \in C(\{u, v\} \in E)$. The node generator implementation is based on the MCSa1 algorithm [25] that exploits graph colouring for bounding/heuristic ordering.

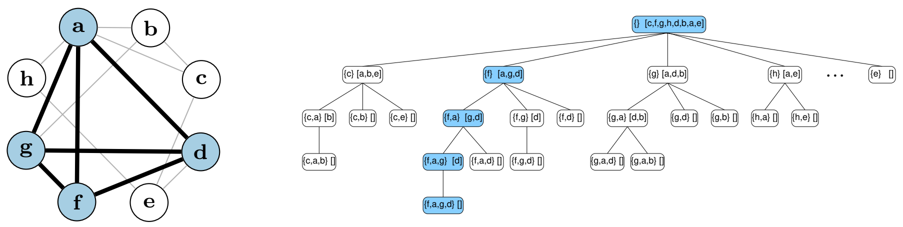


Fig. 4.1: A maximum clique instance. Input graph with clique $\{a, d, f, g\}$ to the left and corresponding search tree to the right. Each tree node displays the current clique and a list of candidate vertices (in heuristic order) to extend that clique.

Maximum clique arises in areas such as computational biology, information retrieval, economics and signal transmission theory [22]. Search instances are drawn from the standard DIMACS challenge instances [10].

k-clique with Finite Geometry. Some applications require a *specifically sized* clique. The k-clique decision search determines whether there is a clique C in a graph G of size k, i.e. $|C| = k$. We apply k-clique to a problem in finite geometry - determining if a *spread* in geometries of the Hermitian variety $H(4, q^2)$ exists[13]. A *spread* is a set of lines L such that every point is incident with exactly one element of L. For $H(4, q^2)$, a spread (if it exists) will have size $q^5 + 1$. Intuitively, a spread forms a partition of the points.

Symmetries in the state space up to depth 3 are broken by pre-processing with GAP [26]. So the clique being searched for is of size q^5+1-3. We consider geometries of the form $H(4, 3^2)$, so $k = 3^5 + 1 - 3 = 241$, and the Maximum Clique node generator is used to generate the search tree.

5 Profiling Irregularity in Exact Combinatorial Search

Parallel exact combinatorial search produces extremely irregular parallelism (Section 2.1). Although the irregularity of a small number of specific parallel searches has previously been investigated, *e.g.* [21, Fig. 4], detailed analysis of irregularity is uncommon, and the extent of the irregularity in most searches is unknown.

To provide detailed information on search task irregularity we have added a small data store on each YewPar compute-node that records key aspects of the parallel search: task runtimes, number of backtracks, number of search tree nodes visited, and total number of tasks spawned. To allow the *shape* of the tree to be investigated, the data is indexed by the tree depth where the task was *spawned*.

Experimental Setup. Measurements are made on the following platforms. YewPar and OpenMP are compiled with gcc 8.2.0 and HPX 1.2.1. **Cirrus** is an HPC cluster comprising 228 compute-nodes, each having twin 18-core Intel Xeon (Broadwell) CPUs (2.1Ghz), 256 GB of RAM and running Red Hat Enterprise Linux Version 8.1. The **GPG Beowulf Cluster** comprises 17 compute-nodes, each having dual 8-core Intel Xeon E5-2640v2 CPUs (2Ghz), 64GB of RAM and running Ubuntu 18.04.2 LTS.

Search Task Runtimes (STR) records the distribution of runtimes for tasks spawned at each depth in the search tree. Much of the variance arises from the structure of the search tree. STR allows us to visualise the distribution of task runtimes throughout a search and provides information about the shape of the search tree for a given search instance.

We illustrate the range and distribution of search task runtimes using a violin plot for the tasks spawned at each search tree depth, excluding the time for spawned tasks to complete. The shape of each violin plot represents the distribution of runtimes, e.g. wide sections correspond to frequent runtimes. The white cross represents the median value, and the black rectangle the interquartile range.

As a basis for comparison we record the STR for a relatively *regular* parallel tree search. This synthetic search enumerates the tree nodes down to depth 30 in a balanced binary tree (so all subtrees are the same size), creating tasks down to depth 8. Figure 5.1 shows the search task runtimes at different depths. Task runtimes at depths 0 to 7 are uniformly small. The tasks at depth 8 do most of the enumeration, and their runtimes have a compact distribution with median 16.4ms and an interquartile range of 16.3ms to 16.6ms.

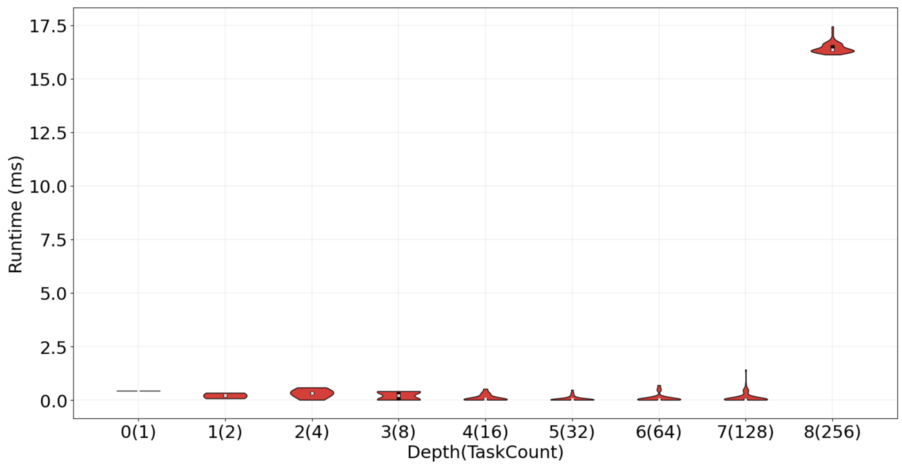


Fig. 5.1: Regular Search Task Runtimes: task runtime distributions for a balanced binary tree search to depth 30 using the Depthbounded skeleton, $d_{cutoff} = 8$. (1 GPG Cluster Compute-Node)

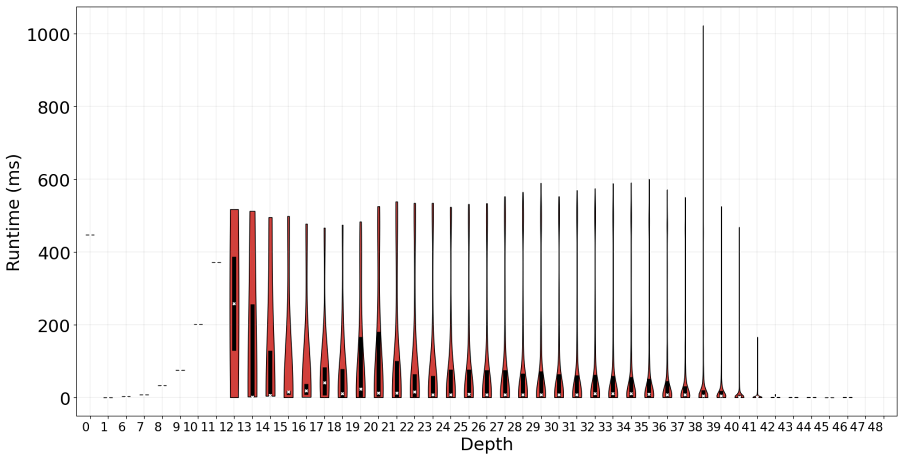


Fig. 5.2: Search task runtime distributions for a Numerical Semigroups genus 48 search, Budget skeleton with $b = 10^7$ backtracks. Depths 2-5 spawn no tasks and are omitted (1 GPG Cluster Compute-Node)

In contrast, Fig. 5.2, Fig. 5.3 and Fig. 5.4 show that the STR distributions for typical combinatorial searches are very different. Figure 5.2 is for a Numerical Semigroups genus 48 search, using the Budget skeleton with a budget of $b = 10^7$ backtracks. The distribution of task runtimes is plotted for each depth in the search tree down to depth 48. Although task sizes increase steadily at depths 1-7, there are few tasks and little variance. This reflects a known result that the

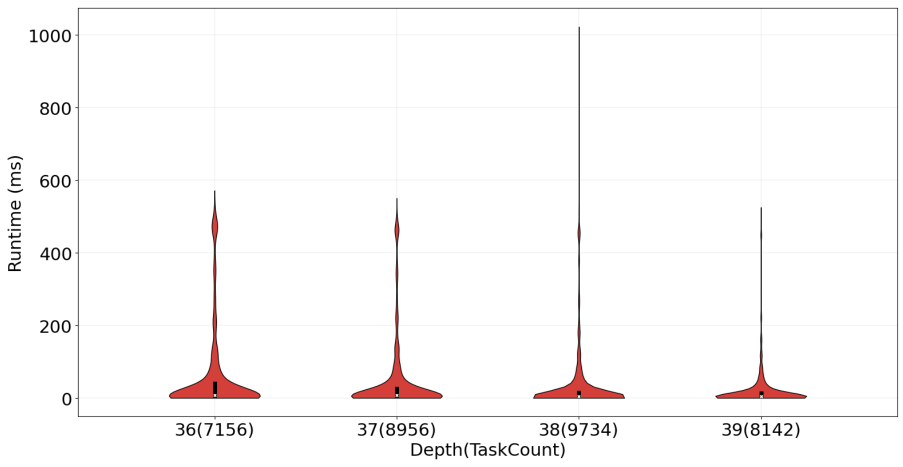


Fig. 5.3: Search task runtime distributions for depths 36 to 39 of a Numerical Semigroups genus 48 search, Budget skeleton with $b = 10^7$ backtracks (1 GPG Cluster Compute-Node)

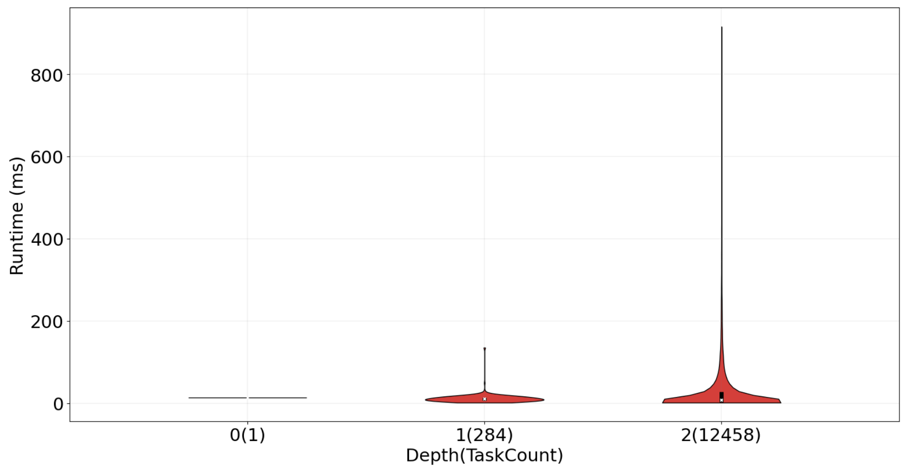


Fig. 5.4: Search Task Runtime Distributions for a Brock400_1 Maximum Clique search, depth 2 Depthbounded skeleton (1 GPG Cluster Compute-Node)

Numerical Semigroups search tree is narrow at low depths [7]. Between depths 12 and 41 there is massive variability in search task runtimes. For example at depth 16 the median task runtime is 19ms, while the interquartile range is 31.5ms, and the maximum task runtime is 499ms.

Figure 5.3 provides more details of the Numerical Semigroups search task runtime distributions at depths 36..39. Not only is the massive variability in runtimes clear, but it is far more apparent that the distributions at these levels,

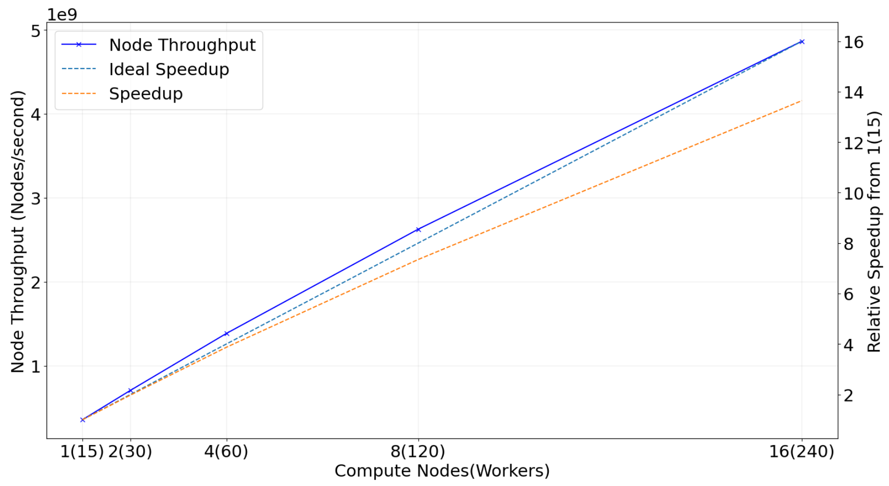


Fig. 5.5: Node throughput and relative speedup from 1(15) GPG Cluster Compute Nodes(workers) for a Numerical Semigroups genus 49 search, Budget skeleton with $b = 10^7$ backtracks

as at other levels, are multi-modal. For example the distributions at levels 36 and 37 both have four clear modes.

Figure 5.4 shows the search task runtime distributions for a Maximum Clique search instance using the Depthbounded skeleton at depth 2 ($d_{cutoff} = 2$). This optimisation search instance is brock400_1 from the DIMACS benchmark suite [10]. Tasks are spawned at only three depths and the vast majority of search tasks, 12458 out of 12753, are generated at depth 2. Depths 1 and 2 both exhibit massive variability in search task runtimes. Most tasks are have short runtimes (less than 70ms), but a small number have much longer runtimes (over 900ms). At depth 2 the median task runtime is 8ms, while the interquartile range is 26ms, and the maximum task runtime is 916ms.

Profiles like Figure 5.2 and Figure 5.4 are a valuable tool as they enable developers to accurately quantify and visualise search task runtimes, a key aspect for tuning parallel performance.

Search Tree Node Throughput records the number of nodes visited by some search task per unit time. It is commonly used as a measure of search speed and, indirectly, the size of the workload[15]. As the number of cores grows, increasing node throughput illuminates how parallelism may reduce search runtime.

YewPar has been extended to record node throughput by counting each node visited during the search using a depth-indexed vector of atomic counters in the profiling data store. To minimise the number of atomic operations each search worker maintains a local counter and only updates the atomic counter in the vector on termination.

