HLPP 2020
13th International Symposium on High-level Parallel Programming and Applications
Porto, Portugal, July 9-10, 2020

Proceedings of the 13th International Symposium on High-Level Parallel Programming and Applications

Porto, Portugal

July 9–10, 2020

Miguel Areias Inês Dutra Ricardo Rocha
(Eds.)

Preface

This volume contains the proceedings of the 13th edition of HLPP, the International Symposium on High-Level Parallel Programming and Applications, initially planned to take place in the Department of Computer Science, Faculty of Sciences, University of Porto, Portugal, during July 9–10, 2020. Due to the Coronavirus disease (COVID-19) outbreak and following the recommendations/guidelines from the World Health Organization and the European Centre for Disease Prevention and Control, the event was made fully virtual, with synchronous/asynchronous online activities, but maintaining the regular publication and presentation activities.

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, 9 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 2020 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 the Núcleo de Tecnologias Educativas at FCUP and the Zoom and Slack platforms 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 2020.

July 2020,

Miguel Areias
Inês Dutra
Ricardo Rocha

Organization

Steering Committee

Alexander Tiskin	University of Warwick, UK
Clemens Grelck	Universiteit van Amsterdam, Netherlands
Frederic Loulergue	Northern Arizona University, USA
Gaétan Hains	Huawei Technologies Paris, France
Kiminori Matsuzaki	Kochi University of Technology, Japan
Quentin Miller	Somerville College Oxford, UK

Program Chairs

Miguel Areias	University of Porto, Portugal
Inês Dutra	University of Porto, Portugal
Ricardo Rocha	University of Porto, Portugal

Publicity Chair

Carlos Ferreira	Polytechnic Institute of Porto, Portugal

Program Committee

Agostino Dovier	University of Udine, Italy
Aleksandar Prokopec	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Ana Lucia Varbanescu	University of Amsterdam, Netherlands
Christoph Kessler	Linköping University, Sweden
Clemens Grelck	University of Amsterdam, Netherlands
Dalvan Griebler	PUCRS/SETREM, Brasil
Frank Pfenning	Carnegie Mellon University, USA
Frederic Loulergue	Northern Arizona University, USA
Frédéric Dabrowski	LIFO - Université d'Orléans, France
Gaetan Hains	Huawei Paris Research Center, France
Herbert Kuchen	University of Münster, Germany
Joel Falcou	Univeristé Paris Sud, France
Kiminori Matsuzaki	Kochi University of Technology, Japan
Kostis Sagonas	Uppsala University, Sweden
Marco Aldinucci	University of Torino, Italy
Massimo Torquati	University of Pisa, Italy
Michel Steuwer	University of Glasgow, UK
Murray Cole	The University of Edinburgh, UK
Peter Kilpatrick	Queen's University Belfast, UK

External Reviewers

Alberto Riccardo Martinelli, Arvid Jakobsson, Breno Menezes, Iacopo Colonnelli, Nina Herrmann, and Thibaut Tachon

Web Page

`https://hlpp2020.dcc.fc.up.pt`

Sponsors

Table of Contents

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

Bounds Checking on GPU

Troels Henriksen

Abstract We present a simple compilation strategy for safety-checking array indexing in high-level languages on GPUs. Our technique does not depend on hardware support for abnormal termination, and is designed to be efficient in the non-failing case. We rely on certain properties of array languages, namely the absence of arbitrary cross-thread communication, to ensure well-defined execution in the presence of failures. We have implemented our technique in the compiler for the functional array language Futhark, and an empirical evaluation on 19 benchmarks shows that the geometric mean overhead of checking array indexes is respectively 4% and 6% on two different GPUs.

Keywords GPU · functional programming · compilers

1 Introduction

Programming languages can be divided roughly into two categories: *unsafe* languages, where programming errors can lead to unpredictable results at run-time; and *safe* languages, where all risky operations are guarded by run-time checks. Consider array indexing, where an invalid index will lead an unsafe language to read from an invalid memory address. At best, the operating system will stop the program, but at worst, the program will silently produce invalid results. A safe language will perform *bounds checking* to verify that the array index is within the bounds of the array, and if not, signal that something is amiss. Some languages perform an *abnormal termination* of the program and print an error message pointing to the offending program statement. Other languages throw an exception, allowing the problem to be handled by the program itself. The crucial property is that the resulting behaviour is well-defined. We use array indexing as the motivating example, but we are concerned with

Troels Henriksen
University of Copenhagen
E-mail: athas@sigkill.dk

all safety checks that can be condensed to a single boolean expression; for example integer division by zero.

Users of unsafe languages are often wary of the run-time overhead of performing safety checks. However, it has been known since even the early days of high-level languages that bounds errors are easy to make and can have disastrous consequences [15], and hence most languages provide at least the option of automatically checking risky operations. In this paper we distinguish *failures* from *errors*. A *failure* is a checked operation that fails in some controlled manner. For example an array index that is discovered to be out-of-bounds. An *error* is a misuse of some low-level API or language construct that causes undefined behaviour. For example, writing to an invalid address. A safe programming language must ensure that anything that would be an error will instead become a failure.[1]

GPUs have long been popular for general-purpose parallel programming, and several high-level languages support compilation to GPU, including Accelerate [5], Lift [19], Julia [4], X10 [7], Harlan [16], APL [17, 12], and SaC [11]. As these are all high-level languages, most of them provide at least the option of performing bounds checking when running on a CPU, but none of them can perform bounds checking in generated GPU code. One important reason is that the most popular GPGPU APIs (OpenCL and CUDA) do not provide good support for abnormal termination of a running GPU kernel.

For example, CUDA provides an `assert()` macro that, if it fails, will terminate the calling kernel. However, it will also invalidate the entire CUDA driver context, meaning that memory that has been copied to the GPU by the current process becomes unavailable. Further, the error message will be printed to the standard error stream, which may be difficult to capture and propagate (e.g. by throwing an exception on the CPU). This means failures cannot be handled in any way other than completely scrubbing the entire GPU state, including even data that was not available to the failing kernel, and restarting from scratch, which is often not acceptable. While a single GPU thread can always terminate itself, this can introduce deadlocks (see section 2.3), which is an error. OpenCL is similar, except there is *no* way for a GPU thread to abnormally terminate an entire running kernel.

Functional array languages [3], where programs consist primarily of bulk operations such as `map`, `reduce`, and rank-polymorphic "vectorised" operators, do not contain indexing errors, as such operations are guaranteed to be in-bounds. However, some algorithms do still require ad hoc indexing, in particular when we use arrays to encode more complicated structures, such as graphs. In particular, parallel "gather" and "scatter" operations have all the same risks as traditional scalar array indexing, and should therefore be checked at run-time. Fortunately, as we shall see, array languages based on bulk operations have certain properties that enable a particularly efficient implementation of run-time safety checks.

[1] The error/failure nomenclature is more or less arbitrary and not standard or common, but the distinction is important for this paper. In C, the term *undefined behaviour* is a close (but not exact) analogue to what we call *errors*.

The contribution of this paper is a compilation strategy for inserting safety checks in GPU code generated by compilers for high-level parallel languages, without relying on support for abnormal termination or error reporting in the GPU API or hardware itself. Our design goals are the following:

Completeness: all possible safety checks that are expressible as a boolean expression can be handled.

Efficiency: overhead must be low, as programmers are quick to turn off safety checks that they believe are detrimental to performance. However, we focus only on performance of the *non-failing* case, as we assume failures are rare and exceptional situations.

Robustness: the GPU driver context must remain operational even in the presence of safety check failures—*errors* must not occur, as far as the GPU programming API is concerned.

Quality of reporting: we must be able to produce accurate information about the source of the failure, phrased in terms of original high-level language (e.g. the failing expression and index) rather than low-level details (e.g. the invalid address).

Note that we are not claiming to safety-check GPU kernels hand-written in low-level languages such as CUDA and OpenCL. Our strategy depends on properties that are straightforward to guarantee in code generated by compilers for deterministic parallel programming languages, but which would not hold for languages that support programmer-written low-level communication between threads.

1.1 Prior Work

Several tools for detecting invalid memory accesses have been implemented for GPUs. Oclgrind [18] presents itself as an OpenCL platform which runs all kernels in an interpreter and reports accesses to invalid memory locations. However, Oclgrind is primarily a debugging tool, as it runs far slower than real hardware. NVIDIA provides the similar tool `cuda-memcheck`, which detects invalid memory accesses, but as the instrumented kernels run with a significant run-time overhead, it is also a debugging tool, and not intended for code running in production.

A more efficient (and less precise) tool is the vendor-agnostic clARMOR [9], which surrounds every allocation with an area of unique *canary values*. If at any point any of these values have been changed, then it must be because of an out-of-bounds write. The overhead is minor (10% on average), but clARMOR can detect only invalid writes, not reads, and cannot identify exactly *when* the invalid access occurred.

All these tools are low-level and concerned with the semantics of OpenCL or CUDA kernel code, and so are not suitable for implementing bounds checking for a high-level language. In particular, they would not be able to live up to our expectations for error messages. Further, all such tools run the risk of *false* *negatives*, where a bounds failure ends up corrupting memory at an address that is valid, but unintended. This cannot be detected by tools that merely verify addresses.

There is also the option of using entirely static techniques to perform bounds checking, such as dependent types [21], which demand that the programmer provides a proof that all indexing is safe before the type checker accepts the program. No checking is then needed at run-time. However, dependently typed programming languages are still an active research topic with regards to both programming ergonomics and run-time performance, and so are not necessarily a good choice in the near future for performance-oriented languages. In either case, such techniques are complimentary to run-time checking: where the programmer is willing to invest the time to provide a proof of safety, we can turn off run-time checks, while keeping checks in the unverified parts of the program.

A closely related problem is *de-optimisation* in the context of JIT compilation, where an assumption made by the JIT compiler may turn out to be false at run-time, and execution must be rolled back in order run a slower interpreted version of the code. Prior work on JIT compiling R to GPU code [10] uses a technique similar to the approach we will be discussing in section 2, but without the optimisations of section 2.2 and section 3, and of course also without error messages.

1.2 Nomenclature and Technicalities

We use OpenCL terminology for GPU concepts. Despite the naming differences with CUDA, the concepts are identical, and our approach works just as well with CUDA as with OpenCL. An OpenCL *work-group* corresponds to what CUDA calls a *thread block*, and is a collection of threads that executes together and may communicate with each other. OpenCL *local memory* corresponds to CUDA *shared memory.*

We are assuming a particularly simple and conventional GPU model, with the GPU operating as a co-processor that merely receives commands and data from the CPU. In particular, we assume a kernel cannot enqueue new kernels, and cannot allocate or free memory. Some real GPUs do have these capabilities, but they have significant performance caveats, are not crucial to GPU programming, and in particular are not used in the code generated by any of the previously mentioned high-level languages.

2 Design and Implementation

As currently popular GPGPU APIs (e.g. OpenCL and CUDA) do not permit abnormal termination of GPU kernels, we need to turn failing executions into normal kernel termination, and somehow communicate the failure back to the CPU. It is important that we do not at any point execute errors, such as reading from invalid memory addresses.

A simple solution is to allocate a single 32-bit integer in GPU memory, which we call `global_failure`, and which we use to track failures. We use the convention that a value of -1 means "no failure", and other values indicate that a failure has occurred. When a failure occurs, the failing GPU thread writes a non-negative integer to `global_failure` and immediately `return`s, which stops the thread. After every kernel execution, we can then copy the value of `global_failure` back the CPU, and if it contains a non-negative value, propagate the failure using conventional CPU mechanisms, such as throwing an exception or printing an error message. This idea is the foundation of our approach, but in this simple form it has significant problems:

1. It is *uninformative*, because simply knowing that the program failed is not enough to provide a good error message. For an array indexing failure, we usually wish to provide the expression that failed, the attempted index, and the size of the array.
2. It is *slow*, because it requires a global synchronisation after every kernel execution, to verify whether it is safe to execute the next kernel. GPUs perform well when given a large queue of work to process at their own pace, not when they constantly stop to transfer 32 bits back to the CPU for inspection, and have to wait for a go-ahead before proceeding.
3. It is *wrong*, because GPU threads are not isolated, but may communicate through barriers or other synchronisation mechanisms. In particular, it is undefined behaviour for a barrier to be executed by *at least one* but *not all* threads. We cannot in general abort the execution of a single thread without risking errors.

We will now explain at a high level how to address these problems, accompanied by a sketch of a concrete OpenCL implementation.

2.1 Better Failure Information

Treating failure as a boolean state, without revealing the source of the failure, is not very user-friendly. Our solution is to assign each distinct *failure point* in the program a unique number: a *failure code*. A failure point is a program location where a safety check is performed. If the check fails, the corresponding failure code is written to `global_failure`. The write is done with atomic compare-and-swap to ensure that any existing failure code is not clobbered. The distinguished value -1 indicates that no failure has yet occurred. We use compare-and-swap to ensure that at most one thread can change `global_failure` from -1 to a failure code. This implies that if multiple threads contain failures, it is not deterministic which of them get to report it. We can only report a single failure to the user, and multiple runs of the same failing program may produce different error messages. During compilation of the original program, we construct a table that maps failure codes to the original source code locations. When we check `global_failure` at run-time on the CPU, we can then identify the exact expression that gave rise to the failure.

To provide human-readable error messages, we associate each failure point with a `printf()`-style format string such as the following:

```
"index %d out of bounds for array of size %d"
```

For simplicity we assume that `%d` is the only format specifier that can occur, but each distinct format string can contain a different number of format specifiers. We then pre-allocate an `int` array `global_failure_args` in GPU memory that is big enough to hold all parameters for the largest format string. When a thread changes `global_failure`, it also writes to `global_failure_args` the integers corresponding to the format string arguments. When the CPU detects the failure after reading `global_failure`, it uses the failure code to look up the corresponding format string and instantiates it with arguments from `global_failure_args`. Note that the CPU only accesses `global_failure_args` when a failure has occurred, so performance of the non-failing case is not affected.

For a non-recursive language, each failure point can be reached through a finite number of different call paths, and stack traces can be provided by generating a distinct format string for each possible path. The recursive case is more difficult, and outside the scope of this paper, but can possibly be handled by simply deciding on a maximum backtrace length, and then representing a failure point as an entire array of source locations, rather than a single one.

2.2 Asynchronous Failure Checking

It is expensive to check `global_failure` on the CPU after every kernel execution. We should do so only when we are, for other reasons, required to synchronise with the GPU. This is typically whenever we need to copy data from GPU to CPU, such as when making control flow decisions based on GPU results, or when we need the final program result.

Simply delaying the check is not safe, as kernel $i+1$ may contain unchecked operations that are safe if and only if the preceding kernel i completed successfully. To address this, we add a prelude to every GPU kernel body where each thread checks `global_failure`. If `global_failure` contains a failure code, that must mean one of the preceding kernels has encountered a failure, and so the all threads of the current kernel terminate immediately. This is an improvement, because checking `global_failure` on the GPU is much faster than checking it on the CPU, and does not involve any CPU/GPU synchronisation. The overhead is a single easily cached global memory read for every thread, which is in most cases negligible, and section 3.5 shows cases where even this can be elided.

This technique means that an arbitrary (but finite) amount of time can pass from the time that a GPU kernel writes to `global_failure`, to the time that the failure is reported to the CPU. Specifically, the time is bounded in the worst case by the time it would have taken the program to finish successfully. We consider this an easy price to pay in return for improving the performance of the non-failing case.

```
 1  kernel sum
 2    (int *global_failure,
 3     int n, int *js,
 4     int m, int *vs,
 5     ...) {
 6    local int sums[GROUP_SIZE];
 7    int gtid = get_global_id(0);
 8    int ltid = get_local_id(0);
 9    int k = get_global_size(0);
10    int acc = 0;
11    for (int i = gtid;
12         i < n;
13         i += k) {
14      int j = js[i];
15      if (j < 0 || j >= m) {
16        *global_failure = 1;
17        return;
18      }
19      acc += vs[j];
20    }
21
22    sums[ltid] = acc;
23
24
25    barrier();
26
27
28
29
30
31    // Perform parallel
32    // reduction of sums...
33  }
```

(a) Incorrect failure handling.

```
 1  kernel sum
 2    (int *global_failure,
 3     int n, int *js,
 4     int m, int *vs,
 5     ...) {
 6    local int sums[GROUP_SIZE];
 7    int gtid = get_global_id(0);
 8    int ltid = get_local_id(0);
 9    int k = get_global_size(0);
10    int acc = 0;
11    for (int i = gtid;
12         i < n;
13         i += k) {
14      int j = js[i];
15      if (j < 0 || j >= m) {
16        *global_failure = 1;
17        goto sync;
18      }
19      acc += vs[j];
20    }
21
22    sums[ltid] = acc;
23
24    sync:
25    barrier();
26    if (*global_failure != -1) {
27      return;
28    }
29    barrier();
30
31    // Perform parallel
32    // reduction of sums...
33  }
```

(b) Correct failure handling.

Fig. 1: OpenCL-like pseudo-code for kernel that sums an indirectly indexed array of integers.

2.3 Cross-thread Communication

In general, a GPU thread cannot safely terminate its own execution, as other threads may be waiting for it in a barrier. Consider fig. 1a, which shows a typical OpenCL summation kernel, where each thread performs a sequential summation of a chunk of the input, followed by by a parallel summation of the per-thread results within the GPU work-group. This requires a *barrier* (line 25) to ensure that all threads have written their result to the shared `sums` array before the parallel reduction takes place.

In this kernel, the sequential part involves indirect indexing, where the array `js` contains values used to index `vs`. These indexes can be out-of-bounds, which on fig. 1a is handled by setting `global_failure` and terminating the thread

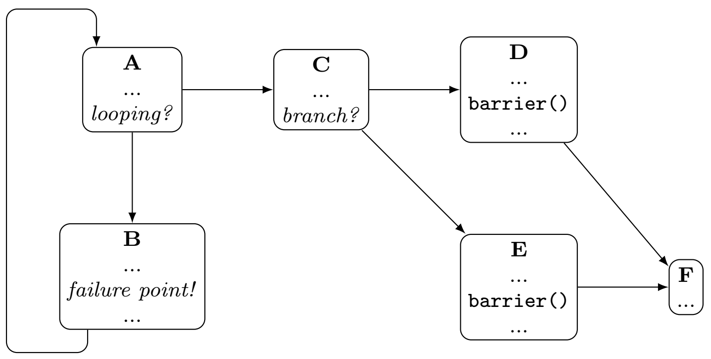


Fig. 2: Control flow graph where the failure point in node **B** cannot know the location of the next barrier.

with `return`. But this is risky, as other threads may already be waiting at the barrier, which will never be reached by the failing thread.

The solution, shown on fig. 1b, identifies the location of the next barrier in the code, places a distinct label just before it (line 24), and then `goto` that label instead of immediately terminating. We call this label/barrier pair a *synchronisation point*. Immediately after the synchronisation point, *all* threads in the work-group check whether any of them have failed, by inspecting `global_failure`, and terminate if so. The barrier implies a memory fence, so the threads within the work-group will have a consistent view of the `global_failure` variable. We also need a barrier immediately after this check, because other failure points may occur in the remainder of the kernel, and these would also set `global_failure`. A kernel may have multiple synchronisation points, each identified with a distinct label. We must ensure that all kernels end with a final synchronisation point, such that there is always a place to jump from a failure point.

Viewed as a control flow graph, the kernel code must have the property that every node that controls a failure point has a postdominator that contains a synchronisation point, and that there are no barriers on the path to the synchronisation point. It is crucial that this is a postdominator, because even non-failing executions must reach it. This property does not hold for arbitrary GPU kernels, but it is straightforward to ensure it when compiling array languages, because all cross-thread communication (and hence, barriers) is implicit in the source program and controlled by the compiler. For example, consider the control flow graph on fig. 2. If a failure occurs in node **B**, where should we jump? The choice of the next barrier is not decided until node **C**. The compiler must insert a synchronisation point in **C** to ensure that there is an unambiguous location to which the failure point can jump.

GPU programmers usually view `goto` with scepticism. Apart from the usual problems [8], unrestricted use of `goto` can cause irreducible control flow, which is in general highly inefficient and sometimes unsupported on SIMT architectures. However, our use of goto to jump to postdominators does not cause irreducible control flow, and as we shall see in section 4, performance on contemporary GPUs is good.

3 Further Optimisations

This section shows additional optimisations and implementation hints that reduce the overhead of failure checking. The sum impact of these optimisations is shown in section 4.

3.1 Avoiding Global Memory Accesses

Some GPU kernels contain significant cross-thread communication within each work-group, which must be interleaved with checking the global_failure variable before every barrier. This can be a bottleneck, as global_failure is stored in global memory, and the kernel may otherwise use mostly the much faster local memory. To address this, we introduce a boolean variable local_failure, stored in local memory, that indicates whether a failure has occurred within the current work-group. When a thread fails, we set *both* global_failure and local_failure, but synchronisation points check only the latter. This is sufficient to ensure safety, as GPU work-groups cannot communicate with each other, and hence cannot be affected by a failure in another work-group. The final code emitted for a failure point is shown on fig. 3c, and the code for a synchronisation point on fig. 3d

3.2 Avoiding Unnecessary Failure Checking

As we discussed in section 2.2, threads in a running kernel initially check global_failure for whether a failure has occurred in a previous kernel. This requires multiple barriers to ensure that threads in the current kernel do not run ahead, fail, and set global_failure before all other threads have had a chance to read its initial value. While barriers are relatively cheap, we have observed that this initial checking still has a cost for very simple kernels. In many cases, we have run-time knowledge that no kernels with failure points have been enqueued since the last time we checked global_failure, and hence checking it is wasteful.

We address this by adding to every kernel another int-typed parameter, failure_is_an_option, that indicates whether *global_failure is potentially set. The value for this parameter is provided by the CPU when the kernel is enqueued.

The full prelude added to OpenCL kernels for failure checking is shown on fig. 3b, and the pertinent kernel parameters on fig. 3a. Note that we still need a barrier to ensure local_failure is properly initialised.

`int *global_failure`
: If pointed-to value is non-negative, the program is in a failing state.

`int failure_is_an_option`
: Whether `*global_failure` is potentially non-negative.

`int *global_failure_args`
: An array of values indicating precisely the failure that has occurred, e.g. the invalid index that was attempted. Never read from a kernel, but only written to.

(a) Parameters added to every kernel. Arguments for `global_failure` and `global_failure_args` are set on the host when the kernel is first created, but `failure_is_an_option` must be set whenever the kernel is enqueued.

```
volatile __local bool local_failure;
if (failure_is_an_option) {
  if (get_local_id(0) == 0) {
    local_failure = *global_failure >= 0;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  if (local_failure) {
    return;
  }
} else {
  local_failure = false;
}
barrier(CLK_LOCAL_MEM_FENCE);
```

(b) The failure handling prelude added to generated OpenCL kernels. See fig. 3a for the meaning of the kernel parameters used.

```
local_failure = true;
if (atomic_cmpxchg(global_failure, -1, k) < 0) {
  // write to global_failure_args...
}
goto sync;
```

(c) The code emitted whenever a failure occurs inside a kernel, where k is a unique non-negative integer identifying the failure, and *sync* a label identifying the next failure synchronisation point (see fig. 3d).

```
sync:
barrier(CLK_LOCAL_MEM_FENCE);
if (local_failure) {
  return;
}
```

(d) Code for a synchronisation point, where *sync* is a distinct label referenced in preceding `goto` statements (see fig. 3c).

Fig. 3: Essential code fragments for our implementation of GPU bounds checking. This lists kernel parameters and code only.

The `failure_is_an_option` parameter corresponds to an ordinary variable maintained by the CPU. It is initially zero, and set whenever we enqueue a kernel that contains failure points. Whenever the CPU synchronises with the GPU, and would normally copy `global_failure` back to the CPU to check its value, we first check `failure_is_an_option`. If zero, that means there is no reason to check `global_failure`. Our motivation is avoiding the latency of initiating a transfer, as copying a single 32-bit word of course takes very little bandwidth. After any CPU–GPU synchronisation where `global_failure` is checked, we reset `failure_is_an_option` to zero.

3.3 Avoiding Synchronisation Points

Many kernels, particularly those corresponding to a `map`, contain no communication between threads, and hence no barriers. For these kernels, failure points can simply `return`.

3.4 Non-Failing Kernels

Many kernels contain no failure points, and are guaranteed to execute successfully. These kernels must still check `global_failure` when they start, because this guarantee may be predicated on the successful execution of previous kernels, but they do not need the `failure_is_an_option` or `global_failure_args` parameters, and their kernel prelude can be simplified to the following:

```
if (*global_failure >= 0) { return; }
```

3.5 Failure-Tolerant Kernels

Some particularly simple kernels are able to execute safely (i.e. error-free) even when previous kernels have failed, typically because they merely copy or replicate memory, possibly with an index transformation. Matrix transposition is an example of such a kernel. For these kernels we can eliminate all failure checking entirely. This is because failures cannot result in memory becoming *inaccessible*; it can only result in the values stored being *wrong*, and these simple kernels are not sensitive to the values they are copying.

4 Experiments

We have implemented the presented technique in the compiler for Futhark [14], a functional array language that can be compiled to OpenCL and CUDA. Apart from checking array indexes, we also check for integer division by zero, as well as arbitrary programmer-provided assertions. These can be handled using the same approach as bounds checking. Futhark is a purely functional language, and so is not suitable for writing full applications. Instead, a compiled Futhark program presents a C API, with Futhark *entry points* exposed as C functions, which are then called by programs written in other programming languages. Futhark does not support exceptions or similar error handling mechanisms, so in the event of a failure, the error message is simply propagated to the return value of the C API, where it is made available for the caller to do with as they wish. One option is of course to print the message to the console and then terminate the entire process, but the GPU and Futhark state remains intact, including the data that was passed to the Futhark entry point, so it is also possible to continue execution with other data.

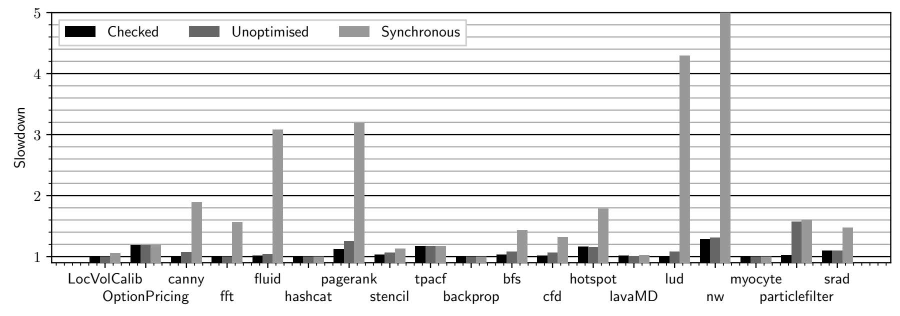
To investigate the efficiency of our implementation technique, we have measured the run-time of a range of Futhark programs compiled with and without bounds checking enabled. The full Futhark benchmark suite[2] contains 41 programs ported from Accelerate [5], FinPar [2], Rodinia [6], and Parboil [20]. Of these, 19 benchmarks require bounds checks in GPU kernels, and are the ones we use in our experiments. See table 1 for a table of the benchmarks and workloads. Since our focus is the relative cost of bounds checking, we do not compare our performance with the original hand-written benchmark implementations. Prior work has shown that Futhark's objective performance is generally good [13], so we consider our results representative of the cost of adding bounds checking to already well-performing code.

4.1 Methodology

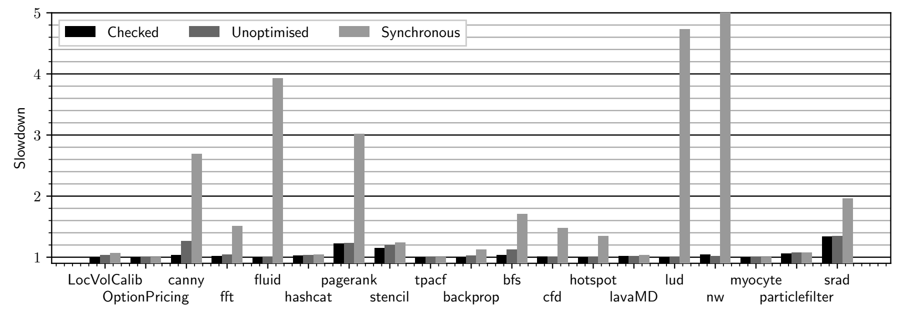
Each benchmark program is compiled and benchmarked with four different ways of handling bounds checks:

Without any checking: our baseline.
Checked: full bounds checking.
Unoptimised: excludes the optimisations from section 3.
Synchronous: `clFinish()` after every kernel enqueuing.

For the latter three we report the relative slowdown compared to the baseline. Our experiments are run on two systems: an AMD Vega 64 GPU where we use Futhark's OpenCL backend, and an NVIDIA RTX 2080 Ti GPU where we use Futhark's CUDA backend. We use the `futhark bench` tool to perform the timing, and the OpenCL backend for code generation. The timing does not include GPU driver setup and teardown, nor does it include copying the *initial* input data to the GPU, nor the *final* results from the GPU. All other CPU–GPU communication is counted. We report the average runtime of 10 runs for each benchmark.



(a) Benchmark results on an AMD Vega 64 GPU with OpenCL. The geometric mean of slowdowns is 1.06 (checked), 1.11 (unoptimised), and 1.61 (synchronous).



(b) Benchmark results on an NVIDIA RTX2080 Ti GPU with CUDA. The geometric mean of slowdowns is 1.04 (checked), 1.07 (unoptimised), and 1.66 (synchronous).

Fig. 4: Runtime slowdown of performing bounds checking compared to not performing bounds checking. See table 1 for the benchmark workloads. *Checked* is the full implementation with the optimisations listed in section 3. *Unoptimised* is without the optimisations. *Synchronous* is with GPU synchronisation after every kernel enqueuing.

4.2 Results

The results are shown on fig. 4. The most obvious conclusion is that synchronous execution can have ruinous overhead; exceeding 5 $\times$ for the *nw* benchmark, and exceeding 2 $\times$ on five other benchmarks on the RTX2080. The most affected benchmarks are structured as a rapid sequence of kernels that each run for at most a few dozen microseconds. Halting the GPU after every kernel, rather than letting it process the queue on its own, adds a significant constant cost to every kernel, which slows down these benchmarks significantly. On the other hand, those benchmarks that run just a few large kernels, such as *OptionPricing*, are not significantly affected. The Vega 64 is slightly less hampered than the RTX 2080 Ti, which is likely because the Vega 64 is relatively slower, so the kernels run for longer on average.