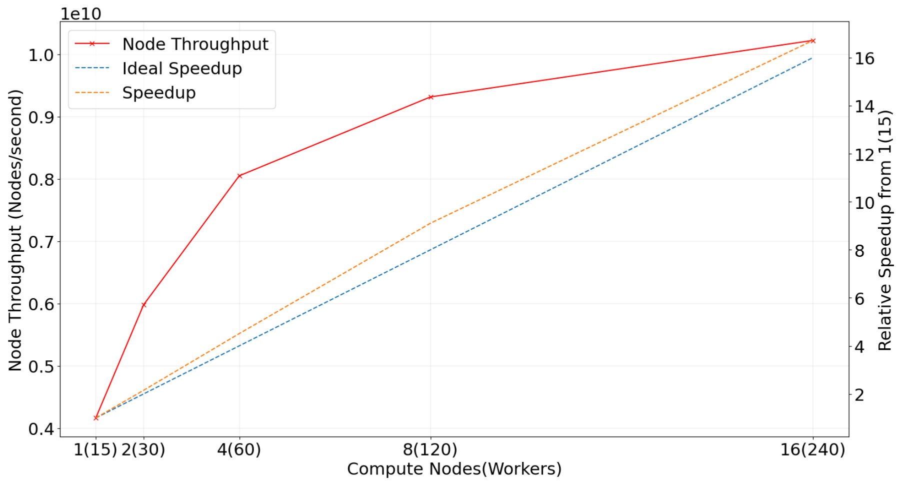


Fig. 5.6: Node throughput and relative speedup from 1(15) GPG Cluster Compute Nodes(workers) for a brock800_2 Maximum Clique search, depth 2 Depthbounded skeleton

Figure 5.5 shows the node throughput and relative speedup for a Numerical Semigroups genus 48 search. This enumeration search again uses the Budget skeleton with a budget b of 10^7 backtracks, and is measured on between 1 and 16 compute nodes of the GPG cluster. This graph, and the other graphs in this section, report throughput as the mean number of nodes visited divided by the median runtime over 5 executions.

Node throughput increases linearly as the number of compute nodes and cores increases, and is closely correlated with the speedup. This is as expected for an enumeration search that has a fixed workload, i.e. must visit exactly the same number of search tree nodes in every execution. So for enumeration searches a higher node throughput directly correlates with speedup.

Figure 5.6 shows the node throughput and speedup for a Maximum Clique search instance using the Depthbounded skeleton at depth 2 ($d_{cutoff} = 2$). This optimisation search instance is DIMACS brock800_2 and is measured on 1 to 16 compute nodes of the GPG cluster.

Both node throughput and speedup increase superlinearly as the number of compute nodes and cores increases, but are not strongly correlated. Speculative search tasks account for the increase in node throughput as the number of cores increases. We believe that the rate of increase of node throughput falls at high core counts because the speculative threads prune much of the search tree. It is, however, not easy to measure how much of the tree is pruned as the pruned subtrees are never generated. The reduced node throughput is again as expected for an optimisation search where pruning reduces the workload.

Search Instance	Sequential C++	Sequential YewPar	Slowdown %	OpenMP (1 Worker) C++	OpenMP (18 Workers) C++	Depth-Bounded YewPar	Slowdown %
brock400_1	252.60	301.08	19.19	295.63	18.89	23.77	25.89
brock400_2	183.38	218.32	19.05	217.96	7.80	10.04	28.74
brock400_3	145.44	174.15	19.74	230.71	4.26	5.66	32.86
brock800_1	3790.72	3868.66	2.06	5914.71	195.48	240.86	23.22
brock800_2	3808.82	3764.54	-1.16	6820.68	232.26	256.52	10.44
brock800_3	3512.92	3511.75	-0.03	4697.50	183.88	207.45	12.82
brock800_4	1316.16	1282.45	-2.56	3560.01	86.87	102.15	17.59
C250.9	1725.04	2112.06	22.44	1756.48	115.72	179.57	55.18
p_hat700-3	1139.00	1278.38	12.24	1020.68	77.52	115.49	48.98
Geo. Mean			9.66				27.63

Table 6.1: Comparing YewPar runtimes (s) and slowdown (%) with Handwritten Maximum Clique Implementations: Sequential and Depth-bounded OpenMP Implementations (Cirrus)

6 Exact Combinatorial Search at HPC Scale

Measuring Parallel Search is Challenging primarily due to the non-determinism caused by pruning, random work-stealing, and finding alternate valid solutions. These can lead to performance anomalies (Section 2.1) that manifest as dramatic slowdowns or superlinear speedups. We control for this by investigating multiple instances of multiple search applications and selecting the median of 5 executions. The experimental setup is as in Section 5.

Sequential and Single Compute-Node Baselines. YewPar's generality incurs some overheads compared to search specific implementations as it decouples search tree generation and traversal. For example, Lazy Node Generators copy search tree nodes (in case they are stolen) instead of updating in-place. We evaluate these overheads on Maximum Clique as a competitive sequential implementation is available [18,17].

Sequential Baseline. The first 4 columns of Table 6.1 show the mean sequential runtimes (over 5 executions) of the 9 DIMACS clique instances [10] that take between 100 seconds and 1 hour to run sequentially on Cirrus. The results show a limited cost of generality, i.e. a maximum slowdown of 22.44%, a minimum slowdown of -2.56%, and geometric mean slowdown of 9.7%. We attribute the small runtime reductions compared with C++ for 2 search instances to optimisations arising from different C++ and C++/HPX compilation schemes.

Parallel Baseline. Parallel execution adds additional overheads, e.g. the YewPar skeletons are parametric rather than specialised, and the distributed memory execution framework is relatively heavyweight on a single compute node. To evaluate the scale of these overheads we compare with a search-specific OpenMP version of the maximum clique implementation. It is imperative that the parallel search algorithm and coordination are almost identical, as otherwise performance anomalies will disrupt the comparison. Hence the Lazy Node Generator is carefully crafted to mimic the Maximum Clique implementation [18], and the OpenMP implementation uses the `task` pragma to construct a set of tasks for each node at depth 1, closely analogous to the DepthBounded skeleton in the YewPar implementation.

There are significant slowdowns for OpenMP using a single worker as the OpenMP scheduler does not preserve the search heuristic, and this is confirmed by printing the task schedule. That is OpenMP provides no guarantee that the search tasks are executed in the order they are spawned, and this illustrates a common issue when using off-the-shelf parallelism frameworks for search [19]. The effect is smaller in the parallel version as the likelihood that at least 1 worker follows the heuristic increases.

Columns 5-7 of Table 6.1 compare the runtimes of the YewPar and OpenMP versions for the DIMACS search instances with 18 search workers on a single Cirrus compute node. We measure the searches on 18 workers/cores rather than on all 36 physical cores available on a Cirrus compute node as experimentation reveals that OpenMP performance reduces above 18 cores. We attribute this to starvation as the depth 1 spawning creates too few tasks to utilise all of the cores. A depth-2 backtracking search would generate far more work, but implementing such a search in OpenMP that is correct, and exactly emulates the YewPar DepthBounded search, is far from trivial especially when trying to maintain search heuristics. The geometric mean slowdown increases to 27.6%, with a maximum slowdown of 55%.

The sequential and single compute node overheads of YewPar are lower on the GPG Cluster. For the same Maximum Clique codes on a slightly larger set of DIMACS instances the mean sequential slowdown is 8.7%, and the slowdown on a single 16-core compute node is 16.6% [5].

We conclude that for these search instances the parallel overheads of YewPar remain moderate, while facilitating the execution of multiple search applications on multiple platforms: multicores, clusters, or HPC systems.

Scaling. As exact combinatorial search problems are NP hard the workloads generated by instances vary greatly, and hence it is not possible to double problem size to measure weak scaling. Hence we report strong scaling, and speedups are relative to execution on a small number of Cirrus compute nodes.

We measure the scaling of searches covering the three search types. Profiling informs the selection of search skeleton and its parameterisation. For example Fig. 5.2 reveals that the depth bound must be at least 12 for the Numerical Semigroups search as there are so few tasks at lower depths. Similarly Fig. 5.4 reveals that a depth bound of 2 for a smaller Maximum Clique instance generates 12K tasks, so we predict that this bound will be ample for 2000 search workers on the larger instance.

The parallel searches we evaluate are outlined in Section 4, and the instances measured are as follows. Numerical Semigroups is an Enumeration search at genus 61, and uses the YewPar Budget skeleton with a budget of 10^7 backtracks. Maximum Clique is an Optimisation search for the DIMACS p_hat1000-3 instance, and uses the Depthbounded skeleton with a depth cutoff of 2 for all measurements other than on 4480 workers where we use a cutoff of 3 to minimise starvation. K-clique is the finite geometry Decision search with k=241 and uses the Depthbounded skeleton with a depth cutoff of 3. This relatively

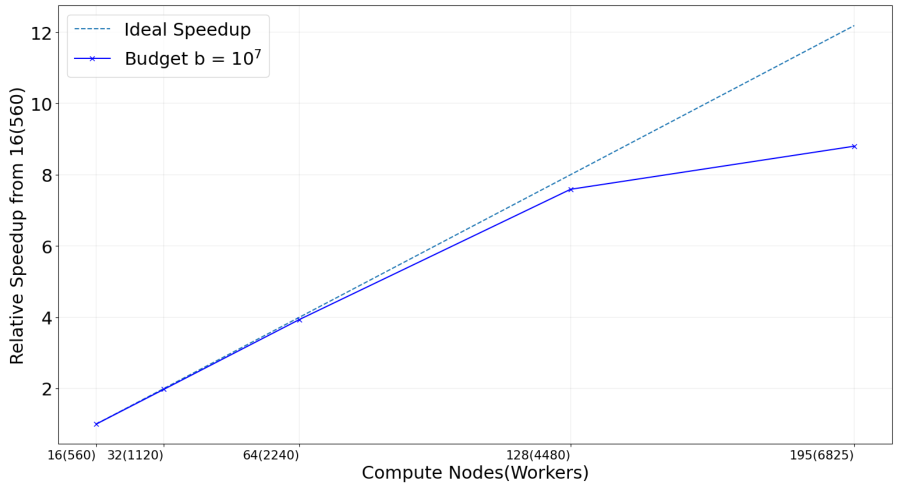


Fig. 6.1: Numerical Semigroups genus 61 search, budget 10^7 backtracks; Speedup relative to 16(560) Cirrus Compute Nodes(workers)

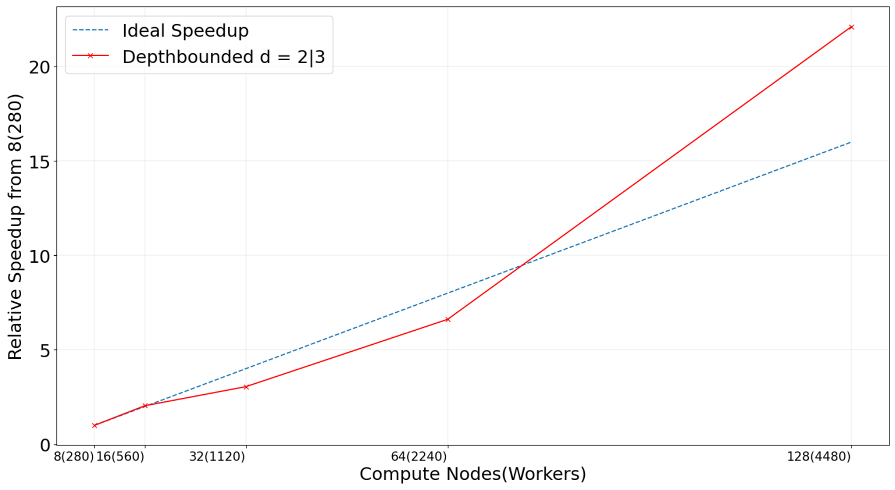


Fig. 6.2: Maximum Clique p_hat1000-3 Search Speedup Relative to 8(280) Cirrus Compute Nodes(workers); Depthbounded with cutoff 2, increased to 3 for 16(4480) to minimise starvation.

high cutoff is selected to generate many search tasks, as searching for a specific clique size induces huge amounts of pruning. While successful decision searches terminate early, here we measure unsuccessful searches that must explore the entire space.

Runtimes Numerical Semigroups runtimes fall from 2649s on 16(560) Cirrus Compute Nodes(workers) to 302s on 195(6825) Compute Nodes(workers).

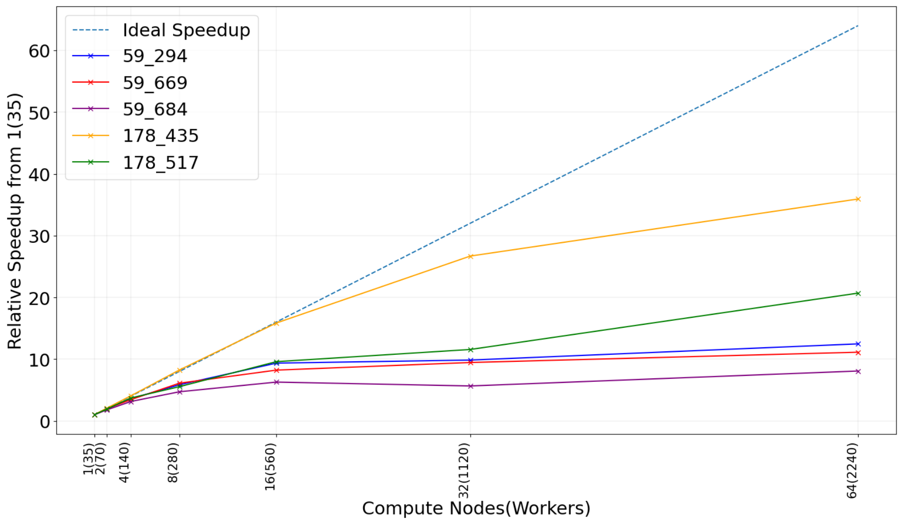


Fig. 6.3: k-clique Finite Geometry Search Speedups Relative to 1(35) Cirrus Compute Node(workers); Depthbounded with cutoff 3

Recall that each YewPar worker is associated with a core. Maximum Clique runtimes fall from 2255s on 8(280) to 102s on 128(4480) compute nodes(workers). For the 178_435 k-clique search, the most significant runtime decrease is from 4563s on 1(35) to 127s on 64(2240) compute nodes(workers). The other k-clique searches have low runtimes: 178_517 has the greatest runtime: 393s on 1(35) and this reduces to 19s on 64(2240) compute nodes(workers). A complete set of runtime and speedup data is available [14].

The results demonstrate that *deploying YewPar on an HPC can dramatically reduce the runtime of different types of combinatorial search compared with state of the art sequential and parallel implementations.* As a further example, the runtimes for a Maximum Clique p_hat1000-3 have fallen from 130.8 hours (Sequential), 4.2 hours (Cilk+) and 3.0 hours (C++ custom threading) on a dual 32-core Intel Xeon E5-2697A (2.6Ghz) [16] to 102s (4480 YewPar workers on Cirrus). As the Xeon has a faster clock speed than both GPG and Cirrus we would expect even longer sequential runtimes on these platforms.

Speedups Figure 6.1 shows the speedups for the Numerical Semigroups genus 61 search. The speedups are relative to execution on 16(560) compute nodes(workers). The relative speedup is near linear up to 4480 workers, with parallel efficiency over 90%. By 6825 workers both speedup and efficiency have declined.

Figure 6.2 shows relative speedups from 8(280) compute nodes (workers) for the Maximum Clique p_hat1000-3 search. Relative speedups increase steadily as the core counts increase up until 4608 cores where super-linear speedups are achieved. Super-linear speedups are common for optimisation searches where pruning can dramatically reduce the workload.

Figure 6.3 shows relative speedups from 1(35) compute nodes (workers) for 5 instances of the k-clique decision search. Speedups vary from instance to instance. The best speedup achieved is for instance 178_435 that has a significantly longer runtime at 1(35) i.e. 4563s. Lower speedups are achieved for instances with lower runtimes on 1(35), and these range from 393s for 178_517 to 113s for 59_684. For these instances runtimes are reduced to between 10s and 20s on 64(2240) compute nodes (workers), and the system is starved of work. So it is likely that far better scaling could be achieved for larger search instances.