[2] `https://github.com/diku-dk/futhark-benchmarks`

Benchmark	Dataset	Benchmark	Dataset
FinPar		**Rodinia**	
LocVolCalib	large	backprop	medium
OptionPricing	large	bfs	graph1MW_6
Accelerate		cfd	fbcorr.domn.193K
canny	512 × 512	hotspot	1024
fft	1024 × 1024	lavaMD	10 boxes
fluid	medium	lud	2048
hashcat	rockyou	nw	large
pagerank	small	myocyte	medium
Parboil		particlefilter	128 × 128 × 10 image, 400000 particles
stencil	default		
tpacf	large	srad	502 × 458 image

Table 1: Benchmarks and datasets used for the measurements on fig. 4. The names of datasets are from the original benchmark sources (hence the inconsistent naming), and have been chosen to be the largest available.

The section 3 optimisations have a relatively small impact on most benchmarks. The largest impact is on *particlefilter* on the Vega 64, where the optimisations reduce overhead from almost 1.6× to essentially nothing. The *bfs* benchmark is an excellent demonstration of bounds checking, as it implements a graph algorithm by representing the graph as several arrays containing indexes into each other. A compiler would have to be *sufficiently smart* to a very high degree to statically verify these index operations. At the same time, most of the GPU kernels are `map` or `scatter`-like operations with no communication between threads, so the section 3.4 optimisation applies readily. The *srad* and *stencil* benchmarks suffer significantly under bounds checking on the RTX 2080 Ti. Both of these are stencil nine-point stencil computations, and are written in a way that prevents the Futhark compiler from statically resolving eight of the nine bounds checks that are needed for each output element.

5 Conclusions

We have demonstrated an implementation technique for implementing checking of array indexing and similar safety checks in GPU kernels generated from high-level array languages, even when the GPU programming API does not support abnormal termination.

Implementing the technique in a mature GPU-targeting compiler took only moderate effort, and our experiments show that the overhead of bounds checking has a geometric average of a relatively modest 6%, counting only those programs where checking is necessary in the first place. This is comparable to other work on bounds-checking C programs [1], although this comparison is admittedly not entirely fair, as a functional array language need not check indexing that arises from operations such as `map` and `reduce`. Nevertheless, our results suggest that bounds checking can be performed by default even in high-performance array languages.

Acknowledgements This research has been partially supported by a grant from the Independent Research Fund Denmark, under the research project *FUTHARK: Functional Technology for High-performance Architectures.*

References

1. Akritidis P, Costa M, Castro M, Hand S (2009) Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th Conference on USENIX Security Symposium, USENIX Association, USA, SSYM09, p 5166
2. Andreetta C, Bégot V, Berthold J, Elsman M, Henglein F, Henriksen T, Nordfang MB, Oancea CE (2016) Finpar: A parallel financial benchmark. ACM Trans Archit Code Optim 13(2):18:1–18:27
3. Bernecky R, Scholz SB (2015) Abstract expressionism for parallel performance. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, pp 54–59
4. Besard T, Foket C, De Sutter B (2019) Effective extensible programming: Unleashing julia on gpus. IEEE Transactions on Parallel and Distributed Systems 30(4):827–841
5. Chakravarty MM, Keller G, Lee S, McDonell TL, Grover V (2011) Accelerating haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, Association for Computing Machinery, New York, NY, USA, DAMP 11, p 314, DOI 10.1145/1926354.1926358, URL `https://doi.org/10.1145/1926354.1926358`
6. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC), Ieee, pp 44–54
7. Cunningham D, Bordawekar R, Saraswat V (2011) Gpu programming in a high level language: Compiling x10 to cuda. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop, Association for Computing Machinery, New York, NY, USA, X10 11, DOI 10.1145/2212736.2212744, URL `https://doi.org/10.1145/2212736.2212744`
8. Dijkstra E (1979) Go to Statement Considered Harmful, Yourdon Press, USA, p 2733
9. Erb C, Greathouse JL (2018) Clarmor: A dynamic buffer overflow detector for opencl kernels. In: Proceedings of the International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, IWOCL 18, DOI 10.1145/3204919.3204934, URL `https://doi.org/10.1145/3204919.3204934`
10. Fumero JJ, Steuwer M, Stadler L, Dubach C (2017) Just-in-time GPU compilation for interpreted languages with partial evaluation. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference

on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017, ACM, pp 60–73, DOI 10.1145/3050748.3050761, URL `https://doi.org/10.1145/3050748.3050761`

11. Guo J, Thiyagalingam J, Scholz SB (2011) Breaking the gpu programming barrier with the auto-parallelising sac compiler. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, Association for Computing Machinery, New York, NY, USA, DAMP 11, p 1524, DOI 10.1145/1926354.1926359, URL `https://doi.org/10.1145/1926354.1926359`
12. Henriksen T, Dybdal M, Urms H, Kiehn AS, Gavin D, Abelskov H, Elsman M, Oancea C (2016) Apl on gpus: A tail from the past, scribbled in futhark. In: Proceedings of the 5th International Workshop on Functional High-Performance Computing, ACM, New York, NY, USA, FHPC 2016, pp 38–43, DOI 10.1145/2975991.2975997, URL `http://doi.acm.org/10.1145/2975991.2975997`
13. Henriksen T, Serup NGW, Elsman M, Henglein F, Oancea CE (2017) Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI 2017, pp 556–571, DOI 10.1145/3062341.3062354, URL `http://doi.acm.org/10.1145/3062341.3062354`
14. Henriksen T, Thorøe F, Elsman M, Oancea C (2019) Incremental flattening for nested data parallelism. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, PPoPP '19, pp 53–67, DOI 10.1145/3293883.3295707, URL `http://doi.acm.org/10.1145/3293883.3295707`
15. Hoare CAR (1981) The emperors old clothes. Commun ACM 24(2):7583, DOI 10.1145/358549.358561, URL `https://doi.org/10.1145/358549.358561`
16. Holk E, Newton R, Siek J, Lumsdaine A (2014) Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. SIGPLAN Not 49(10):141155, DOI 10.1145/2714064.2660244, URL `https://doi.org/10.1145/2714064.2660244`
17. Hsu AW (2019) A data parallel compiler hosted on the gpu. PhD thesis, Indiana University
18. Price J, McIntosh-Smith S (2015) Oclgrind: An extensible opencl device simulator. In: Proceedings of the 3rd International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, IWOCL 15, DOI 10.1145/2791321.2791333, URL `https://doi.org/10.1145/2791321.2791333`
19. Steuwer M, Remmelg T, Dubach C (2017) Lift: A functional data-parallel ir for high-performance gpu code generation. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, IEEE Press, CGO 17, p 7485
20. Stratton JA, Rodrigues C, Sung IJ, Obeid N, Chang LW, Anssari N, Liu GD, Hwu WmW (2012) Parboil: A revised benchmark suite for scientific

and commercial throughput computing. Center for Reliable and High-Performance Computing 127
21. Xi H (2007) Dependent ml an approach to practical programming with dependent types. J Funct Program 17(2):215286, DOI 10.1017/S0956796806006216, URL `https://doi.org/10.1017/S0956796806006216`

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters

August Ernstsson · Johan Ahlqvist ·
Stavroula Zouzoula · Christoph Kessler

Abstract We present the third generation of the C++ based open-source skeleton programming framework SkePU. Its main new features include new skeletons, new data container types, support for returning multiple objects from skeleton instances and user functions, support for specifying alternative platform-specific user functions to exploit e.g. custom SIMD instructions, generalized scheduling variants for the multicore CPU backends, and a new cluster-backend targeting the custom MPI interface provided by the StarPU task-based runtime system. We have also revised the smart data containers' memory consistency model for automatic data sharing between main and device memory. The new features are the result of a two-year co-design effort collecting feedback from HPC application partners in the EU H2020 project EXA2PRO (2018–2021), and target especially the HPC application domain and HPC platforms. We evaluate the performance effects of the new features on high-end multicore CPU and GPU systems and on HPC clusters.

Keywords High-level parallel programming · Heterogeneous computing · Skeleton programming · Co-design approach · Cluster computing

1 Introduction

The recently started slowdown of Moore's Law implies, for the foreseeable future, that further performance growth in high-performance computing (HPC) critically depends on efficiently utilizing hardware resources, leveraging even more heterogeneity in the form of accelerators such as GPUs and scaling up to even higher degrees of cluster-level parallelism. This leads to programmability and portability issues on the software side. High-level programming using algorithmic skeletons is a promising approach to bridge this widening gap.

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

SkePU is a C++ based open-source skeleton programming framework for heterogeneous parallel systems. Based on an already modern and type-safe programming interface, we have redesigned SkePU to especially target the HPC application domain. The resulting third generation of SkePU presented in this paper is the result of a two-year co-design effort taking into account the feedback from HPC application partners in the EU H2020 project EXA2PRO (2018–2021), and aims at striking a good balance between improved programmability for HPC applications and decent performance and scalability on HPC platforms while keeping the strict portability approach of SkePU. The main new features include new skeletons, new data container types for multi-dimensional data and scalable data movement at distributed execution, support for returning multiple objects from skeleton instances and user functions, support for specifying optional, also platform-specific variants of user functions to exploit e.g. custom SIMD instructions, generalized scheduling variants for the multicore CPU backends, and a new cluster-backend targeting the custom MPI interface provided by the StarPU task-based runtime system. We have also revised the smart data containers' memory consistency model for automatic data sharing between main and device memory. We evaluate the performance effects of the new features on high-end multicore CPU and GPU systems and on HPC clusters.

The remainder of this paper is organized as follows. Section 2 introduces the main concepts in SkePU (as already available in SkePU 2), surveys the extensions, and sketches the main steps leading towards SkePU 3. Section 3 presents the new skeletons added in SkePU 3, as well as new features and modernized interfaces of existing ones. Section 4 presents new container types in SkePU 3. Section 5 explains the new coherence model of SkePU 3, and Section 6 presents the principles for execution on clusters. Section 7 contains preliminary results. Related work is discussed in Section 8, and Section 9 concludes.

2 Towards SkePU 3

2.1 Skeleton Programming Fundamentals

(Algorithmic) Skeletons [5, 6] are generic high-level programming constructs based on higher-order functions such as *map*, *reduce*, *scan* etc. that can be instantiated by plugging in sequential problem-specific code parameters, so-called *user functions* and that implement a frequently occurring, often domain-specific, characteristic *pattern* of control and data dependence for a possibly parallel, distributed or heterogeneous target platform. Especially in the context of heterogeneous systems, it is common that a skeleton comes with multiple implementations (called *back-ends*) for different target platforms, such as back-ends for sequential, multi-threaded, message-passing or accelerator execution. Skeletons can be realized as libraries or as (often, embedded) domain-specific languages (DSLs) atop a sequential programming language, where C++ is

most common today, see also Section 8 for an overview of further skeleton programming environments.

Skeleton instances are thus the result of composing multiple software artifacts (a skeleton and one or several user functions), but can be used (invoked) in the same way as traditional hand-written functions.

While skeletons are high-level abstractions of *computation*, they usually also work on high-level abstractions for collections of operand *data*, often in the form of STL-like *data containers* that encapsulate and transparently manage their elements internally [9].

The foremost objective of skeleton programming is improved *programmer productivity* compared to explicit parallel programming, i.e., to make writing programs for parallel, distributed and heterogeneous systems as easy as well-structured sequential programming—where the available set of skeletons fits. Portability at a high level of abstraction is also important, especially in the context of heterogeneous computing systems, where an obvious tuning possibility is the automated selection of the fastest backend depending on the execution context [8]. The price of abstraction might be a certain loss in efficiency compared to explicitly parallel code written by system experts, however the abstraction might even lead to higher performance where the better structuring and the knowledge of dependence patterns can enable automated optimizations.

2.2 A Short History of SkePU

SkePU (1) was introduced in 2010 [11] as a skeleton programming library for heterogeneous single-node but multi-accelerator systems, from the beginning designed for portability to include single- and multi-GPU backends for the C-based OpenCL and for CUDA (which then only partly supported C++), and had thus been technically based on C++03 and on C preprocessor macros as the interface to user functions.

SkePU 2, introduced in 2016 [15], was a major revision of the SkePU [11] library, ushering in modern C++ to the skeleton programming landscape. Rebuilding the interface from the ground up, the skeleton set was updated to be variadic, leaving the old fixed-arity skeletons from SkePU 1 behind. Variadic skeleton signatures was the first main motivator of SkePU 2: *flexible* skeleton programming.

This rewrite also took the opportunity to integrate patched-on functionality in SkePU 1 into the core design of the programming model. One such example is the absorption of SkePU 1 `MapArray` into the basic SkePU 2 `Map`. `MapArray` was a dedicated skeleton in SkePU 1 created as a clone of `Map` with the ability to accept an auxiliary, random-accessible array operand into the user function, allowing deviations from the strictly functional map-style patterns when demanded by the target application. This was one of the first lessons from practical experience [21] that skeleton patterns are not always perfectly suited to algorithms in real-world application code.

Table 1: Overview of SkePU Features

	SkePU 1 (2010) [11]	**SkePU 2** (2016) [15]	**SkePU 3** (2020)
API based on	C, C++ (pre-2011), C preprocessor	C++11, Precompiler (clang)	C++11, Precompiler (clang, mcxx)
Skeletons	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapArray`, `MapOverlap`, `Generate`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `Call`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `MapPairs`/–`Reduce`, `Call`
User functions as	C preprocessor macros Not type-safe	Restricted C++ functions Type-safe	Restr. C++ functions, plus multi-variant user functions Type-safe
Containers	`Vector<>`, `Matrix<>`	`Vector<>`, `Matrix<>`	`Vector<>`, `Matrix<>`, `Tensor3<>`, `Tensor4<>`, `MatRow<>`
Platforms supported	CPU (C, OpenMP), GPU (CUDA, OpenCL)	CPU (C++, OpenMP), GPU (CUDA, OpenCL)	CPU, GPU, hybrid CPU/GPU, StarPU-MPI, ...
Scheduling (OpenMP)	Static	Static	Static, Dynamic
Memory model	Sequential consistency	Sequential consistency	Weak consistency (default), optionally sequential cons.

SkePU 2 also introduced the *pre-compiler*, lifting SkePU from its humble origins as a pure template include-library into a full-fledged *compiler framework*. This, together with the effort to push the C++ type system farther than most, if not all, comparable frameworks enabled the second main motivator of SkePU 2: *type-safe* skeleton programming.

Table 1 gives a synopsis of the different features of SkePU versions.

2.3 SkePU 3 Overview

SkePU provides data-parallel *skeletons*, all of which are of arbitrary arity and polymorphic in the operand container shape and element type. Skeleton instances accept operands that are statically grouped (by a template parameter) into element-wise accessible and random-accessible parameters [15]. The `Map` skeleton computes every result container element by element-wise application of a user function f to the corresponding (for element-wise accessed operands) or possibly any (for other operands) elements; `MapOverlap` applies a stencil function in one or several dimensions of an element-wise accessed operand that can also access elements in a limited neighborhood of the corresponding operand container element(s); `Reduce` applies reduction for a binary associative user function; `MapReduce` provides a combination of `Map` and `Reduce`; `Scan` computes generic prefix-sums for the provided binary associative user function; and `Call` simply calls the user function for each position of the output container; in combination with multi-variant user functions it also provides a portable escape mechanism to invoke explicitly parallelized code where the available skeletons do not fit (well) [14]. The new skeletons added or generalized in SkePU 3 (cf. Tab. 1) are described in Section 3.

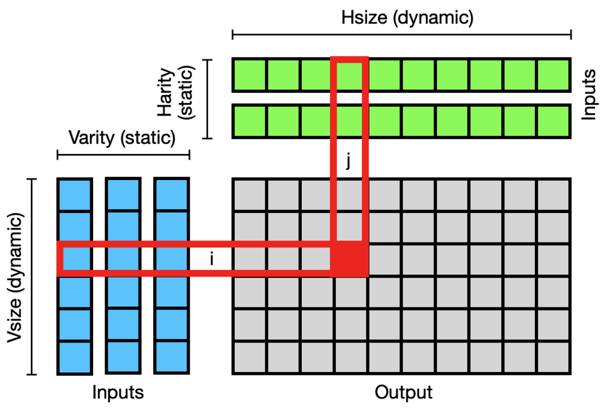


Fig. 1: `MapPairs` computes a Cartesian mapping of 1D vectors into a 2D space.

SkePU allows to select the backend for each skeleton instance call statically or dynamically. A tuning mechanism allows for automated backend selection depending on a call's operand sizes and locations. Hence, it might be statically unknown where a skeleton call will execute.

Operands to skeleton instances are to be passed in *data containers*, which are STL-like, generic collection abstract data types like `Vector` and `Matrix` that encapsulate C++ array-type data. We call them *smart* containers [9] because they transparently perform data transfer and memory management for their elements in (heterogeneous) systems with distributed memory, as well as global optimizations for data locality [13]. Using C++ iterators, skeleton instance calls may also operate on a proper subset of a container's elements only. New containers in SkePU 3 (see Tab. 1) are described in Section 4.

3 Skeleton Set and Interface Extensions

This section covers all major changes to the skeleton API in SkePU 3, except for the multi-variant user function feature which is already detailed in [14]. We begin with `MapPairs` and `MapPairsReduce`, which are variants of `Map` resp. `MapReduce` on a 2D domain that are explicit about the access pattern of lower-dimensional (1D) operands.

3.1 MapPairs

The `MapPairs` skeleton, added as an additional top-level skeleton in SkePU 3, applies a Cartesian product-style pattern from *two* `Vector<T>` sets (note that the templated type may differ across these vectors). Each vector set may contain an arbitrary number of vector containers, similar to the variadicity of

Map. All of the vectors in a set are expected to be of the same size. Each Cartesian combination of vector set indices generates one user function invocation, the result of which is an element in a Matrix. As in Map, there is an optional Index2D parameter in the user function signature to access this index. An example is shown in Listing 1.

Listing 1: Using the MapPairs skeleton.

```
// MapPairs user function
int mult(int a, int b) { return a * b }

// MapPairs skeleton instantiation and usage
// size_t Vsize, Hsize defined here
auto outerproduct = skepu::MapPairs(mult);

skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
skepu::Matrix<int> res(Vsize, Hsize);
outerproduct(res, v1, h1);
```

Advanced and more flexible use of MapPairs can be carried out similarly to other SkePU skeletons. For instance, it retains flexibility of Map with regard to variadicity (5-way variadic, compared to Map being 4-way variadic): (1) resulting outputs (see Sect. 3.3), (2) element-wise-V (”vertical”, column-aligned) input arguments, (3) element-wise-H (”horizontal”, row-aligned) input arguments, (4) random-access input arguments, and (5) uniform input arguments.

A MapPairs instance of higher order would look like in the example below:

```
auto pairs = skepu::MapPairs<3, 2>(...);
```

This instance will accept three vertical and two horizontal input vectors.

3.2 MapPairsReduce

MapPairsReduce is the combination of a MapPairs followed by a row-wise or column-wise reduction over the generated matrix elements. Like MapPairs it supports arbitrary arities of the vertical and horizontal input Vector groups (<0,0> and up). It returns a Vector containing the row-wise or column-wise sums, where the summing dimension is specified as in 2D Reduce.

As a small example, we use the force calculation phase of Nbody simulation using this new skeleton. In SkePU 2, using the more general Map skeleton, it could be written as shown in Listing 2.

Listing 2: Nbody simulation code using Map.

```
auto nbody_init = skepu::Map<0>(init);
auto nbody_simulate_step = skepu::Map<1>(move);

nbody_init(particles, np);

for (size_t i = 0; i < iterations; i += 2)
{
        nbody_simulate_step(doublebuffer, particles, particles);
        nbody_simulate_step(particles, doublebuffer, doublebuffer);
}
```

In SkePU 3 (Listing 3), the use of `MapPairsReduce` eliminates the explicit loop required in the user function (`move`, omitted here for space). Moreover, this formulation intrinsically avoids the double-buffering requirement for the existing approach, and hence memory pressure is reduced.

Listing 3: Nbody simulation code using `MapPairsReduce` in SkePU 3.

```
auto nbody_init = skepu::Map<0>(init);
auto nbody_influence = skepu::MapPairsReduce<1, 1>(influence, sum);
auto nbody_update = skepu::Map<2>(update);

nbody_init(particles, np);

for (size_t i = 0; i < iterations; ++i)
{
        nbody_influence(accel, particles, particles);
        nbody_update(particles, particles, accel);
}
```

The total number of user-function calls is now quadratic in the number of particles `np` (an inherent property of MapPairs), compared to linear before, so good performance is even more reliant on compiler optimization, in particular, user function inlining.

Due to different ways of computing the reductions in the two versions, the outputs are not exactly identical but match to about 4-5 significant decimal digits. This is expected due to the use of (single-precision) floating point numbers.

3.3 Multi-valued Return in Map Skeletons

SkePU 3 introduces tuple-like return functionality for cases where a single skeleton instance requires multiple (element-wise) output containers. This way, multiple return values can be computed by the same user function, operating on the inputs in one sequence, potentially improving data locality compared to two separate skeleton invocations after each other. Although the values are returned in a tuple-like manner, the output containers are completely separate objects (see Fig. 2). This distinguishes this new feature from the existing use of custom `struct`s as (inputs or) return values, as those are stored in array-of-records format. To use this feature, we specify the return type in the user function signature as `skepu::multiple<[basic_type, ...]>`, i.e., analogous to `std::tuple`. Then, at the site of the `return` statement, we construct this compound object by `skepu::ret([expression, ...])`.

Listing 4 shows an example of a user function utilizing this:

Listing 4: User function with multi-valued return.

```
skepu::multiple<int, float>
multi_f(skepu::Index1D index, int a, int b, skepu::Vec<float> c, int d)
{
  return skepu::ret(a * b, (float)a / b);
}
```

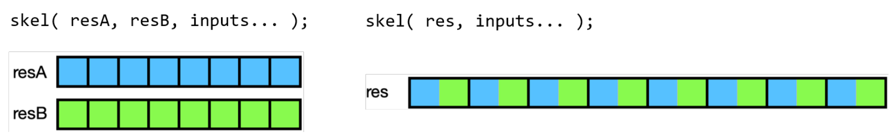


Fig. 2: Difference in return value storage between using multi-valued return (left) and single-value (by manually managed array-of-struct) return (right).

The skeleton instance declaration and invocation follow the syntax of ordinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order. Listing 5 gives an example.

Listing 5: Using multi-valued return with Map in SkePU 3.

```
skepu::Vector<int> v1(size), v2(size), r1(size);
skepu::Vector<float> e(1);

auto multi = skepu::Map<2>(multi_f);

multi(r1, r2, v1, v2, e, 10);
```

Multi-valued return statements are available in the skeletons which follow the typical map pattern: `Map`, `MapPairs`, and `MapOverlap` .

3.4 Dynamic Scheduling with OpenMP Backends

In SkePU 2, all skeletons, in particular the `Map` based skeletons, assumed an equal load distribution of the user function executions over the entire range of input container elements. Some applications may however exhibit an irregular workload distribution instead, especially in CPU-affine computations and sometimes even in combination with very short input vectors.

For these cases, SkePU 3 adds in all skeletons (except `Scan` and `Call`) an option for dynamic scheduling in the OpenMP backend.

```
spec.setSchedulingMode( skepu::Backend::Scheduling::Dynamic );
```

Other supported `Scheduling` modes in the OpenMP backends are `::Guided` (for guided self-scheduling), `::Auto` (for auto-tuned scheduling as implemented in the used OpenMP compiler), and of course `::Static` which remains the default scheduling mode.

In addition, a chunk size locally overriding the default chunk size (defined for each scheduling mode as in OpenMP) can be set:

```
spec.setCPUChunkSize(8);
```

Performance evaluation results for three load balancing / nondeterministic-time benchmarks are given in Section 7 and show in each case some improvement over the static SkePU 2, in spite of some overhead for the dynamic scheduling modes; speedups for unbalanced workloads up to 60% (for Mandelbrot-set computation) have been observed.

3.5 Revised Syntax for MapOverlap

Experiences from SkePU users, and in particular the application of SkePU in teaching, has showed that the syntax for `MapOverlap` user functions is one of the more challenging aspects of SkePU. In SkePU 2, a stencil operator was specified as in the following example (here, for a rectangular $2oi + 1 \times 2oj + 1$ stencil implemented by 2 nested loops):

Listing 6: A MapOverlap user function in SkePU 2.

```
float over_2d( int oi, int oj, size_t stride, const float *r,
               const skepu::Mat<float> stencil )
{
  float res = 0;
  for (int i = -oi; i <= oi; ++i)
    for (int j = -oj; j <= oj; ++j)
      res += r[y*(int)stride+x] * stencil.data[(i+oi)*ox + (j+oj)];
  return res;
}
```

For SkePU 3 we have redesigned and simplified the programming interface for specifying stencil operators. The above stencil computation looks as follows in SkePU 3:

Listing 7: New syntax for MapOverlap user function in SkePU 3.

```
float over_2d( skepu::Region2D<float> r, const skepu::Mat<float> stencil )
{
  float res = 0;
  for (int i = -r.oi; i <= r.oi; ++i)
    for (int j = -r.oj; j <= r.oj; ++j)
      res += r(i, j) * stencil(i + r.oi, j + r.oj);
  return res;
}
```

4 New Data-Containers

The availability of smart containers, previously restricted to vector and matrix types, has a significant effect on the usability of a skeleton programming framework. Even though a basic one-dimensional data set can be used to emulate more complex data representations, doing so at a framework level rather than on the user level provides more information to the implementation about access patterns, thus bringing increasing opportunities for optimizing communication- and memory access patterns; while also providing a more intuitive user interface and reduced application code size for users.

In SkePU 3 this is recognized on two levels: new multi-dimensional *tensor* containers, as well as a new ”proxy” container in user functions for accessing a single *row* from a matrix.

4.1 Tensors

The SkePU container set is extended with *tensors*, which are higher-dimensionality containers. In SkePU 3 there are tensors of three and four dimensions,

```
float func( T a, T b, MatRow<T> mr, Mat<T> m )  { ... }
```

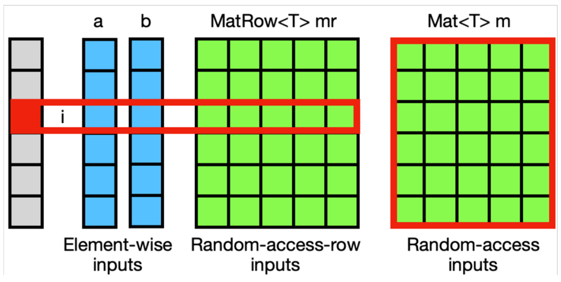


Fig. 3: Element accessibility for `MatRow` vs. `Mat` parameters in a user function.

complementing the existing 1D ”vector” and 2D ”matrix”. Smart container dimensionality in SkePU 3 is therefore fixed by the framework, though their sizes in each dimension is user-defined.

The interfaces for these containers are virtually identical to those of the other containers, differing in the obvious ways of naming and element access as detailed below. The full set of smart containers in SkePU 3 now covers up to four-dimensional structures; see Listing 8 for their definitions.

Listing 8: Smart container set in SkePU 3.

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
skepu::Tensor3<float> t3(dim1, dim2, dim3);
skepu::Tensor4<float> t4(dim1, dim2, dim3, dim4);
```

The set of `Index` object types in SkePU, usable in e.g. user function signatures to identify the index of the element being operated on, is likewise extended with 3D and 4D equivalents (Listing 9):

Listing 9: Index types corresponding to each smart container.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

Tensors are available in the skeleton API as element-wise inputs to `Map`, `Reduce`, `MapReduce`, `Scan`, and `MapOverlap`. They are also accessible freely user functions as proxy objects, where applicable. In some skeleton configurations the dimensionality of element-wise inputs is irrelevant by design, though in Map-based skeletons it can be accessed by using `Index` parameters.

4.2 MatRow Container Proxy

SkePU has since version 2 offered flexible parameter lists for user functions, including *random-access* containers (implemented as lightweight *proxy objects*) in addition to the default element-wise inputs. While this allows for powerful expressivity, very little about the access patterns of these random-access containers is known to SkePU, and performance may thus not always be ideal.

One very common pattern when using `Matrix` as a random-access container parameter is that each user function invocation is only interested in one row of the matrix. This pattern is seen in matrix-vector multiplication and similar multi-reduction-style computations. To improve SkePU performance in these cases, SkePU 3 introduces a new proxy object, `MatRow<T>`. Bridging the gap between element-wise mapped and random-access container arguments, this proxy type when used in a `Map` skeleton instance that maps over vectors (i.e., the result container(s) of the skeleton are `Vector`), makes available one single row of the argument matrix container to the user function, see Fig. 3.

As an example, matrix-vector multiplication using `MatRow<T>` may be implemented as in Listing 10:

Listing 10: Matrix-vector multiply using MatRow in SkePU 3.

```
template<typename T>
T arr(const skepu::MatRow<T> mr, const skepu::Vec<T> v)
{
  T res = 0;
  for (size_t i = 0; i < v.size; ++i)
    res += mr.data[i] * v.data[i];
  return res;
}
```

Compared to the closest corresponding SkePU 2 implementation below (still valid in SkePU 3), which only offers the more generic `Mat` proxy container, the code is more succinct and there is more information about the access pattern available to SkePU.

Listing 11: Matrix-vector multiply in the SkePU 2 style.