7 Conclusions

We report the first ever study of generic combinatorial search at HPC scale, i.e. 100+ compute nodes and 4000+ cores. The study demonstrates the capacity of the YewPar search framework to scale to HPC.

We have demonstrated *generic* high performance combinatorial search, i.e. that a variety of exact combinatorial searches can be easily parallelised for HPC using YewPar. Complete implementations of sophisticated state-of-the art parallel searches require only around 500 lines of code. Previously (1) just a few searches have been individually hand-crafted for HPC scale e.g. [9,4]; and (2) the genericity of YewPar has only been demonstrated on a modest cluster (100s of cores) by parallelising seven searches [5]. Here we exhibit HPC-scale searches using different YewPar skeletons and covering the three search types: optimisation, enumeration, and decision (Section 4).

We have presented **a new mechanism for profiling key aspects of generic parallel combinatorial search in YewPar.** The extreme irregularity of parallel combinatorial search has only rarely been measured, and then only for specific search applications, e.g. [21]. We exhibit profiles that quantify the irregularity of many search applications in the generic YewPar framework. Although implemented for YewPar and in HPX the profiling techniques do not depend directly on either: these are generic techniques and measures valuable for *any* parallel combinatorial search framework. Search task runtime profiling aids parallelisation by providing information on aspects like the huge differences in search task runtimes, mean task runtime, and the radically different (and frequently multi-modal) task runtime distributions at each search tree depth, e.g. Fig. 5.2. Profiling node throughput quantifies the dramatic differences in parallel behaviour between enumeration searches with fixed workloads, e.g. Fig. 5.5, and optimisation searches with variable workloads, e.g. Fig. 5.6 (Section 5).

We demonstrate, for the first time, generic exact combinatorial searches at HPC scale. Baselining against state-of-the-art sequential C++ and C++/OpenMP implementations on 9 standard (DIMACS) search instances shows that the generality of YewPar incurs a mean sequential slowdown of 9%, and a mean parallel slowdown of 27.6% on a single 18-core compute node (Table 6.1). Guided by the profiling we effectively parallelise seven standard instances of the three searches, and systematically measure runtime and relative speedups at scale. We show how *deploying YewPar on an HPC can deliver dramatic reductions in runtime compared with state of the art hand crafted search implementations, both sequential and parallelised at smaller scale.* For example reducing the p_hat1000-3 sequential search from 131 hours to just 102 seconds using YewPar on a 4480 cores (Fig. 6.2).

Comparing different search types shows similar speedup and scaling characteristics to smaller-scale parallel search [5], e.g. pruning in the Maximum Clique optimisation search reduces workload and hence delivers super-linear speedups up to 128(4480) compute-nodes(workers) (Fig. 6.2). The maximum relative speedups we achieve for the Numerical Semigroups enumeration search are near-linear up to 192(6825) compute-nodes(workers) (Fig. 6.1), and sub-linear for five k-clique decision searches on up to 64(2240) compute-nodes(workers) (Fig. 6.3). It is likely that far better scaling can be achieved for k-clique, and other decision searches, if suitable instances can be found (Section 6).

Ongoing Work. Currently determining good parameters for a search instance, like depth cutoff or backtrack budget, entails a parameter sweep. Ongoing work seeks to determine whether we can use pre-execution profiling to predict parameters, e.g. is backtracks-per-second-per-worker sufficient to determine an appropriate budget for search instances? We would also like to explore whether performance can be improved by extending YewPar to use the profiling metrics to dynamically adapt the search, e.g. a compute node with ample work may increase the depth cutoff to provide more search tasks.

Acknowledgements This work was supported EPSRC grants MaRIONet (EP/P006434), STARDUST (EP/T014628) and S4: Science of Sensor Systems Software (EP/N007565).

References

1. Alba, E., et al.: MALLBA: A Library of Skeletons for Combinatorial Optimisation. In: Euro-Par, Paderborn, Germany, August, 2002, Proceedings (2002)
2. Archibald, B.: Skeletons for Exact Combinatorial Search at Scale. Ph.D. thesis, University of Glasgow (2018), http://theses.gla.ac.uk/id/eprint/31000
3. Archibald, B., Maier, P., Stewart, R., Trinder, P.: Implementing YewPar: A Framework for Parallel Tree Search. Euro-Par, Gottingen, Germany (2019)
4. Archibald, B., et al.: Sequential and parallel solution-biased search for subgraph algorithms. In: CPAIOR 16th Thessaloniki, Greece, June, 2019 (2019)
5. Archibald, B., et al.: YewPar: skeletons for exact combinatorial search. In: PPoPP '20:, San Diego, California, USA, February, 2020. ACM (2020)
6. Barucci, V., et al.: Maximality properties in numerical semigroups and applications to one-dimensional analytically irreducible local domains, vol. 598. American Mathematical Soc. (1997)
7. Fromentin, J., Hivert, F.: Exploring the tree of numerical semigroups. Math. Comp. **85**(301) (2016)
8. Galea, F., Le Cun, B.: Bob++: a framework for exact combinatorial optimization methods on parallel machines. In: International Conference High Performance Computing & Simulation (HPCS) (2007)
9. Heule, M.J.H., Kullmann, O.: The science of brute force. Commun. ACM **60**(8) (2017)

10. Johnson, D.J., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October, 1993. American Mathematical Society (1996)
11. Kaiser, H., et al.: HPX: A Task Based Programming Model in a Global Address Space. In: ICPGASPM 2014, Eugene, OR, USA, October, 2014 (2014)
12. Kehrer, S., Blochinger, W.: Development and operation of elastic parallel tree search applications using TASKWORK. In: Ferguson, D., Muñoz, V.M., Pahl, C., Helfert, M. (eds.) Cloud Computing and Services Science - 9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2-4, 2019, Revised Selected Papers. Communications in Computer and Information Science, vol. 1218, pp. 42–65. Springer (2019). https://doi.org/10.1007/978-3-030-49432-2_3, https://doi.org/10.1007/978-3-030-49432-2_3
13. Klein, A., Storme, L.: Applications of finite geometry in coding theory and cryptography. Information Security, Coding Theory and Related Combinatorics **29**, 38–58 (2011)
14. MacGregor, R.: Generic High Performance Exact Combinatorial Search [Data Repository]. https://doi.org/10.5281/zenodo.4270336
15. Maher, S.J., Ralphs, T.K., Shinano, Y.: Assessing the effectiveness of (parallel) branch-and-bound algorithms. arXiv preprint arXiv:2104.10025 (2021)
16. McCreesh, C.: Solving hard subgraph problems in parallel. Ph.D. thesis, University of Glasgow (2017)
17. McCreesh, C.: Sequential MCsa1 Maximum Clique Implementation (2018), https://github.com/ciaranm/sicsa-multicore-challenge-iii/c++/
18. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms **6**(4) (2013)
19. McCreesh, C., Prosser, P.: The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound. TOPC **2**(1) (2015)
20. Menouer, T., et al.: Mixing Static and Dynamic Partitioning to Parallelize a Constraint Programming Solver. International Journal of Parallel Programming **44**(3) (2016)
21. Otten, L., Dechter, R.: AND/OR branch-and-bound on a computational grid. J. Artif. Intell. Res. **59** (2017)
22. Pardalos, P., Xue, J.: The maximum clique problem. Journal of Global Optimization **4** (04 1994)
23. Pietracaprina, A., et al.: Space-efficient parallel algorithms for combinatorial search problems. J. Parallel Distrib. Comput. **76** (2015)
24. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch & bound. In: ICSOFT, Setúbal, Portugal, September, 2006 (2006)
25. Prosser, P.: Exact algorithms for maximum clique: A computational study. Algorithms **5**(4) (2012)
26. The GAP Group: GAP - Groups, Algorithms, and Programming, Version 4.8.7 (2017), https://www.gap-system.org/Releases/4.8.7.html

Noname manuscript No.
(will be inserted by the editor)

Interruptible Nodes: Reducing Queueing Costs in Irregular Streaming Dataflow Applications on Wide-SIMD Architectures

Stephen Timcheck · **Jeremy Buhler**

Received: date / Accepted: date

Abstract Streaming dataflow applications are an attractive target to parallelize on wide-SIMD processors such as GPUs. These applications can be expressed as a pipeline of compute nodes connected by edges, which feed outputs from one node to the next. Streaming applications often exhibit irregular dataflow, where the amount of output produced for one input is unknown *a priori*. Inserting finite queues between pipeline nodes can ameliorate the impact of irregularity and improve SIMD lane occupancy. The sizing of these queues is driven by both performance and safety considerations – relative queue sizes should be chosen to reduce runtime overhead and maximize throughput, but each node's output queue must be large enough to accommodate the maximum number of outputs produced by one SIMD vector of inputs to the node. When safety and performance considerations conflict, the application may incur excessive memory usage and runtime overhead. In this work, we identify properties of applications that lead to such undesirable behaviors, with examples from applications implemented in our MERCATOR framework for irregular streaming on GPUs. To address these issues, we propose extensions to support *interruptible nodes* that can be suspended mid-execution if their output queues fill. We illustrate the impacts of adding interruptible nodes to the MERCATOR framework on representative irregular streaming applications from the domains of branching search and bioinformatics.

Keywords Irregular · Dataflow · Streaming · Queueing · Interrupt · SIMD

This work was supported by National Science Foundation award CNS-1763503.

Stephen Timcheck
Washington University in St. Louis, St. Louis, Missouri, USA
E-mail: stimcheck@wustl.edu

Jeremy Buhler
Washington University in St. Louis, St. Louis, Missouri, USA
E-mail: jbuhler@wustl.edu

1 Introduction

Streaming computations arise in numerous domains, including bioinformatics [1], astrophysics [13], data integration [2], network packet inspection [10], branch-and-bound search [6], and decision cascades in machine learning [14]. Applications of this type process a long stream of independent data items, which makes them amenable to running on wide-SIMD devices such as GPUs. Streaming applications can be expressed as a pipeline of *compute nodes* connected by *edges*, where incoming data is processed by a node and then sent to the next node via a connecting edge. High-level programming support for streaming is important to manage node execution and data storage on edges transparently to the application designer; in this work, such support is provided by MERCATOR [3,12], a framework we previously constructed to support streaming applications on GPUs.

Many streaming applications of interest — including those mentioned above — exhibit *irregular dataflow*. In an irregular application, the number of output data items produced per input to a node is not fixed *a priori* but rather varies dynamically and unpredictably for each data item. Irregular data flow means that data must be queued on edges between nodes, both for parallelism (i.e., accumulating a full SIMD vector of inputs to the next node) and for safety if a node generates more output than the next node can consume at once. The sizes of inter-node queues should be carefully chosen based on considerations of average and worst-case node behavior [12], and scheduling of an application's nodes must then be cognizant of inter-node queue occupancy to ensure safe and efficient execution [9].

A basic property of streaming pipelines in MERCATOR is that execution of a compute node is *uninterruptible*: once the node begins to consume a SIMD vector of inputs, it must finish before another node can be scheduled to execute. This behavior arises because existing GPU runtimes do not support preemptive scheduling of compute kernels or of functions within one kernel. As a result, for each node, there is a minimum safe size for its output queue, namely the space needed to hold the most output that could be generated by one vector of input. This safety constraint must override queue sizing decisions driven by average-case performance considerations, which can result in applications that allocate much more queue space than they typically use and that may be inefficiently scheduled.

In this work, we first identify performance and memory usage problems that appear in irregular streaming applications due to safety constraints arising from uninterruptible nodes. We then describe modifications to our MERCATOR framework that enable application programmers to cooperatively support suspension and resumption of a node, which requires saving and restoring its execution state. Finally, we benchmark some representative applications to evaluate the impact of interruptibility on application performance and memory usage.

The rest of the paper is divided as follows. Section 2 examines related work, while Section 3 describes our application model. Section 4 looks at examples of irregular streaming applications and the impact minimum queue size restrictions have on their performance and memory usage. Section 5 describes MERCATOR's new interruptible node facility and the challenges of implementing interruptible nodes. Section 6 empirically evaluates interruptible nodes on two applications, NQueens and BLAST. Finally, section 7 concludes and considers future work.

2 Related Work

Our own prior work on MERCATOR includes the design of its node scheduler [9] and techniques for reducing overhead caused by switching from one pipeline node to another during execution [12]. This work focuses on a mechanism to allow a node to suspend and later resume execution and shows how, in the context of a wide-SIMD execution model for streaming pipelines, such a mechanism can have important benefits for throughput and for memory usage.

Prior work in scheduling multiple tasks on the GPU includes work on cooperative CPU-GPU scheduling. Hyoseung et al. [5] considered a single GPU shared between multiple non-preemptive tasks that must be scheduled sequentially. The CPU determines which GPU task to run at any given time. Kato et al.'s TimeGraph system [4] similarly includes a CPU-side scheduling mechanism for GPU tasks, each of which may have multiple components, and can make scheduling decisions based on task priority. MERCATOR also manages multiple non-preemptively executing tasks, in the form of different compute nodes in a pipeline, but the nodes along with their scheduler are all functions within one GPU kernel. Hence, we cannot use the facilities that might be used by CPU-side schedulers, such as multithreading or timer interrupts. Moreover, nodes communicate through the pipeline edges between them, which raises a different set of scheduling considerations than for independent tasks.

Other work has investigated preemptive scheduling of multiple kernels on a shared GPU, which would be desirable for, e.g., GPU virtualization and would also help ameliorate the problems we identify with non-preemptive node execution in MERCATOR. One such system, Chimera [8], assumes the existence of hardware support for kernel preemption (which was simulated using GPGPU-Sim) and focuses on how to lower the throughput and latency impacts of context switching between kernels. While MERCATOR does not consider applications with strong latency constraints, our prior work also focuses on reducing throughput impacts, specifically the frequency of required inter-node switches (which, unlike in the case of independent tasks, are unavoidable for nodes in a streaming pipeline with finite queues). The present work furthers the goal of switching reduction by using node interruptibility to enable optimizations that further reduce switches and so improve throughput.

Much like Chimera, FLEP [15] seeks to enable kernel preemption on the GPU, with a focus on speeding up high-priority kernels as well as fairly distributing time between kernels. The authors design preemption both for the entire GPU and for specific processors on the GPU. Unlike Chimera, FLEP's preemption does not assume hardware support but rather is achieved partly via compiler-side transformations on kernel code, wrapping the kernel's interior with conditions to exit on setting a variable available to the CPU. FLEP further uses timing information gathered from applications to estimate preemption overhead and guide decisions on when to preempt a kernel. While MERCATOR's scheduling decisions are not motivated by priorities, the present work must also contend with code transformations to enable nodes to sus-

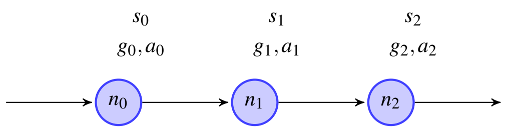


Fig. 1: A simple pipeline application topology. Node n_0 feeds into n_1, and n_1 feeds into n_2. Node n_i has service time s_i, average gain g_i, and maximum gain a_i.

pend at certain strategic points so that another node can run. We presently offer only low-level facilities that enable application developers to write preemptible code manually, but future work will consider the feasibility of automated, higher-level program transformations to support node suspension and resumption.