```
template<typename T>
T arr(skepu::Index1D row, const skepu::Mat<T> m, const skepu::Vec<T> v)
{
  T res = 0;
  for (size_t i = 0; i < v.size; ++i)
    res += m.data[row.i * m.cols + i] * v.data[i];
  return res;
}
```

There is no change in syntax of skeleton instantiation or skeleton invocation needed for this feature to apply.

The performance benefit of using `MatRow` (where applicable) instead of the more general `Mat` container proxy comes from significantly reduced operand data transfer volume when executing over distributed memory scenarios, both in multi-GPU execution and in cluster execution: the communication pattern with `MatRow` is a scatter operation, while with `Mat` it is a broadcast.

Matrix-row user function proxy containers are available in user functions for `Map`, `MapReduce`, and `MapOverlap` skeleton instances that satisfy the above requirements.

5 Consistency Model

Experiences from users of SkePU 2 demonstrated that the dual-mode model of SkePU can be a bit challenging to adapt to. Like for instance GPU programming models, SkePU programs execute code in one of two modes, in GPU programming parlance "host" and "kernel" mode. In SkePU, these are represented by being either outside or inside of the *dynamic scope* of a skeleton user function. While syntactically highly similar, the capabilities in each mode are very different. Host SkePU code is effectively like any C++ environment, as it is the goal of the framework to be possible to embed in existing C++ applications. This means that the programmer can use any C++ constructs and idioms such as classes, dynamically allocated structures, etc.

Inside a user function, however, the environment is effectively a single-threaded, no-side-effects, C-like land[1].

These differences also mean that the memory (coherency) models are different in the two views. SkePU handles memory consistency at the boundary—during entry and exit of a skeleton invocation and the user function evaluation. Inside the user function, side effects are not allowed and therefore random memory reads are disabled, and the coherency model is straightforward.

SkePU 3 *deprecates* the angle bracket `[ ]`-notation for smart container element read/write access *outside* user functions.[2] This is part of a simplification of the coherency mechanisms for manual element access from the host (CPU) side. Instead, the programmer should *flush* the whole container instead before doing single-element accesses of user function data, see below.

Instead of angle brackets, the parentheses `( )`-notation is extended to higher dimensionality. This syntax accepts one index argument for each dimension of the underlying container. The indices count must equal container dimensionality, otherwise there is a compile-time error. Formally, the access syntax is `container(i,[j, [k, [l]]]) [= value];`

Hence, there is no longer a coherency-satisfying single-element access mechanism to SkePU smart containers except inside user function proxy objects (`Vec<T>`, `Mat<T>`, etc). However, optional runtime checks outside user functions can be (re-)activated for parenthesis accesses by setting a compiler flag, e.g., for debugging purposes or for backwards compatibility with code written for SkePU 2.

A common pattern in SkePU applications is that smart containers are used for a computationally intensive part of the application, and the data is then either handed over to a non-SkePUized section, or serialized e.g. to disk. To accommodate this pattern, it is important that there is a way to ensure

[1] The reason for this is to preserve compatibility with as many accelerator environments as possible, such as OpenCL C or even FPGAs.

[2] In SkePU 2 (and SkePU 1), the bracket operator is a protected container access, which outside user functions checks for the accessed element's state in the data container's metadata (updated or invalid) and, if necessary, triggers a (bulk) data movement to update the container's copy in host memory from a currently valid device copy. All bracket accesses thus incur runtime overhead for the check.

consistency of the local container contents. SkePU 3 provides this through the `flush` operation to complete the new consistency model.

Flushing smart container data can be performed on smart container instances or collectively by a variadic free function. Either approach accepts a flush mode enum argument providing options, e.g. if the remote data buffers should be cleaned up or not, as seen in Listing 12.

Listing 12: Examples of using the flush operation.

```
skepu::Vector<int> v1(n), v2(n);
skepu::Matrix<int> m1(n, n), m2(n, n);

v1.flush(); // FlushMode::Default
m1.flush(); // FlushMode::Default

skepu::flush(v2, m2); // FlushMode::Default

v1.flush(skepu::FlushMode::Dealloc);
m1.flush(skepu::FlushMode::Dealloc);
skepu::flush<skepu::FlushMode::Dealloc>(v2, m2);
```

The `flush` (member) functions are known symbols to the precompiler, so the presence or absence of flush operations in SkePU source code is subject to static analysis and optimization.

6 Cluster Backend

SkePU 3 provides two different modes of using cluster resources:

- *Outer MPI mode*: the application code already contains explicit MPI code for cluster-level parallel execution, using SkePU only locally on each node for execution of skeletons on multicore CPU and/or accelerators.
- *Inner MPI mode*: The application does not contain any MPI (nor other parallelization) code. If an environment for MPI parallel execution is available (usually, multiple nodes on a cluster), then skeletons can transparently execute in parallel across these nodes if selecting the MPI backend.

Outer and Inner MPI mode are mutually exclusive, i.e., for applications that are pre-parallelized using explicit MPI code the MPI backends of all skeletons are disabled.

The implementation of inner MPI parallelism is technically based on generating StarPU task code using the MPI interface of the StarPU runtime system [3], which detaches each node's generated send and receive operations into special CPU "codelets" that are exposed to StarPU as separate tasks for dynamic scheduling [2]. Distributed variants of the smart data-containers (`Vector`, `Matrix` etc.) with the same interface as the node-local counterparts come with default distributions, and each cluster node runs one copy of the SkePU executable atop a local instance of StarPU in SPMD style. Execution over distributed container operands follows the "owner computes rule", stating that each node only executes those operations that calculate (write) elements it owns (i.e., are part of its local partition of the result container).

For using inner MPI parallelism, no syntactic changes in SkePU code are required, thus following SkePU's strict portability principle. The illusion of a single SkePU process performing all the work on a single node even with the MPI backend is maintained by implementing the `Reduce` skeleton by an MPI Allreduce operation so that the reduction result is available on each of the SPMD processes. The weak memory consistency model of SkePU 3 (cf. Sect. 5) applies also to distributed containers: the programmer must explicitly `flush` (i.e., gather) them back to the master (i.e., the rank 0 process) before the most recent values of elements of remote partitions can be accessed by a read access on the master, or after a write access by the master.

The only remaining issue in SPMD execution is that I/O operations need be protected from being executed everywhere. To make sure that such code is executed only by the SPMD master process (and distributed data to be output is automatically flushed and gathered/scattered to/from before/after the access, respectively), such code should be guarded by the new construct

```
skepu::external (
  [ skepu::read(rdcontlist),] [&]() {
     ...
  } [, skepu::write(wrcontlist)]
);
```

where the optional arguments `skepu::read()` and `skepu::write()` list container objects that may be read from resp. written to main memory in the framed code block (`...`). This semi-automatic solution with an explicit framing construct allows to not depend on static analysis by the precompiler, which may not be feasible in the context of separate compilation and using libraries.

7 Performance evaluation

Figure 4 shows SkePU 3 performance results for an embedded ODE solver from the *Libsolve* library[3] [17], solving the Brusselator 2D-MIX problem with 7 stage vectors for four different system sizes ($N = 250$, 500, 750, 1000 rows) on a server with 12 cores Xeon(R) CPU E5-2630L and a K20c GPU, with pre-selected single-node CPU and GPU backends respectively. The solver core uses 9 different skeleton instances (of `Map`, `Reduce` and `MapReduce`) with an average of 63 calls to skeleton instances per time step; it iterates over 1976 time steps in total for the largest scenario in Fig. 4, for which it performs 124,532 calls to skeleton instances in total.

Figure 5 shows performance results for the Nbody scenario of Section 3.2 using the OpenMP backend, taken on the same server. There is a slight increase in execution time, although too small to account for an inlining issue (discussed in Section 3.2). A likely explanation for the slowdown is due to the change in memory access pattern. Depending on the environment, the more

[3] Libsolve repository: https://github.com/UBT-AI2/rk

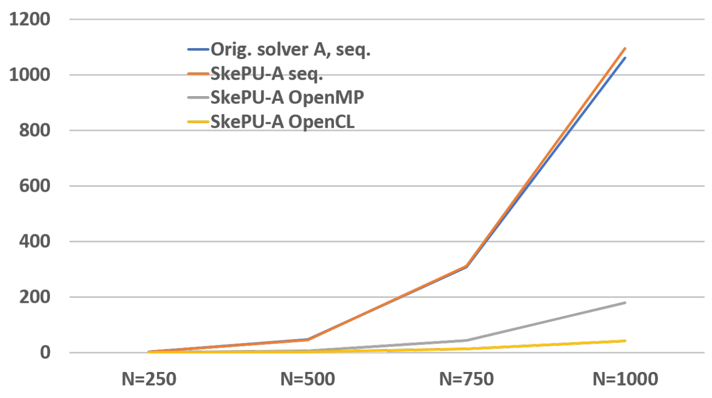


Fig. 4: Execution times (seconds) of the SkePU 3 port of Variant A of the Embedded Runge-Kutta ODE solver implementation in the *Libsolve* library [17], solving the Brusselator 2D-MIX problem for 4 different system sizes.

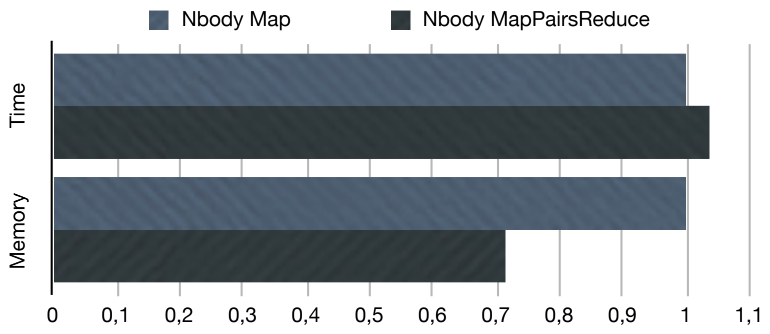


Fig. 5: Normalized execution time and memory footprint for two variants of Nbody: the `Map` variant (Listing 2) and `MapPairsReduce` variant (Listing 3).

significant improvement in memory footprint might be enough to prefer the MapPairsReduce variant.

Execution time results for SkePU 3 ports of PARSEC benchmarks Blackscholes and Streamcluster on the same server can be found in Figures 6 and 7. The results show that the SkePU abstraction overhead compared to the hand-multithreaded PARSEC code is small (Blackscholes) or very small (Streamcluster), and that SkePU provides further targets for free (here, OpenCL for Blackscholes). The Streamcluster benchmark also exhibits a common problem encountered in SkePU-izing legacy C/C++ code: arrays containing a pointer-based data structure (e.g., a directed graph), if packaged e.g. in a `Vector` container, work very well with the OpenMP backend but are not portable to execution on e.g. a GPU with a different address space, as host addresses are not portable to device memory. For such cases, more advanced container types (e.g., directed graphs) would be required, which is left for future work.

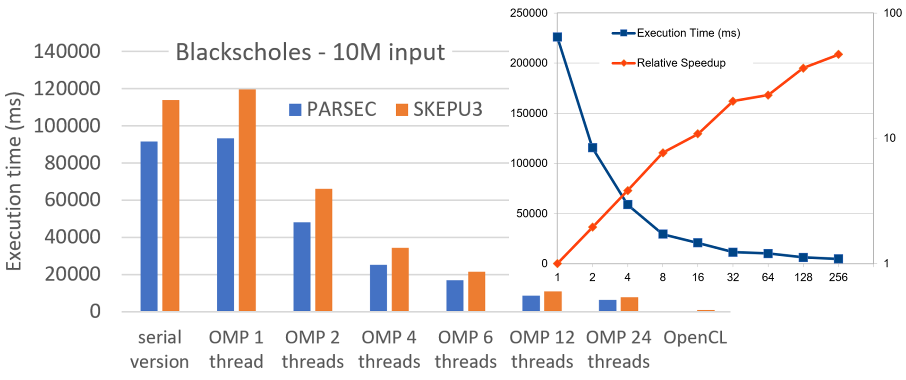


Fig. 6: Execution time (ms) of the SkePU 3 port of the PARSEC benchmark Blackscholes on its largest input set. Left: Time with serial, OpenMP, OpenCL backends in SkePU and for manually multithreaded code in PARSEC. Right: Time and speedup with the StarPU-MPI backend on the cluster of Figure 9.

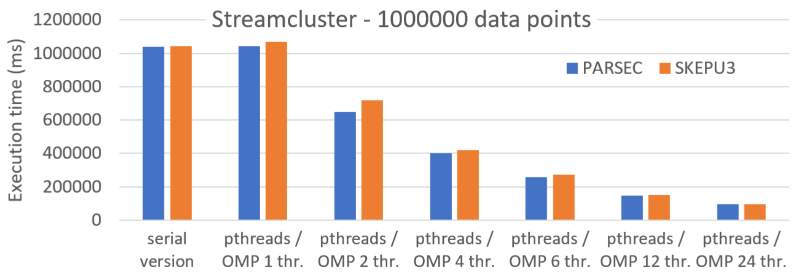


Fig. 7: Execution times of the SkePU 3 port of the PARSEC benchmark Streamcluster on 10^6 data points.

For the same machine, Figure 8 shows the positive performance effect of using dynamic scheduling in three data-parallel benchmarks with irregular workload, in spite of the runtime overhead of dynamic scheduling: (1) Generating a 1024×1024 Mandelbrot image using the SkePU 3 `Map` OpenMP backend with different scheduling modes. Dynamic scheduling (chunksize 16) outperforms the static default mode. (2) Lexicographic reduction finding the maximum among 10^8 date/time tuples. `Guided` dynamic scheduling (chunksize 8) outperforms the static default mode. (3) Counting prime numbers using `MapReduce` where dynamic scheduling performs best. Results for the sequential CPU and OpenCL backends are provided as reference.

Figure 9 shows the scaling behavior of the SkePU 3 port of a brain simulation mini-application [19] performing 200 time steps with 90000 neurons and dense synapse connectivity using up to 32 nodes (each node having two Xeon Gold 6130 with 16 cores each) of the *Tetralith* cluster at NSC Linköping. The version that uses the *MatRow* container proxy benefits from more scalable communication compared to using the default *Mat* container. For comparison,

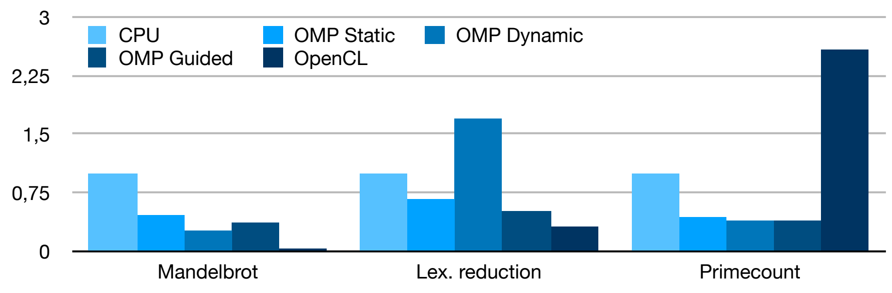


Fig. 8: Execution time (normalized to the sequential CPU backend time) for three irregular-load benchmarks.

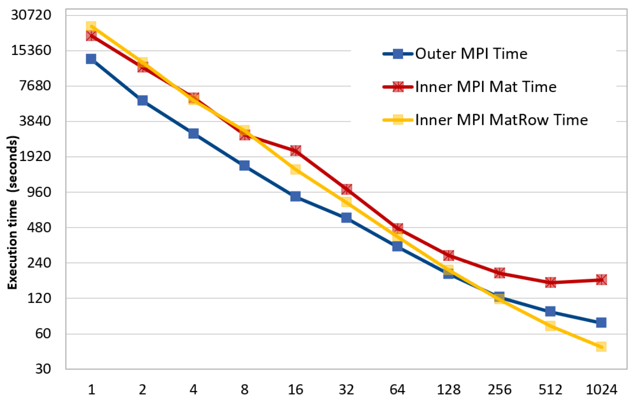


Fig. 9: Execution time (in seconds, logarithmic scale) of the SkePU 3 port of a brain simulation mini-application [19] performing 200 time steps with 90000 neurons using up to 32 nodes (each with 32 cores) of the Tetralith cluster. "Outer MPI" refers to a manual MPI parallelization using plain MPI, the two "Inner-MPI" versions use SkePU's StarPU-MPI backend instead.

Table 2: Microbenchmark results of vector initialization, seconds.

	With GPU backends	Without GPU backends
Sequential consistency `v[i]`	0.899	0.308
Weak consistency `v(i)`	0.313	0.310

the diagram also shows a manual MPI parallelization of the SkePU code (i.e., outer-MPI SkePU) where the communication structure corresponds to that of the MatRow version. While the SkePU version of MatRow scales and performs best for larger numbers of MPI ranks, we also see that there is an execution time overhead of using SkePU with the StarPU-MPI based backend of up to a factor of 2 as long as running on a single cluster node ($\leq$32 MPI ranks).

To illustrate the motivation behind the change of consistency model for SkePU smart containers (Sect. 5), we have measured the execution time through a microbenchmark. Allocating and initializing the elements of a SkePU vector using a simple for-loop results in the numbers in Table 2. If the SkePU application is compiled without either GPU or CUDA backends there is no appreciable overhead, but as soon as those device copies are present it is approximately 3x faster to use non-managed access operators.

8 Related work

The skeleton approach to high-level programming of parallel systems has been introduced by Cole in 1989 [5,6]. Since then, many academic skeleton programming frameworks have been presented, and the concept also increasingly found its way into commercial/industrial-strength programming environments, such as Intel TBB for multi-core CPU parallelism, Nvidia Thrust or Khronos SYCL for GPU parallelism, or Google MapReduce and Apache Spark for cluster-level parallelism over huge data sets in distributed files.

While early skeleton programming environments attempted to define and implement their own programming language, library-based and DSL-based approaches have, by and large, been more successful, due to fewer dependencies and lower implementation effort. Frameworks for skeleton programming became practically most effective in combination with (modern) C++ as base language. Moreover, the approach was fueled by the increasing diversity of processing hardware with upcoming multi-core and heterogeneous parallelism since about 2005.

Among the C++ based skeleton programming environments for heterogeneous systems, we find mostly library-based ones, e.g. SkePU 1 [11], SkeTo [18], SkelCL [23], GrPPI [10] or pre-compiler based, such as SkePU 2 [15] and Musket [20]. FastFlow [7], originally designed for multicore CPU execution, added support for GPU and distributed execution [1] later. SkelCL targeted OpenCL for single- and multi-GPU systems with explicit data distribution. Muesli [4] initially supported MPI cluster execution and added support for GPU execution later [12]. MPI execution of skeletons is also supported e.g. in Musket.

The Allpairs skeleton [22] in SkelCL can be considered as a variant of `MapPairs` that accepts matrix operands only; any reduction needs be implemented as part of the user function in Allpairs (i.e., by nesting), while we provide the combination `MapPairsReduce` (i.e., chaining). `MapPairs` and `MapPairsReduce` specifically support *multiple* separate *1D* vector operands in both dimensions, as requested for use with the *MetalWalls* application by EXA2PRO project partner CNRS.

Multiple return values of skeletons and user functions is inspired by Python and was also found useful in SkePU-izing *MetalWalls* to avoid duplicated work resulting from using multiple skeleton calls for different result containers. We are not aware of any other skeleton programming framework supporting multiple return values from skeletons and user functions where parameters are passed explicitly. Musket [20], which also uses a precompiler, requires (except for the `this` object) using global container variables in user functions.

9 Conclusions and future work

We have presented the design of the third generation of the SkePU skeleton programming framework for heterogeneous systems and HPC clusters. The new features are the result of a co-design effort together with HPC application partners in the EXA2PRO project, and are geared towards improving programmability, flexibility, better performance, or several of these aspects, while keeping SkePU source code strictly portable and compatible with sequential C++11. We provide an early evaluation of the new features w.r.t. performance; further results will be reported in the final version.

Future work will, beyond performance improvements in the implementation, conduct performance studies on the four main EXA2PRO applications being ported to SkePU 3, and further extend the set of SkePU 3 example programs. A survey of how SkePU 3 embeds in the EXA2PRO high-level programming model can be found in [16]. SkePU 3 has already been made interoperable with multi-variant functions (”components”) [16], which provide a flexible escape mechanism for expressing parallelizations of computations where no skeleton fits (well) or for using accelerator types for which no appropriate SkePU backend is available (yet). Moreover, the new alternative precompiler being developed for SkePU 3 (which is technically based on the BSC Mercurium compiler) will allow for static transformations of skeleton groups, which is for now only supported to a limited degree as a runtime optimization for special skeleton sequences [13].

The SkePU 3 source code (with the clang-based precompiler) is publically available under a modified 4-clause BSD license at `https://skepu.github.io`.

Acknowledgements This work has been partly funded by EU H2020 project EXA2PRO (801015) and by the Swedish National Graduate School in Computer Science (CUGS). We thank all project partners in EXA2PRO for feedback that led to the design of SkePU 3. We also thank the National Supercomputing Centre (NSC) and SNIC for access to their HPC computing resources (SNIC 2016/5-6).

References

1. Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in FastFlow. In *Euro-Par 2012: Parallel Processing Workshops*, LNCS 7640, pages 47–56. Springer, 2013.
2. Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 298–299. Springer, 2012.
3. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
4. Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Münster skeleton library Muesli - a comprehensive overview, 2009. ERCIS Working Paper No. 7.
5. Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

6. Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation.* Pitman and MIT Press, Cambridge, Mass., 1989.
7. Marco Danelutto and Massimo Torquati. Structured parallel programming with ”core” FastFlow. In *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.
8. Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In *Proc. 4th International Workshop on Multicore Software Engineering*, pages 25–32. ACM, 2011.
9. Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *Intern. Journal of Parallel Programming*, 44(3):506–530, 2016.
10. 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.
11. Johan Enmyren and Christoph W Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.
12. Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. J. of High Perf. Computing and Networking*, 7:129–138, 2012.
13. August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019. e5003 cpe.5003.
14. August Ernstsson and Christoph Kessler. Multi-variant user functions for platform-aware skeleton programming. In *Proc. of ParCo-2019 conference, Prague, Sep. 2019, in: I. Foster et al. (Eds.), Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36, IOS press*, pages 475–484, March 2020.
15. August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, pages 1–19, 2017.
16. Christoph Kessler, August Ernstsson, Suejb Memeti, and Johan Ahlqvist. Embracing heterogeneity for exascale computing: The EXA2PRO high-level programming model. Proc. Work-in-progress session at PDP’20 conference, Västerås, Sweden, Report SEA-SR-55-4, Johannes-Kepler Univ. Linz, Austria, March 2020. ISBN 978-3-902457-55-4.
17. Matthias Korch and Thomas Rauber. Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. *J. Parallel Distributed Comput.*, 66(3):444–468, 2006.
18. Kiminori Matsuzaki and Kento Emoto. Implementing fusion-equipped parallel skeletons by expression templates. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2010.
19. Sotirios Panagiotou, August Ernstsson, Johan Ahlqvist, Lazaros Papadopoulos, Christoph Kessler, and Dimitrios Soudris. Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU. In *Proc. SCOPES’20.* ACM, May 2020.
20. Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proc. Symposium on Applied Computing (SAC’19)*, pages 1534–1543. ACM, 2019.
21. Oskar Sjöström, Soon-Heum Ko, Usman Dastgeer, Lu Li, and Christoph Kessler. Portable parallelization of the EDGE CFD application for GPU-based systems using the SkePU skeleton programming library. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer, editors, *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proc. of ParCo-2015 conference, Edinburgh, UK, Sep. 2015.*, pages 135–144. IOS Press, April 2016.
22. Michel Steuwer, Malte Friese, Sebastian Albers, and Sergei Gorlatch. Introducing and implementing the AllPairs skeleton for programming multi-GPU systems. *International Journal of Parallel Programming*, 42(4):601–618, 2013.
23. Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL–a portable skeleton library for high-level GPU programming. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS’11)*, 2011.

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

Integrating Node-Level Parallelism Abstractions into the PGAS Model

Pascal Jungblut · Karl Fürlinger

Abstract The Partitioned Global Address Space (PGAS) programming model brings intuitive shared memory semantics to distributed memory systems. Even with an abstract and unifying virtual global address space it is, however, challenging to use the full potential of different systems. Without explicit support by the implementation node-local operations have to be optimized manually for each architecture. A goal of this work is to offer a user-friendly programming model that provides portable performance across systems. In this paper we present an approach to integrate node-level programming abstractions with the PGAS programming model. We describe the hierarchical data distribution with *local patterns* and our implementation, MEPHISTO, in C++ using two existing projects. The evaluation of MEPHISTO on HPC systems shows that our approach achieves competitive scalability on the evaluated systems while requiring only minimal changes to port it from a CPU-based system to a GPU-based one using a CUDA back-end.

Keywords PGAS · Parallel Computing · Programming Models

1 Introduction

Porting performance critical software to new architectures is a challenging task. Programming abstractions like OpenMP provide means to decouple algorithms from implementation details to ease the transition. Yet, the many combinations of system configurations, back-ends and implementations force programmers to modify their code for acceptable performance. In this paper we present the integration of abstractions for node-level parallelism into the Partitioned Global

MNM-Team
Ludwig-Maximilians-Universität (LMU) München
Computer Science Department
Oettingenstr. 67, 80538 Munich, Germany
E-mail: pascal.jungblut@nm.ifi.lmu.de
E-mail: karl.fuerlinger@nm.ifi.lmu.de

Address Space (PGAS) programming model. We describe the challenges of locality, load distribution, data movement and configuration and our methods to overcome them. We evaluate the approach on a range of platforms and configurations.

First we briefly describe PGAS and the need for abstractions of node-level parallelism in a distributed system. Then we go over the challenges of such an integration and present our approach. We implemented our approach using the DASH PGAS library and Alpaka for node-level parallelism. Afterwards we present details of an implementation and evaluation results.

1.1 Partitioned Global Address Space

In PGAS each process dedicates a part of its memory to a virtual, global address space. In this global space, all processes may access memory locations on remote locations. Each object in the global space is *owned* by one process or more general a rank on which it is stored. An object is globally addressable unit. Although all global memory is accessible from all nodes, it is desirable to operate only on the local data. This is called the *owner-computes* execution pattern and stems from higher latency and possibly lower bandwidth for remote accesses compared to local ones. Note that remote accesses may also refer to other NUMA-domains on the same node.

There exist several PGAS implementations, both as dedicated programming languages like Unified Parallel C (UPC), Coarray Fortran and Chapel or as a library for existing languages like Global Arrays or Hierarchically Tiled Arrays (HTA). Although PGAS does not dictate how remote objects are accessed, many libraries use one-sided message passing. One-sided implies that only one of the communicating partners is active, i.e. it initiates the transfer, passes all necessary parameters and monitors the progress. These one-sided operations are often implemented using Remote Direct Memory Access (RDMA) so the target of an operation can be truly passive. Hence the semantics are often similar to shared memory programming where threads may read and write arbitrary shared memory at any time.

Figure 1 shows a distribution of an array across nodes. It also includes node-local private memory that cannot be read from other processes. Here, node 2 holds a private integer.

Libraries as well as programming languages provide data structures and algorithms that are designed to perform well within a PGAS context. This often includes the maximization of local accesses and resorting to topology-aware strategies.

1.2 Abstractions for Node-level parallelism

PGAS processes can be mapped to nodes, NUMA-domains, cores, or SMT threads. It can be beneficial to assign processes to NUMA-domains and use

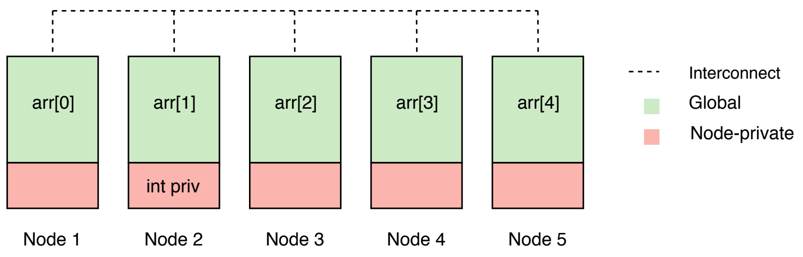


Fig. 1: An array of 5 elements distributed across 5 nodes using the PGAS model.

shared memory parallelism inside these to avoid costly cross-NUMA accesses. If we take this idea one step further the algorithms may use the hierarchical structure of the hardware to maximize locality. This is, however, not natively supported by all implementations. For example UPC only supports the concept of processes (called “threads”) whereas HTA supports data placement that reflects the system’s hierarchical hardware structure.

Programming for accelerators like GPUs and more specialized devices like TPUs or FPGAs often requires either low-level and/or vendor-specific languages and libraries like CUDA or OpenCL. These give programmers a high level of control, are often tailored to the requirements of the hardware and thus offer a lot room for optimization. However, the portability is limited: even for languages that are not vendor specific, executing the same code on other hardware necessitates a compatible implementation. Higher-level models like SYCL and Alpaka build on top of these but hide the specific *back-end*. This potentially allows users to migrate code more easily from one to another platform. Even on the same hardware this can be useful, because it allows a fast evaluation of all supported back-ends. As a side-effect these abstractions use a single-source model. This means that the whole program, including the kernels, are written in one programming language, e.g. C++.

We introduce the combination of abstractions for node-level parallelism and the PGAS programming model. This approach offers flexibility for porting codes to new distributed memory architectures. Our main contributions are:

1. A unified, hierarchical system for data placement across nodes and compute units
2. A flexible distribution of the workload
3. An interface to integrate shared memory parallelism with the PGAS model

To have fine grained control over threads and accelerators, programmers often have to resort to manually extending the PGAS environment, similar to an MPI+X approach that is employed in message passing solutions. We want to provide a usable, flexible and abstract integration of the aforementioned abstractions. For that we allow the implementations to dynamically transfer the ownership of data to an accelerator or other processing elements. The details of this idea are described in Section 2.3 and 2.4.

We first detail our approach. In Section 3 we describe the C++ implementation using two existing libraries along with an evaluation in Section 4. Related approaches are discussed in Section 5 and finally we conclude this paper and offer an outlook for future work.

2 Approach

To integrate node-level parallelism kernel acceleration into the PGAS programming model, we extended the latter to allow the distribution of local work to accelerators. Supported algorithms can hand over the control along with a task to *executors* that will gain ownership of (some part of) the data. During this phase it is prohibited for any other entity than the executor to access the data, so data that is needed outside of an executor must be copied before. Accelerators then work asynchronously on the tasks. If an accelerator and the host do not share the same memory space, data must be moved to the accelerator's memory before the execution starts. Conversely the results from the execution must eventually be copied back to the host. However, skipping redundant copies from accelerator to host or vice-versa is an optimization available in case a coherent shared memory space is available.

2.1 Definitions

Within one node there can be several *types* of processing elements (PE). We define that two PEs have the same type if they have the same computational capabilities and memory space, e.g. two cores on the same CPU have the same type while a CPU-core and a GPU-thread have different ones. Performance differences from manufacturing or changes in frequency are not considered in this paper. These could nonetheless be included if the focus was more on load balancing. Processing elements or *groups of the same type* of processing elements on a node l are called *entities* E_l. Entities can be freely defined as long as the constraints above are honored. For example it might be useful to define the PEs of CPU socket, a NUMA-domain or simply a group of cores as one entity. Each PE of an entity E is called an *instance* e_i.

2.2 Requirements

Both the PGAS implementation as well as the node-level abstraction need to fulfill some requirements for our approach to work. This is the case for most of the widely used software. We go over some of it in Section 5. The PGAS implementation needs to support two features:

– **Persistent and predictable data layout**: the runtime is not allowed to move allocated memory from one process to another. The software may support policies for the global data layout or let the user specify it manually.

- **Explicit synchronization**: the interoperability of the local and global part requires support for synchronization between processes.

On the node-level the requirements are:

- **Non-blocking memory management and kernel invocation**: our approach assumes that kernel invocation and memory movement may both be non-blocking. To our knowledge this is supported natively by all widely used implementations.
- **Separate memory space or coherence with PGAS**: the node-level abstraction must either support memory spaces per device or we expect the memory to be coherent with the global memory view, e.g. CUDA-aware MPI.

The requirements for the PGAS implementation are motivated by the support of unaware local implementations: if the data was moved during the execution of the local framework, this would lead to race conditions. On the node we must assume a symmetric situation where the two systems (local and global) are unaware of each other. Thus, it is required to either have explicit support for separate memory spaces or to provide a coherent view of the node-local memory to the PGAS library.

2.3 Data placement

A crucial part of the interaction between the node-level back-end and the PGAS environment is the strategy for data placement. How inter-node and intra-node distribution of data is configured has a large influence on the achievable performance and compatibility. The owner-computes model already motivates the usage of favorable layouts for many hardware topologies. The strategy to avoid costly remote accesses implies that data locality is a priority.

PGAS implementations let the user specify data placement configurations to varying degrees. Some like UPC derive the data distribution from the number of elements and the number of nodes. Others allow more fine grained control up to Hierarchically Tiled Arrays where each hierarchy level represents a hardware level. DASH implements the *pattern concept* which lets the user map each element of a container to an arbitrary location in global memory. We extend this pattern concept to work with entities instead of processes (i.e. MPI ranks). As a basis we use the pattern concept as described in previous work [7]. The mapping of data onto global memory location is a three-step process: first, each element in a container is assigned to a location in the global index space. Second the index space is divided into blocks which are finally distributed across units. This approach is suited best for a one-to-one mapping from processes to accelerators, but is not flexible enough for more complicated setups. Current HPC-systems may have two or three accelerators per CPU-socket.

We extended the (global) patterns that describe the mapping from the global domain space to global indices by *local patterns*. A local pattern P_l

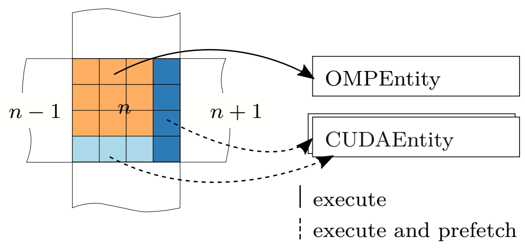


Fig. 2: 2-dimensional local blocks of memory on node n are executed on different entities. The local pattern describes the exact assignment (indicated by color) to entities. Depending on the memory spaces the blocks will be prefetched or copied to the entities memory. Here, two instances of CUDAEntity exist which represent two distinct GPUs.

maps node-local blocks B_n as defined by the global pattern P onto entities: $P_l : B_n \mapsto E$. Note that the local patterns only consider node-local blocks so this mapping is independent from the global memory layout. Additionally, the mapping of a block does not imply any data movement. Only when the data is requested to be *owned* by an algorithm, data may be moved to an entity's memory space.

The motivation for this hierarchical approach is two-fold. First each unit n may define a different pattern for its blocks B_n. This could be due to a heterogeneous architecture or for algorithmic optimizations. Second, for the same blocks multiple local patterns may be defined, because no data movement takes place. This is especially useful to balance the load individually per algorithm. For example, if there are $b_l = |B_n|$ local blocks, one local pattern may assign $w_{CPU} \cdot b_l$ blocks to the CPU and $w_{GPU} \cdot b_l$ blocks to the system's GPUs and change w_{CPU} and w_{GPU} between invocations. A pattern may define an arbitrary amount of blocks b_l for a container.

Local blocks that are assigned to an entity with a separate memory space must be copied before usage; either explicitly by issuing an appropriate call or implicitly by allocating memory in a compatible memory space. For example all recent CUDA-enabled devices support *unified memory* to transparently migrate pages from the host memory space to the CUDA-device's space. If this is not available or wanted, for example due to limitations of the PGAS implementation, it is possible to track local blocks that were already copied to an entity's memory space. The runtime may then eliminate redundant migrations. Unified memory may have a negative performance impact, since some runtime has to keep track of page faults and migrate memory pages on demand. To counter this, we can use prefetching when an entity requests ownership of a block, e.g. using `cudaMemPrefetchAsync`. However, the evaluation shows that excessive prefetching may have a negative impact on the scalability due to congestion on the memory bus.

Figure 2 shows a local pattern of a 2-dimensional array on node n. It contains 4×4 contiguous blocks of memory. The top-left ones are mapped to the OMPEntity while the bottom and right blocks are mapped to two separate instances of the CUDAEntity (i.e. two CUDA-enabled GPUs). Exemplary the seven blocks assigned to CUDAEntity will be prefetched before the kernel execution is started.

For the evaluation we implemented a flexible local pattern. Here b_i denotes the i-th local block of a node-local process with index p and t the total number of instances of the mapped entity E. The local pattern maps b_i to $e_k \in E$.

– An *identity*-pattern with $k = i$. This is only valid if $|b_l| = t$.
– A *round-robin*-pattern with $k = i - t\lfloor \frac{i}{t} \rfloor$. Trivially, if $|b_l| = t$ this is equivalent to the identity pattern.
– An *x-per-process*-pattern with $k = i - x\lfloor \frac{i}{x} \rfloor + px$.

The x-per-process-pattern is useful for cases where the number of node-local processes is less than the available entities. It distributes x consecutive local blocks to each entity. The evaluation in Section 4 contains an example with the DGX-1 system where the total runtime is optimal for four processes per node with eight GPUs.

2.4 Computation

The definition of a local pattern does not imply any data movement. Transfers only happen when an algorithm schedules work on local blocks. The execution of an algorithm is split into three phases:

1. **Initialization**: Allocate memory, copy missing blocks to the desired memory space and initialize local variables.
2. **Computation**: Pass ownership to assigned entities and execute kernels using the executors.
3. **Finalization**: Copy the result buffers to the host memory space and release ownership of the local blocks.

During the computation phase, accesses to non-owned blocks are only allowed if the owning entity does not use the same memory space, i.e. there exists a separate copy on the device memory of the mapped entity. For all other accesses the memory must be copied beforehand. Algorithm 1 shows how the execution is scheduled in more detail. This strategy is also employed in pure PGAS applications, e.g. by copying the ghost cells in a stencil application in chunks to avoid the latencies for element-wise remote accesses.

The loops over the entities and the blocks mapped to them may be executed concurrently to avoid unnecessary blocking.

3 Implementation

To test our approach we implemented it using two already existing libraries. The PGAS library used here is DASH. For the compute-intensive kernel operations,

Algorithm 1 Execution of local blocks on their assigned entities

```
Initialize
$entities \leftarrow E_l$
for all $entity \leftarrow entities$ do
    $b_e \leftarrow BlocksForEntity(entity)$
    for all $b_{e_n} \leftarrow b_e$ do
        Block until $entity$ gets ownership over $b_{e_n}$
        if $entity$ shares memory space with $b_{e_n}$ then
            Prefetch $b_{e_n}$ to $entity$
        else if Block $b_{e_n}$ currently not in $entity$'s memory then
            Copy $b_{e_n}$ to $entity$
        else
            Do nothing
        end if
        Execute kernel on $entity$ with $b_{e_n}$
        Release ownership of $b_{e_n}$
    end for
end for
```

we chose Alpaka. Both are pure C++ libraries that require a C++14 compatible compiler. We will briefly describe both frameworks in the following section and how we integrated them as MEPHISTO[1].

3.1 DASH

DASH is a C++14 PGAS library that implements distributed data structures and optimized algorithms similar to the Standard Template Library (STL). It is build on top the DASH run time (DART) which supports a range of distributed memory abstractions like one-sided MPI, OpenSHMEM or GASPI. The containers like `dash::array` and `dash::map` are compatible with their STL counterparts, so they can be used with STL algorithms. However, the STL is not optimized for the PGAS environment: DASH algorithms minimize remote access and may employ low-level implementations for better performance. For example `dash::transform_reduce` will use the reduce implementation of the underlying technology, e.g. `MPI_Reduce`. Listing 1 shows a simple program using a DASH array and two algorithms to generate a sorted array of random numbers.

```
dash::Pattern<2> blocked(8, 12, dash::BLOCKED, dash::NONE)
auto array = dash::array<double>(blocked);
dash::generate(array.begin(), array.end(), random_gen);
dash::sort(array.begin(), array.end(), array.begin());
```

Listing 1: A DASH array with 8×12 elements is created where the first dimension is specified as blocked. The distributed array is filled with random numbers and sorted using PGAS-aware algorithms.

The data placement is specified with the DASH patterns which map the global index space to processes. Listing 1 shows how a distributed array is

[1] `https://github.com/pascalj/mephisto`

allocated with the data layout defined by the 2-dimensional `dash::Pattern`. It is then filled with random numbers using `dash::generate` algorithm and finally sorted by `dash::sort`.

3.2 Alpaka

The Abstraction Library for Parallel Kernel Acceleration (Alpaka) [15] is a C++14 header only meta-programming library for node-level parallelism. It supports several back ends like C++ `std::thread`, OpenMP, Boost Fiber or CUDA. Alpaka relies heavily on compile-time configuration with types: it offers a consistent interface across all back-ends. It is possible to switch the back-end, e.g. from `std::thread` to CUDA RT, by changing one C++ type.

The programming model is similar to the one CUDA offers, i.e. the work is split up into multiple threads per block and blocks per grid. There are some conceptual extensions to support different platforms: a kernel executed on a CPU thread typically yields the best performance when it operates on many contiguous memory locations, negating the overhead of thread management. In contrast the programming model for GPUs encourages a mapping of one element per thread. Alpaka offers an additional *execution layer* that allows looping over elements. For a GPU-based back-end the loop-size would typically be one and, since it can be set at compile time, the loop is optimized out or expresses vectorization. Listing 2 shows a simple kernel invoked by Alpaka. To switch to an execution on a CUDA-based GPU only `AccCpuSerial` has to be changed to `AccGpuCudaRt` and the work division, i.e. the number of elements per thread and threads per block, should be adapted. Alpaka's blocks do not correspond to local blocks as described in Section 2.1 but to the concept known from CUDA.

```
using Dim = alpaka::dim::DimInt<3>;
using Acc = alpaka::acc::AccCpuSerial<Dim, size_t>;
// ...
// MyKernel is a C++ functor
MyKernel kernel;
alpaka::kernel::exec<Acc>(queue, workDiv, kernel);
```

Listing 2: Invocation of an Alpaka kernel. Type of accelerator is set at compile time using only types. Here `workDiv` configures the decomposition into grids, blocks and thread-elements.

Alpaka provides queues that are conceptually similar to CUDA streams. A queue belongs to one accelerator, i.e. one particular device, and schedules kernels to execute. Both blocking and non-blocking queues are provided. In this integration we exclusively use the non-blocking queues so the synchronization can be managed by MEPHISTO.

3.3 Integration

This work aims to integrate PGAS with node-level abstractions for portable performance between system configurations. As a prototype we integrated Alpaka and DASH into MEPHISTO. In this section we describe the technical details of the approach outlined in Section 2.

To implement the local patterns we extended the existing patterns in DASH. A local pattern inherits from a global pattern and extends it with one essential method:

```
template<typename Entity>
std::vector<block_spec> LocalPattern::blocks_for_entity(
  const Entity &e);
```

Listing 3: Method to retrieve the local blocks for an entity.

This method can be implemented for each entity, for example `CudaRTEntity` or `NumaEntity`. Assigning a new local pattern is a non-collective operation so no synchronization is required. With `blocks_for_entity` the DASH algorithms can retrieve the mapping of local blocks to entities and gain ownership.

We used the concept of C++'s *execution policies* and *executors* to allow users to specify on which entities an algorithm should execute. Execution policies can be used to relax the guarantees given by the STL, e.g. to have `std::for_each(std::execution::par_unseq, begin, end, f)` apply `f` in parallel and possibly vectorized over `[begin, end)`. The policies can be extended by executors to specify *where* an algorithm is executed. At the time of writing none of the executor proposals have been standardized. We extended one of them[2] to integrate an `AlpakaExecutor` that can be attached to an execution policy. An `AlpakaExecutor` can be created for each entity. It holds state, e.g. a thread pool, and may exist across algorithm invocations.

Inside the algorithm's implementation the three phases are executed as described in Section 2. We obtain the blocks with `blocks_for_entity` for each entity and start the kernels using Alpaka. In our implementation the execution devices from Alpaka are directly mapped to entities.

```
auto local_blocks   = pattern.blocks_for_entity(entity);
auto nlocal_blocks  = local_blocks.size();
std::vector<result_t> lres{nlocal_blocks};
for(int i = 0; i < nlocal_blocks; i++) {
  executor.request_ownership(entity, local_block[i]);
  executor.execute_kernel(reduction_kernel(user_func), entity,
    local_block[i], &lres[i]); // releases ownership
}
executor.synchronize();
result_t lresult = std::reduce(lres.begin(), lres.end(),
  result_t{}, user_func);
return reduce_global(lresult, user_func);
```

Listing 4: Simplified excerpt from the MEPHISTO-enabled `dash::reduce` algorithm.

[2] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r12.html`

Listing 4 shows a simple invocation of Alpaka inside of DASH. For our prototype we extended DASH's `transform`, `reduce` and `transform_reduce` which have equivalent semantics as the STL variants. The prototype also provides three predefined Alpaka-enabled executors for CPU (serial, OpenMP) and CUDA. The code resembles the three phases outlined in Section 2.4: all blocks for the current entity (`entity`) are loaded and the ownership is requested. In this case this is a blocking call, but the request for ownership may be wrapped into a `std::future` to asynchronously wait and start executing the kernel after that. The call to `execute_kernel` hands over the control to Alpaka. It calculates a reasonable work division, i.e. the dimensions of elements per thread, threads per block and the number of blocks, based on the input size and the characteristics of the entity. The algorithm synchronizes after all local blocks have been reduced by the entities. This is a node-local barrier. The results of all entities are then reduced again and finally a global result is calculated using the existing global `dash::reduce`.

3.4 Usage

Listings 2 and 4 show internals of the integration that are transparent to a user of the library. The interface of the containers and algorithms is very similar to DASH programs like the example in Listing 1. The changes required to offload the invocation of a DASH-algorithm to an entity are small:

– Specify a local pattern for the container.
– Extend the call to the algorithm with an execution policy.

To extend the example from Listing 1 we add a round-robin local pattern and pass the policy with the executor attached to the algorithms.

```
1 using Entity = mephisto::entity::CUDA;
2 dash::Pattern<2> blocked(8, 12, dash::BLOCKED, dash::NONE)
3 mephisto::local_pattern::round_robin<Entity> pattern(blocked);
4 mephisto::execution::par_unseq<Entity> par_unseq();
5 auto array = dash::array<double>(pattern);
6 dash::generate(par_unseq, array.begin(), array.end(), random_gen);
7 dash::sort(par_unseq, array.begin(), array.end(), array.begin());
```

Listing 5: MEPHISTO-enabled version of Listing 1.

Listing 5 shows the same example with MEPHISTO enabled. Line 1 specifies the entity that should be used within the executor. Only this line would need to be changed to offload to other entities. Lines 3 and 4 create the local pattern and initialize an execution policy. The execution policy is passed to both algorithms as a first parameter, similar to the standard C++ execution policies.

4 Evaluation

To evaluate our approach and its portability, we implemented a global transform-reduce operation that supports offloading with Alpaka. The data structures are allocated and managed by DASH. Where applicable they are allocated using unified memory allocators so that GPUs can access data without having to copy it first.

We evaluated the prototype using several systems:
SuperMUC-NG at LRZ is based on a Intel Xeon 8174 (Skylake-SP) with 48 cores at 2.7 GHz. The benchmarks were compiled with Intel ICC 19.0.5.281 and executed with Intel MPI 2019. Each node has 96GB of memory.
HAWK of the HLRS consists of 5,632 nodes with two AMD EPYC 7742 CPUs at 2.25 GHz with 64 cores each. We tested several compilers and OpenMP-implementations and found ICC 19.1.0.166 and AOCC (Clang) 2.1.0 with the best consistent results. The reported results were compiled with ICC.
DGX-1 P100 from NVIDIA contains eight Tesla P100 GPUs with 16 GB HBM2 cross-connected with NVLink. The host CPUs are two 20-core Intel Xeon E5-2698. **DGX-1 V100** is very similar to DGX-1 P100: it contains Tesla V100 with 16 GB HBM2 instead. The code was compiled with ICC 19.0 for the host code and the CUDA back-end with version 10.2 on both systems.

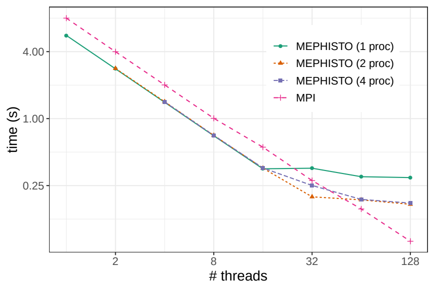
4.1 Reduction

As a micro-benchmark we implemented a reduction using `dash::transform_reduce` and observed the scaling behavior as well as the portability across architectures. The operation takes a unary function `unary(elem)` for the transform and a binary operation `binary(accum, elem)` for the reduction. One can lower the number of slow memory accesses by computing `binary(acc, unary(elem))` for each element in a block and use a tree reduction for the block, grid, entity level and finally the global level for the global result. For a given array *arr* of size a in this scenario the `transform_reduce` computes $\sum_{i=0}^{a-1} \frac{arr_i}{{arr_i}^2+1}$.

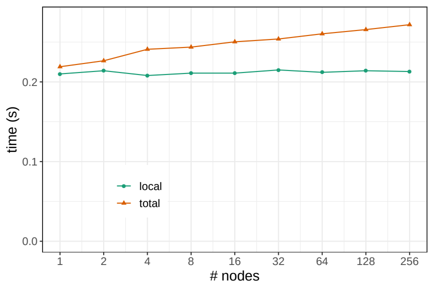
Figure 3 shows the total run time of a reduction for 10 GB total (strong scaling) and 20 GB per process (weak scaling) on up to 256 nodes in HAWK and SuperMUC-NG. In both strong scaling studies the overhead of MEPHISTO's back-end (OpenMP in this case) becomes visible as we add more threads per process. Due to Alpaka's optimized reduction kernel and its zero-overhead abstractions the total run time is slightly lower than a pure MPI implementation up until 32 threads per process.

Because PGAS is often used to program distributed systems as well, we evaluated the same implementation with 20 GB per process. For HAWK (fig. 3b) we chose the fastest combination of 4 processes each with 32 threads per process. The graph also shows the purely local portion of the computation, up until the global reduction. As expected it stays nearly constant regardless of the number of total nodes. The same effect can be seen on SuperMUC in figure 3d. Here we used 32 threads and one process per node so all node-level parallelism was managed by Alpaka. For the final global reduction MEPHISTO uses an `MPI_Allreduce` so the growing overhead can be traced back to that call.

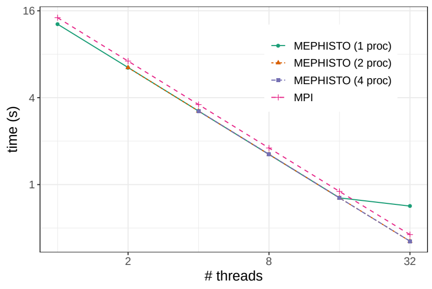
One central motivation of MEPHISTO is the optimal support for different architectures and thus portability. To demonstrate the feasibility and portabil-



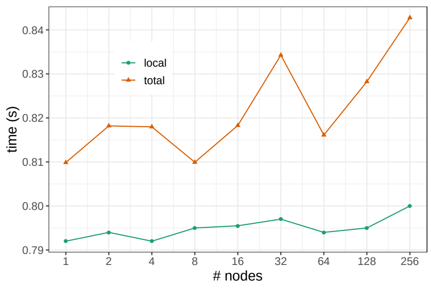
(a) HAWK: strong scaling with 10 GB.



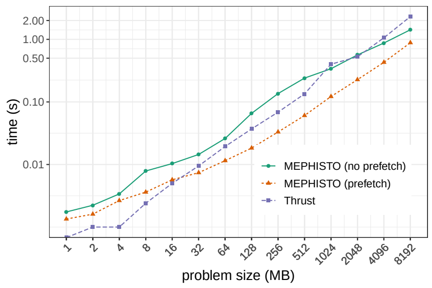
(b) HAWK: weak scaling with 4 processes and 20 GB per process.



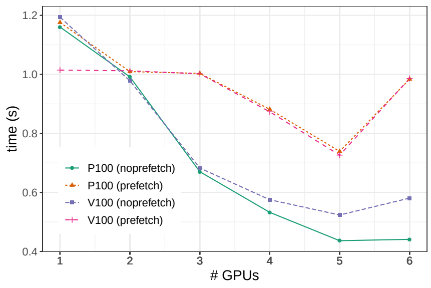
(c) SuperMUC-NG: strong scaling with 10 GB.



(d) SuperMUC-NG: weak scaling with 4 processes and 20 GB per process.



(e) One Nvidia P100



(f) DGX-1 V100 vs P100

Fig. 3: Evaluation of MEPHISTO's transform-reduce implementation. The top figures show shows strong scaling on a shared memory node of HAWK (a) and weak scaling with 20GB per process on the same system (b). The figures below show the same benchmarks on SuperMUC-NG. Figure e shows a comparison between Thrust and MEPHISTO with and without prefetch enabled whereas Figure f shows the scaling on a different number of GPUs.

ity of the approach we also evaluated the transform-reduce implementation on Nvidia's DGX-1 nodes with P100 and V100. Figure 3e shows the execution time for a transform-reduce over the problem size on one P100. We compare MEPHISTO (with and without prefetching) to Thrust [1]. This library comes bundled with CUDA and implements STL-like algorithms optimized for the execution on CUDA-enabled GPUs. We used `thrust::transform_reduce` to perform the same operation as with MEPHISTO.

Block-wise prefetching has a positive effect on the performance across all problem sizes. However, especially for small problem sizes Thrust's implementation is the fastest alternative. Note that Thrust's containers manage memory on their own and do not rely on unified memory. Therefore we include data movement between host and device but no allocation in the run time which reduces Thrust's run time compared to MEPHISTO. Scaling up from one GPU per node to up to six as depicted in Figure 3f reveals two characteristics. First, prefetching does not offer a speed-up in all scenarios. Especially when the number of concurrently used GPUs is large, prefetching has a clear negative effect on the performance. Second, the parallel efficiency is not ideal with this problem size of 10GB. Further investigation indicates that the PCI-e link between the host memory and the GPUs becomes congested. The effect is amplified when all GPUs start prefetching whole blocks during the execution.

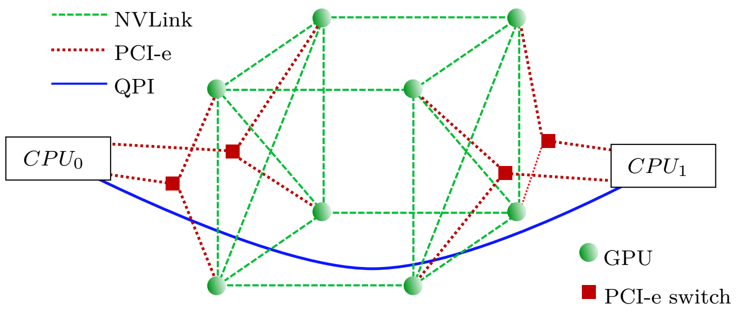
During the evaluation it became clear that the architecture of DGX-1 limits the throughput for a single process. Figure 4a shows the topology of the DGX-1: each CPU is connected to four P100/V100 via two PCI-e switches. The CPUs in turn are connected with Intel's QuickPath Interconnect (QPI) and the GPUs with NVLink. With one process pinned to CPU_0, all blocks assigned to GPU 4, 5, 6 and 7 must be sent over QPI to the second socket and PCI-e to the GPUs. With the *x-per-process*-pattern the benchmark can be started with one, two, four or eight processes and with eight, four, two or one GPU per process. The evaluation in Figure 4b shows the benefit of each configuration, especially when working with prefetching. With one process assigned to each GPU the PCI-e lanes become congested. Even with two processes (one per CPU) there is still a higher runtime than with a 4×2 configuration. This matches the topology directly and shows the need for flexible process-to-accelerator assignments.

4.2 MiniMD

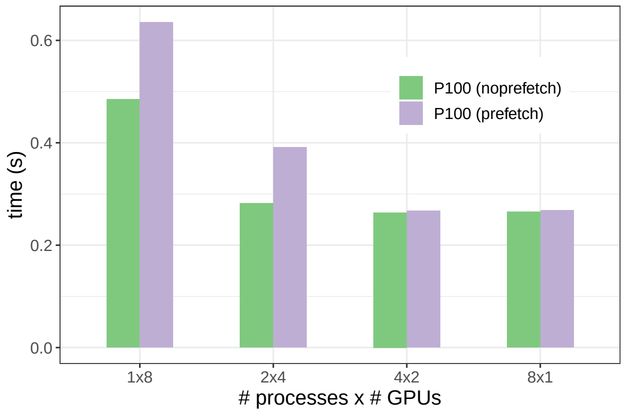
As a more complex case we ported MiniMD [5] to DASH and then used MEPHISTO to add shared memory parallelism. This step required only some minimal changes in the code:

– Choose one of the pre-configured entities (e.g. using OpenMP) to be used.
– Specify which calls to DASH's algorithms should be performed in parallel using executors.
– Optional: change the data layout using DASH's patterns.

MiniMD is a molecular dynamic proxy application that mimics the workload of LAMMPS [14]. The 3d-space is decomposed into a configurable number of



(a) Simplified DGX-1 architecture [10]



(b) Runtime of MEPHISTO's transform-reduce combinations of the number of processes and GPUs per process.

Fig. 4: DGX-1 architecture and the resulting performance differences. The x-per-process local pattern allows flexible mapping of GPU.

cells. Each cell holds a dynamic number of atoms. To update the position and velocity of each atom, the force of the surrounding atoms is calculated. MiniMD configures a cut-off distance r_{cutoff} and only calculates forces of neighbors inside this radius. Because the space domain is already split into cells, only the cells within r_{cutoff} have to be considered, reducing the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for n atoms. For our evaluation we used the default configuration of the reference implementation that requires each cell to consider at maximum 27 neighbor cells. After a fixed amount of iterations, the atoms need to be re-assigned to cells due to changed positions. The atoms are stored in a `dash::NArray` with four dimensions: three dimensions representing cells and one for atoms in each cell. An `NArray` is static in size so we re-allocate when a cell overflows during the binning process. At process borders the bins are mirrored after each binning phase and the positions and velocities of atoms are updated during each iteration.

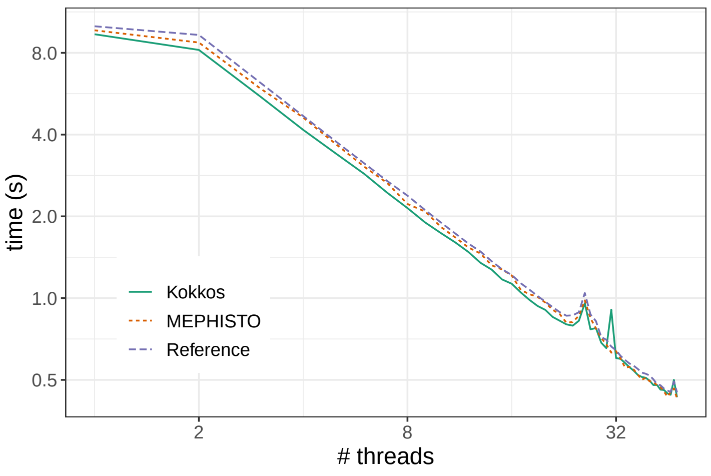


Fig. 5: Comparison of the shared memory runtime of the reference, Kokkos and MEPHISTO implementations on SuperMUC-NG.

For the updates on atoms we are using global pointers that asynchronously fetch updated positions on each iteration. The computation of the neighbor forces is done using `dash::transform` with a `par_unseq` execution policy attached with an `AlpakaExecutor`.

Figure 5 shows the comparison of the reference implementation and the Kokkos and MEPHISTO ports on one node of SuperMUC-NG with an increasing number of threads. The anomalies at around 24 threads point to a NUMA-related issue, although we observed similar spikes for two socket-pinned MPI processes (with half the number of threads) on all variants. Here all variants perform similar. We tested several other combinations of processes and threads and all performed similar or minimally slower.

5 Related Work

A large number of frameworks, compiler extensions and dedicated languages have been developed for shared and distributed systems. We highlight the most relevant related works and discuss how they relate to Alpaka, DASH and MEPHISTO.

5.1 Shared memory abstractions for parallelism

Similar to Alpaka is Kokkos [6], an open source abstract interface for shared memory programming. For the details on the differences between both, refer to [15]. SYCL[3] is cross platform and built on top of OpenCL that supports

[3] `https://www.khronos.org/sycl/`

potentially a wide range of accelerators. In contrast to OpenCL, it is also single-source. SYCL is a part of Intel's OneAPI approach for heterogeneous, parallel programming. Thrust [1] is a library that implements STL-like algorithms with a CUDA, Thread Building Blocks or OpenMP back-end. Because it ships its own data structures, the compatibility with most PGAS implementations will be limited. OpenMP itself is an open standard for shared memory programming. Version 4.5 introduced the *target* environment with data regions which can be used to allocate device memory and offload computation to accelerators. However, it does not yet support explicit memory access to the different memory types of accelerators (e.g. block- and grid-wide memory). Very similar to OpenMP is OpenACC, also an open standard, which provides access to block-shared memory. All of these offer no integration with distributed memory paradigms.

5.2 PGAS implementations

PGAS is either implemented as a programming language or in a library for an existing language. UPC [4], Coarray Fortran [13], Chapel [2], and X10 [3] are languages with built-in PGAS support. All have configurable data placement with respect to the nodes, making it usable for our approach. However, these languages require a special compiler that limits the portability. High-level libraries such as DASH, Global Arrays or UPC++ provide PGAS support without the need for a special compiler. They ship with distributed data structures and algorithms. There are also more low-level ones such as GASPI or one-sided MPI. These are used as a provider by the aforementioned libraries. The integration of the shared memory abstractions with our approach is possible with all of these, because the only requirements for the PGAS implementation are user-defined data layouts and a form of synchronization between processes. All of the above offer both.

5.3 Hybrid approaches

There has been some work on combining shared memory parallelism in a PGAS environment, much like MPI+X with the message passing model. The MiniMD application was ported to PGAS in [11] using UPC and POSIX threads. Jose et al. implement a PGAS runtime to achieve considerable speedups due to shared memory support [8] These focus on specific implementation whereas MEPHISTO is agnostic to the back-end. Most similar to our work is [12]. The authors combine already mentioned directive-based OpenACC with XcalableMP to XcalableACC, enabling offloading to accelerators. It requires a dedicated compiler but offers accelerator-to-accelerator communication. HPX [9] supports distributed job scheduling but uses *Active Global Address Space*, thus violating our requirement that allocated memory is not moved by the runtime.

6 Conclusion

6.1 Limitations

The approach, aside from the restrictions outlined in Section 2, requires users to be aware of race conditions when working with a shared memory space. Currently there exists no explicit method to synchronize during the execution. However, it is possible to use either synchronization from both the PGAS and the node-level abstraction's implementation.

Further, our prototype implementation currently only implements unified memory for the CUDA back end. This in turn requires CUDA-aware MPI to provide a coherent view of the data to MPI.

6.2 Summary

In this paper we present our approach to integrate node-level abstractions for parallelism with the PGAS programming model in a user-friendly way. It extends existing methods for data distribution to include *local patterns* that map contiguous memory blocks to processing resources. Further, it includes a simple execution model using these patterns to execute kernels on *entities*, heterogeneous processing elements. We combine two existing projects as MEPHISTO to achieve flexible kernel acceleration and offloading in distributed systems with partitioned global memory. The evaluation of the approach on different CPU and GPU architectures shows competitive performance when compared to industry-standard implementations.

6.3 Outlook

In the future we want to focus on automatic load balancing (auto tuning) between entities. For now we only tested the execution on one entity at a time, albeit with multiple instances (threads/GPUs) of each. When the host and multiple entities may execute during a single invocation of an algorithm, the benefits of smart load balancing seem worthwhile investigating to minimize the idle time for each entity.

Another more complex endeavor is the scheduling of kernels on executors independently of the invocation of their containing algorithms. This could bring benefits due to better cache usage and the removal of barriers. For example, we extended DASH's `transform_reduce` to support heterogeneous hardware with Alpaka. Part of the speed-up we gained over two separate calls to `transform` and `reduce` comes from the temporal locality the combined implementation provides. A runtime could detect adjacent calls to `transform` and `reduce` and instead execute the compact version instead, similar to kernel fusion.

References

1. Nathan Bell and Jared Hoberock. Chapter 26 - Thrust: A productivity-oriented library for CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359 – 371. Morgan Kaufmann, Boston, 2012.
2. Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
3. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.
4. UPC Consortium et al. Upc language specifications v1. 2. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2005.
5. Paul Crozier and Steven Plimpton. miniMD v. 1.0. Technical report, Sandia National Laboratories, 2009.
6. H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
7. Tobias Fuchs and Karl Fürlinger. Expressing and exploiting multi-dimensional locality in DASH. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 341–359. Springer, 2016.
8. Jithin Jose, Sreeram Potluri, Hari Subramoni, Xiaoyi Lu, Khaled Hamidouche, Karl Schulz, Hari Sundar, and Dhabaleswar K Panda. Designing scalable out-of-core sorting with hybrid MPI+PGAS programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–9, 2014.
9. Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.
10. Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.
11. Mingzhe Li, Jian Lin, Xiaoyi Lu, Khaled Hamidouche, Karen Tomko, and Dhabaleswar K Panda. Scalable MiniMD design with hybrid MPI and OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–4, 2014.
12. Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *2014 First Workshop on Accelerator Programming using Directives*, pages 27–36. IEEE, 2014.
13. Robert W Numrich and John Reid. Co-array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM New York, NY, USA, 1998.
14. Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1993.
15. Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. Alpaka–an abstraction library for parallel kernel acceleration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 631–640. IEEE, 2016.

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

On Single-Valuedness in Textually Aligned SPMD Programs

Frédéric Dabrowski

Abstract We propose a formal definition of single-valuedness in the context of SPMD programs. In this definition, the concomitance of the computation of values at distinct processes relies on a logical time, which is induced by textually aligned program points. We show how textual alignment and single-valuedness can be used to ensure proper use of collectives in SPMD programs. We focus on synchronization barriers and Direct Remote Memory Access (in BSP style) and sketch the analysis implemented in our prototype analyser.

Keywords SPMD · BSP · Collective primitives · textual alignment · single-valuedness, · formal semantics · static analysis

1 Introduction

In the SPMD programming model [2,11], a collection of parallel processes executes a *Single Program on Multiple Data*. Unlike the *Single Instruction, Multiple Data* (SIMD) [12] model, where all processors execute the same instructions at the same pace, the SPMD model allows replicated processes to follow distinct flows of control. Several communication means may be proposed by SPMD programming languages (MPI[13], OpenMP[27], BSPlib[34], ...). The most popular are *Direct Remote Memory Access* (DRMA) and message passing. In both cases, *collective operations* (or *collectives* for short) play a major role. They expose a simple synchronization scheme: all processes execute the same sequence of collectives, performing a global synchronization for some of them. Broadcasts, reductions and global barriers are examples of collectives. However, behind the apparent simplicity of the model, the ability to execute distinct instruction streams with no restriction may lead to programming errors.

Univ. Orléans, INSA Centre,Val de Loire,
LIFO EA 4022 Orléans, France
E-mail: frederic.dabrowski@univ-orleans.fr

To preclude these types of errors, one can introduce a strict separation at the programming language level between global and parallel flows of control. The former produces a single instruction stream, which every process follows. The latter produces multiple instruction streams free of collectives [26,25,14]. It is noteworthy that this distinction had already been present in the early definition of the SPMD model, quoting [9]: "the participating processes follow a different parallel flow of control, but all the processes follow the same global flow of control". However, SPMD programs are most often written in general programming languages using libraries, the result being that the two flows are mixed up (e.g., MPI, implementations of the BSP model [16,36]). This simple observation highlights the need for tools capable of reconstructing the global control flow in library based implementations.

Current practices show that global barriers are usually *textually aligned*, which means that all processes synchronize on the same textual occurrences. In other words, their use is confined to global control flow. The same applies to other kinds of collective operations. This model not only simplifies programming, but also it is a prerequisite for some program analysis [21,5,4]. In previous work [19,6,8], we formally defined textual alignment and prove that enforcing textual alignment of synchronization barriers is a sufficient condition to avoid deadlocks. In this paper, we consider another relevant property: *single-valuedness*. This property states that an expression occurring in the text of a program evaluates to the same value at all processes. For some collectives, not only processes must execute the same instruction but also they must execute it on the same data. For example, in MPI all processes must pick the same source process when performing a broadcast. By combining textual alignment and single-valuedness, one can enforce this stronger property. We propose a formal definition of single-valuedness and show how these two properties can be used to prove the correctness of programs using collectives, not only global synchronization points but also more general collectives. In particular, this combination solves a problem raised by Aiken et al. in [1,15]; namely defining the notion of computing the same value at programs points occuring between physical synchronization points. The primary idea is that all textually aligned program points can be seen as logical barriers. In particular, we show how to check the correctness of DRMA operations à la BSP. More details will be given in Section 2. In [3], we propose sufficient condition to ensure correct use of barriers, however this work did not propose a formal definition of single-valuedness. In this paper we provide such a definition based on textual alignement and propose a simpler semantics.

In [1,15], Aiken and Gay introduced the concept of structural correctness in non textually aligned programs. Unlike our work it only considers parameterless primitives. Their work had been used for the design of the Titanium language [35,10]. A later proposal introduced textually aligned barriers in Titanium by revisiting structural correctness. This proposal was finally replaced by a dynamic approach [22] after it has been observed that it was flawed [20, 22]. A recent work also considers dynamic approaches to the problem of textual alignment of collectives [23]. In [18], the authors consider an empirical static analysis for detecting Multi-Valued expressions, which is used to lower the number of dynamic checks. Barrier checking for non-textually aligned is also studied in [37].

In Section 2 we present a subset of the BSPlib language. Is supports drma communications. In Section 3 we present BSP_{DRMA} a toy languages that mimics the features of this BSPlib subset. We define its operational semantics and the two kinds of errors we aim to rule out, deadlocks and stack mismatch. Sections 4 and 5 introduce formal definitions of textual alignment and single valuedness. We show how they can be used to rule out the two kind of errors introduced in section 3. In Section 6 we sketch the prototype checker we are developing. In section 7 we conclude and sketch future work;

2 BSP

In the Bulk Synchronous Parallel (BSP) model, a static number processes progress at the same pace, synchronizing on global barriers. Between two consecutive synchronizations, processes perform local computations and issue communication requests. These requests will be handled at the time of the next synchronization barrier. Communication requests are either message-passing requests or direct remote memory access requests. Several implantation of the BSP model exists (BSPlib[34], BSML[28], BSPOnMPi[30], MultiCoreBSP[32],...). Here we focus on BSP DRMA primitives as proposed by the BSPlib and provide a short description of their behavior.

Each process has access to its identifier and to the total number of process through the functions

`bsp_pid_t bsp_pid(void)` and `bsp_pid_t bsp_nprocs(void)`

The execution of a program proceeds in successive steps, which are local computation steps issuing communication requests, terminated by global synchronization barriers. Requests issued during a step are served at the end of this step (Figure 2). Synchronization barriers are performed by a collective call to