3 Application Model

In this section, we more formally describe streaming dataflow applications and how we map them onto a wide-SIMD execution platform. Our target for MERCATOR applications is an NVIDIA GPU running applications written in the CUDA language; however, this platform's properties and limitations are typical of other wide-SIMD targets such as AMD GPUs running OpenCL. Details of MERCATOR's usage and application mapping beyond those described here may be found in [12].

3.1 Application Mapping

An application is represented as a pipeline of *compute nodes* $n_0, n_1, \ldots$ with successive nodes connected by dataflow *edges* as shown in Figure 1. Each compute node n_i consumes a vector of up to v inputs at a time and produces a variable number of outputs per input for the downstream node n_{i+1} to process later. The number of outputs produced per input to a node, which we call its *gain*, may vary dynamically in a data-dependent fashion up to some known maximum.

An edge between two nodes has a finite *queue* in which data produced by the upstream node is stored until it can be consumed by the downstream node. We assume that queue sizes are fixed for the duration of an application's execution, or at least for the time needed to process a large number of inputs, due to the high cost of dynamic memory allocation on our target platform. When a node n_i begins to consume input from its upstream queue, it does so in SIMD vectors of up to v items at a time until either its upstream queue empties or it cannot consume another vector of inputs without potentially overflowing the remaining output space in its downstream queue. At that point, n_i must yield control to a global *node scheduler*, which selects other nodes to execute until n_i again has both available input data and available output space.

A GPU platform typically contains multiple processors, each of which may support multiple, asynchronous, non-communicating execution contexts (*GPU blocks* in CUDA). MERCATOR runs an independent replica of the application's pipeline within each context, with all contexts pulling data competitively from a single shared input stream, running asynchronously in parallel, and writing to a single shared output stream. Each context's pipeline replica has its own set of queues and its own scheduler instance that runs nodes *sequentially* within that context. In what follows, we focus on the behavior and memory usage of *one* pipeline replica, which processes SIMD vectors but whose nodes are sequentially scheduled, with the understanding that a GPU executing an application may run (and allocate queue memory for) hundreds of pipeline replicas concurrently.

Finally, we emphasize that node scheduling is *non-preemptive*: once a node starts to consume a vector of inputs, it cannot yield to the scheduler until those inputs have been completely processed and any outputs from them emitted downstream. This lack of preemption is a limitation of our target platform – CUDA does not support preemptive scheduling of different GPU kernels or of different functions within a single GPU kernel. Consequently, a node cannot safely consume a vector of inputs unless it has space for the most output it could possibly generate from these inputs in its downstream queue; otherwise, it might either overrun that queue or deadlock the application because it cannot finish execution.

3.2 Application Performance and Queue Optimizations

A node n_i's behavior is characterized by its *service time* s_i, *average gain* g_i, and *maximum gain* a_i. The service time s_i is the average time n_i takes to process a vector containing between 1 and v inputs, which is assumed to be constant due to the SIMD target architecture. The gain of a node defines how many data items are output (on average for g_i, or in the worst case for a_i) for each item input to n_i. The average number of outputs from n_i per input to the *first* node n_0 in the pipeline is n_i's *average cumulative gain*, computed as $G_i = \prod_{k=0}^{i} g_k$.

Our performance metric of interest for streaming dataflow applications is *throughput*. Throughput depends on the node scheduler, which should schedule nodes so as to ensure that they have full vectors of input ready to consume whenever possible. Moreover, because switching execution between nodes incurs runtime overhead, scheduling should ideally ensure that a node can run for as long as possible before an empty input queue or full output queue requires switching to another node. Irregular dataflow forbids *a priori* computation of a static optimal schedule as in regular streaming dataflow models [7,11], but MERCATOR uses a scheduling policy, Active-Full/Inactive-Empty (AFIE) [9], that ensures that nodes run with full input vectors and limits inter-node switches to within a small constant factor of the fewest possible even under a clairvoyant schedule.

Even with AFIE scheduling, application throughput is sensitive to the relative sizes of inter-node queues. For example, a node with high average gain benefits from a larger output queue because it can write more outputs before filling the queue and forcing a return to the scheduler. Given a fixed total budget of queue memory for one pipeline replica and the average cumulative gains G_i of each node, one may formulate the problem of how to allocate the budgeted memory among the queues in the pipeline so as to minimize the expected number of times the application must return to the scheduler. By solving this problem analytically, we may perform *queue space distribution* [12] to minimize scheduling overhead.

Finally, it may be that the overhead of queueing and dequeueing data between two adjacent pipeline nodes n_i, n_{i+1} exceeds the benefit to SIMD occupancy obtained by having the queue in the first place. In such cases, it may be better to *merge* the two nodes into one that performs their combined computations for each vector of inputs to n_i. Merging analysis [12] relies on knowledge both of the average gain g_i and of the service time s_i for each node.

4 Impact of Minimum Safe Queue Sizes

The need to enforce minimum safe sizes to accommodate uninterruptible nodes has consequences for application performance and resource utilization. In this section, we identify these consequences and illustrate them through two representative irregular streaming applications from the domains of branching search and bioinformatics. A key feature in these applications is a large gap between a node's average gain, which is most relevant for performance analysis, and its maximum gain, which determines the minimum queue size needed for safety.

4.1 Example Applications

In this section and Section 6, we study two irregular streaming applications whose pipelines exhibit the impacts of minimum safe queue sizes: NQueens and BLAST.

NQueens. The NQueens application enumerates all possible ways of placing N queens on an $N \times N$ chess board such that no queen can attack another. The problem can be solved using a *branching search tree*, in which a node at level i of the tree determines all the feasible ways to place a queen on row i of the board given fixed placements of queens on rows $1 \ldots i-1$. Similar branching structures appear in branch-and-bound combinatorial optimization and other tree traversal applications.

To make NQueens a streaming application, we create a pipeline of N nodes, where node n_i enumerates feasible ways to place the $i+1$st queen. An initially empty board is passed to the first node, which outputs N partial boards, each with a possible placement of a queen on the first row. For each such board, the second node places a queen in all feasible ways on the second row and passes the resulting partial boards to the third node, and so on until all complete feasible boards are enumerated. Node n_i can produce up to $a_i = N - i$ outputs per input, though some of these possibilities are infeasible and so are discarded.

Our benchmark computation enumerates all feasible solutions for $N = 18$. To ensure adequate parallelism to occupy all GPU blocks, we precompute feasible placements of the first four queens on the CPU and pass these partial boards as the input stream to a GPU pipeline of 14 nodes.

BLAST. The Basic Local Alignment Search Tool for nucleotide sequence [1] performs pattern matching in a large DNA database. A small DNA *query* sequence is compared to the database to identify substrings that match it to within a small edit distance. The application extracts successive substrings of length $k = 8$ from the database and compares them to a hash table of all k-mers in the query; when a k-mer matches, the locations of all matching k-mers in the query are enumerated, and each such match is further verified using a series of increasingly complex filters to retain the small fraction of matches that are biologically significant. BLAST is representative of a large class of decision filter cascades that can be implemented as irregular streaming applications; other examples include Viola-Jones face recognition [14] and Snort packet inspection [10].

The four nodes of the BLAST pipeline check the hash table, enumerate the positions of matches in the query, and implement successive filters. Of particular interest to our work is node n_1, which is responsible for enumerating k-mer matches and can list up to 16 matching query positions for each database position. We test BLAST on a query of length 30 Kbases from a bacterial genome against a database containing two copies of the human genome, equalling 6.4 Gbases.

4.2 Memory Bloat

For a node n_i that processes up to v inputs at once and emits at most a_i outputs per input, the minimum safe size for its output queue is $a_i v + v - 1$ slots [9]. Clearly, $a_i v$ slots are needed to accommodate the node's maximum outputs from one input vector; the remaining $v - 1$ slots are needed to accommodate a residue from prior runs of less than one full vector-width of items whose consumption may have been deferred in hopes of obtaining a full vector later. In short, the minimum safe queue size scales linearly with a node's maximum gain.

In contrast, the *ideal* queue size for a node is one that minimizes the overhead incurred by the node scheduler, which is proportional to the frequency with which the scheduler must be called to switch between nodes. In [12], we showed that given a fixed total amount of memory devoted to queues, the fraction of that memory that should be allocated to a node's output queue to minimize switching scales roughly as the square root of a node's *average cumulative gain*.

When the minimum safe size for a queue exceeds its ideal size for a given memory budget, we say that the queue is *bloated*. The larger the gap between a queue's average cumulative gain and its (individual) maximum gain, the greater the bloat of its output queue.

As an example, Table 1 illustrates the ideal and safe queue sizes for all 14 nodes of the NQueens application when using SIMD vectors of size 128. Nodes early in the pipeline have large maximum gains and hence large minimum queue sizes, since the branching search for feasible boards at early stages has few constraints. In contrast, the average cumulative gain is *smallest* for nodes early in the pipeline, growing rapidly with greater tree depth except at the highly constrained final stages. Moreover, all nodes individually have average gains less than (and mostly less than half) their maximum gains.

In the table, we consider a total queue memory allocation of 600 MB across 368 pipeline replicas, which was determined to be on the low end of a reasonable total queue memory allocation based on how much memory NQueens uses for input and output streams and how much memory was available on our GPU. The output queues for nodes early in the pipeline have smaller ideal sizes than their minimum safe sizes and hence are bloated. The queues for nodes 0 and 1 are bloated by more than a factor of ten. The application designer must therefore either increase its memory allocation to accommodate the required bloat or take away memory from later nodes' queues, incurring more scheduling overhead as a result.

Table 1: Gains and implied queue sizes of NQueens application with 128-wide SIMD vectors and a target allocation of 600 MB for all queues.

Node	Max Gain	Avg. (Cumulative) Gain	Safe Queue Size (Items)	Ideal Size (Items)
0	14	8.87 (8.87)	1919	44
1	13	7.46 (66.13)	1791	120
2	12	6.18 (408.62)	1663	298
3	11	5.12 (2094.02)	1535	674
4	10	4.20 (8792.57)	1407	1381
5	9	3.40 (29853.91)	1279	2545
6	8	2.71 (80990.22)	1151	4191
7	7	2.14 (173673.81)	1023	6138
8	6	1.66 (288936.00)	895	7917
9	5	1.27 (366991.64)	767	8922
10	4	0.94 (344898.36)	639	8649
11	3	0.66 (227664.25)	511	7027
12	2	0.41 (94298.90)	383	4523
13	1	0.19 (18367.82)	255	3992

The problem of bloat may be exacerbated by optimizations that attempt to merge adjacent nodes. In the case of NQueens, analysis of service times and SIMD occupancy according to [12] suggests that merging nodes 0 and 1 and eliminating the queues between them could be beneficial for throughput. However, merging two nodes with maximum gains a and a' results in a combined node with maximum gain $a \cdot a'$. For this example, the minimum safe size for the merger of nodes 0 and 1 is $14 \cdot 13 \cdot 128 + 127 = 23423$ entries – nearly 200 times the ideal queue size of 120. Adding this space to the ideal allocation shown would increase the application's overall queue memory usage by roughly 40%.

In short, when an application's nodes have a large maximum gain but a small average cumulative gain, the resulting constraint on queue sizes can lead to substantial bloat that increases memory requirements and forces deviation from the throughput-ideal pattern of queue sizing.

4.3 Pessimistic Scheduling Behavior

Even when bloat is not a substantial concern, the disparity between a node's average and maximum gain can incur additional costs to execution. To illustrate the issue, consider the queue allocations for the BLAST application shown in Table 2. The total memory allocation is much smaller than for NQueens (only 32 KB per pipeline, for 368 pipelines) because of the need to reserve as much GPU memory as possible for BLAST's sequence database. Given the application's overall memory budget and SIMD width, the minimum queue sizes do not incur substantial bloat except at the last node; overall, bloat accounts for only a small fraction of the application's overall queue space usage (either ideal or minimum safe).

Table 2: Gains and implied queue sizes of BLAST application with 128-wide SIMD vectors and a target allocation of 11.5 MB for all queues.

Node	Max Gain	Avg. (Cumulative) Gain	Safe Queue Size (Items)	Ideal Size (Items)
0	1	0.38 (0.38)	255	1552
1	16	1.92 (0.73)	2175	2151
2	1	0.03 (0.02)	255	392
3	1	0.000009 (0.0000002)	255	1

However, we observe that node n_1, the enumeration node, exhibits a large disparity between its maximum gain (16) and its individual average gain (roughly 2). This node cannot consume a vector of input unless it has at least $16 \cdot 128 = 2048$ slots free in its output queue; otherwise, the vector's output might overrun the queue in the worst case. However, the node's actual gain averages 246 outputs from one input vector. Hence, after consuming one input vector, the node typically must yield to the scheduler and cannot be run again until its output queue is emptied. Yet the node's queue is large enough to hold more than 8 input vectors' worth of "typical" output!

Hence, even disregarding bloat, a node whose maximum gain greatly exceeds its average gain typically leaves most of its output queue unused. If that space could safely be used without fear of overrun, the node would encounter a full output queue (and hence would need to return the scheduler) much less often.

Both bloat and pessimistic scheduling behavior are driven by nodes with maximum gains that far exceed their average individual or cumulative gains. This disparity is traceable to a basic limitation of our model: *because nodes must process their inputs without interruption*, they need enough space to write the maximum possible amount of output each time they run. In the next section, we describe a method to remove this limitation.

5 Interruptible Nodes

To overcome performance and resource issues caused by large minimum queue sizes, we extend the MERCATOR framework with support for *interruptible nodes*. We first

```
__device__ void
MyNode::run(int x) {
  i = 0;
  while (i < M) {
    int v = f(x, i);
    push (v, v > 0);
    ++i;
  }
}
```

Fig. 2: A MERCATOR node function that iterates over its input to produce up to M outputs per input item. Although the code appears sequential, it runs concurrently on an entire SIMD vector of inputs, each of which maps to the variable x in a different CUDA thread.

describe the basic idea of interruptible nodes and why they address the problems identified in the previous section, then describe how we implement them given the limitations of our target platform.

5.1 Semantics of Interruptible Nodes

The key observation underlying interruptible nodes is that, while a node may produce > 1 output per item in its input vector, it cannot actually enqueue more than v items (the width of one SIMD vector) at a time. More specifically, a node in a MERCATOR application emits items to its downstream queue by calling a function `push()`, which takes a vector of items and a per-SIMD-lane flag indicating whether each item is valid (and so should be emitted). Because `push()` cannot emit more than v items at once, A node that may emit multiple outputs from a single input must call `push()` multiple times in one run.

As an example, Figure 2 illustrates CUDA code for a MERCATOR application node `MyNode`. Each SIMD lane of `MyNode` takes an integer input x and produces up to M outputs. The ith potential output for a SIMD lane is computed from the input by a function `f()`, and the result is pushed downstream iff it is a positive value. This node's maximum gain is M, but its average gain depends on the inputs and the properties of the function `f()`.

A single call to `push()` is safe so long as the downstream queue has at least v slots available to receive outputs. This is true no matter how large the node's maximum gain is. Hence, in a node that may call `push()` several times, we wish to make each such call a *yield point* – if the push would fail due to insufficient queue space, the node should be suspended, and control should return to the scheduler. Once sufficient space is available, the node may resume execution from the point of suspension.