```
void bsp_sync(void)
```

Processes can communicate by accessing the memory of other processes. To refer to a remote memory location processes rely on a mapping between local addresses. This mapping is built programmatically. Processes collectively push memory locations on a distributed stack. As the stacks have the same size at every processes, they can be seen as a global stack of p-tuples (where p is the number of processes). Elements of a tuple are the physical adresses at each process of a replicated logical adress. For example in (Figure 2) both process 0 and process 1 have pushed the address of the variable `y` which is a at process 0 and b at process 1. In this case the address a of process 0 is mapped to the address b of process 1 (and vice versa). Pushing a memory location on the stack is done by a call to

```
void bsp_push_reg(const void* addr, bsp_size_t size)
```

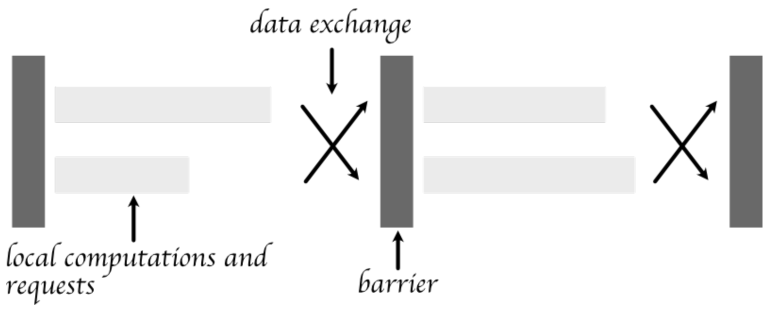


Figure 1 Execution of a BSP program

where `addr` is the local memory location and `size` is the length of the buffer. A line of the global stack can be removed by a collective call to

```
void bsp_pop_reg(const void* addr)
```

Finally, a process can read or write to a remote location by calling the following functions

- `void bsp_put(bsp_pid_t pid, const void * src, void *dst, bsp_size_t offset, bsp_size_t size)`: writes `size` bytes from the location `src` to the offset `offset` of the remote location `dst`.
- `void bsp_get(bsp_pid_t pid, const void * src, void *dst, bsp_size_t offset, bsp_size_t size)`: reads `size` bytes from the offset `offset` of the remote location `src` to the location `dst`.

Example 1 Here, each process writes its id in the memory of its right neighbor.

```
x = bsp_pid();
y = 0;
bsp_push_reg(&y,sizeof(int));
bsp_sync();
bsp_put((pid + 1) % bsp_nprocs(), &x, &y, 0, sizeof(int));
bsp_pop_reg(&y),
bsp_sync();
```

First each process pushes the address of variable `y` and perform a barrier to update the stack (`bsp_put_reg` and `bsp_pop_reg` issue requests that are served at the end of the current step). Then each process requires a write of the value of `x` to the distant register matching the address of `y` in the stack. Finally, processes pop the addresses and perform a barrier to realize the communication.

In this paper we consider two kinds of errors:

- deadlocks: occur when one process is blocked on a barrier while another process is terminated (we only consider partial correctness)
- stack mismatch: two processes pop locations occurring in distinct lines of the stack. As we have seen each line is a mapping between remote addresses, processes must thus agree on which line to remove.

In the next section we present a simple language and its semantics. This language is used to formalize these kinds of errors.

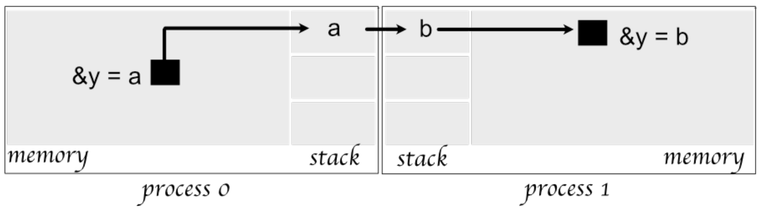


Figure 2 Two processes registering physical addresses into their stacks

3 bsp$_{\mathbf{drma}}$

The language BSP$_{\text{DRMA}}$ is a variation of the While language, extended by a minimal set of primitives dedicated to BSP-like DRMA programming. It supports global synchronization, dynamic register allocation, push and pop operations and update of remote registers. It is akin to the subset of BSP of Section 2, besides a few limitations whose aim is to simplify the presentation. Dynamic allocation is supported only in the context of DRMA communications, there is no heap allocated structures. We use the name *register* instead of memory location to reflect this. Registers hold data of size one. We modify the communication scheme to avoid non determinism due to concurrent write to remote registers. More precisely, messages received by a process are buffered on a per source process basis and can be read separately by the target process. These restrictions are made without loss of generality:

- in practice, shared memory locations are used to exchange contiguous data (dynamic structures are serialized, processes don't exchange pointers),
- concurrent writes are communication errors that we do not deal with in this paper.

3.1 Syntax

We consider a countable set of variables *Var* that we note x, y, z, ... An expression a is a term built from integers, variables, arithmetic operations and the constants *pid* and *nprocs*. The two constants denote respectively the current process id and the total number of processes. Statements are decorated with labels taken in a countable set *Lab*, elements of which are noted ℓ (possibly with subscript). The syntax of the language is given in Figure 3.1. Each label occurs at most once in a statement. When not necessary labels are omitted. The instruction `skip` does nothing and returns control to the rest of the computation. An instruction $x := a$ stores the value of the expression a in the variable x. Sequences, conditionals and loops behave as usual.

- global synchronizations are performed by `sync`. It is a blocking instruction that must be performed collectively by all processes. Pending requests are realized at the time the synchronization occurs.

$$a ::= pid \mid nprocs \mid x \mid \ldots \qquad \text{(expressions)}$$

$$i ::= \texttt{skip} \mid x := a \mid \texttt{sync} \mid \texttt{init}\ x \mid \texttt{push}\ x \mid \texttt{pop}\ x \mid x[a] \leftarrow y$$

$$s ::= [i]^\ell \mid s; s \mid [\texttt{if}\ a]^\ell\ \{s\}\ \{s\} \mid [\texttt{while}\ a]^\ell\ \{s\} \mid [\texttt{with}\ x\ \leftarrow\ y[a]]^\ell\ \{s\} \quad \text{(statements)}$$

Figure 3 Syntax of BSP$_{\text{DRMA}}$

- a fresh register is allocated and stored in the variable x by the instruction `init` x.
- a register stored in x is pushed (resp. poped) by the instruction `push` x (resp. `pop` x).
- An instruction $y[a] \leftarrow x$ performs a request to update the register paired with the register stored in y at process a. The new value is that of x.
- an instruction [`with` $x \leftarrow y[a]$] $\{s\}$ stores in x the last value written by processor a in the register paired with the register stored in y and executes s. The value was written at the previous step. If no such value exists, s is dropped and the control returns immediately to the rest of the computation.

As stated before, remote writes in BSP$_{\text{DRMA}}$ differs a bit from what can be found in BSP. Here a register is rather like a buffer in which other processes can write at a reserved position. Let's rephrase the example of the previous section in BSP$_{\text{DRMA}}$ to illustrate this.

```
x := pid;
init z; push z; sync;
z[(pid + 1) mod nprocs] ← x;
pop x; sync
[with y ← z[(pid + 1) mod nprocs]] {skip};
```

We use the local variable y to store the value written in z by the right neighbor of the process. Specifying the emitter rather that reading the last written value (of a scheduling dependent process) rules out non determinism.

3.2 Semantics

We give an operational semantics for our language as a small-step transition system. The semantics records execution paths (sequences of labels) followed by processes during the execution. These annotations will be used in the definitions of textual alignment and single-valuedness. They have no effect on the behavior of programs and could be erased.

3.3 Definitions

A path pt is a finite sequence of labels. A register is a triple (u, pt, ℓ) where u belongs to a countable set of names $\mathcal{U}$, pt is a path and ℓ a label. A value $v \in \mathcal{V}$ is either an integer or a register. An environment E is a mapping from variables to values.

The semantics of expressions is given by a function $[\![.]\!] : \mathit{Env} \times \mathit{nat} \to \mathcal{V}$ where the second parameter is the processes id at which the evaluation takes place. The special constant *pid* returns the process id so we have $[\![\mathit{pid}]\!](E, i) = i$. The special constant *nprocs* returns the number of processes so we have $[\![\mathit{nprocs}]\!](E, i) = p$. Unlike the process id, the number p of process, which is unique for a given execution, is left implicit to improve readability. The semantics of the rest of expressions is as usual. For the sake of simplicity, we assume that $[\![.]\!](E, i)$ is a total function.

The semantics is a transition system over global states, which consists of vectors of size p where p is the number of processes. Components of vectors are processes states. They have the form of tuples $(s, (E, S, B, R), pt)$ where, at process i,

- s is either a statement or the termination symbol $\bullet$
- E is the environment of process i
- S is the contribution of process i to the stack
- B is the buffer of process i; it is a function mapping registers and process ids to values. It is a partial function. If $B(u, j) = v$ then process j has requested an update of u with value v at i.
- R is the history of requests performed by process i since the beginning of the current step. Requests, noted r, are defined by

$$\begin{array}{rll} r ::= & \mathit{push}(u, pt, \ell) & \text{push request} \\ & \mid \mathit{pop}(u, pt, \ell) & \text{pop request} \\ & \mid \mathit{write}((u, pt, \ell), i, v) & \text{message request} \end{array}$$

  A $\mathit{write}((u, pt, \ell), i, v)$ denotes a write of value v at the location paired with u at process i. We note $R_1 \cdot R_2$ the concatenation of R_1 and R_2.
- pt is the sequence of labels crossed by i since the beginning of the execution.

As usual a context C denotes the rest of the computation, it has the form of a statement with a hole. Given a context C and a statement s we note $C[s]$ for the result of placing s in the hole in C. Contexts are defined by the grammar:

$$C ::= [\,] \mid [\,]; s$$

We generalize the notation to the termination symbol $\bullet$ by the equations: $[\bullet] = \bullet$ and $[\bullet]; s = s$. Given a function f, We note $f[x \mapsto v]$ the function defined by

$$f[x \mapsto v](y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

If f is partial, we note $dom(f)$ its definition domain. We note $\Gamma \cdot \gamma$ a sequence Γ extended with the element γ.

A global transition, $v \to v'$ moves from one global state to the next. Figure 4 gives the two rules that define global transitions.

- Rule local specifies individual computation steps. The vector is updated according to the result of a transition of the picked component (see below). Note that process may execute `sync` instructions occurring at distinct labels.
- Rule sync specifies global steps which occur when all process reach a synchronization barrier. Communications requested during the computation step are served. All components of the vector are updated according to the result. The definition of $\triangleright$ is given in Figure 5. We note $R^{\downarrow}$ (resp. $R^{\uparrow}$) the sequence, in order, of registers pushed (resp poped) in R.

In both cases we record the labels crossed by processes. Other rules specify local transitions. They have the form

$$pt, \ell \vdash_i s, E, B \to s', E', r$$

where i is the process id, pt is the sequence of labels crossed so far by i and ℓ is the current label. Executing s with environment E and buffer B leads to the statement s' and the environment E' performing the request r (or ϵ if no request is performed).

- The `skip` instruction, assignment, conditional and loops behave as usual (rules skip, assign if$_1$, if$_2$,while$_1$ and while$_2$).
- An instruction `init` x generates a fresh register stored in variable x (rule init). The new register is annotated with pt and ℓ. We assume a function *fresh* that maps a buffer to a fresh register, two buffers with the same domain are mapped to the same register.
- An instruction `push` x (resp. `pop` x) performs a request to push (resp. pop) the register (u, pt, ℓ) stored in x. A request $push((u, pt, \ell))$ (resp. $pop((u, pt, \ell))$) is issued (rules push and pop).
- An instruction $x[a] \leftarrow y$ performs a request to write the value v stored in y to a remote register paired with the register stored in x . The value j of a is the target process. A request $write(u, j, v)$ is issued (rule send).
- A statement [`with` $x \leftarrow y[a]]^{\ell}$ $\{s\}$ reads the message sent to the current process to the register stored in y and stores it in x (rule receive$_1$). The control is then returned to the statement s. If no such message exists, the control is simply returned to the rest of the computation (rule receive$_2$).

Given an environment E, the initial state $init(E)$ is $(E, \epsilon, \emptyset, \epsilon)$. We note $E \vdash s \leadsto_i (s', st, pt)$ if $\langle (s, init(E), \epsilon), \ldots, (s, init(E), \epsilon) \rangle \to^* v$ and $\pi_i(v) = (s', st, pt)$. The relation $\leadsto_i$ is the projection on process i of an execution. We note $\to^*$ the reflexive transitive closure of v and say that v' is reachable from v if $v \to v^*$. Local transitions are deterministic and so are global transitions, thanks to the `with` construct. The semantics is deterministic in the sense defined below.

$$\frac{}{pt, \ell \vdash_i [\texttt{skip}]^\ell, E, B \to \bullet, E, \epsilon}\ \text{skip}$$

$$\frac{[\![a]\!](E, i) = v}{pt, \ell \vdash_i [x := a]^\ell, E, B \to \bullet, E[x \mapsto v], \epsilon}\ \text{assign}$$

$$\frac{\mathit{fresh}(B) = u}{pt, \ell \vdash_i [\texttt{init}\ x]^\ell, E, B \to \bullet, E[x \mapsto (u, pt, \ell)], \epsilon}\ \text{init}$$

$$\frac{E(x) = (u, pt, \ell)}{pt, \ell \vdash_i [\texttt{push}\ x]^\ell, E, B \to \bullet, E, \mathit{push}(u, pt, \ell)}\ \text{push}$$

$$\frac{E(x) = (u, pt, \ell)}{pt, \ell \vdash_i [\texttt{pop}\ x]^\ell, E, B \to \bullet, E, \mathit{pop}(u, pt, \ell)}\ \text{pop}$$

$$\frac{E(y) = v \qquad E(x) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j}{pt, \ell \vdash_i [x[a] \leftarrow y]^\ell, E, B \to \bullet, E, \mathit{write}(u, j, v)}\ \text{send}$$

$$\frac{E(y) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j \qquad B(u)[j] = v}{pt, \ell \vdash_i [\texttt{with}\ x\ \leftarrow\ y[a]]^\ell\ \{s\}, E, B \to s, E[x \mapsto v], \epsilon}\ \text{receive}_1$$

$$\frac{E(y) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j \qquad B(u)[j]\ \mathit{undefined}}{pt, \ell \vdash_i [\texttt{with}\ x\ \leftarrow\ y[a]]^\ell\ \{s\}, E, B \to \bullet, E, \epsilon}\ \text{receive}_2$$

$$\frac{[\![a]\!](E, i) \neq 0}{pt, \ell \vdash_i [\texttt{if}\ a]^\ell\ \{s_1\}\ \{s_2\}, E, B \to s_1, E, \epsilon}\ \text{if}_1$$

$$\frac{[\![a]\!](E, i) = 0}{pt, \ell \vdash_i [\texttt{if}\ a]^\ell\ \{s_1\}\ \{s_2\}, E, B \to s_2, E, \epsilon}\ \text{if}_2$$

$$\frac{[\![a]\!](E, i) \neq 0}{pt, \ell \vdash [\texttt{while}\ a]^\ell\ \{s\}, E, B \to s, E, \epsilon}\ \text{while}_1$$

$$\frac{[\![a]\!](E, i) = 0}{pt, \ell \vdash [\texttt{while}\ a]^\ell\ \{s\}, E, B \to \bullet, E, \epsilon}\ \text{while}_2$$

$$\frac{\begin{array}{c} pt, \ell \vdash_i s, E, B \to s', E', r \\ \pi_i(\boldsymbol{v}) = (C[s], (E, S, B, R), pt) \qquad \pi_i(\boldsymbol{v'}) = (C[s'], (E', S, B, R \cdot r), pt \cdot \ell) \end{array}}{\boldsymbol{v} \to \boldsymbol{v'}}\ \text{local}$$

$$\frac{\begin{array}{c} \boldsymbol{w} \rhd \boldsymbol{w'} \\ \forall i. \pi_i(\boldsymbol{v}) = (C_i[[\texttt{sync}]^{\ell_i}], \pi_i(\boldsymbol{w}), pt_i) \qquad \forall i. \pi_i(\boldsymbol{v'}) = (C_i[\bullet], \pi_i(\boldsymbol{w'}), pt_i \cdot \ell_i) \end{array}}{\boldsymbol{v} \to \boldsymbol{v'}}\ \text{sync}$$

Figure 4 Dynamic Semantics

We have $w \triangleright w'$ where $\pi_i(w) = (E_i, S_i, B_i, R_i)$ $\pi_i(w') = (E'_i, S'_i, B'_i, R'_i)$ if

- $E'_i = E_i$
- $S'_i = (S_i / R_i^{\uparrow}) \cdot R_i^{\downarrow}$
- $B'_i[u](j) = \begin{cases} v & \text{if } \mathit{write}((u', (pt', \ell')), i, v) \in R_j \text{ and } \mathit{pos}(S_i, u) = \mathit{pos}(S_j, u') \\ \textbf{error} & \text{if } \mathit{write}((u', (pt', \ell')), i, v) \in R_j \text{ and } \mathit{pos}(S_i, u) \neq \mathit{pos}(S_j, u') \\ \text{undefined} & \text{otherwise} \end{cases}$

  If several messages have the same source and same target, we take the last one.
- $R'_i = \epsilon$

and no **error** occurs. We note Γ/Γ' for

$$\begin{aligned} \Gamma/\epsilon &= \Gamma \\ \Gamma \cdot \gamma / \Gamma' \cdot \gamma' &= \begin{cases} \Gamma/\Gamma' & \text{if } \gamma = \gamma' \\ (\Gamma/\Gamma' \cdot \gamma') \cdot \gamma & \text{otherwise} \end{cases} \\ \epsilon/\Gamma \cdot \gamma &= \textbf{error} \\ \textbf{error} \cdot \gamma &= \textbf{error} \end{aligned}$$

Figure 5 Exchange

Indeed, local transition are deterministic and "scheduling" choice are not significant. Moreover, thanks to the `with` constructs, communications are also deterministic.

Lemma 1 *Let v, v_1 and v_2 be vectors. If $v \to^* v_1$ and $v \to^* v_2$ then there exists v' such that $v_1 \to^* v'$ and $v_2 \to^* v'$.*

3.4 Programming errors

As stated before, we intend to rule out two kinds of errors: deadlocks and stack mismatches. In this section we introduce their formal definitions. A deadlock occurs when a process is blocked at a barrier waiting for a terminated process. All processes are stuck (we do not consider infinite loops).

Definition 1 A deadlock occurs in v, if the following property holds

$$\mathit{deadlock}(v) = \exists i, j. \pi_i(v) = (C[\texttt{sync}], -, -) \wedge \pi_j(v) = (\bullet, -, -).$$

A statement is *well synchronized* if no deadlock occurs in any state reachable from an initial state.

Example 2 The following program is not well synchronized because some processes perform less synchronizations than others

$$x := \mathit{pid}; \texttt{while } x > 0 \texttt{ do } x := x - 1; \texttt{sync done}$$

More precisely all processes but 0 (which is terminated) are stuck on the first barrier. Thanks to determinism, deadlocks are reproducible and are easily observed by programmers.

A stack mismatch occurs when two processes perform incompatible push and pop requests. Intuitively, requests are compatible if all processes perform the same number of push/pop request and if ”concomitant” pop requests refer the same positions in the stacks. Intuitively, a mapping between remote registers, as defined by stacks can be removed but it cannot be modified (see example 3).

Definition 2 A stack mismatch occurs in v if there exists i and j such that $\pi_i(v) = (C_i[\texttt{sync}], (-, S_i, -, R_i), -)$, $\pi_j(v) = (C_j[\texttt{sync}], (-, S_j, -, R_j), -)$ and

$$\begin{array}{lr}
|R_i^{\downarrow}| \neq |R_j^{\downarrow}| & \vee \\
|R_i^{\uparrow}| \neq |R_j^{\uparrow}| & \vee \\
(\exists k.(R_i^{\uparrow})_k = u_i \wedge (R_j^{\uparrow})_k = u_j \wedge \mathit{pos}(S_i, u_i) \neq \mathit{pos}(S_j, u_j)) &
\end{array}$$

A statement is *well matched* if no stack mismatch occurs in any state reachable from an initial state.

Example 3 The following program is not well matched because process 0 tries to pop the first line while other processes try to pop the second line.

$\texttt{init}\ x; \texttt{init}\ y; \texttt{push}\ x; \texttt{push}\ y; \texttt{sync}; \texttt{if}\ \mathit{pid} = 0\ \texttt{then}\ \texttt{pop}\ x\ \texttt{else}\ \texttt{pop}\ y\ \texttt{end}; \texttt{sync}$

But the following statement is well matched (processed may pop any line).

$$\texttt{init}\ x; \texttt{init}\ y; \texttt{push}\ x; \texttt{push}\ y; \texttt{sync}; \texttt{pop}\ x; \texttt{sync}$$

Remember that we only consider programming errors related to misuses of collectives. In particular, we don't consider local errors such as trying to pop or to use non pushed registers In the next section, we will show how to use textual alignment and single-valuedness to define a programming methodology that rules out the two kind of errors we have introduced.

4 Textual Alignment

Instances of textually aligned labels are crossed by all processes at the same pace, at least from a logical point of view. Intuitively, textually aligned code blocks could be executed in a pure SIMD mode. We will return to this remark later. Some programs are obviously classified as textually aligned, whereas others require more explanations to justify their classification.

Example 4 Consider the following statements written in C.

```
    if (bsp_pid() > 0) {sync()} else {sync()}

    if (bsp_pid() < bsp_nprocs() ) {sync()} else {sync()}

    x = 0;
    while (x<bsp_nprocs()) {if (x = bsp_pid()) sync(); x = x +
  1}

```

```
    x = 0;
    while (x < 3) {if (x == 2) sync; x = x + 1}

```

The first statement (line 1) is clearly not textually aligned. Processes execute *sync* instructions that occur in distinct branches. Obviously, label equality is the weaker reasonable condition. On the opposite, it is obvious that the second statement (line 3) should be considered textually aligned. In the third statement (lines 5,6), all processes perform the same number of iterations of the loop. Yet, they call the *sync* primitive at distinct iterations. Although the behaviors of all processes are observationally equivalent, this statement should not be considered as textually aligned (think of loop unrolling). On the opposite, the last statement (lines 8,9) is textually aligned.

Intuitively, a label ℓ is said to be textually aligned if, whenever a process reaches a *textual occurrence* of ℓ, other processes will eventually reach the same occurrence (we consider partial correctness only). An obvious way to distinguish occurrences of a label ℓ is to consider the set of execution paths leading to ℓ. However, for our purpose, this definition is not appropriate as exemplified by the following statement:

$$\texttt{if}\ b\ \texttt{then}\ x := 0\ \texttt{else}\ x := 1\ \texttt{end}; x := x + 1$$

In this case, we would like to consider that whichever branch is taken, the same textual occurence of the last assignment is reached. We note Δ_l the functions that retain, from a path, the labels of loops surrounding ℓ in a program statement s (we omit the statement which is always clear from the context). Obviously, the information extracted by Δ_ℓ is sufficient to distinguish distinct occurrences of ℓ in the execution trace of processes.

Definition 3 A label ℓ in a statement s is *textually aligned* if for all E, if exists $i < p$ such that $E \vdash_i s \rightsquigarrow (s', st_i, pt_j)$ where $entry(s') = \ell$ then for all $j < p$, there exists st_j and pt_j such that $E \vdash_j s \rightsquigarrow (s', st_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$.

This definition relate local executions rather that global executions. This is because, in general, the execution of the same textual occurrence of a label at distinct processes may be separated by arbitrarily many synchronizations. Altough, in this paper, we will consider programs with textually aligned barriers, this property cannot be assumed a priori. For such programs it will be the case that same textual occurrences of labels are always reached during the same steps by all processes. Indeed, instances of textually aligned label occur in the same order in distinct processes as stated by the following lemma.

Lemma 2 *Let ℓ_1 and ℓ_2 be two textually aligned program points in s and let E be an environment.*

- $E \vdash_i s \rightsquigarrow (s_i, st_i, pt_i)$ *and* $E \vdash_i s \rightsquigarrow (s'_i, st'_i, pt'_i)$
- $E \vdash_j s \rightsquigarrow (s_j, st_j, pt_j)$ *and* $E \vdash_j s \rightsquigarrow (s'_j, st'_j, pt'_j)$

- *$entry(s_i) = entry(s_j) = \ell_1$ and $entry(s'_i) = entry(s'_j) = \ell_2$*
- *$\Delta_{\ell_1}(pt_i) = \Delta_{\ell_1}(pt_j)$ and $\Delta_{\ell_2}(pt'_i) = \Delta_{\ell_2}(pt'_j)$*

then if $pt_i \prec pt'_i$ we also have $pt_j \prec pt'_j$ where $\prec$ is the prefix order.

We omit the proof of this intermediate result, the interested reader can refer to our previous Coq[33] developments [7].

Proposition 1 *If all barriers of a statement are textually aligned then this statement is well-synchronized.*

Proof Let i and j be two process ids. We prove that for all $n > 0$ if $E \vdash_i s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_i, pt_i)$ where i has crossed n barriers then there exists a vector v such that $\langle \ldots (s, init(E), \epsilon) \ldots \rangle \to^* v$, $\pi_j(v) = (C[[\texttt{sync}]^\ell], w_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$. The proof is by induction on n. Suppose that $E \vdash_i s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_i, pt_i)$, then by hypothesis we have $E \vdash_j s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$. We have to prove that these two local state are part of the same synchronisation. Suppose that the local state of $(C[[\texttt{sync}]^\ell], w_j, pt_j)$, which we call A_j, corresponds to another synchronization (otherwise we are done). We distinguish two cases, whether A_j occurs in a synchronization preceding $A_i = (C[[\texttt{sync}]^\ell], w_i, pt_i)$ or not.

- Suppose A_j was part of a previous synchronization. Then we have a previous state of i of the form (s'_i, w'_i, pt'_i) such that $pt'_i \prec pt_i$ and, by induction hypothesis, $\Delta_\ell(pt'_i) = \Delta_\ell(pt_j)$. It comes $\Delta_\ell(pt_i) = \Delta_\ell(pt'_i)$ which is incompatible with $pt'_i \prec pt_i$.
- Now suppose A_j was not part of a previous synchronization. We assumed A_j is not part of same synchronization as A_i. Then there exists a state $A'_j = (C'[[\texttt{sync}]^{\ell'}, w'_j, pt'_j)$ which is part of the same synchronization as A_i and $pt'_j \prec pt_j$. But by textual alignment, there exists $A'_i = (C[[\texttt{sync}]^{\ell'}], w'_i, pt'_i)$ such that $\Delta_{\ell'}(pt'_i) = \Delta'_\ell(pt_j)$. We have $pt_i \prec pt'_i$ otherwise A_i and A'_i denote the same local state and then we have $\ell = \ell'$ and $\Delta_\ell(pt_j) = \Delta_\ell(pt_i) = \Delta_\ell(pt'_i) = \Delta_\ell(pt'_j)$ which is leads to a contradiction. Finally we get a contradiction by Lemma 2.

From this result it is immediate that assuming a deadlocks leads to a contradiction. □

In this section we have shown how textual alignment can be used as a sufficient condition to ensure correctness of parameterless collectives such as global synchronization barriers. More elaborated collectives require not only the execution of the same instruction but also coherency in the actual values the instruction is used with. This is the case, for example, of the broadcast instruction in MPI for which all processes must agree on the source process. In the next section, we consider single-valuedness and show how it can be used to prove correctness of *push* and *pop* instructions.

5 Single Valuedness

As defined in the literature, a variable is single-valued if all processes map it to the same value at the same time. As the authors says in [1],

> [...] 'at the same time" is a slippery notion in a setting with asynchronous execution. Only at global synchronization points (i.e., barriers, broadcasts, and the start and end of execution) is it possible to assert anything useful about the state of all processes,

In this section, we claim that textual alignment leads to a more effective notion of time. We show that any textually aligned label can be use to state properties on the state of all processes. Textually aligned program points are logical synchronization points. They can be seen as common clock ticks, of which synchronization barriers are simply special cases.

Single-valuedness can be defined in term of Leibniz equality. However, in the context of BSP_{DRMA} we need a bigger equivalence relation. Indeed, we will propose a correctness criterion for which it is necessary to identify registers allocated at the "same time". This is why registers are annotated in the semantics. Two values v_1 and v_2 are similar, noted $v_1 \backsimeq v_2$, if one of the following equations is satisfied:

- $v_1 = v_2$ if v_1, v_2 are integers and $v_1 = v_2$
- $(u, pt, \ell) = (u', pt', \ell')$ if $\ell = \ell'$ and $\Delta_\ell(pt) = \Delta_\ell(pt')$

Definition 4 A variable x in a statement s is *single-valued* at label ℓ if for all E, if exists $i < p$ and $j < p$ such that $E \vdash_i s \rightsquigarrow (s_i, st_i, pt_i)$ and $E \vdash_j s \rightsquigarrow (s_j, st_j, pt_j)$ where $entry(s_i) = entry(s_j) = \ell$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$ then $E_i(x) \backsimeq E_j(x)$.

The notion of single value easily extends to expressions by requiring that all variables are single-valued and that *pid* doesn't occur in the expression.

Example 5 Here, the pushed variable is single-valued in the first statement but not in the second.

$$\texttt{init}\ x; \texttt{push}\ x$$

$$\texttt{if}\ b\ \texttt{then}\ \texttt{init}\ x\ \texttt{else}\ \texttt{init}\ x\ \texttt{end}; \texttt{push}\ x$$

Next we show how to use the single-valuedness property to enforce a proper use of collectives operating on the stack. Intuitively, if *push* and *pop* instructions are textually aligned, all processes perform the same list of *push*/*pop* instructions (up-to parameters). This is sufficient for *push* instructions but not for *pop* instructions because all concomitant *pop* instructions must refer to the same line in the stack. This requirement is met by enforcing single-valuedness of stacked values.

Proposition 2 *If all barriers in s are textually aligned and if* push/pop *instructions in s are textually aligned and single-valued then s is well matched.*

Proof We prove a stronger property which is that for every reachable state v there exists R, R' and S such that for all process i we have $\pi_i(v) = (C_i[\mathtt{sync}], (-, S_i, -, R_i), -)$ and

$$R_i^{\downarrow} \backsimeq R \wedge R_i^{\uparrow} \backsimeq R' \wedge S_i \backsimeq S$$

where $\backsimeq$ is applied peer to peer. Because barriers are textually aligned, we can reason on a single step and conclude by induction on the number of barriers crossed so far. So suppose the property holds at the beginning of the step. By hypothesis and by Lemma 2, push/pop instructions at ℓ that occurs in i at path pt_i also occurs, in same order, in j at path pt_j and $\Delta_\ell(pt) = \Delta_\ell(pt_j)$. By the single value hypothesis we have request that are performed in the same order with compatible (with respect to $\backsimeq$) values. Moreover, the stacks are equal because of the hypothesis (pushed/pop values were equals at the end of the previous step). It there is no previous step, the initial state trivially satisfy the conditions. $\square$

We have showed how to combine textual alignment and single-valuedness to define sufficient conditions to ensure correctness of DRMA collectives in a BSP like language. As stated before, we have focused on issues related to the collective nature of *push* and *pop* instructions. It is still possible for a program to get stuck because processes collectively fail, for example by trying to *pop* a non valid register. Such behaviors can be ruled out by simple local correctness properties. We briefly discuss this issue in the next section.

6 Analysis

We sketch the design principles underlying our prototype BSP analyser for C programs[29]. It guarantees that a validated program contains no errors related to the use of collectives. We use the Frama-C[31] toolbox to check that calls to the following functions are textually aligned and that the parameter `addr` is single-valued.

- `void bsp_sync(void)`
- `void bsp_push_reg(const void* addr, bsp_size_t size)`
- `void bsp_pop_reg(const void* addr)`

Frama-C is a platform dedicated to C source code analysis which can be extended by new plugins written in OCaml. It provides basic tools: abstract syntax tree, control-flow graphs, program-dependence graph and a dataflow analysis engine among others. Our prototype handles all BSP functions of section 2. Given a C program it computes a control flow graph decorated with the following information

- program point with data-dependencies on the process id

- program point with control-dependencies on pid-dependent values

The analysis proceeds in two steps, both of which extensively rely on the program dependence graph generated by Frama-C.

- First, we compute a safe approximation of the set of functions that may return pid-dependent values. This is done by an inter-procedural dataflow analysis which propagates dependencies between function definitions. A fix-point iteration is performed. Each step propagates dependencies through edges of the program dependence graph. It start with the singleton {`int bsp_pid(void)`}
- Second, we proceed to a trivial intra-procedural analysis to infer the above mentionned information from the previous step.

From the decorated control flow graph it is easy to extract safe approximations of the following properties.

- single-valuedness of branching conditions
- single-valuedness of push and pop instructions
- textual alignment of all program points (a program point is textually aligned as soon as all surrounding branching are single-valued)

Indeed, the analysis captures an over approximation of the intended properties. As an example of loss of precision, it is possible for a value to be pid-dependent and yet to be single valued. Consider the condition at label ℓ in the following example.

$$\texttt{if } [pid < nprocs]^{\ell} \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}$$

Remember, that our prototype does not address local correctness of calls to functions operating over the stack. It is then possible for a program to try to pop a memory location that is not in the stack. Conceptually, this feature can be added to our prototype quite easily. It requires a simple dataflow analysis computing abstract stacks of reachable states. In practice, we observed that they are never two distinct memory locations allocated at the same program point into the stack. Thus, we argue that it is enough to have abstractions that abstract exactly, for each allocation point, the last memory location allocated at that point. More precisely, a dataflow analysis can abstract values as follows: the last memory location (u, pt, ℓ) allocated is abstracted by the label ℓ and all value are abstracted by $\top$. This is a simple kill-gen analysis where an abstract memory location ℓ (generated by a memory allocation) is turned into $\top$ (killed) as soon as a new memory location is allocated ℓ. Adding this feature to our prototype requires to extend the value analysis EVA[24] of Frama-C to discriminate the last memory locations allocated at each program point. We expect to add this feature to our prototype in a short time.

Another issue not handled by our prototype is the detection of programs in which the sizes of shared buffers are not sufficient to receive messages. To rule out this kind of error one could use a high-precision bound analysis. Bound checking for array accesses can be done with the EVA plugin of Frama-C.

7 Conclusion

We have formally defined sufficient conditions to ensure correctness of collective DRMA communications à la BSP. These sufficient conditions rely on the formal definitions of two important properties, namely textual alignement and single-valuedness. The formal definition of textual alignment improves on our previous work in term of simplicity. As far as we know this is the first formal definition of single-valuedness based on textual alignment. We have shown that the latter permits to define the former not only at synchronization points but also at all textually aligned points. Indeed, the latters behave as synchronization points at the logical level. Building on these results we have sketched the principles of a static analysis and we have presented our prototype checker. In future work, we expect to extend our checker with an appropriate value analysis. This will allow to rule out local errors as well. Another direction is to consider more collective communication scheme like broadcast, map, reduces and others. We also expect to study the interaction of our analysis with the PARCOACH [17] dynamic checker. In this context, we expect our analysis to reduce the overhead of the instrumentation.

References

1. Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.
2. Larbey F. Auguin M. Opsila: an advanced simd for numerical analysis and signal processing. In *Microcomputers: developments in industry, business, and education, Ninth EUROMICRO Symposium on Microprocessing and Microprogramming*, pages 311–318, Madrid, 1983.
3. Wadoud Bousdira, Arvid Jakobsson, and Frederic Dabrowski. Safe Usage of Registers in BSPlib. In *SAC 2019*, Limassol, Cyprus, April 2019.
4. Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. *An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection*, pages 106–120. Springer International Publishing, Cham, 2017.
5. C. Chen, W. Huo, L. Li, X. Feng, and K. Xing. Can we make it faster? efficient may-happen-in-parallel analysis revisited. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 59–64, Dec 2012.
6. Frederic Dabrowski. Textual Alignment in SPMD Programs. In *SAC '18: Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, Pau, France, April 2018.
7. Frédéric Dabrowski. Jlamp 2019, coq artefact. `https://github.com/DabrowskiFr/coq-jlamp2018`.
8. Frédéric Dabrowski. A denotational semantics of textually aligned spmd programs. *Journal of Logical and Algebraic Methods in Programming*, 108:90 – 104, 2019.
9. Frederica Darema. *SPMD Computational Model*, pages 1933–1943. Springer US, Boston, MA, 2011.
10. Hilfinger P. N. (editor), Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium Language Reference Manual, Version 1.16.8. Technical Report UCB//CSD-04-1163x, Computer Science, UC Berkeley, 2004.
11. Darema F. Spmd model: past, present and future, recent advances in parallel virtual machine and message passing interface. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science*, Santorini/Thera, Greece, 2001.

12. Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.
13. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1.
14. F. Gava and F. Loulergue. A static analysis for bulk synchronous parallel ml to avoid parallel nesting. *Future Generation Computer Systems*, 21(5):665 – 671, 2005. Parallel computing technologies.
15. D. Gay. *Barrier Inference.* PhD thesis, University of California, Berkeley, 1998.
16. Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Comput.*, 24(14):1947–1980, December 1998.
17. Pierre Huchant, Emmanuelle Saillard, Denis Barthou, Hugo Brunie, and Patrick Carribault. PARCOACH Extension for a Full-Interprocedural Collectives Verification. In *Second International Workshop on Software Correctness for HPC Applications*, Dallas, United States, November 2018.
18. Pierre Huchant, Emmanuelle Saillard, Denis Barthou, and Patrick Carribault. Multivalued expression analysis for collective checking. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 29–43, Cham, 2019. Springer International Publishing.
19. Arvid Jakobsson, Frédéric Dabrowski, Wadoud Bousdira, Frédéric Loulergue, and Gaetan Hains. Replicated synchronization for imperative {BSP} programs. *Procedia Computer Science*, 108:535 – 544, 2017. International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.
20. A. Kamil. Problems with the titanium type system for alignment of collectives. unpublished note, 2006.
21. Amir Kamil and Katherine Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199, Berlin, Heidelberg, 2006. Springer-Verlag.
22. Amir Kamil and Katherine Yelick. *Enforcing Textual Alignment of Collectives Using Dynamic Checks*, pages 368–382. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
23. Andreas Knüpfer, Tobias Hilbrich, Joachim Protze, and Joseph Schuchart. *Dynamic Analysis to Support Program Development with the Textually Aligned Property for OpenSHMEM Collectives*, pages 105–118. Springer International Publishing, Cham, 2015.
24. CEA List. Theevaplug-in. http://frama-c.com/download/frama-c-eva-manual.pdf, 2019.
25. Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005.
26. Frédéric Loulergue and Gaétan Hains. *Functional parallel programming with explicit processes: Beyond SPMD*, pages 530–537. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.
27. OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.
28. Software. BSML: Bulk synchronous parallel ml, a library for BSP programming in OCaml. `https://traclifo.univ-orleans.fr/BSML/`.
29. Software. Bspcheck, a c static analyzer for bsp programming. `https://github.com/DabrowskiFr/bspcheck`.
30. Software. BSPOnMPI, a platform independent software library for developing parallel programs. `http://bsponmpi.sourceforge.net/`.
31. Software. Frama-c, an extensible and collaborative platform dedicated to source-code analysis of c software. `https://frama-c.com/`.
32. Software. MultiCoreBSP, BSP programming on modern multicore processors. `http://www.multicorebsp.com/`.
33. The Coq Development Team. Coq. `https://coq.inria.fr`.
34. Mick van Duijn, Koen Visscher, and Paul Visscher. BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model. `http://bsplib.eu/`.

35. Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.
36. A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. Multicorebsp for c: A high-performance library for shared-memory parallel programming. *Int. J. Parallel Program.*, 42(4):619–642, August 2014.
37. Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

A Divide-and-conquer Parallel Skeleton for Unbalanced and Deep Problems

Millán A. Martínez · Basilio B. Fraguela ·
José C. Cabaleiro

Abstract The Divide-and-conquer (D&C) pattern appears in a large number of problems and is highly suitable to exploit parallelism. This has led to much research on its easy and efficient application both in shared and distributed memory parallel systems. One of the most successful approaches explored in this area consists in expressing this pattern by means of parallel skeletons which automate and hide the complexity of the parallelization from the user while trying to provide good performance. In this paper we tackle the development of a skeleton oriented to the efficient parallel resolution of D&C problems with a high degree of unbalance among the subproblems generated and/or a deep level of recurrence. Our evaluation shows good results achieved both in terms of performance and programmability.

Keywords Algorithmic skeletons · Divide-and-conquer · Template metaprogramming · Load balancing

1 Introduction

Divide-and-conquer [1], hence denoted D&C, is a strategy widely used to solve problems whose solution can be obtained by dividing them into subproblems, separately solving those subproblems, and combining their solutions to compute the one of the original problem. In this pattern the subproblems have the same nature as the original one, thus this strategy can be recursively applied to them until base cases are found. The independence of the subproblems

Millán A. Martínez · Basilio B. Fraguela
Universidade da Coruña, CITIC, Computer Architecture Group. 15071. A Coruña. Spain
Tel: +34-881 011 219
E-mail: {millan.alvarez, basilio.fraguela}@udc.es

José C. Cabaleiro
Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. 15782. Santiago de Compostela, Spain
Tel: +34 881 816 421
E-mail: jc.cabaleiro@usc.es

allows exploiting parallelism in this pattern, and the fact that it has a well defined structure allows expressing it by mean of algorithmic skeletons [6], which automate the management of typical patterns of parallelism [19]. Since skeletons hide the implementation details and the difficulties inherent to parallel programming from the user, they largely simplify the development of parallel versions of these algorithms with respect to manual implementations. In fact several parallel skeletons for expressing D&C problems have been proposed in the literature, either restricted to shared memory systems [18,10,8] or supporting distributed memory environments [7,3,9,15,5,11]. In addition to the large number of problems that exhibit a D&C pattern, properly designed D&C skeletons can be used to express or implement other very common patterns such as map or reduce [12]. This wide applicability makes it extremely interesting in our opinion to develop highly optimized skeletons for this pattern.

In this paper we present a novel C++ parallel skeleton for the resolution of D&C problems in shared memory environments. Our proposal improves upon the existing solutions when considering applications with a large degree of unbalance among the subproblems generated and/or a large depth in the recursion of the algorithm. As we will see, these situations can sometimes lead the existing skeletons to either provide suboptimal performance or even be inapplicable. Our skeleton, called *parallel_stack_recursion* is an evolution of the *parallel_recursion* skeleton proposed in [10] after a complete redesign and reimplementation. Our new implementation is available at https://github.com/fraguela/dparallel_recursion together with the material published in [10] and [11].

The rest of this paper is organized as follows. The next section discusses the related work. Then, Section 3 reviews the key aspects and main problems of the D&C skeleton *parallel_recursion*. Our solution to these problems is presented in Section 4, where the new skeleton and its implementation details are explained. The evaluation of the performance and programmability of the new implementation is presented in Section 5. Section 6 is devoted to our conclusions and future work.

2 Related Work

The divide-and-conquer parallel pattern is supported by a number of skeletons in the literature. Like ours, some of them are restricted to shared memory systems, the advantage with respect to the distributed system counterparts being the reduced communication and synchronization costs as well as the easier load balancing. In this category we find the Java-based *Skandium* library [18], which follows a producer-consumer model in which the tasks are pushed and popped from a shared ready queue by the participating threads. The C++ *DAC* parallel template [8] supports three different underlying runtimes (OpenMP [4], Intel TBB [22] and FastFlow [2]) who take the responsibility of balancing the workload among the threads. These two proposals have in common that the skeleton only needs to be fed the input data and functions for identifying base cases, splitting non-base cases, combining partial results and solving base cases, which are indeed the basic components of a D&C algorithm.

Our work derives from the C++ *parallel_recursion* skeleton [10], explained in detail in Section 3, which relies on the Intel Threading Building Blocks (TBB) runtime [22] for tasking and load balancing. This skeleton allows to optionally provide a partitioner object that helps the runtime to decide when the resolution of a non base problem must be solved sequentially or in a parallel fashion. This is in contrast with the previously discussed approaches, which always apply parallelism to the resolution of every non base case. They can, though, mimic a similar behavior at a higher programming cost by identifying as base cases also those non basic problems whose parallelization is not worthy and including a sequential D&C implementation for their resolution in the function that takes care of the base cases. As we will see, even with this higher degree of control, *parallel_recursion* is not well suited to problems that exhibit a large degree of unbalance. The partitioner concept of [10] is inspired by the TBB, which offers several higher order functions, many of which support partitioners, as well as lower level mechanisms to exploit D&C parallelism, although it lacks a specific skeleton for this parallel pattern.

In the distributed memory realm we find *eSkel* [7], which provides parallel skeletons for C on top of MPI, including one for D&C. The API is somewhat low-level because of the limitations of the C language, which leads for example to the exposure of MPI details. A template library for D&C algorithms without this problem is *Muesli* [5], which is designed in C++ and built on top of MPI and OpenMP, although the latter has only been applied to data-parallel skeletons, so that its D&C skeleton is only optimized for distributed memory.

Lithium [3] is a Java library specially designed for distributed memory that provides, among others, a parallel skeleton for D&C. The implementation is based on macro data flow instead templates and extensively relies on runtime polymorphism. This is in contrast with *Quaff* [9], whose task graph must be encoded by the programmer by means of C++ type definitions which are processed at compile time to produce optimized message-passing code. These static definitions mean that tasks cannot be dynamically generated at arbitrary levels of recursion and, although the library allows skeleton nesting, it has the limitation that this nesting must be statically defined.

Finally, there are D&C skeletons specifically oriented to multi-core clusters, as they combine message-passing frameworks in order to support distributed memory with multithreading within each process. This is the case of [14], which supports load balancing by means of work-stealing. The proposal though is only evaluated with very balanced algorithms and unfortunately, contrary to ours, it is not publicly available. Furthermore, their balancing operations always involve a single task, which, as we will see, can be very inefficient. Another work in this area is *dparallel_recursion* [11], an evolution of [10] in which the shared memory portion relies on [10] and offers thus the same behavior.

It is interesting to notice that while skeletons have been traditionally directly used by programmers, their scope of application is growing thanks to very promising novel research. Namely, the development of techniques and

```
template<typename T, int N>
struct Info : Arity<N> {
  bool is_base(const T& t) const; //base case detection
  int num_children(const T& t) const; //number of subproblems of t
  T child(int i, const T& t) const; //get i−th subproblem of t
};

template<typename T, typename S>
struct Body : EmptyBody<T, S> {
  void pre(T& t); //preprocessing of t before partition
  S base(T& t); //solve base case
  S post(T& t, S *r); //combine children solutions
};
```

Listing 1: Class templates with pseudo-signatures for the info and body objects used by `parallel_recursion`

tools to identify computational patterns and refactor the codes containing them [16,17] not only simplifies the use of skeleton libraries by less experienced users but it can even lead to the automatic parallelization of complex codes on top of libraries of skeletal operations.

3 The *parallel_recursion* skeleton

In this section we will describe the D&C algorithm template *parallel_recursion*, including the limitations that led us to propose a new alternative in this field.

3.1 Syntax and semantics

Specifying a D&C algorithm requires providing functions to decide whether a problem is a base case or, on the contrary, it can be subdivided into subproblems, to subdivide a non-base case, to solve a base case, and to obtain to solution of a non-base problem by combining the solutions of its subproblems. The analysis performed in [10] noticed that the two first functions are mostly, and often exclusively, related to the nature of the input data structure to process, while the two latter ones more strongly relate to the computation being performed. For this reason, *parallel_recursion* relies on two separate objects to provide these two groups of elements. We now describe in turn the requirements for these objects, which are modeled by the C++ class templates `Info` and `Body` shown in Listing 1.

The object that describes the structure of the problem is called the *info* object and it must belong to a class that provides the member functions `is_base(t)`, which indicates whether a given problem `t` is a base case or not, `num_children(t)`, which gives the number of subproblems in which a non-base problem can be subdivided, and finally `child(i, t)`, which returns the `i`-th child of the non-base problem `t`. As shown in Listing 1, the class `Info` for this object must derive from a class `Arity<N>` provided by the library, where `N` is either the number of children of every non-base case of the problem, when it is fixed, or the identifier `UNKNOWN` when this value is not a constant. In the first case, `Arity<N>` automatically provides the `num_children` function member so that users do not need to implement it.

We call *body* object the one that provides the computations. Its class must provide the functions of the class template `Body` shown in Listing 1. Here, `base(t)` provides the solution for a base case `t`, while `post(t, r)` receives in the pointer `r` the array of solutions to the subproblems in which a non-base problem `t` was subdivided so that combining them, maybe with some additional information from `t`, it can compute the solution to the parent problem `t`. The object class must also support a function member `pre(t)` that allows performing computations on the problem `t` before even checking whether it is a base problem or not, as it was found to be useful for some D&C problems. The library provides a class template `EmptyBody<T,S>` that can be used as base case for the body object classes, where `T` is the type of the problems and `S` is the type of the solutions, although this it not required. The main advantage of `EmptyBody` is that it provides empty implementations of all the body functions required, so that deriving a class from it avoids writing unneeded components.

Besides the input problem and the two aforementioned objects, the skeleton accepts a fourth optional argument called the *partitioner*. Its role is to indicate when parallelism should be applied during the execution of the skeleton. The partitioner can be of three different classes. If the programmer provides a partitioner from the `simple_partitioner` class, the skeleton will parallelize the resolution of any non-base problem. This behavior is the only possible one in the other shared-memory skeletons we know of [18,8]. Creating, scheduling and synchronizing parallel tasks for every level of a D&C recursion tree may be very inefficient, particularly when the tree contains many computations of small size. For this reason, two other alternatives are supported. If the partitioner belongs to the `auto_partitioner` class, the skeleton will apply some heuristics in order to try to generate a number of parallel tasks that keeps busy the threads available while allowing for some load balancing, in case the tasks were not of an identical size. In general it is impossible for the skeleton to know in advance the size of the tasks, and even the number of children of each subproblem. Therefore a third possibility is to use a `custom_partitioner`, in which the user controls when to apply parallelism by means of a `do_parallel(t)` member function that she must provide in the *info* object. The function must return a boolean that indicates whether a problem `t` should be solved using parallelism or not.

Listing 2 illustrates the use of the skeleton in a problem consisting in adding the value `val` stored in a tree of nodes of type `tree_t` in which each node has a variable number of children whose pointers are stored in a `std::vector<tree_t*>` called `children`. The base cases are identified by the `is_base` member function of the info object, whose class is `TreeAddInfo`,