Making a node interruptible addresses the performance and resource concerns raised in the previous section. Critically, it is no longer necessary to bloat a node's output queue to accommodate its maximum gain a_i, because the node can be suspended if it would otherwise overrun the queue. The only size constraint on the output queue is that it hold at least $2v-1$ entries – the minimum needed by the AFIE scheduler for safety given pushes of size up to v items. Moreover, if (as in our BLAST example) a node's output queue size significantly exceeds its vector width times its average gain, the node will likely be able to consume multiple vectors of input without filling the queue and returning to the scheduler. Should the node exhaust its queue space while processing a vector, it can now be suspended and resumed later.

5.2 Implementation Challenges for Interruptibility

A MERCATOR application is specified using a high-level pipeline description that produces a CUDA skeleton, with stub functions for each node that are filled in by the application developer. These functions are compiled together with MERCATOR's node scheduler and other runtime support code to form a single GPU kernel that consumes a stream of inputs stored in GPU global memory. Adding interruptible node semantics to this model is challenging due to the limitations of CUDA and so requires cooperation from the application developer.

Ideally, CUDA device code would support a facility for saving execution state in a way that can be resumed later, analogous to `setjmp/longjmp` in C or continuations in functional languages. In the absence of such a facility, we chose to provide a minimal set of extensions to let an application recognize when node suspension is required and communicate a decision to suspend to the MERCATOR runtime. The application developer then implements state saving and restoring as part of the node's code.

We extended MERCATOR's runtime in two ways. First, the `push()` function now returns a boolean value to indicate if the *next* call to `push()` might fail due to insufficient (i.e., $< v$ items) downstream queue space. Second, a node now returns a boolean value to the MERCATOR runtime to indicate whether it finished processing its input vector (and so can immediately be run again with another vector if one is available) or had to suspend in the middle of processing a vector. In the latter case, when a node resumes after interruption, the runtime will invoke it with the *same* input vector that it was processing when it suspended execution. MERCATOR guarantees that a suspended node will not be called again until it can successfully complete at least one push operation of up to v items and so make progress.

The application developer's code for a node is responsible for detecting that the next call to `push()` may fail, saving its state in order to suspend itself, and later restoring this state and resuming execution when it is called after a suspension. This code may take advantage of MERCATOR's per-node state facility, which lets the developer declare state variables that can be initialized at application load time and then read and written from within a node. Figure 3 illustrates a modification of the node in Figure 2 to support suspension and resumption.

Even this simple example illustrates the challenges of user-directed suspension and resumption. The state variable is shared by all CUDA threads, so writes to it must be protected by block-wide synchronization calls to ensure that all threads see a consistent value. Real applications may need to store multiple pieces of state in order to resume execution. The more complex the control structure of the node (e.g., a `push()` inside nested loops), the more challenging it is to transform the code to

```
__device__ void
MyNode::init() {
    if (threadIdx.x == 0)
      getState()->i = 0;
    __syncthreads();
}

__device__ bool
MyNode::run(int x) {
  int i = getState()->i;
  bool canContinue = true;
  while (i < M && canContinue) {
    int v = f(x, i);
    canContinue = push(v, v > 0);
    ++i;
  }
  __syncthreads();
  if (threadIdx.x == 0)
    getState()->i = (i == M ? 0 : i);
  __syncthreads();
  return (i == M);
}
```

Fig. 3: Modification of a node to support suspension and resumption. The current iteration *i*, which is the only state variable that must be stored on suspension, is initialized to 0 at application start and is then read from stored state each time the node is run. When a push indicates that the downstream queue is full, the loop is interrupted and its current state stored. If fewer than *M* iterations have completed, the node returns *false* to the MERCATOR runtime to indicate that it should be suspended.

behave correctly in the presence of suspension and resumption. Future work should investigate whether a CUDA language compiler can be extended to perform the transformations needed for node interruptibility or to implement an efficient `setjmp`-like facility.

Finally, we note that the code transformations needed to support interruptibility themselves introduce overhead, in the form of additional state reads and writes and additional synchronization. The cost of this overhead must be weighed against the savings from fewer invocations of the node scheduler when evaluating the performance impact of interruptible nodes.

6 Empirical Evaluation

To evaluate the quantitative impacts of interruptible nodes on application performance and storage in MERCATOR, we implemented interruptible node support as described in the previous section, then modified the code of our example applications (BLAST and NQueens) to support saving and restoring of node state. We then investigated the behavior of these applications on an NVIDIA RTX 2080 GPU using CUDA 11.2. Applications were run using inputs as described in Section 4.1 with a

SIMD vector width of 128 and used 368 pipeline replicas (the maximum number of CUDA blocks permitted on our GPU given the applications' register usage) to fully occupy all processors of the GPU. All reported running times represent the average over 50 trials.

6.1 Reduction of Scheduling Overhead

We first compared the NQueens application without interruptible node support ("NoInterrupt") to a version in which all 14 nodes of the pipeline were made interruptible ("AllNodeInterrupt"). We did this comparison for a range of target allocations for total queue memory, from 600 to 1400 MB summed over all queues in all replicas. For the NoInterrupt implementation, any queue bloat required for safety was allocated over and above this target value. For the AllNodeInterrupt implementation, we allocated the same total amount of memory as for the corresponding NoInterrupt version but redistributed the excess previously used for bloat across all the application's queues so as to minimize switching overhead, according to the analysis of [12].

Figure 4 shows that the net impact of interruptibility on performance was negative – the additional cost and complexity of saving and restoring node state far outweighed any savings from overhead reduction. This result was consistent over a range of possible targets for total memory allocated to queues.

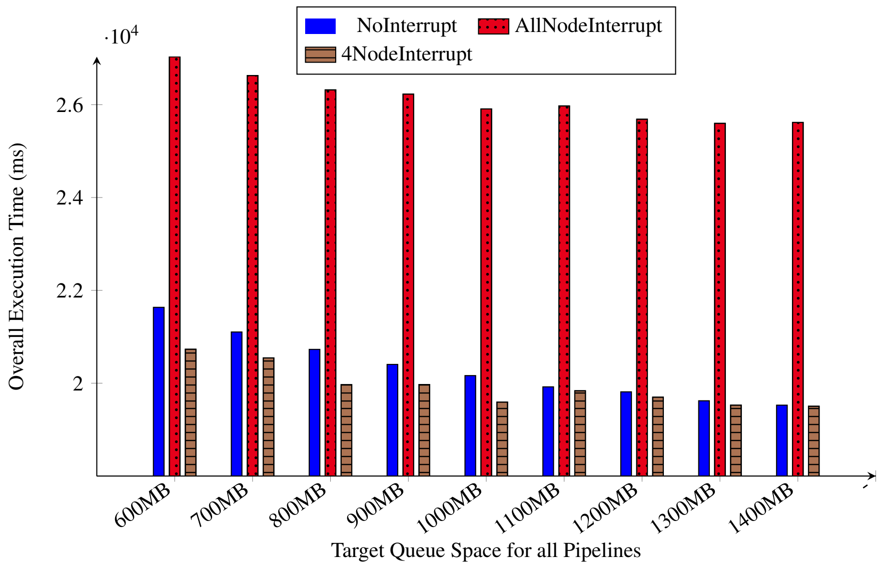


Fig. 4: Total execution time for NQueens for different target allocationsfor total queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±150 ms.

Recall from Table 1 that only the first few nodes of the NQueens pipeline exhibited output queue bloat. To reduce the cost of interruptibility, we modified the AllNodeInterrupt implementation so that only the first four nodes of the pipeline were interruptible; the remaining nodes were left uninterruptible. This modified implementation ("4NodeInterrupt") exhibited a statistically significant performance *improvement* over the uninterruptible version for target allocations up to 1100 MB.

As shown in Figure 5, redistribution of space previously needed for bloat in the first four nodes of NQueens had a salutary effect on scheduler overhead. Nodes near the middle of the the pipeline, which have the largest average cumulative gain (i.e., process the most data) benefit the most from larger output queues through reduction in scheduler calls, which we believe to be the primary source of performance improvement. This benefit diminishes as the total memory allocated to queues grows, since bloat (and hence the memory redistributed to other queues) is the excess of a queue's minimum safe size, which is fixed, over its optimal target size, which grows with the overall memory allocation. Moreover, larger queues reduce the absolute number of times the scheduler is called, which further reduces the benefit of the redistribution optimization. Hence, we see that the difference in running time between NoInterrupt and 4NodeInterrupt decreases with increasing target allocation.

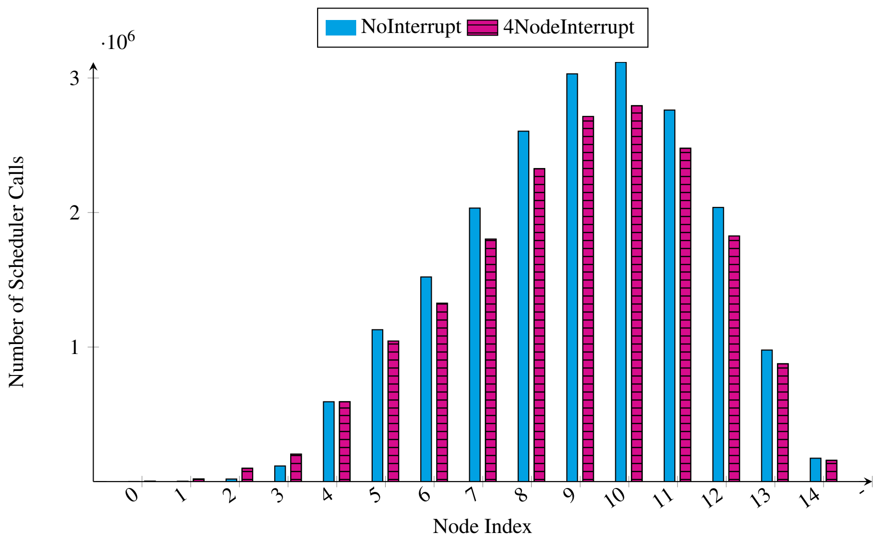


Fig. 5: Number of scheduler calls from each node for NQueens given 600MB total queue space for all pipelines.

We then investigated whether the same strategy of targeted interruptibility was effective for the BLAST application. Recall from Table 2 that BLAST was not significantly bloated even at a relatively small target allocation of queue space; however, we identified node 1, which has a max gain of 16 but an average gain of only 2, as having a queue that was significantly underutilized. We therefore made only this node interruptible and compared the performance of the modified application ("Interrupt") to that of the original, uninterruptible version ("NoInterrupt").

Figure 6 shows that at the smallest target allocation (11.5 MB across all queues), targeted interruptibility had a large beneficial effect on running time. As Figure 7 shows, making node 1 interruptible greatly reduced the number of times it was forced to yield to the scheduler, as would be expected given that the node can now safely consume multiple vectors of input before filling its output queue. Allowing this queue to fill also reduced the frequency with which the following node, node 2, had to yield to the scheduler due to an empty input queue.

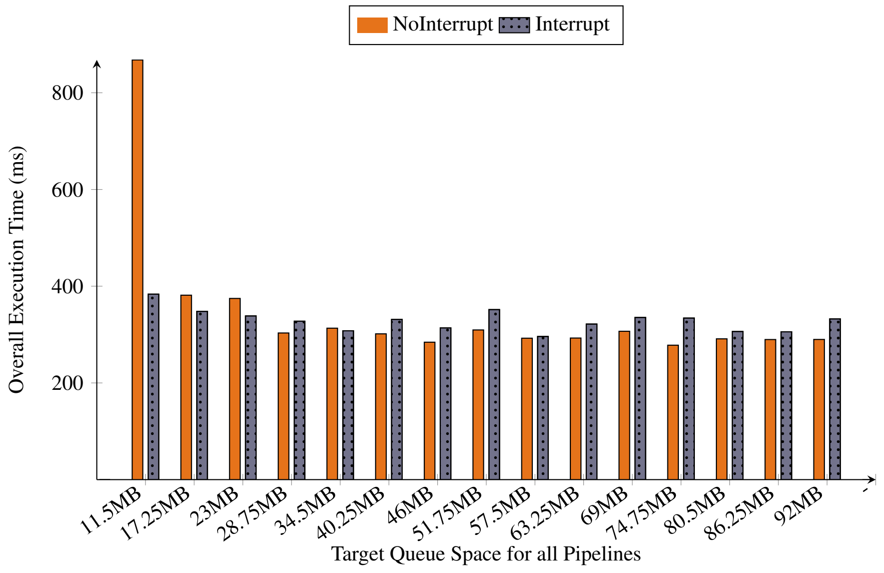


Fig. 6: Total execution time for BLAST for different target allocations of queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±50 ms.

Once again, the benefits of interruptibility were highest at small target allocations, as larger allocations (e.g., 92 MB, as shown in the figure) are naturally large enough to permit node 1 to write multiple input vectors' worth of worst-case output to its output queue before yielding. Overall, we observed no statistically significant difference in performance in the interruptible vs. uninterruptible implementations for target allocations larger than 11.5 MB.

6.2 Combining Interruptibility with Node Merging

As discussed in Section 3, node merging to eliminate queue overhead is a potentially useful pipeline transformation. However, for nodes with large maximum gains, merg-

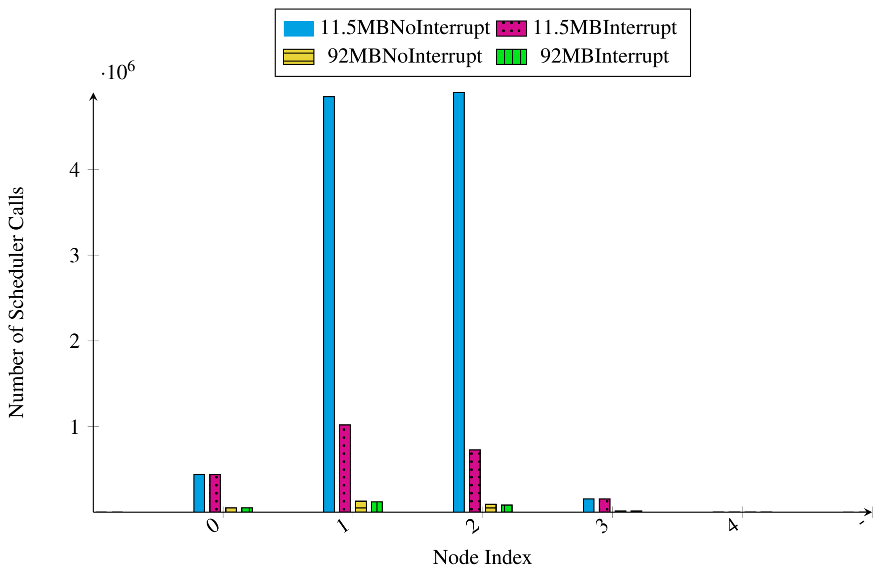


Fig. 7: Number of scheduler calls from each node for BLAST at two different target queue space allocations.

ing can result in excessive queue bloat for the merged node's output queue – bloat that can be ameliorated by making the merged node interruptible.

We investigated the impact of merging nodes 0 and 1 of the NQueens application, which our analysis in [12] suggested was potentially beneficial to performance. Without interruptibility, merging these two nodes increased the application's actual queue memory usage by 16-30% beyond the target, as shown in Figure 8. Making the merged node interruptible eliminated this excess memory usage. The resulting implementation exhibited running time similar to that of the unmerged version.