```
struct TreeAddInfo: public Arity<UNKNOWN> {
  bool is_base(const tree_t *t) const { return t->children.empty(); }

  int num_children(const tree_t *t) const { return t->children.size(); }

  tree_t *child(int i, const tree_t *t) const { return t->children[i]; }
};

struct TreeAddBody: public EmptyBody<tree_t *,int> {
  int base(tree_t * t) { return t->val; }

  int post(tree_t * t, int *r) { return std::accumulate(r, r + t->children.size(), t->val); }
};

int r = parallel_recursion<int>(root, TreeAddInfo(), TreeAddBody(), auto_partitioner());
```

Listing 2: Reduction on a tree using `parallel_recursion`

as those nodes whose vector of children is empty. These nodes just contribute their stored value `val` to the reduction, as shown in the member function `base` of the body object, whose class is `TreeAddBody`. The `num_children` member function of the info object provides the correct variable number of children of each node, each child being obtained by means of the `child` member function of the same object. Finally, the `post` member function of the body object uses the `std::accumulate` function to add the values returned by all the children of the node together with the value `val` stored in the node itself. The last line in Listing 2 illustrates the invocation of the skeleton with the root of the tree and applying an `auto_partitioner` to take the decisions on parallelization.

3.2 Implementation and limitations

The implementation of `parallel_recursion` heavily relies on templates and static parallelism, which are resolved at compile time, in order to avoid the costs associated to runtime polymorphism. As for parallelism, it uses the low level API of the Intel TBB [22] to build and synchronize the parallel tasks. This also means that the library relies on the TBB scheduler to balance the workload among the threads, which is achieved by means of work-stealing.

At the top level, the skeleton proceeds generating new parallel tasks to solve the children subproblems of each non-base problem as long as the partitioner in use decides that parallelism must be applied. However, when the partitioner decides that a given problem must be solved sequentially, the skeleton assigns the solution of that problem to a purely sequential highly optimized code that relies on the info and body objects to perform the computation, but which never checks again the possibility of creating new parallel tasks within the resolution of that problem. As a result, the task graph generated by the skeleton takes the form of a tree that grows with new tasks as long as the partitioner in use recommends doing so, and which once this is not the case, reaches leaf tasks. Each leaf task recursively solves in a sequential fashion an independent D&C problem.

The aforementioned strategy is very successful in many situations, but it presents two main limitations. First, there is the issue of load balancing. As seen in [10], the skeleton can provide good performance for some unbalanced problems by generating more tasks than threads and letting the TBB scheduler balance them. However, sometimes this does not work well because the unbalance among tasks generated at high levels of the D&C tree may be too big to keep all the threads busy, while generating parallel tasks down to the level needed to attain this balance could be very detrimental to performance. In addition, even if we wished to assert as much control as possible by means of a custom partitioner, it might not be possible to estimate whether it is interesting or not to parallelize a given problem with the information available.

The second limitation is related to the implementation of the serial computations performed by the skeleton when it decides not to parallelize a D&C problem. They follow a very efficient and simple recursive strategy that relies on stack memory for the recursive calls. Unfortunately, stack memory is much more limited than other kinds of memory, and if this recursion is very deep it can be easily exhausted, breaking the program. As in the case of the load balancing problem, this can be solved by reducing the size of the sequential tasks applying parallelism up to deeper levels in the D&C tree. However, often this does not solve the stack memory problems either, as this limitation also exists in the case of the parallel computations of this skeleton. The reason is that the frame in which a parallelized problem is considered remains in the stack until the lower level tasks it generates finish and return their results, which also implies continuous growth in the depth of the stack memory as deeper and smaller parallel tasks are generated. Furthermore, the excessive parallelization may have important additional costs due to the overheads associated to the creation, scheduling and synchronization of the tasks.

4 A new D&C algorithm template

Given the nature of the problems of the `parallel_recursion` skeleton described in Section 3.2, our first approach to solve them was to try to minimize the changes required following an incremental strategy. Namely, we designed new partitioners that allowed spawning new parallel tasks from tasks that such partitioners had decided to run sequentially at some point, something that the original skeleton did not support. Unfortunately, the results obtained were unsatisfactory, which led us to consider a complete redesign and reimplementation in a form of a new algorithm template which we call `parallel_stack_recursion`. We now discuss in detail the strategy followed by this parallel skeleton together with its interface and a very useful auto-tuning feature for its most critical parameter.

4.1 Implementation strategy

We observed that the cost of the creation and management of parallel tasks and the decision on when to build them in order to balance the workload could imply large overheads, particularly in algorithms in which the core computation was relatively lightweight. As a result we decided to build our new algorithm template so that it would have a single parallel task per thread, and to base the load balancing on the ability of such tasks to steal pending work from other tasks, which should be cheaper. This largely simplified the structure of the parallel execution, in particular avoiding the requirement of `parallel_recursion` to perform an efficient scheduling of parallel tasks, which this skeleton obtained by relying on the excellent scheduler of the Intel TBB framework. As a result, the parallelization of `parallel_stack_recursion` was just based on the C++11 facilities for multithreading, thus eliminating the dependency on Intel TBB. Since the skeleton uses a single task per thread, both words will be used interchangeably in what follows.

In order to enable the load balancing among the threads, the library could simply rely on a shared queue where all the threads could place and retrieve problems to process such as the one proposed in [18]. However that strategy implies the need for synchronizations on the queue every time a thread requests a new problem to process or tries to insert new pending problems. For this reason we designed a data structure in which each thread has its own container of pending problems, where it places the new problems it generates and from which it obtains the problems it processes. The load balancing is achieved in this structure by stealing pending work from the containers of other threads when the current thread cannot find work in its own container. Such steals of course must be performed with proper synchronization on the container of the victim thread.

The use of the proposed containers to keep the problems also solves the second issue of `parallel_recursion` related to the limitation of the stack memory, as the data of our containers are stored in the heap and there are no longer recursive calls within the skeleton. Rather, each thread works in a simple cycle in which, once a new problem is obtained, it is processed in a single step if it is a base case, while non base problems can be also subject to an optional processing, after which they are decomposed in children problems that are stored in the container to be considered later. In either case, the problem is then deallocated and the skeleton proceeds to obtain a new problem to process.

As for the problem containers, given the recursive nature of D&C algorithms, and in order to enhance locality, using stacks seemed the most natural and performant option. As a result of this selection, since each thread will be always pushing and popping problems from the top of its stack, it was clear that work stolen by another thread should be taken from the opposite part of the stack, namely its bottom. Despite this adequate design decision, if work stealing could happen at any moment in the private stack of a thread, this thread would always have to use synchronization mechanisms whenever it accessed its stack in order to make sure it suffered no conflicts with work steals. This would clearly strongly degrade the performance with respect to unsynchronized accesses. In order to avoid this problem, the work stack of each thread is divided in two dynamic sections:

– At the top we find the *local section*, which is exclusively reserved for the owner thread. Since it is only accessible by its owner, work cannot be stolen form this portion of the stack and the owner thread can therefore push and pop problems in an unsynchronized fashion there. Each thread is expected to work the vast majority of the time on this portion of the stack.
– Just below there is the *shared section*, from which other threads can steal work when they run out of it, and in particular, from the bottom of this section. As a result, and as its name implies, accesses to this region must always be synchronized.

While steals could always take place with a granularity of a single problem, there are two reasons why in general it is more beneficial to steal chunks of several problems. The first one is that the computing cost associated to a single problem may be too small, and thus stealing only one problem will often be insufficient to keep reasonably busy doing useful work the thread that initiated the process. The second one is that each steal has non negligible costs, as it involves not only examining the stacks of several threads until finding one with stealable problems in the shared section, but also locking the victim stack and transferring the stolen problems from it to the destination stack. It is therefore desirable to amortize this cost among several stolen problems. For these reasons our skeleton supports a parameter called *chunkSize* that controls the number of problems stolen in a steal process. This way, once the user defines this variable, a thread will only attempt to steal work from another thread when the shared section of its stack has at least *chunkSize* elements, and that will be the amount of problems stolen.

Our library also uses the chunk size to decide when to migrate work between the local and the shared sections of a stack. This way, when a local section has a size s of at least two chunk sizes, $(\lfloor s/chunkSize \rfloor - 1) \times chunkSize$ elements of the bottom of the local section are moved to the shared section. Conversely, if a local section becomes empty, the associated thread checks whether the shared section has elements. If this is the case, the *chunkSize* problems at the top of the shared section (notice that the size of the shared section is always a multiple of *chunkSize*), and thus just next to the just emptied local section, are moved to this latter section. If on the contrary the shared section has no data, the thread will try to steal work, namely *chunkSize* problems, from another stack. The data movements between –always consecutive– sections of the same stack are very cheap because, beyond the required synchronization, as they always affect the shared section, they do not imply any actual data movement, but rather a modification of pointers that indicate the limits of the sections on the stack.

A final issue to consider is what to do when a thread cannot find work to steal from any other thread. In this case, it spinlocks, waiting for a signal that is activated each time that any thread releases a chunk to its shared section. At that point, the thread retries the steal process. If at any time it is detected that all threads are in the spinlock waiting for work, this means that all the problems have already been processed and the execution of the algorithm finishes.

4.2 Interface

One of the aims in the design of *parallel_stack_recursion* was to minimize the changes in its interface so that it were as similar as possible to that of *parallel_recursion*. Indeed the new algorithm template can use exactly the same info objects and with the same semantics as the original *parallel_recursion* algorithm. There are some changes however in the body and partitioner objects supported, which we now explain in turn.

4.2.1 Body object

Given the implementation described in Section 4.1, a problem is destroyed as soon as its children are generated, which makes it impossible to use the `post` member function described in Section 3.1. It would have been possible to use still that interface, at the cost of more memory and CPU consumption, by storing somewhere subdivided problems until all their subproblems are processed, and by adding in the internal data structures of the library components to associate each problem with its subproblems and their solutions. However, when we analyzed the highly unbalanced problems for which the new skeleton was useful, we found that the reduction operations of their D&C algorithms were not only associative and commutative, but also that we could not find situations in which it were necessary to process together a problem with the solution of its subproblems. Therefore, although it would have been perfectly possible to use exactly the same body objects as *parallel_recursion*, for efficiency reasons we propose a new `post` function that covers all the problems we found and is in fact easier to write than the original one. Its signature is

```
void post(const S& local_result, S& global_result)
```

where `S` is the type of the results, `local_result` is a partial result such as the one obtained by processing a base problem by means of the `base` member function, and `global_result` is the global result with which the local result must be reduced. A restriction with this design is that while with the original `post` function the non base problems could contribute to the computation of the global result, this is impossible now. The reason is that since partial results are only obtained from the `base` functions applied to base problems and their reductions performed by `post` invocations, the intermediate nodes of the tree have no mechanism to contribute to the global result beyond the results of its children. While this is enough in many problems, whose results are obtained by combining only the results obtained at the leaves of the D&C recursion tree, in some algorithms the internal nodes may also have a contribution to the result. For this reason, the body objects of our skeleton support a

```
struct NewTreeAddBody: public EmptyBody<tree_t *,int, true> {
  int base(tree_t * t) { return t->val; }

  void post(int local, int& global) { global += local; }
};

int r = parallel_stack_recursion<int>(root, TreeAddInfo(), NewTreeAddBody());
```

Listing 3: Reduction on a tree using `parallel_stack_recursion`

`S non_base(const T& t)`
member function that computes a partial result from a non base node `t`. The `EmptyBody` template introduced in Section 3.1 provides a default implementation of this function that just invokes the `base` member function. Finally, since only some problems benefit from the `non_base` function, the `EmptyBody` template now supports a third optional argument which is a boolean that indicates whether this function should be used, when true, or not, when false.

Given the explanations above, the problem expressed in Listing 2 using `parallel_recursion` can be rewritten using `parallel_stack_recursion` as shown in Listing 3. The listing does not include the `TreeAddInfo` class because it is identical. As for the body class, since the internal nodes of this D&C recursion tree also contribute to the result, it derives from an instantiation of the `EmptyBody` class template whose third argument is true. The member function `non_base` is not implemented though, as its default implementation relies on the `base` member function, which suffices in this case, as both base and non base nodes contribute their `val` value to the global result. Altogether we can see that the interface is very similar and somewhat simpler than that of `parallel_recursion`.

4.2.2 Partitioner

The second change reflected in the interface pertains to the partitioner. In `parallel_recursion` this object decides when to switch from parallel computation, by generating and synchronizing new parallel tasks, to sequential computation, by entering a sequential recursive computation. In `parallel_stack_recursion` however, there is always a single task per thread that iteratively works on its container stack, sometimes stealing work from other stacks. Therefore the role of the partitioner was redefined. Namely, in this skeleton it chooses whether a given problem taken from the stack must be processed using the aforementioned strategy based on dynamic stacks described up to this point or, on the contrary, it must be solved by means of a recursive sequential computation unrelated to the stack container analogous to those offered by `parallel_recursion`. The second alternative means that all the computations are directly performed in the thread that took the problem from its stack, making impossible the stealing of portions of this D&C recursion tree by other threads. It has the advantage however that the computation can be faster because every interaction with the container stack is avoided and replaced with a direct optimized recursive execution that relies on the stack memory of the thread. This can be particularly advantageous for problems whose computations are very simple.

In `parallel_stack_recursion`, the *simple* partitioner gives place to the default behavior that relies on the container stacks for all the processing, while the *custom* partitioner allows to programmatically choose between the default behavior and the optimized sequential resolution for every problem taken from the stack. This partitioner must be used with caution, since it can cause the same problems of unbalance and excessive stack memory usage that the new skeleton intends to avoid.

Regarding the *automatic* partitioner, we must remember that the effort to develop this new skeleton derives from the impossibility to find adequate work decompositions in terms of overhead incurred and load balancing in the original skeleton for very irregular problems. As a result, it seemed of little use to support any automatic partitioner in the new skeleton, since there are no simple heuristics that allow to obtain good performance in the irregular unbalanced problems it is oriented to. This is why we can notice that Listing 3 does not use the automatic partitioner used in Listing 2.

4.3 Chunk size auto-tuning

As we have seen, our skeleton only introduces one quantitative parameter, called *chunkSize*, which controls the granularity of the steals among threads as well as the movements between the sections of a stack. This is one of the most important parameters that influences performance. Usually, a program has a set of consecutive chunk sizes that provide good performance and, as one moves away from these values, the performance begins to decrease, sometimes very quickly. The reason is that if the chunk size is too small, the threads consume too much time performing continuous steals, while if it is too large, there are fewer steals and some threads remain idle for too long. Unfortunately, there is no a universal value for this parameter that guarantees good performance for all cases, as the best value depends on many factors such as the type of D&C problem, its implementation, and even on the processor architecture where the execution is performed.

Figure 1 illustrates the comments we have just made by representing the performance of the benchmarks used in our evaluation in Section 5 when they are parallelized using different chunk sizes for `parallel_stack_recursion`. The performance is measured as the speedup achieved with respect to an optimized sequential execution when running each problem using 24 threads, i.e. one per core, in the system used in our experiments, also described in Section 5, and a simple partitioner. We can see that the performance is basically

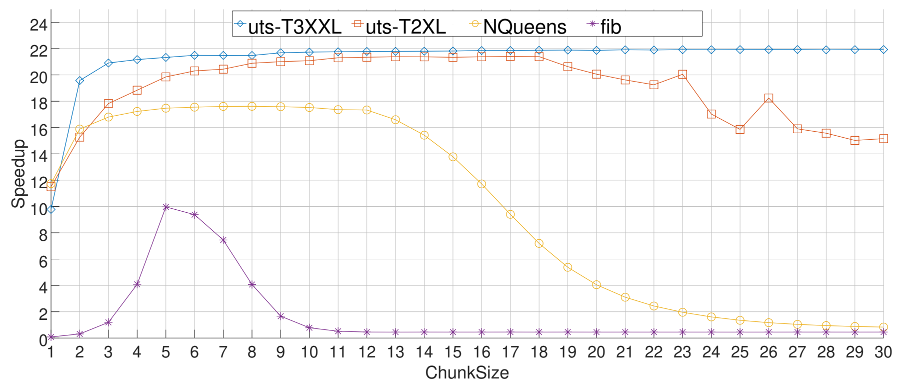


Fig. 1: Performance of several benchmarks in a 24 core system using the `parallel_stack_recursion` library with the *simple* partitioner as a function of the *chunkSize*.

Table 1: Benchmarks used. n stands for nodes and h for heights.

Name	Problem Size	Seq. Time
uts-T3XXL	Binomial tree, 2793M n, h 99049	519.867 s
uts-T2XL	Geometric [cyclic branch factor] tree, 1495M n, h 104	457.092 s
N Queens	16 x 16 board	178.696 s
fib	54st Fibonacci number	332.491 s

a concave downward curve (inverted U shape) with respect to the chunk size because of the aforementioned problems when the chunks too large or too small. Interestingly, while benchmarks such as *fib* present a very limited number of chunk sizes that provide good performance, others reach near optimal performance for a large range of values.

Given the importance of this parameter and the difficulty to predict a priori a good value for it, users should perform tests in order to choose a reasonable value for their executions. While doing this manually is not particularly difficult, it is tedious and it represents additional work that can be automated. For this reason, another contribution of our new library is an auto-tuning framework that automatically searches for the best chunk size for a given problem and environment. The framework allows to control the search process, for example, the amount of time of the search or the size of the number of tests.

5 Evaluation

Our evaluation relies on the benchmarks described in Table 1, which includes their sequential runtime in the system used in the experiments. The *uts* (Unbalanced Tree Search [21]) benchmark processes unbalanced trees of different shapes and sizes. The two main types of trees it suports are binomial and geometric. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be earned by choosing to move one node over another for load balance: the expected work at all nodes is identical. In geometric trees however the expected size of the subtree rooted at a node increases with proximity to the root. The *N Queens* benchmark solves the N Queens puzzle problem, which computes on how may ways can n chess queens be placed on an $n \times n$ chessboard so that no two queens threaten each other. Finally, *fib* implements the recursive algorithm to compute the *n-th* Fibonacci number. Although this is an inefficient method, it is often used in the literature of D&C and unbalanced algorithms. The enormous simplicity of its computations is particularly interesting when evaluating a parallel skeleton like ours, since it is in this kind of algorithm where the overheads of the library can be more clearly observed.

We will first analyze the performance of the skeleton, which will be followed by a study on the programmability advantages it offers. In all cases, it will be compared to the sequential version, a version developed using OpenMP, and another one based on the `parallel_recursion` algorithm template. While other approaches could not be compared for space reasons, it must be noted that `parallel_recursion` was successfully compared to other implementations in the single node experiments in [11], thus providing an approximate indirect comparison to our new proposal.

5.1 Performance evaluation

All the measurements were taken in a server with 128 GB of memory and two 2.5 GHz Intel Xeon E5-2680v3 with 12 cores each, totaling 24 cores. The codes were compiled with g++ 6.4.0 and the optimization level O3. As we will see, we measured the performance when using 6, 12 and 24 cores, always using a single thread per core. We measured separately the performance obtained using the different kinds of partitioners supported by each skeleton, tuning the user-provided function used by the custom partitioner separately for the two skeletons. The `parallel_stack_recursion` chunk size used was obtained by means of a separate auto-tuning configured to use just 10% of the runtime of the sequential version. This time is not included in the performance measurements. Further tests proved that in our experiments the performance of the chunk size obtained following this strategy was always almost identical to that of the optimal chunk size.

In all the figures, the benchmarks using `parallel_stack_recursion` will be labeled as *spar*, those using `parallel_recursion` will be known as *par*, and those that are implemented with *OpenMP* will be labeled as *omp*.

The *uts* benchmark allows generating unbalanced trees that follow different distributions and have different sizes and shapes depending on the arguments to the binary. The *T3XXL* tree is predefined in the *uts* distribution package, while *T2XL* has been added as example of a geometric tree with a circular

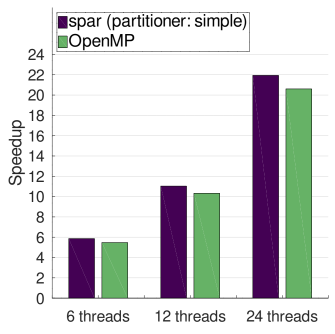


Fig. 2: Performance results of *uts-T3XXL* benchmarks

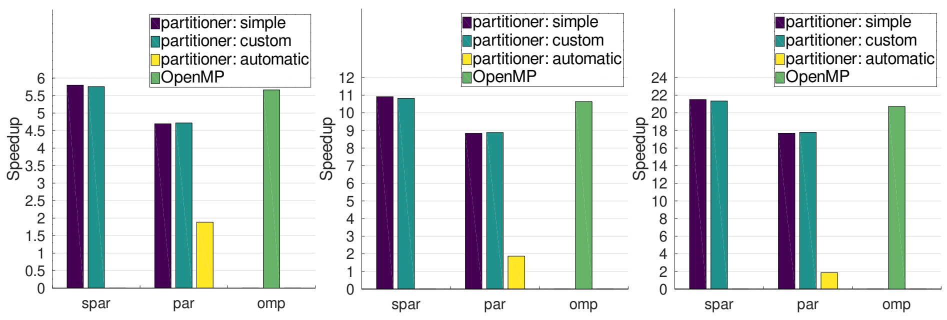
factor branch, its *uts* parameters being `-t 1 -a 2 -d 26 -b 7 -r 220`. The performance obtained on these trees is now discussed in turn.

We consider that *T3XXL* is the most challenging benchmark tried, since this binomial tree has a very large depth and extreme imbalance. Figure 2 shows the speedup achieved by our skeleton and the standard OpenMP implementation of the benchmark for 6, 12 and 24 threads, taking as baseline the standard sequential implementation. All the executions with `parallel_recursion` failed with a stack overflow error no matter the partitioner used, further confirming the interest of our new proposal. The figure only shows the results of `parallel_stack_recursion` with a *simple* partitioner because we were unable to find a custom partitioner that performed better. As we can see, our skeleton, despite strongly reducing the complexity of the code with respect to the OpenMP implementation as shown in Section 5.2, systematically offers between 6.5% and 7.1% better performance than the manually optimized OpenMP implementation.

It deserves to be mentioned that in experiments using smaller binomial trees so that `parallel_recursion` would not break, it consistently offered clearly worse performance than `parallel_stack_recursion` and OpenMP.

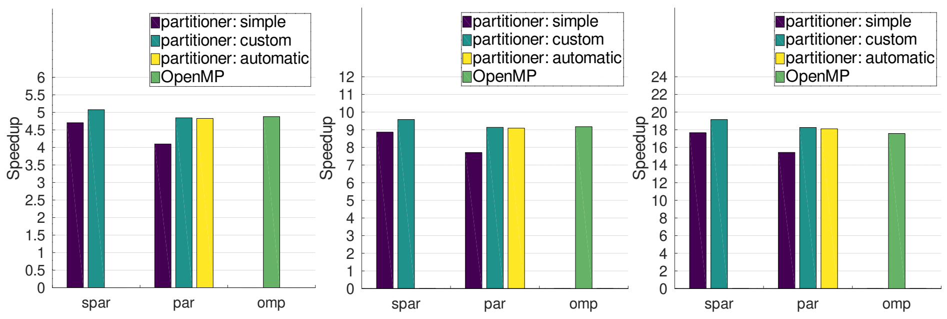
T2XL is a geometric tree with a cyclic branch factor, which makes somewhat difficult to balance its processing. As we can see in Figure 3, the use of the *simple* partitioner in `parallel_stack_recursion` is enough to obtain the best performance, there being no advantage in the use of a *custom* partitioner. The speedups obtained by `parallel_recursion` are always lower, and it is particularly interesting that the use of the *automatic* partitioner provides very bad results for this benchmark. As for OpenMP, despite the high programming cost of developing by hand this optimized code, it performs about 3% slower than our skeleton.

Figure 4 shows the results of *N Queens*. The larger complexity of the computations of this benchmark allows the *simple* partitioner to obtain quite good results, although usually worse than those of OpenMP. The exception happens at 24 threads, where fact that our new proposal always performs slightly



(a) 6 threads (b) 12 threads (c) 24 threads

Fig. 3: Performance results of *uts-T2XL* benchmarks

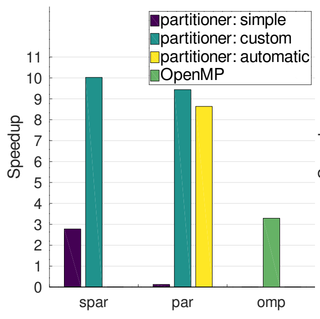


(a) 6 threads (b) 12 threads (c) 24 threads

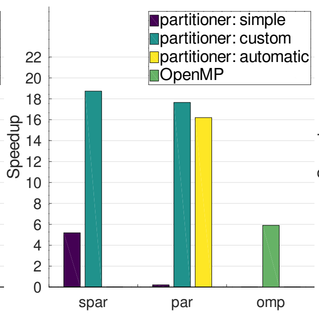
Fig. 4: Performance results of *N Queens* benchmarks

better than `parallel_recursion` allows it to reach the same performance as OpenMP. As expected, both skeletons improve their performance when a tuned *custom* partitioner is used, our new proposal systematically offering better performance than the other implementations. This way, it is consistently $\sim 4.8\%$ faster than `parallel_recursion` and $\sim 4.4\%$ faster than OpenMP, except for 24 threads, where its advantage grows to 9%.

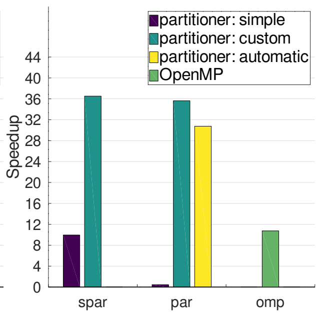
Figure 5 shows the performance of all the parallel implementations of *fib* developed. As mentioned before, this is a particularly challenging benchmark given the extremely lightweight nature of all the individual functions that conform it as a D&C algorithm. This is clearly reflected in the poor performance of both skeletons when a *simple* partitioner is used, as the consideration of every single Fibonacci number computation as a separate task to be managed leads to much overhead. Our new skeleton is considerably more efficient than `parallel_recursion` in this situation, being in fact 22 times faster when 24 threads are used, and reaching a performance similar to that



(a) 6 threads



(b) 12 threads



(c) 24 threads

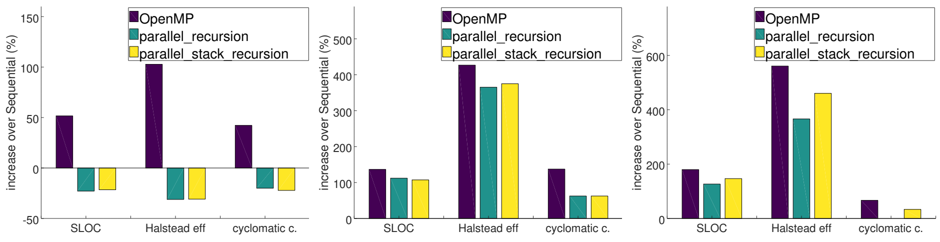
Fig. 5: Performance results of *fib* benchmarks

of OpenMP. When smarter partitioners that perform as sequential computations the calculations of Fibonacci numbers under some threshold are used, the performance of both skeletons grows considerably. In this scenario our new proposal also consistently outperforms `parallel_recursion` across all the parallel executions, although only for a small margin that decreases as the number of threads grow. Namely, the speedup of the new skeleton with respect to `parallel_recursion` goes from 6.2% for 6 threads to 2.5% for 24.

As for the absolute performance, both skeletons achieve superlinear speedups, which are in addition much higher than that of the OpenMP version, even when it also applies the strategy of computing as sequential tasks the Fibonacci numbers below a threshold for which we searched for the optimal value. Both behaviors are related to the fact, already observed and discussed in [11], that the object code that the compiler generates from our skeleton is much more efficient than the one it generates from the typical recursive implementation used by the sequential and OpenMP versions.

5.2 Programmability comparison

The best approach to measure and compare the programmability of different options is probably to rely on the observations and results from a group of programmers with a similar degree of expertise when trying to apply them. This is seldom possible, thus, our study relies on three approximate metrics of this kind. The first one is the number of source lines of code (SLOC) excluding comments and blank lines. Its value strongly depends on the programming style used and lines can widely vary in terms of complexity. A more precise metric is the Halstead programming effort [13], which estimates the complexity of the program through a formula that takes into account the number of unique operands, unique operators, total operands and total operators found in the code. The last metric computed is the cyclomatic complexity [20], defined as $V = P + 1$, where P is the number of predicates or decision points in a program.



(a) *uts* benchmark (b) *N Queens* benchmark (c) *fib* benchmark

Fig. 6: Growth of the programmability metrics of the parallel implementations with respect to the sequential one.

Figure 6, shows the relative growth of all the metrics in the parallel versions with respect to the sequential counterpart. The measurements were performed in all the cases on the whole application. As expected, given their similar API and semantics, the metrics are very similar for the two skeletons considered, which is interesting given the more complex behavior and better performance of our new proposal. The differences come basically from the changes in the `post` and `non_base` functions, although the latter one is only required in *uts*.

It is interesting that, despite relying on compiler directives, whose API is usually terser than that of libraries, the OpenMP version yields consistently worse programmability metrics than the skeletons. This way, depending on the metric used, the OpenMP version requires between 83% and 192% for more effort than `parallel_stack_recursion` for *uts*, between 28% and 72% for *N Queens* and between 14% and 25% for *fib*. An important reason for this in the case of the last two benchmarks is the need to write two versions of the algorithm in order to obtain the best performance, something which was already observed in [11]. The first version is the one invoked by the user and it contains the OpenMP directives as well as tests to decide whether the solution of a problem should rely on task parallelism or be performed as a sequential task. The second implementation is purely sequential and it takes care of these latter tasks avoiding any overhead associated to the parallelization. In the case of *uts* the sequential and OpenMP versions are much more complex than the ones based on skeletons because they have to explicitly create and manage data structures to perform the processing of the trees than in the skeleton implementations are implicitly provided by the library runtime. It is also interesting that sometimes, despite the better performance observed in Section 5.1, our skeleton offers slightly better programmability metrics than `parallel_recursion`. This is mostly associated to the simpler `post` method of `parallel_stack_recursion`, which by being restricted to a single element, avoids loops and computations on numbers of children that are required in the analogous method of `parallel_recursion`.

6 Conclusions

Divide-and-conquer is a very important pattern of parallelism that appears in many problems and can be used to implement other very relevant patterns. This makes very relevant and useful the development of tools that allow the easy and optimized implementation of this pattern, one of the best solutions being algorithmic skeletons. In this paper we have introduced `parallel_stack_recursion`, a C++ algorithmic skeleton that implements this pattern in shared memory with a focus on problems with large levels of recursion and/or high degree of unbalance. While our proposal is a complete redesign of `parallel_recursion`, a highly optimized skeleton for the same pattern, it manages to keep an almost identical interface.

Our evaluation shows that indeed the new skeleton can be applied in situations in which `parallel_recursion` breaks due to stack memory limitations, which justifies by itself its development. Furthermore, the new skeleton is on average 10.4% faster than `parallel_recursion` in the benchmarks that the latter one supports, and 4.9% faster than optimized OpenMP implementations if we disregard the *fib* benchmark, in which the compiler gives an unfair advantage to our skeletons. The maximum speedups however can go up to 22% when compared to `parallel_recursion` and 9% when compared to the OpenMP baseline, again discarding *fib*. Despite these performance advantages, the evaluation shows that the development effort associated to our new proposal is consistently similar to that of `parallel_recursion` and noticeably better than that of OpenMP. This latter observation is particularly true in the case of our largest benchmark, *uts*, in which versions not based on a skeleton have to manually define and manage data structures in order to support the highly irregular processing and the load balancing it needs to attain good performance.

As future work we plan to develop a version of this skeleton optimized for systems such as current multi-core clusters, whose optimal exploitation involves the usage of distributed and shared memory programming paradigms.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (TIN2016-75845-P, TIN2016-76373-P and the predoctoral Grant of Millán Álvarez Ref. BES-2017-081320), and by the Government of Galicia (Xunta de Galicia, Spain) co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04 and ED431C 2018/19) as well as under Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019–2022, refs. ED431G2019/01 and ED431G2019/04). We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280. John Wiley & Sons, Ltd (2017)
3. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Gener. Comput. Syst. **19**(5), 611–626 (2003)
4. Board, O.A.R.: OpenMP application program interface version 5.0 (2018)
5. Ciechanowicz, P., Kuchen, H.: Enhancing Muesli's data parallel skeletons for multi-core computer architectures. In: 12th IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2010), pp. 108–113. IEEE (2010)
6. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1989)
7. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389–406 (2004)
8. Danelutto, M., De Matteis, T., Mencagli, G., Torquati, M.: A divide-and-conquer parallel pattern implementation for multicores. In: Proc. 3rd Intl. Workshop on Software Engineering for Parallel Systems, SEPS 2016, pp. 10–19. ACM (2016)
9. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Computing **32**(7-8), 604–615 (2006)
10. González, C.H., Fraguela, B.B.: A generic algorithm template for divide-and-conquer in multicore systems. In: Proc. 12th IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2010), pp. 79–88. IEEE (2010)
11. González, C.H., Fraguela, B.B.: A general and efficient divide-and-conquer algorithm framework for multi-core clusters. Cluster Computing **20**(3), 2605–2626 (2017)
12. Gorlatch, S., Cole, M.: Parallel skeletons. In: Encyclopedia of Parallel Computing, pp. 1417–1422. Springer (2011)
13. Halstead, M.H.: Elements of Software Science. Elsevier (1977)
14. Hosseini Rad, M., Patooghy, A., Fazeli, M.: An efficient programming skeleton for clusters of multi-core processors. Int. J. Parallel Program. p. 1094–1109 (2018)
15. Karasawa, Y., Iwasaki, H.: A parallel skeleton library for multi-core clusters. In: Proc. 2009 Intl. Conf. on Parallel Processing (ICPP'09), pp. 84–91. IEEE (2009)
16. von Koch, T.J.K.E., Manilov, S., Vasiladiotis, C., Cole, M., Franke, B.: Towards a compiler analysis for parallel algorithmic skeletons. In: Proc. 27th Intl. Conf. on Compiler Construction, CC 2018, p. 174–184 (2018)
17. Kozsik, T., Tóth, M., Bozó, I.d.: Free the conqueror! refactoring divide-and-conquer functions. Future Gener. Comput. Syst. **79**(P2), 687–699 (2018)
18. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: Proc. 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing (PDP 2010), pp. 289–296. IEEE (2010)
19. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2004)
20. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **2**, 308–320 (1976)
21. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In: Languages and Compilers for Parallel Computing (LCPC 2006), pp. 235–250. Springer Berlin Heidelberg (2006)
22. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

High-Level Parallel Ant Colony Optimization with Algorithmic Skeletons

Breno A. de Melo Menezes · Nina Herrmann · Herbert Kuchen · Fernando Buarque de Lima Neto

Abstract Parallel implementations of swarm intelligence algorithms such as the Ant Colony Optimization (ACO) have been widely used to shorten the execution time when solving complex optimization problems. When aiming for a GPU environment, developing efficient parallel versions of such algorithms using CUDA can be a difficult and error-prone task even for experienced programmers. To overcome this issue, the parallel programming model of *Algorithmic Skeletons* simplifies parallel programs by abstracting from low-level features. This is realized by defining common programming patterns (e.g. map, fold and zip) that later on will be converted to efficient parallel code. This exempts the programmer from low-level programming aspects. In this paper, we investigate how algorithmic skeletons in the form of a programming library (namely *Musket*) can cope with the development of a parallel implementation of ACO and how that compares to low-level implementations. Our experimental results show that *Musket* suits the development of ACO. Besides making it easier for the programmer to deal with the parallelization aspects, *Musket* generates high performance code with similar execution times when compared to low-level implementations.

Keywords Algorithmic Skeletons · Ant Colony Optimization · High Performance Computing

B.A. de Melo Menezes, N. Herrmann, H. Kuchen
University of Muesnter. Leonardo-Campus 3. 48149 Muenster - Germany
Tel.: +49 251 83-38267
E-mail: {breno.menezes,nina.herrmann,kuchen}@uni-muenster.de

F. Buarque de Lima Neto
University of Pernambuco. Rua Benfica, 455. Recife, 50720-001 - Brazil
Tel.: +55 (81) 3184-7548
E-mail: fbln@ecomp.poli.br

1 Introduction

Nature inspired metaheuristics have been widely used to solve complex optimization problems [1]. When tackling combinatorial problems, Ant Colony Optimization (ACO) is one of the best known algorithms [2]. Initially proposed by Dorigo, it is inspired by the social behavior of ant colonies when searching for new sources of food and their ability of finding the shortest path between the colony and the food [3]. ACO was initially created to solve problems such as the Traveling Salesman Problem (TSP), achieving satisfactory results.

ACO has a higher computational cost when compared to other metaheuristics. Considering the TSP problem, the number of nodes to be visited in the tour increases and the number of possible tours grows exponentially as the number of nodes in the graph gets bigger. In order to explore more of these possibilities in the search space, more ants are needed in the colony and, therefore, the computational costs increase substantially. Knowing that a considerable part of the computations are performed during path construction and that the runtime might be negatively affected when tackling a bigger instance of TSP with several ants, the application of improvements becomes mandatory in order to run the algorithm in a reasonable amount of time without losing the quality of the solutions.

Parallel implementations of ACO have been introduced aiming for different high-performance hardware, such as multi-core CPUs and GPUs. Low-level frameworks such as OpenMP, MPI and CUDA provide many tools for programmers, assisting the development of such parallel versions of the algorithm. Nevertheless, lots of skills and experience are necessary in order to come up with highly optimized code, especially when combining these frameworks. The tools provided by CUDA help programmers to develop parallel programs for GPUs, even though some expertise is necessary to generate high performance code. Programmers must be aware of data transfers, synchronization points and many other issues that turn this task difficult and error prone.

Aiming to ease the development of such parallel algorithms, high-level parallelization tools provide means to profit from the use of high-performance hardware without the issues inherent to low-level programming. For example, some tools allow the use of fixed structures which are often used in programming, known as algorithmic skeletons. Skeletons represent common operations, such as *map, zip* and *reduce*. Those can be used in a program and will be converted to parallel code. This way, the programmer's job is to translate the methods from the original algorithm into predefined operations available in the high-level tool which is responsible for the parallelization.

Musket (Muenster Skeleton Tool for High-Performance Code Generation) is a approach based on a Domain Specifcic Language (DSL) created to speed-up the development of parallel programs [4]. By using it, programmers are able to create code by first writing it in the DSL *Musket* and then converting the program into parallel CPU or GPU code. Created as a general purpose tool, *Musket* has already been applied and tested in several problems including metaheuristics, presenting promising performance results compared to other parallelization approaches [5].

In this paper, we investigate the use of *Musket* to create a parallel GPU version of ACO in order to understand and compare how it relates to a low-level implementation in terms of performance and development complexity. The identification of positive and negative sides of using the general purpose structures available in *Musket* may also serve as a base for the future development of the framework.

Our paper is organized as follows: The basics about ACO are displayed in Section 2. In Section 3 we give an overview of related work. The description of *Musket* and how it was applied in this work can be found in Section 4. Experiments are detailed in Section 6 together with the results. In Section 7 we put together the conclusions of this work and point out the future work.

2 Ant Colony Optimization

ACO is a metaheuristic in which artificial ants cooperate among each other in order to solve complex discrete optimization problems. In the case of TSP, the objective is to find the shortest tour in a graph, starting from a random node, visiting each node once and only once and coming back to the original node [6, 7]. In order to solve such a problem, each ant in the colony tries to create a tour and at the end of each iteration they share their success through pheromone deposits. More successful ants, the ones that generated shorter tours, deposit more pheromone on the visited edges. Pheromone is what will attract other ants in the following iteration, helping them to generate similar tours based on the success of ants from previous iterations.

The process described above can be divided into two steps which compose the execution of ACO, namely tour construction and pheromone deposit. During the tour construction, each ant must create a tour starting at a random node. The decision where to go next from the current node i is done in a probabilistic manner, taking into account the distance and the amount of pheromone between the actual node and a candidate node. The probability is calculated as shown in Equation 1.

$$p_{i,j} = \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \epsilon N}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}} \quad \forall j \epsilon N \tag{1}$$

where α and β are the parameters used to determine the influence of pheromone quantity and distance between the nodes over the probability, respectively. $\tau_{i,j}$ is the pheromone at the edge from node i to j, $\eta_{i,j}$ is $1/d_{i,j}$, where $d_{i,j}$ is the distance from node i to j. N is the set of unvisited nodes that can follow node i. $p_{i,j}$ is the probability that the ant goes from node i to j. The actual node i and visited nodes have the probability equal to zero.

Once each ant has created its tour, the fitness of each ant will be equal to the total distance traveled. Afterwards, the pheromone update shall take place.

The first step is the pheromone evaporation where each edge loses a certain quantity of pheromone according to the following assignment (Equation 2).

$$\tau_{i,j} := (1 - \rho) * \tau_{i,j} \tag{2}$$

where, $\rho \in [0..1]$ defines the evaporation rate and $\tau_{i,j}$ is the amount of pheromone between nodes i and j. ρ is used to control the amount of pheromone, enabling the algorithm to focus more on new trails. After the evaporation, it is time for each ant to deposit pheromone on the tour it has created, according to the following assignment (Equation 3):

$$\tau_{i,j} := \Delta t + \tau_{i,j} \tag{3}$$

where $\Delta t = 1/q_k$, and q_k is the length of the round tour of ant k. By doing so, ants that generated shorter tours deposit more pheromone at visited edges than the ones that generate longer tours.

This process is repeated through as many iterations as needed. Although the pheromones are used to attract ants and help them create tours similar to a previous successfully ones, the probabilistic way of choosing the next step allows ants to create distinct paths and therefore generate diversity.

2.1 GPU-ACO

Aiming a GPU environment, we present here one possible implementation of ACO in parallel using CUDA. The steps that compose the ACO algorithm as described above are quite simple and the general process makes the algorithm quite suitable for parallelization. Even so, extra care is required when dealing with the same steps in a parallel way.

The CUDA framework makes it possible to run sequential instructions on the CPU, while the computational intensive tasks can run on the GPU in parallel. In our approach, the whole algorithm runs on the GPU and, therefore, operations such as routing and pheromone updates are declared as CUDA kernels. One positive point about this approach is that no data transfer between host and device is necessary along the iterations. These data transfers are performed at the beginning (reading data and copying to GPU) and at the end of the execution (retrieving results from GPU to host). A general view of the proposed implementation is represented in Algorithm 1.

The first step of this implementation is the initialization of the structures that compose the problem. It includes reading the data from a file that contains the coordinates of each city present in the graph of the TSP instance and the initialization of other variables. Afterwards, the data can be copied to the GPU and, already on the device side, other data structures necessary to run ACO can be created directly on the GPU, i.e. random number generators, distance matrix, pheromone matrix and route matrix. After everything is created and initialized properly, the algorithm can loop through its iterations and perform the tour constructions and pheromone updates.

Algorithm 1 Pseudo-code GPU-ACO of TSP