7 Conclusion and Future Work

Irregular streaming dataflow applications have great potential for wide-SIMD parallelization, but this potential can be realized only by inserting queues in the application pipeline. The "right" sizes for these queues are determined by potentially conflicting design considerations: performance, which favors certain relative queue sizes to reduce scheduling overhead, and safety, which imposes often severe minimum queue size requirements when a node can produce many outputs per input in the worst case. We have shown that the pressure of safety on queue size can be ameliorated by selectively making nodes *interruptible*, and that doing so can be a net positive for throughput and/or queue memory usage.

Interruptibilty works best when targeted to nodes with large maximum gains but much smaller individual or cumulative average gains. The space saved from such node's output queues can be removed from the application, decreasing its memory

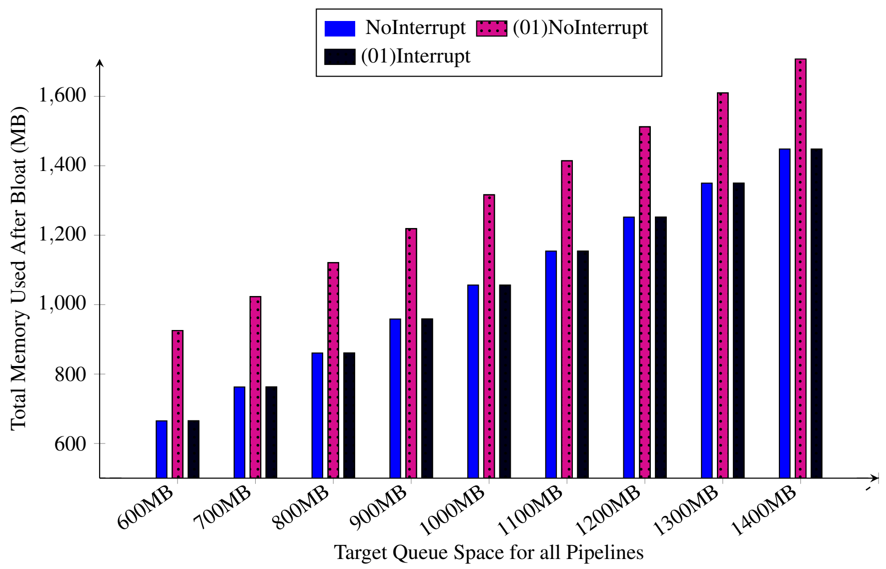


Fig. 8: Total actual memory used for each target allocation in NQueens before and after merging nodes 0 and 1.

usage, or redistributed to other nodes in the pipeline, potentially increasing throughput. Even when queue sizes do not change much after interruptibility, a node whose average gain is far below its maximum gain can benefit from reduced scheduler overhead when it is made interruptible. These benefits are most readily seen when the overall target allocation of queue space to the application is smaller, since changing queue sizes and allowing greater queue occupancy have the largest impact when the total available queue space is small.

In the future, we hope to obtain more accurate measurements of the overhead of interruptibility, in particular the cost of saving and restoring state. Timing these operations, which take place inside a function called within the CUDA kernel, is challenging, particularly because they may involve operations by multiple GPU threads that run asynchronously. We also plan to better model potential *increases* in node switching overhead when the bloat is removed from a node's output queue. Accounting for these effects would allow us to better predict whether making a node interruptible is likely to be beneficial to throughput overall and to direct the effort of optimization accordingly.

Another avenue for investigation is whether the burden of interruptibility on the application developer — in particular, the need to extensively rewrite code to support interruptions — can be reduced. Ideally, the CUDA runtime would provide support to implement suspension and resumption of nodes with appropriate saving of state in between. Because of GPUs' very large register files, naively saving all register state for a block when suspending might have a prohibitive cost in time and memory; hence, it may be preferable to leverage compiler analysis or user-provided variable tagging to identify and save only live data at the point of suspension. Alternatively, GPU support for hardware preemption would greatly aid interruptibility and might change the preferred realization of streaming applications on the GPU entirely. The impacts of hardware preemption for irregular streaming could be investigated now on multicore CPUs, which have robust preemptive multithreading as well as increasingly large SIMD vector widths.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3), 403–410 (1990)
2. Cabrera, A.M., Faber, C.J., Cepeda, K., Derber, R., Epstein, C., Zheng, J., Cytron, R.K., Chamberlain, R.D.: DIBS: A data integration benchmark suite. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, p. 25–28. Association for Computing Machinery, New York, NY, USA (2018)
3. Cole, S.V., Buhler, J.: MERCATOR: A GPGPU framework for irregular streaming applications. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 727–736 (2017)
4. Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y., et al.: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: 2011 USENIX Annual Technical Conference (USENIX ATC 11) (2011)
5. Kim, H., Patel, P., Wang, S., Rajkumar, R.R.: A server-based approach for predictable GPU access control. In: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10. IEEE (2017)
6. Kolesar, P.J.: A branch and bound algorithm for the knapsack problem. Management science **13**(9), 723–735 (1967)
7. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**, 1235–1245 (1987)
8. Park, J.J.K., Park, Y., Mahlke, S.: Chimera: Collaborative preemption for multitasking on a shared GPU. ACM SIGARCH Computer Architecture News **43**(1), 593–606 (2015)
9. Plano, T., Buhler, J.: Scheduling irregular dataflow pipelines on SIMD architectures. In: Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing, WPMVP'20, pp. 1–9. Association for Computing Machinery, New York, NY, USA (2020)
10. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration, LISA '99, p. 229–238. USENIX Association, USA (1999)
11. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: R.N. Horspool (ed.) Compiler Construction, pp. 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
12. Timcheck, S., Buhler, J.: Reducing queuing impact in streaming applications with irregular dataflow. Parallel Computing **109**, 102863 (2022)
13. Tyson, E., Buckley, J., Franklin, M., Chamberlain, R.: Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **595**, 474–479 (2008)
14. Viola, P., Jones, M.: Robust real-time object detection. In: International Journal of Computer Vision (2001)
15. Wu, B., Liu, X., Zhou, X., Jiang, C.: FLEP: Enabling flexible and efficient preemption on GPUs. ACM SIGPLAN Notices **52**(4), 483–496 (2017)

Noname manuscript No.
(will be inserted by the editor)

SMSG: profiling-free parallelism modeling for distributed training of DNN

Haoran WANG · Thibaut TACHON · Chong LI · Sophie ROBERT · Sébastien LIMET

Received: date / Accepted: date

Abstract The increasing size of deep neural networks (DNNs) raises a high demand on distributed training. Hybrid parallelism, which combines different parallelism strategies, has been proved to an efficient option than using either data parallelism or operator parallelism a single parallelism. Good hybrid parallelism strategies could be found by a human expert, but it takes time to design such suitable strategies, and it is not easy to adapt them to new coming DNNs or variations of the existing DNNs. Thousands of new types of DNNs have come out each recent year. Therefore, automating parallelism strategy generation is crucial and desirable for DNN designers. Recently, some automatic searching approaches have been studied to free the experts from the heavy parallel strategy conception. However, these approaches all rely on a numerical cost model, which requires heavy profiling results that lack portability. The profiling work needs to be done again, whether the network structure or hardware architecture changes. Besides, the large number of operators and possible partition dimensions of each operator also complicate the profiling tasks. As a result, these profiling-based approaches cannot lighten the strategy generation work.

Our intuition is that there is no need to predict the actual execution time of the distributed training but to compare the relative value of the cost of different strategies for choosing a proper parallel strategy. From the previous numerical cost models, which consider the operator as the primitive item, SMSG proposes as an alternative a symbolic cost model based on the communication and computation semantic. With the symbolic model, we decouple the parallel algorithm from hardware characteristics. SMSG defines cost functions for each kind of operator to quantitatively evaluate the amount of data for computation and communication. Heavy profiling tasks are avoided. Besides, to control

H. WANG
20 quai du point du jour, 92100 Boulogne-Billancourt, France
Tel.: +33-640711437
E-mail: wanghaoran19@huawei.com

the high searching complexity caused by the redistribution cost, a functional transformation using the Third Homomorphism theorem was applied to reduce the searching complexity. Experiments show that SMSG can find good hybrid parallelism strategies to generate an efficient training performance similar to the state of the art. Moreover SMSG covers wide varieties of DNN models with good scalability. SMSG provides as well good portability when changing training configurations that a profiling-based approach cannot.

Keywords Distributed Training · Deep Network Networks · Symbolic Cost Model · Functional Transformation · Performance Analysis

1 Introduction

The size of DNN models has been scaling up dramatically in recent years. Many gigantic models achieved remarkable accuracy in domains like natural language processing (NLP) [1], computer vision [2], recommendation systems [3], etc. The execution of such a gigantic model requires tremendous computational and memory resources [1]. To be able to train gigantic models in a distributed way, different *parallelism*, like *data parallelism*[4], *operator parallelism*[5] and *pipeline parallelism*[6], have been proposed. Different parallelisms were initially designed for different basic model structures. Mixing parallelisms [7] could provide not only a more general approach for modern complex structures but also a more efficient solution to train a DNN model than applying single parallelism.

Efficient mixing parallelism with *hybrid strategies* could be designed by a human expert [7]. However, it is time- and labour-consuming to find a well-behaviour strategy for each given DNN model. Experts usually spend months to decide an efficient parallel strategy for a new DNN model. Therefore, systematically generating efficient strategy generation is crucially desired by the DNN model researchers.

Several strategy searching approaches, like [8–13], have been proposed and could provide effective hybrid strategies for their targeted DNN models. However, these approaches exhibit poor generality for new-coming DNN models because they are all based on a profiling-based cost model. The number of operators in the modern DNN model is more than 1000 [1], and the number of possible partition dimensions increases exponentially w.r.t the number of devices [13]. Therefore, the profiling work takes a lot of time [8]. Besides, when the shapes or the sub-structures of a DNN change or deal with a different hardware environment, the profiling tasks need to be prepared again. All these time-costly tasks result in the poor generality of these approaches.

To avoid the limitations of the profiling-based approaches, we proposed a profiling-free approach to expand the generality of automatic strategy searching in this paper. Our main contributions are the followings. (i) Based on the computation and communication semantics, we build a symbolic cost model to quantitatively evaluate the relative cost instead of predicting the execution time of different parallelism strategies. In this model, the cost is separated into two parts: the hardware parameters and the computation and communication data quantity. We decouple the hardware characteristics from the parallel execution details with this model. Therefore, we only need to profile the hardware parameters rather than the execution time of each operator under uncountable configurations.

(ii) Inspired by the Homomorphism theory, we derive the strategy generation from the symbolic cost model and reduce the exponential complexity caused by the redistribution cost between operators. Therefore, the searching time is restricted to an acceptable range, which contributes to the generality of our approach.

To validate the generality of SMSG, we conducted the following experiments. We choose Expert-Designed strategies for targeted DNN models as the best performance baseline, and we choose TensorOpt [8], the SOTA method of operator-level strategy searching, as the automatic searching approach baseline. First, we compare the strategy quality of ResNet[2] on different hardware architectures. With careful profiling, both SMSG and TensorOpt can find the strategy with similar end-to-end performance as the Expert-Designed strategy. However, the experiments also show that the profiling-based approaches like TensorOpt find strategies with sub-optimal performance without carefully profiling results. Moreover, we tested SMSG on varieties of DNNs including ResNet50/101/152, Wide&Deep [3], Bert [14], GPT-3 [1], and T5 [15]. The similar performance of the Expert-Designed strategy validates the generality of SMSG.

2 Modeling the cost of distributed training

2.1 Distributed strategies searching

The objective of DNN training is to find proper parameters to predict results from new inputs. A training iteratively executes *Forward Propagation* (FPG) and *Backward Propagation* (BPG). A FPG computes a batch of inputs using current parameters to predict results; a BPG starts from the derivative of the last operator back to the derivative of the first operator, computes the gradients and updates the parameters according to the Loss. The input data of a DNN model is always processed in a batch, i.e. dozens of or hundreds of data items are executed simultaneously in an iteration step.

A DNN model could be represented as a *computational graph*, which is a directed acyclic graph (DAG), where vertices are the operators and edges connected to the operators signify the dataflow direction in the graph. *Operators* are DNN-level mathematical computations such as Matrix Multiplication (MatMul), Convolution (Conv), Element-wise operator (Ele-W, e.g. Add/Mul), etc. The data of DNN training is *tensor* structured as a multi-dimension array. The dimension of the tensor, which organizes data (e.g. an image, a sentence, or a vector of features) in a batch, is called *batch dimension*.

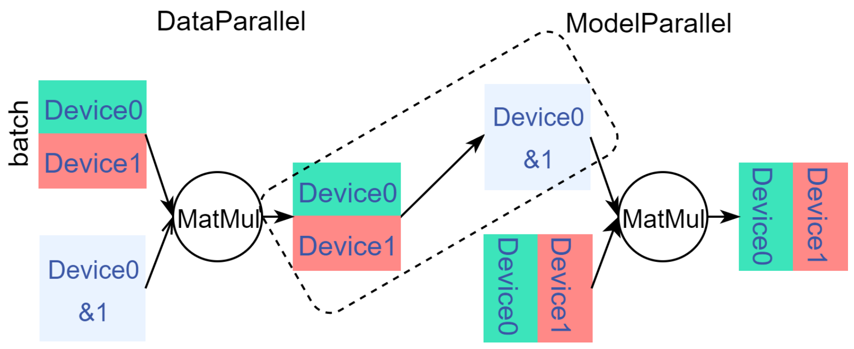


Fig. 1 Simplest computational graph with DP/MP examples

DNN training can be parallelized differently. In the paper, we take the following the most used parallelisms as examples:

– *Data parallelism* partitions a batch into several mini-batches onto multi-devices while each device owns the same entire DNN models. Data parallelism is efficient for the DNNs with a small parameter size because there is no extra communication cost except the parameter updating at the end of each *step* (training iteration). The cost of parameter updating corresponds to the q_p in Section 2.3. However, gigantic state-of-the-art models can have trillions of parameters [1] where data parallelism suffers from high parameter updating costs.
– *Operator parallelism* which splits the other dimension except for the batch dimension of the operator, which causes extra intra-communication which corresponds to the q_c. The number of possible dimensions of operator parallelism for a single operator is enormous, so the analysis of the intra-communication cost is complicated. Besides, different dimensions partitioned for connected operator generates data redistribution, which corresponds to the q_r.

Fig. 1 shows a computational example composed by two MatMul. The first MatMul is partitioned along its batch dimension, which is applied Data Parallelism, while the second one is partitioned along the vertical dimension of its second input tensor, which is one kind of model parallelism. It can be conducted from this graph that the output of the first MatMul is the same tensor as the first input of the second MatMul. Data redistribution is generated because of the mismatch of the data.

Deep Learning (DL) frameworks [16–18] can accelerate the development of DNN models by systematically generating execution code from a computational graph. Advanced frameworks can even automatically insert communication for distributed training from parallelism strategies. A parallelism strategy denotes along which dimensions the tensors of an operator are partitioned, as shown in Fig. 3.

Deciding a suitable hybrid parallelism strategy is difficult for the following reasons: (i) Different types of operators may prefer different parallelism strategies. For example, it is not suitable to partition the kernel tensor of a Conv op because the shape of the kernel is usually $3 * 3$ or $5 * 5$. A common practice for Conv op is to partition it along its *batch dimension* (data parallelism) or *channel dimension* (one of the possible operator parallelism strategies). [2] (ii) Besides, the shape of the operators also affects the strategy choices. [8] A MatMul op is more suitable for data parallelism when its parameter tensor is small and should be configured as operator parallelism when its parameter tensor is very large. (iii) The operators are not executed separately. They are all connected in the computational graph via the edges. [19] The output tensor of an operator is also the input tensor of its successive operator. If the parallel strategies are different for these two operators, the tensor needs to be redistributed in the cluster, which generates additional cost and should also be taken into consideration.

Due to these difficulties, it is time and labour-consuming to concept accurate parallelism strategies according to researchers' expertise and experiments. Moreover, expert-designed hybrid parallelism usually returns bad performances when migrated to new DNN models. Therefore, systematic strategy searching has become a crucial research topic.

2.2 Profiling-based modeling

Many parallelism strategy generators have been proposed recently: OptCNN [11], ToFu [13], TensorOpt [8], etc. OptCNN supports operator-level (data parallelism and operator parallelism) parallel strategy searching automatically for each layer (a group of operators to perform one object) in CNN with a numerical cost model and a dynamic programming algorithm. ToFu and TensorOpt extend OptCNN to generate strategies for each operator. The results of these works show the feasibility and the potential of automatic parallelism strategy generation.

The principal idea of distributed cost modelling is to describe the time consummation of the distributed training. Related works[8,12] try to build a general cost model to predict the actual execution time. They compare the predicted time of different parallelism strategies of the given DNN on a specific hardware platform and choose the strategy with the lowest predicted time. Let us take OptCNN [11] as an example since other solutions are the extensions of it.

A computational graph $G = (V, E)$ is defined by the vertices set V such that $v_i \in V$ is an operator and the edges set E such that e_{ij} represents the dataflow direction between v_i and v_j. The hybrid parallelism strategy $\mathcal{S}$ of the Graph G is defined as the set of operator-level strategies. Let Op an operator, if n denotes its number of input tensors, $S_{Op} = \{s_i, 1 \le i \le n\}$ where $s_i = [x_{d1}, x_{d2}, ...]$ is a set of integer to define how to partition each tensor of d_i dimension.

The predicted global execution time T is defined as follows:

$$T(G, \mathcal{S}, D) = \sum_{v_i \in V} (t_e(v_i, S_{v_i}, D) + t_p(v_i, S_{v_i}, D)) + \sum_{e_{ij} \in E} t_r(e_{ij}, S_{v_i}, S_{v_j}, D) \quad (1)$$

– $t_e(v_i, S_{v_i}, D)$ is the time to execute the operator v_i under its strategy. It includes the time for local computation and the communication time caused by the partition strategy. t_e is an average time measured from several executions of the operator v_i profiled with its strategy under hardware environment D.
– t_p denotes the parameter updating time at the end of each iteration (after backward propagation). Parameter updating is usually implemented by all-reduce, requiring the same profiling method as t_e.
– t_r, the redistribution cost between two connected operators v_i, v_j, is usually estimated by the multiplication of the data size and the known communication bandwidth.

This modelling methodology describes the total cost generated from the distributed training. Experiments show that optimal hybrid strategies are found in this way. However, OptCNN can only search layer-level strategies; Tofu/TensorOpt searches operator strategies only for small DNNs. The main limitation of these modelling methods is that they require inevitable preparing work to profile the operator under different configurations. New types of DNNs come out today, which also carries out new types of operator and partition dimensions. This modelling methodology becomes unrealistic for the following reasons: (i) A computational graph may contain thousands of operators. Even the operators could be classed into dozens of types, but the profiling works are required when the shape of the operators changes. (ii) For one operator, all the dimensions of its tensors are splittable, thus causing a polynomial search space, making the profiling work heavy. (iii) Profiling results of an operator are heavily dependent on the hardware. Re-profiling is needed when the type, number of the accelerator, or cluster connections change.

Due to the heavy preparation work, these profiling based modelling methods lack portability and generality. As a result, a more general method for new DNNs with different environments is demanded.

2.3 Profiling-free modeling

The profiling task is inevitable for these approaches because, from an AI expert's point of view, the operator is the basic unit of performance modelling that cannot be split. As a result, $t_e(v_i, S_{v_i}, D), t_s(v_i, S_{v_i}, D)$ are unsplittable and the values could only be estimated through profiling.

Our insight is to model the cost through a parallel computing model like the bridging model[20]. We model the computation and communication costs caused by parallel strategy instead of the execution time of operators and edges. With such a modelling methodology, the parallel execution analysis can be decoupled with the hardware environment, which is critical in avoiding the heavy profiling task. Besides, we do not need to compare the real predicted time to evaluate the performance of hybrid parallelism strategies $\mathcal{S}$. The relative value could compare the performance.

Based on what has been presented above, we propose the following symbolic cost model as a metric to compare the performance of two strategies $\mathcal{S}$ on a computational graph G:

$$\begin{aligned} C(G,\mathcal{S}) = &\sum_{v_i \in V} (w \times q_x(v_i, S_{v_i}) + g \times (q_c(v_i, S_{v_i}) + q_p(v_i, S_{v_i})))+ \\ &\sum_{e_{ij} \in E} g \times q_r(e_{ij}, S_{v_i}, S_{v_j}) \end{aligned} \tag{2}$$

In this model, the variables could be classified into two categories as follows:

– **Profiled hardware parameters:** w, g denote the real-time computation capacities of the accelerator. The calibrated FLOPS and bandwidth usually cannot be fully used, and these two values are obtained through profiling the hardware environment.
– **Data quantity function without profiling:** $q_x(v_i, S_{v_i}), q_c(v_i, S_{v_i})$ and $q_p(v_i, S_{v_i})$ are respectively the computation quantity, intra-communication caused by parallelism and parameter updating communication for an operator v_i under the strategy S_{v_i}. $q_r(e, S_{v_i}, S_{v_j})$ denotes the quantity of data redistribution caused by conflicting strategy of two connected operators.

The hardware feature and parallel costs are separated with this symbolic modelling method. The computation and communication capacities w, g can be estimated by profiling the hardware. The quantities q_x, q_c, q_p and q_r can be symbolically analysed without profiling.

Various new DNNs have come out in recent years, but they are all based on 20+ computational operators (e.g. MatMul, Conv, Etc.) and 100+ element-wise operators. These operators can be classed into 20+ types. We analyze the semantics of these 20+ operator types and build the symbolic cost model for each operator type under different strategies. As for the possibilities of partition dimension, even though there are many dimensions for each operator, only a few are practical for parallel training after semantic studying. For example, Conv operator has seven possible partition dimensions including *batch, input_channel, input_height, input_weight, output_channel, kernel_height, kernel_weight*, but only *batch* and two *channel*s are practical because the other dimensions generates super large communication cost. For partitioning kernel dimension, the communication is hard to be implemented because the shape is usually petite, like 3×3. As a result, we define the cost model for the 20+ types of operators with limited possible partition dimensions, which could be generally applied to any DNN.

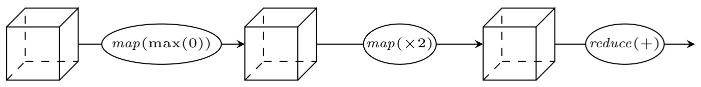


Fig. 2 Minimal neural network example

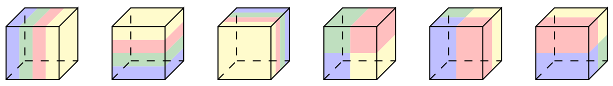


Fig. 3 Possible distribution of a 3D tensor over 4 devices

Our symbolic model can help find the optimal strategy because it is essentially the same as the profiling-based model. In fact, the two models describe the same parallel cost in different scope, $t_e(v_i, S_{v_i}, D)$ in profiling-based model is actually the summation of $w \times q_x(v_i, S_{v_i}) + g \times q_c(v_i, S_{v_i})$ in our symbolic model. $t_p(v_i, S_{v_i}, D)$ equals to $g \times q_c(v_i, S_{v_i})$ and $t_r(e, S_{v_i}, S_{v_j}, D) = g \times q_c(e, S_{v_i}, S_{v_j})$. Thanks to the symbolic modeling, the heavy profiling works are avoided. We can quantitatively evaluate the cost of computation, intra communication and redistribution with the defined symbolic cost model. We only profiles the communication/computation capacity of the hardware. However, because we don't use the real profiled value, the redistribution cost of the symbolic cost invokes high search complexity. Section 3 proposes a method to reduce the strategy generation complexity. Although this complexity reduction comes at the cost of decreasing the quality of the strategy found, we present a scheme to mitigate the decrease in quality. This scheme can be seen as a heuristic-based greedy approach in which we prove that the heuristic-based vertex reordering preserves the graph cost thanks to its definition as a homomorphism.

3 Functional transformation and reduction

The cost of a vertex depends on its own strategy (computation amount q_x, intra-communication q_c and parameter updating q_p) but also of the strategy of its neighbours (redistribution q_r). Choosing a strategy for a vertex will influence its neighbours that will influence their neighbours until the whole graph recursively. It is impossible to find the optimal distribution strategy for real-life deep neural networks in a reasonable amount of time.

Example We illustrate the complexity of the problem through the following minimal example. Consider a toy neural network represented on fig. 3 of 3 operators on 3-dimensional tensors that we aim to distribute over four devices. Operators may be, for example, a RELU followed by an element-wise twofold increase followed by the sum of all elements. Possible distributions of a 3D tensor over four devices are all illustrated on fig. 3, and the number is 6. As there are three operators, the total number of possibilities for this whole tiny graph is $6^3 = 216$.

We plan to cut the complexity of this problem by taking decisions based on local contexts and not questioning them afterwards, which is a greedy method. To mitigate the difference between our cost and the optimal one, we treat vertices by order of decreasing importance. This way, the most critical vertices will have a strategy that benefits them the most. Although the less important ones may not benefit from the best strategy, the global impact on performances will be smaller. In order to justify this reordering, we formulate our algorithm as a homomorphism that presents the benefit of being computable in any order.

To do so, we will first introduce how the cost may be computed from a homomorphism of a vertex list in sec. 3.3. However, the data-flow representation of the computation is not a list but a directed acyclic graph (DAG). Morihata [21] showed that the homomorphism theory (and especially its third theorem) might be extended to trees. The only difference between a tree and a DAG lies in the number of parents (outputs) that may be more than 1 in a DAG which leads to the existence of several possible paths from one vertex to another. To remove the existence of these different paths, we propose to consider the spanning tree of the DAG that would select only one of those paths for each case.

3.1 Notations

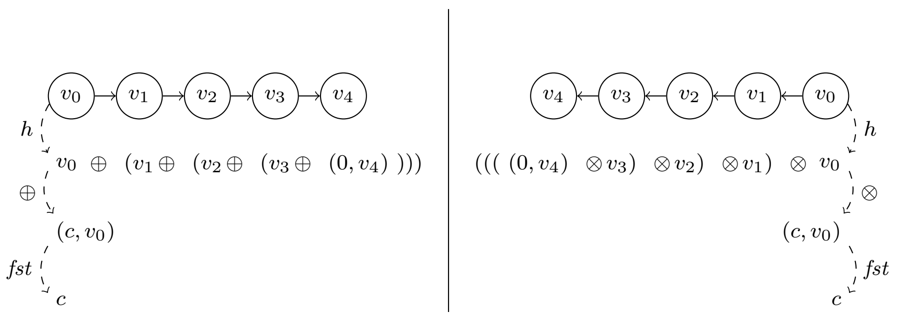
S is the set of all strategies of all vertices in G. We note s_i the strategy of vertex v_i in S. The cost of the computational graph was given by equation (3).

$$\begin{aligned} cost_{op}(v_i, s_i) &= w \times q_x(v_i, s_i) + g \times \big(q_c(v_i, s_i) + q_p(v_i, s_i)\big) \\ cost_{rdst}(v_i, v_j, s_i, s_j) &= g \times q_r(e_{ij}, s_i, s_j) \end{aligned} \tag{3}$$

This way, equation (3) can be rewritten

$$C(G, S) = \sum_{v_i \in V} cost_{op}(v_i, S_{v_i}) + \sum_{(v_i, v_j) \in E} cost_{rdst}(v_i, v_j, S_{v_i}, S_{v_j}) \tag{4}$$

Remark that if $S_{v_j} \notin S$ (because v_j has not been seen yet) then $cost_{rdst}(v_i, v_j, S_{v_i}, S_{v_j}) = 0$. Assume function $strt(v)$ gives the set of possible strategies for a given vertex v. The strategy generation consists of taking for each vertex the strategy that minimizes its cost. It may be expressed recursively as



v_0
v_1
v_2
v_3
v_4
h
$v_0 \oplus (v_1 \oplus (v_2 \oplus (v_3 \oplus (0, v_4))))$
$\oplus$
(c, v_0)
fst
c
$(((((0, v_4) \otimes v_3) \otimes v_2) \otimes v_1) \otimes v_0$
$\otimes$

(a) Leftward processing of $cost_{all\_rdst}$ (b) Rightward processing of $cost_{all\_rdst}$

Fig. 4 Leftward and rightward processing of $cost_{all\_rdst}$ over a vertex list

$$\begin{aligned}
cost_v(v_i, s_i, S, G) &= cost_{op}(v_i, s_i) + \textstyle\sum_{(v_i,v_j)\in E} cost_{rdst}(v_i, v_j, s_i, s_j) \\
search(v_i, S, G) &= S \cup \{s_i\} \text{ such that } s_i \in strt(v_i) \text{ minimizes } cost_v(v_i, s_i, G, S) \\
S_0 &= \emptyset \\
S_{i<|V|} &= search(v_i, S_{i-1}, G)
\end{aligned} \tag{5}$$

The remaining of this section will express equations (5) as a homomorphism. As a preliminary, we introduce the following notations taken from [22]. $hom(\oplus, f, a)(l)$ is a homomorphism that maps function f on each element of l before reducing with the binary operator $\oplus$ whose first application will be with initialization element a. For example

$$hom(\oplus, f, a)([x, y, z]) = a \oplus f(x) \oplus f(y) \oplus f(z)$$

We note α *list* the type of list of elements of type α. Function will be noted in curry notation. For example, $f : A \to B \to C$ is the function f, that when applied to an argument of type A will produce a function of type $B \to C$ that when applied to an argument of type B will produce a value of type C. We use $+\!\!+ : \alpha\ list \to \alpha\ list \to \alpha\ list$ as the concatenation operator. The function composition is noted $(f \circ g)(x) = f(g(x))$. To access elements of a couple, we use $fst(x, y) = x$ and $snd(x, y) = y$.

3.2 Redistribution cost as a homomorphism

We define the leftward operation $cost_{all\_rdst}$ to compute all redistribution cost of a linear graph (list). A leftward operation is a recursive operation for which the elements are added to the left (input) side (see fig. 4(a)). To be a leftward operation, $cost_{all\_rdst}$ needs to be defined in the form

$$\begin{aligned}&h([v] \mathbin{+\!\!+} x) = v \oplus h(x)\\&\text{with}\\&\qquad h \ : \ \alpha \ list \to \beta \ list\\&\qquad \oplus \ : \ \alpha \to \beta \to \beta\end{aligned}$$

We instantiate this form with the operation $cost_{all\_rdst}$ leftward

$$\begin{aligned}cost_{all\_rdst} &= fst \circ h\\h([v] \mathbin{+\!\!+} x) &= v \oplus h(x)\\h([v_0]) &= (0, v_0)\\v_i \oplus (c, v_j) &= (c + cost_{rdst}(v_i, v_j, s_i, s_j), v_i)\end{aligned} \tag{6}$$

Suppose that S, for which $s_i, s_j \in S$, is a global constant fixed for the whole cost computation. As edges direction have no influence on the redistribution cost, function $cost_{all\_rdst}$ is also computable rightward (in the output direction, as illustrated in fig. 4(b)). A rightward operation must respect the form

$$\begin{aligned}&h(x \mathbin{+\!\!+} [v]) = h(x) \otimes v\\&\text{with}\\&\qquad h \ : \ \alpha \ list \to \beta \ list\\&\qquad \otimes \ : \ \beta \to \alpha \to \beta\end{aligned}$$

We instantiate this form with the operation $cost_{all\_rdst}$ rightward

$$\begin{aligned}cost_{all\_rdst} &= fst \circ h\\h(x \mathbin{+\!\!+} [v]) &= h(x) \otimes v\\h([v_0]) &= (0, v_0)\\(c, v_j) \otimes v_i &= (c + cost_{rdst}(v_i, v_j, s_i, s_j), v_i)\end{aligned} \tag{7}$$

Thus, by the third homomorphism theorem [22], $cost_{all\_rdst}$ is a homomorphism because it is defined both leftward (eq. (6)) and rightward (eq. (7)).

3.3 Cost and strategy generation as a homomorphism

The intra-communication cost is defined directly as the homomorphism

$$cost_{all\_op} = hom(+, cost_{op}, 0)$$

Hence, the whole cost of the graph may be defined as the addition of the two homomorphisms

$$cost_l(l) = cost_{all\_op}(l) + cost_{all\_rdst}(l)$$

The two may also be computed together as

$$\begin{aligned}
cost_l &= fst \circ h \\
h([v] +\!\!+\, x) &= v \oplus h(x) \\
h([v_0]) &= (cost_{op}(v_0, s_0), v) \\
v_i \oplus (c, v_j) &= (c + cost_{rdst}(v_i, v_j, s_i, s_j) + cost_{op}(v_i, s_i), v_i)
\end{aligned} \tag{8}$$

and symmetrically for the rightward notation.

$$\begin{aligned}
cost_l &= fst \circ h \\
h(x +\!\!+ [v]) &= h(x) \otimes v \\
h([v_0]) &= (cost_{op}(v_0, s_0), v) \\
(c, v_j) \otimes v_i &= (c + cost_{rdst}(v_i, v_j, s_i, s_j) + cost_{op}(v_i, s_i), v_i)
\end{aligned} \tag{9}$$

In this case too, the third homomorphism theorem tells us that $cost_l$ is a homomorphism because it is defined both leftward (eq. (8)) and rightward (eq. (9)). Now that we have shown how the cost may be formulated as a homomorphism, the strategy generation may be in turn written leftward

$$\begin{aligned}
S &= fst \circ h \\
h([v] +\!\!+\, x) &= v \oplus h(x) \\
h([v_0]) &= \big(search(v_0, \emptyset, []), [v_0]\big) \\
v_i \oplus (S, L) &= \big(search(v_i, S, L), v_i +\!\!+\, L\big)
\end{aligned} \tag{10}$$

and rightward

$$\begin{aligned}
cost_l &= fst \circ h \\
h(x +\!\!+ [v]) &= h(x) \otimes v \\
h([v_0]) &= \big(search(v_0, \emptyset, []), [v_0]\big) \\
(S, L) \otimes v_i &= \big(search(v_i, S, L), v_i +\!\!+\, L\big)
\end{aligned} \tag{11}$$

The strategy generation being a homomorphism allows us to state that the vertex list (linear graph) may be processed in any order. This homomorphism may be extended to trees thanks to the work of Morihata [21]. Trees differ from DAGs in the sense that a node may have several parents in the case of a DAG. To address this issue, we treat the DAG as its spanning tree (only pick one parent of each node) while computing the redistribution cost with respect to all of the DAG edges. This allows us to treat any DAG cases, but our proof that the reordering of the vertices preserves the cost cannot be extended to DAGs.

4 Evaluation

4.1 Experiment environment

DNN models: We evaluate SMSG on real-world DNN models:

- Computer Vision:
  - ResNet50 with Cifar10 dataset,
  - ResNet50/101/152 [2] with ImageNet dataset,
  - Fully Convolution Network (FCN) [23];
- Recommendation Systems:
  - Wide&Deep [3] with Criteo dataset;
- Neural Language Processing:
  - BERT [14],
  - PanGu-Alpha 2.6B and 13B [24] (B stands for billion, which signifies the size of parameters),
  - T5 [15] (Text-to-Text Transfer Transformer) with dedicated text dataset respectively.

Evaluation metric: We choose the average *step time* to compare the performance of the hybrid parallel strategies. Step time is the training time of one batch of data, including the FPG and BPG, which is inversely proportional to throughput. Shorter step time denotes better performance.

Hardware environments: The experiments are conducted on an Atlas900 AI cluster [25]. Each Atlas node is composed of eight Ascend910 accelerators. Our experiments test until 32 accelerators where the four nodes are connected with a 32-port switch. All the Ascend910 clusters are interconnected directly, even from a different node. We also implemented an 8 NVIDIA-V100 GPU cluster as a control group to show the better portability of our approach.

Deep Learning Framework: Our experiments are conducted on MindSpore, which supports automatically lancer distributed training with a given strategy. The strategy found by SMSG will be taken as an input for Mindspore, and the training will be conducted on this DL framework.

Searching Algorithm: SMSG offers the ability to modeling the cost and compute it with the homomorphism. As for the searching algorithm, we implemented a linear-complexity searching algorithm D-Rec [26] for the experiments in this section.

Baseline: We choose Expert-Designed strategies for each specific real-world DNN model as the baseline and compare its step time with that of the strategy found by SMSG. To show the better portability of SMSG than the profiling-based approaches, we choose TensorOpt [8] as the competitive approach, which is the SOTA approach in 2021.

4.2 Generality

We tested the quality of the hybrid parallel strategy found by SMSG in an extensive range of real-world DNN models. Table 1 shows the average step times of training varieties of DNN models with Expert-Designed strategies and the strategies found by SMSG. The last column shows the performance percentage of SMSG to Baseline (high than 100% denotes a better performance). It can be found that the minimum performance percentage of SMSG to the baseline

Performance: step time/ms (8 Ascends)				
DNN models		**Baseline**	**SMSG**	Percentage
CV	ResNet50-cifar	48.58	45.91	105.83%
	ResNet50-ImageNet	57.53	61.18	94.03%
	ResNet101-ImageNet	86.73	93.38	92.88%
	ResNet152-ImageNet	120.57	127.46	94.59%
	FCN	485	512	94.72%
Rec.Sys.	Wide&Deep	21.6	22.38	96.51%
NLP	BERT	110.63	122.38	90.40%
	PanGu-Alpha 2.6B	4826	4876	98.91%
	PanGu-Alpha 13B	13990	13988	100.01%
	T5	1288	1279	100.70%

Table 1 Performance on varieties of DNN models

is 90.40% for the BERT model. The experiments show that SMSG can find good hybrid strategies for varieties of real-world DNN models with a correct performance higher than 90% to the Expert-Designed strategies.

The hardware parameters and quantity functions are separated thanks to the profiling-free modelling. SMSG only profiles the communication and computation of the 8 Ascends cluster one time (it takes a few minutes), and the hardware parameter values can be generally used for all the DNN models. The DNN models are different compositions of the 20 kinds of operators. SMSG shows its generality in searching for varieties of DNN models with the predefined quantity functions and the one-time profiled cluster parameters.

4.3 Portability

The experiments in this subsection demonstrate the portability of SMSG and the profiling-based approach when the training environments are changed. For the profiling-based approaches like TensorOpt, profiling the operators under different parallel configurations of typical DNN models usually takes more than one day. On the contrary, for SMSG, profiling a hardware configuration takes only some minutes. In this section, the results of SMSG shown in Table 2 and Table 3 are obtained with the profiled communication and computation capacity on targeted hardware architecture because it only takes some minutes. However, for the TensorOpt, we kept one profiling base and varied the training configurations to show the impact of the profiling data. We choose two typical DNN models, ResNet152-ImageNet and Wide&Deep, to test.

Table 2 shows the percentage performance of TensorOpt and SMSG w.r.t. the number of cluster devices. Both for ResNet and Wide&Deep, it can be easily conducted that with the increase in devices numbers, the quality of strategies found by TensorOpt decreases because they do not have enough profiling data of the possible partition dimensions, so they missed the optimal strategies. However, the profiling-free approach SMSG can keep a good strategy quality because of the leveraged profiling time.

Performance: Percentage w.r.t the Baseline			
DNN models	Dev. Num.	TensorOpt (Profiled with 8 Ascend)	SMSG
ResNet152-ImageNet	8	93.16%	94.59%
	16	79.15%	110.74%
	32	63.25%	90.17%
Wide&Deep	8	96.23%	96.51%
	32	59.96%	102.82%

Table 2 Portability w.r.t. the scale of cluster

Performance: Percentage w.r.t the Baseline				
Profiling Base	DNN models	TensorOpt (8 Ascends)	TensorOpt (8 GPUs)	SMSG
8 GPUS	ResNet152-ImageNet	62.12%	99.18%	99.53%
	Wide&Deep	49.55%	98.25%	98.33%
8 Ascend	ResNet152-ImageNet	98.16%	71.56%	94.59%
	Wide&Deep	97.23%	66.89%	96.51%

Table 3 Portability w.r.t. hardware architecture

Same conduction can be made from Table 3 that searching the strategy with different profiled data from a different architecture for TensorOpt, the decrease of strategy quality is evident, while SMSG keeps good results. The cost model of TensorOpt is based on the profiled execution time of operators on the actual hardware. The execution time of an operator with the same parallel strategy is different on GPUs and Ascends. That is why the strategy quality of TensorOpt decreases when executed on GPUs with profiling data on Ascends. Heavy profiling tasks (a few days) limit the portability of these profiling-based approaches, while SMSG with a lightened profiling job is more practical.

5 Conclusion

This paper proposes SMSG, a profiling-free strategy generation method for distributed DNN training. The main idea of SMSG is that its symbolic cost model is built based on the relative cost instead of the execution time. The hardware parameters and similar data quantity functions are separated. The cost functions can be optimized independently from the hardware. Besides, we introduce homomorphism to re-organize the redistribution cost so that the searching algorithm does not have graph topology dependency. These two contributions offer us the generality and portability for the DNN parallel strategy generation.

References

1. T. Brown, B. Mann, N. Ryder, et al., in *Advances in Neural Information Processing Systems*, vol. 33 (Curran Associates, Inc., 2020), vol. 33, pp. 1877–1901. URL

https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

2. K. He, X. Zhang, S. Ren, J. Sun, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778. DOI 10.1109/CVPR.2016.90
3. H.T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, H. Shah, in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (Association for Computing Machinery, New York, NY, USA, 2016), DLRS 2016, p. 7–10. DOI 10.1145/2988450.2988454. URL https://doi.org/10.1145/2988450.2988454
4. A. Krizhevsky, I. Sutskever, G.E. Hinton, in *Advances in neural information processing systems* (2012), pp. 1097–1105
5. J. Dean, G.S. Corrado, R. Monga, et al., in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Curran Associates Inc., Red Hook, NY, USA, 2012), NIPS'12, pp. 1223—-1231
6. Y. Huang, Y. Cheng, A. Bapna, O. Firat, M.X. Chen, D. Chen, H. Lee, J. Ngiam, Q.V. Le, Y. Wu, Z. Chen, *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism* (Curran Associates Inc., Red Hook, NY, USA, 2019)
7. M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, arXiv preprint arXiv:1909.08053 (2019)
8. Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, F. Yu, IEEE Transactions on Parallel and Distributed Systems **33**(8), 1967 (2022). DOI 10.1109/TPDS.2021.3132413
9. S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, W. Lin, in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Association for Computing Machinery, New York, NY, USA, 2021), PPoPP '21, p. 431–445. DOI 10.1145/3437801.3441593. URL https://doi.org/10.1145/3437801.3441593
10. J.M. Tarnawski, D. Narayanan, A. Phanishayee, in *Advances in Neural Information Processing Systems*, vol. 34, ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, J.W. Vaughan (Curran Associates, Inc., 2021), vol. 34, pp. 24,829–24,840. URL https://proceedings.neurips.cc/paper/2021/file/d01eeca8b24321cd2fe89dd85b9beb51-Paper.pdf
11. Z. Jia, M. Zaharia, A. Aiken, in *Proceedings of Machine Learning and Systems*, vol. 1, ed. by A. Talwalkar, V. Smith, M. Zaharia (2019), vol. 1, pp. 1–13. URL https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf
12. Z. Jia, S. Lin, C.R. Qi, A. Aiken, (PMLR, 2018), *Proceedings of Machine Learning Research*, vol. 80, pp. 2274–2283. URL http://proceedings.mlr.press/v80/jia18a.html
13. M. Wang, C.c. Huang, J. Li, (Association for Computing Machinery, New York, NY, USA, 2019), EuroSys '19. DOI 10.1145/3302424.3303953. URL https://doi.org/10.1145/3302424.3303953
14. J. Devlin, M.W. Chang, K. Lee, K. Toutanova, in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (Association for Computational Linguistics, Minneapolis, Minnesota, 2019), pp. 4171–4186. DOI 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423
15. C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P.J. Liu, Journal of Machine Learning Research **21**(140), 1 (2020). URL http://jmlr.org/papers/v21/20-074.html
16. M. Abadi, P. Barham, J. Chen, et al., in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (USENIX Association, Savannah, GA, 2016), pp. 265–283. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi
17. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., in *Advances in neural information processing systems* (2019), pp. 8026–8037
18. *MindSpore*. https://www.mindspore.cn/
19. H. Wang, C. Li, T. Tachon, H. Wang, S. Yang, S. Limet, S. Robert, in *European Conference on Parallel Processing* (Springer, 2021), pp. 201–216

20. L.G. Valiant, Journal of Computer and System Sciences **77**(1), 154 (2011)
21. A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, SIGPLAN Not. **44**(1), 177–185 (2009). DOI 10.1145/1594834.1480905. URL https://doi.org/10.1145/1594834.1480905
22. J. Gibbons, Journal of Functional Programming **6**(4), 657–665 (1996). DOI 10.1017/S0956796800001908
23. A. Schwing, R. Urtasun, (2015)
24. W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang, C. Li, Z. Gong, Y. Yao, X. Huang, J. Wang, J. Yu, Q. Guo, Y. Yu, Y. Zhang, J. Wang, H. Tao, D. Yan, Z. Yi, F. Peng, F. Jiang, H. Zhang, L. Deng, Y. Zhang, Z. Lin, C. Zhang, S. Zhang, M. Guo, S. Gu, G. Fan, Y. Wang, X. Jin, Q. Liu, Y. Tian, CoRR **abs/2104.12369** (2021). URL https://arxiv.org/abs/2104.12369
25. Atlas900. https://e.huawei.com/en/products/cloud-computing-dc/atlas/atlas-900-ai
26. H. Wang, C. Li, T. Tachon, H. Wang, S. Yang, S. Limet, S. Robert, in *European Conference on Parallel Processing* (Springer, 2021), pp. 201–216