```
1: $initialize\_ACO$
2: $copy\_data\_to\_Device()$;
3: $initialize\_GPU$
4: $calculate\_distance\_kernel <<< n\_blocks, n\_threads >>> (...)$;
5: $create\_random\_generators\_kernel <<< n\_blocks, n\_threads >>> (...)$;
6: $n\_threads = warp\_size$;
7: $n\_blocks = n\_ants/warp\_size$;
8: while ($iterations < n\_iterations$) do
9:     $tour\_construction\_kernel <<< n\_blocks, n\_threads >>> (...)$;
10:    $pheromone\_evaporation\_kernel <<< n\_cities, n\_cities >>> (...)$;
11:    $pheromone\_deposit\_kernel <<< n\_ants, n\_cities >>> (...)$;
12: end while
13: $copy\_results\_to\_Host()$;
14: $clear\_GPU()$;
```

The tour construction is the first step in an iteration and also the most demanding task of ACO. In order to create a parallel version of it, we consider the straight forward approach where each ant creates its tour independently in a thread. The number of CUDA blocks will be determined by dividing the number of ants in the colony by the desired number of threads per block, meaning that it can be changed and adapted according to the necessity and capabilities of the hardware. The simplicity to implement this approach is one of its positive aspects. Once there is a sequential implementation of ACO, it is quite easy to port it to CUDA using this approach.

After the tour construction phase, the pheromone update is broken down into two different steps. The first one is the pheromone evaporation, in which each connection in the graph loses a certain amount of pheromone as explained before in Equation 2. In our parallel implementation, one CUDA block is generated for each city and, inside this block, one thread is generated for each other city in order to decrease the amount of pheromone (Listing 1).

```
__global__ void evaporation_kernel(double* c_phero) {
    int edge_index = blockIdx.x * blockDim.x + threadIdx.x;
    double RO = 0.5; //Evaporation rate = 50%

    if(blockIdx.x !=  threadIdx.x){  // no edge from city_i to itself
        c_phero[edge_index] = (1 - RO) * c_phero[edge_index];
    }
}
```

Listing 1: Evaporation kernel

The pheromone deposit phase starts which one CUDA block assigned to one ant in the colony. Each thread inside a block is responsible for updating an edge visited by the ant in the tour constructed previously. As the pheromone matrix is stored in the GPU's global memory and it is being updated by different threads, racing conditions might appear. In order to overcome that, the pheromone deposit is performed using CUDA's atomic operations, guaranteeing the integrity of the data without losing performance.

The process is repeated for a number n of iterations. At the end, the best results are copied to the host and the execution is ended. With such a simple approach it is already possible to achieve a considerable speedup when compared to sequential implementations.

3 Related Work

Low-level parallel implementations of ACO have been already investigated in literature. The approaches include mainly the introduction of data parallelism into the code, like in the work of Uchida et al. [8]. Other works focus on improvements in the algorithm that would favor data parallelism. Cecilia et al. developed a new mechanism called I-Roulette in order to enhance parallelism during the path creation process. Furthermore, they introduce strategies for parallelization of the pheromone update process suitable for GPU architectures [9].

Another approach to reduce the execution time of the algorithm is to improve the parallelization itself, instead of modifying the algorithm and fitting it to the hardware. The idea is that, given ACO's characteristics and given how the processes are structured in a GPU, for example, different levels of parallelization can be used in order to make a better use of all processing power of the hardware [10]. Also, in another work, the same authors investigate the use of atomic operations to enhance the process of updating the pheromone matrix [11]. The results indicate that different levels of parallelism can be useful according to the size of the problem and that the use of atomic operations can speedup the pheromone update phase.

Rieger et al. introduced *Musket*, a DSL for parallel programming [4]. Their idea is to offer a language with algorithmic skeletons built in and with a syntax similar to C++ in order to help programmers to write high performance distributed parallel programs without the need of expertise in low-level frameworks. High performance low-level code for different architectures (Multi-core CPUs, GPUs or clusters) is generated from Musket files. The authors point out the benefits of using a DSL compared to other high-level approaches. Also, among some examples, the Fish School Search (FSS) metaheuristic is used as a benchmark. Further analysis of FSS and *Musket* are done by Wrede et al. [5]. Both studies show the possibility of using such general purpose tools for the application in the metaheuristics field. The major criticism of using high-level frameworks has been the possible loss in performance. This papers serves to evaluate the performance of a skeleton based implementation and a hand written implementation previously discussed [10] [11].

4 Musket

Musket (Muenster Skeleton Tool for High-Performance Code Generation) is a framework based on a DSL which enables programmers to develop parallel applications and generate optimized code without requiring knowledge about low-level programming languages and frameworks. For interested readers the code can be found in a public repository [12].

The syntax of *Musket* is based on C++ which is widely used for high performance computing. It was defined using the Xtext framework and it includes a parser and an editor that can be incorporated to Eclipse [13]. In this way, programmers can use helpful resources such as syntax highlighting, code completion and validation. Creating parallel programs in *Musket* is simplified in multiple ways. Common parallel programming structures are simplified by representing them as skeletons. Moreover, the division and allocation of data structures to distinct processes is done by the code generator which transforms Musket code to C++ with CUDA operations in case GPU code is generated. Furthermore, it is totally abstracted from specifying the number of threads to be started. Those and other advantages become more apparent by illustrating an exemplary program (Listing 2).

A *Musket* program is divided into four parts, namely *meta-information*, *data structure declaration*, *user function declaration* and *main program declaration.* The *meta-information* block (lines 1-5) specifies for which type of hardware code should be generated. Firstly, the platform argument distinguished between a program for GPUs or CPUs. In case of stating multiple platforms multiple programs are created. For our application context only the GPU code generator is required. Afterwards, the number of processes, cores, and GPUs which shall be used are specified. Information about the targeted architecture is essential to generate a distribution for data structures which is efficient and to organize the parallel execution of the skeletons.

In *Musket* global data structures are declared before writing functions in the *data structure declaration* block (line 7). On the one hand, for each additional data structure type which is offered, the effort for the implementation rises. On the other hand, it is possible to include additional information for the data structures e.g. on the data distribution (see line 7). Here, the elements of the array are distributed among the GPUs. Available distribution modes are *dist* for distributed, *copy* to make data structures available for all processes, and local which is a specific form of copy where no global copy needs to be created. Similar to the specification of arrays, matrices are created, which require the same parameters despite having two parameters for the size, to specify the number of rows and columns.

The *user function declaration* part (lines 9-11) includes custom user functions which will be invoked by skeletons and then executed among the nodes and cores available. Inside user functions programmers can make use of control structures, such as *if-else statements* and *for loops*, moreover, a selection of C++ library methods and external functions are available. Global structures

```
 1  #config PLATFORM GPU CPU_MPMD
 2  #config PROCESSES 4
 3  #config CORES 24
 4  #config GPUS 4
 5  #config MODE release
 6
 7  array<int,16384,dist> a;
 8
 9  int double_values(int i){
10      return i + i;
11  }
12
13  main{
14      mkt::roi_start();
15      a.mapInPlace(double_values());
16      mkt::roi_end();
17  }
```

Listing 2: Musket Code Sample

declared in the previous section can be used either with a local index or a global index. Moreover, local variables can be created.

In the last part, similar to C programs, a main function is declared which defines the overall structure of the program. In the example, lines 13-17 contain the *main program declaration*. There, general instructions of the program must be listed using control structures, musket functions, and the parallelization instructions in the form of algorithmic skeletons. Musket functions are typically used functions for writing parallel programs which do not need to be executed in parallel, for example measuring the run time and getting maximal and minimal values of data types. The example starts with starting the time measurement. Wrapping such functions relieves the user of the framework to think about target specific functions. In order to simplify parallelization, *Musket* offers (different versions of) the Skeletons *fold*, *map*, *reduce* (which in contrast to fold only accepts a selection of commonly used reduction operators), *zip*, *gather*, *scatter* and *shift partition*. For *zip* and *map*, in place and index variants and their combinations are available. The given example doubles the value of every element of the array. The implementation of the ACO algorithm will show how those structures can also be used for more complex programs.

The written DSL code will then be transformed into low-level code. With each program generated (in case of multiple platforms) CMake Files and execution scripts are generated. The generated code is not meant to be further adjusted.

5 Our Proposal

ACO is a metaheuristic which is suitable for parallelization. Many tasks, such as the path construction, are independent of each other and each ant is able to create its own path without the interference from other ants. Even though, the task of creating a parallel version for it can be quite challenging. A few steps require extra care since reduction steps are executed before performing general calculations. For example, as shown on Equation 1, where the probability is calculated using the product of the amount of pheromone and the distance divided by the sum of all products. Furthermore, steps like the pheromone-update phase include changes which shall be performed by each ant in the colony over the pheromone matrix, which is a structure used by the whole colony. Therefore, the programmer must be careful to avoid race conditions and perform the right data transfers without compromising performance.

In order to overcome such difficulties, *Musket* is helpful. Generally, high-level frameworks have the advantage that the user does not need expertise in the specific area (in this context parallel programming). Musket DSL code is also more concise than e.g. C++ code with calls to a (skeleton) framework.

However, using a DSL also has its disadvantages. The developer of the DSL has to decided which functionalities are essential. Missing necessary functionalities limit the user of the DSL. While, in contrast, including too much functionalities increases the complexity of the code generator and confuses inexperienced users.

In order to evaluate the usability and practicability of a high-level framework for the exemplary case of the ACO algorithm, a *Musket* version will be compared to a hand written version. The aspects of major interest are the performance of the programs and the complexity of the syntax and structure provided by the different approaches. As part of the comparison the creation process of a program implementing the ACO algorithm in *Musket* will be described, to illustrate advantages and disadvantages of using a high-level framework.

5.1 Musket-ACO

In the following, interesting aspects of the high-level implementation of ACO will be be discussed. It is assumed that the general process of the ACO algorithm from the low-level implementation is known [10]. The implementation of the main method is explained and supplemented with selected user functions to illustrate how closely the implementation is to the original algorithm defined in Algorithm 1.

Similar to the low-level program the high-level program first calculates the distance for each city to each other city in the map and saves the results in a data structure (Listing 3, line 3). This means that for example for a map of 4 cities 16 distances are calculated (including the 0 distance for itself). Those calculations are parallelized with a map skeleton, applied to the distance data structure. The generated code starts one thread for each calculation since the calculations for each entry are independent of each other.

In the next step of the algorithm a fixed number of closest cities for each city is calculated. This speeds up the route calculation in the subsequent step

```
1  main{
2      mkt::roi_start();
3      distance.mapIndexInPlace(calculate_distance());
4      city.mapIndex(calculate_iroulette());
5      for (int i = 0; i < iterations; i++){
6          antss.mapIndex(route_kernel2());
7          bestroute = d_routes_distance.reduce(min);
8          antss.mapIndex(update_best_sequence(bestroute));
9          antss.mapIndex(update_delta_phero());
10         city.mapIndex(update_phero());
11     }
12     mkt::print(distance);
13     mkt::roi_end();
14 }
```

Listing 3: Main Method of Musket Program

since a promising subset of close cities can be checked first. Therefore, when searching for a possible next city not the whole data structure storing the distance to all possible cities is checked but a significantly smaller data structure with the closest cities. For e.g. 442 cities this means that instead of selecting possible next cities from a data structure which has 442^2 entries a data structure of the size $x * 442$ is checked. Depending on x which is is the fixed number of selected cities this can significantly decrease the size of the data structure chosen and therefore reduce the time to select promising next candidates. The low level implementation has used 32 cities for this purpose. For the comparison of the programs the same value is used.

The implementation of the calculation is shown in Listing 4. The variable `IROULETE` is the number of closest cities which should be calculated. Each iteration of the for loop in line 2 calculates one entry for the closest city. Initially, no city is selected as the next closest city (line 5). Then it is checked whether the city already belongs to the closest cities (line 9-11), if not it is checked whether the distance is smaller than the distance to the previous city. After all cities have been checked, the city with the smallest distance is written to the `d_iroulette` data structure. Since the calculation requires to be executed for each city, the user function is called with a map.

For calculating the routes, a map skeleton is used (Listing 3 line 6). The algorithm is parallelized for ants, hence for each ant the map operation is conducted. Based on the amount of pheromone deposited and on random starting points, one ants calculates one route. Since the user function of the route kernel is long, it is not displayed in a separate listing. For the exact procedure of calculating one route take a look at [10].

The next step of the algorithm serves to identify connections which are likely to be part of a short route. As ants leave pheromone on paths where they find sources for food, connections which are used in short routes are marked with a high amount of pheromone and connections between cities which are rarely used are marked with a low value of pheromone. This requires to calculate the total distance of each route and update the pheromone of each

```
int calculate_iroulette(int cityindex, int value){
    for(int i = 0 ; i< IROULETE ; i++) {
        double citydistance = 999999.9;
        double c_dist = 0.0;
        int cityy = -1;
        for(int j = 0 ;j<ncities;j++){
            bool check = true;
            for(int k = 0 ; k < i ; k++){
                if(d_iroulette[cityindex * IROULETE + k] == j){
                    check = false;
                }
            }
            if(cityindex != j){
                if (check == true) {
                    c_dist = distance[(cityindex * ncities) + j];
                    if(c_dist < citydistance){
                        citydistance = c_dist;
                        cityy = j;
                    }
                }
            }
        }
        d_iroulette[cityindex * IROULETE + i] = cityy;
    }
    return value;
}
```

Listing 4: User function of Musket Program to calculate the closest cities

city connection based on the quality of routes they are involved in. The step of calculating routes and updating pheromone are repeated as can be seen in Listing 3 lines 5-11. With each iteration the result is likely to improve. However, the effect of increasing the number of iterations on the quality of the result is not part of our work as we discuss the applicability of high level approaches rather than the quality of the algorithm.

The explained steps of calculating the length of the routes and updating the pheromone according to the routes resulted in multiple skeleton calls (Listing 3 lines 7-10). First, in line 7 and 8 the best route is identified and the path of the best route is saved. The best route is identified by using a reduce skeleton which finds the minimum value of all route distances (line 7). Depending on that value, firstly, the best sequence is identified with a map skeleton which checks for the route of each ant whether it is the best route and in case it finds the best route writes the best route to the corresponding data structure (line 9). The process of updating the pheromone had to be split in two skeletons. This is necessary in order to assure that synchronization takes place. First, the pheromone is updated based on all entries of possible routes. This means that some distances between cities are updated multiple times. Only after all updates are finished the second step of updating the pheromone can be executed which is to build the average of the old and new value. Moreover, extremely big or small values are removed as part of the user function in line

10. This is essential to prevent that ants get stuck for one route, since it has an extremely high pheromone value, and other routes are not taken into account.

In total the implementation did not vary significantly from the original algorithm. Only minor adjustments had to be made. The most outstanding deviation is the implementation of finding the minimum value of the routes and updating the pheromone. This change has the advantage that part of the operations are executed in parallel.

6 Our Case Study

The comparison between both parallel implementations of ACO used in this work can be done from different perspectives. As we propose the use of a high-level parallelization, the first point to analyse is the applicability of the tool. As mentioned in the previous section, the skeletons available in *Musket* suffice to create a high-level program of ACO. Another aspect which can be discussed is the usability of *Musket*. In Musket, 215 lines where needed compared to the 374 lines of the low-level implementation. For this comparison the code methods which read the data from files were excluded in both programs.

Most of the difference between the codes from both version comes from the *main* method, since kernel calls do not have to be written in *Musket*. But most importantly it is abstracted from all data transfers, which consume several code lines in the low-level program. Moreover, it should be considered that the lines-of-code metric is admittedly debatable since it misses to evaluate how complex the written lines are. In addition to requiring only 57% of the lines of code compared to the low-level program, *Musket* abstracts from complex decisions such as choosing the number of threads or moving data from the CPU to the GPU. Therefore, it could be argued that creating a *Musket* program does not only require less lines of code but additionally is written faster since the programmer does not need to think as much as in a low-level implementation using pure CUDA.

Another aspect to be evaluated in this work is the runtime. In order to test this aspect in both parallel implementations mentioned in this work, a NVIDIA GeForce RTX 2080 Ti accelerator containing 4352 CUDA cores, 11 GB memory and running CUDA 7.5 was used. The programs and results are publicly available [14].

The experiments performed in this work include instances of TSP taken from popular repositories with different graph sizes [15,16]. The selected TSP instances have different numbers of vertices so that the performance of low- and high-level ACO implementations can be evaluated on various problem sizes (Table 1). Unfortunately, the selection from the different repositories does not grow linearly in size. While some maps are very close in size, e.g. qa194 and d198, other maps have big differences, e.g. d1291 and pr2392.

For experimental purposes, each version was tested using different colony sizes (1024, 2048, 4096 and 8192 ants) for all TSP instances in order to simulate different levels of computational load. An important remark is that in this

Instance	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
# vertices	38	194	198	318	442	783	1002	1173	1291	2392

Table 1 TSPLIB

work, the fitness achieved is not relevant and the focus of the analysis is rather on the execution time. Since in essence both implementations represent the same algorithm and just vary in the parallelization approach and both achieve similar fitness values when using the same setup.

Aiming a fair comparison between both approaches, the execution times registered in the experiments are denoted in seconds and represent the whole execution of the algorithm, including the initialization process and data transfers between host and GPU. The runtimes are the average of 30 runs excluding the first runs due to the warm up of the GPU.

The results show that for the smaller problems, where less resources are needed, both implementations present very similar, almost identical, results. On the other hand, when more resources are needed, the low-level version tends to scale better and provide shorter execution times. Figure 1 puts the values beside each other graphically for an easier comparison. The values for the last and biggest map are excluded in this graph since they impede the readability and will be discussed afterwards.

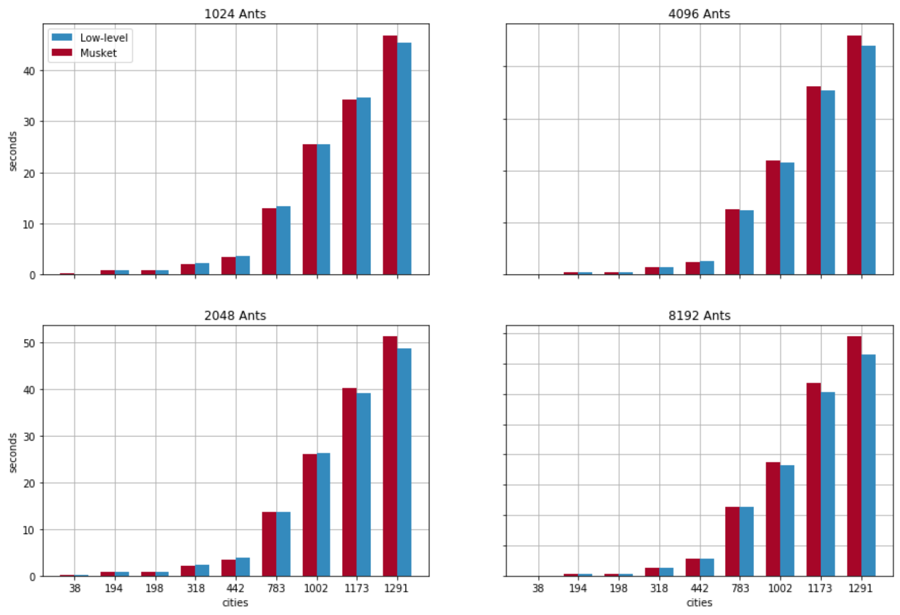


Fig. 1 Execution times comparison

Intuitively, the graph underlines how close the runtime values from the low-level program and the *Musket* program are. Also, as the TSP instance increases in size, the low-level program tends to be slightly faster, especially with higher values for the colony size.

The execution times follow a similar pattern also for the biggest problem tackled (pr2392). Figure 2 illustrates this. This pattern appears in all test cases and is directly connected to how the programs are organized. In the low-level version, the operations are executed specifically for a certain task, against general purpose operations present in the *Musket* program.

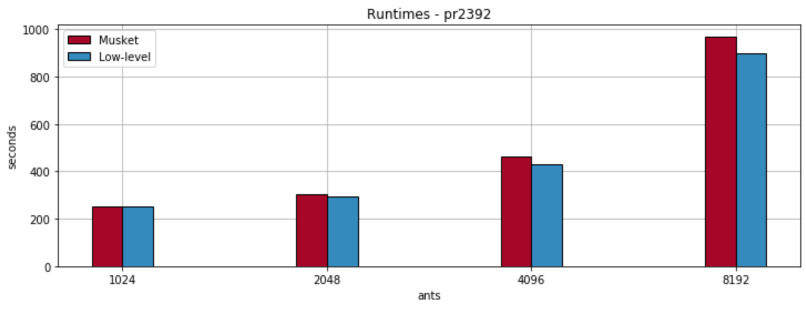


Fig. 2 Execution times for pr2392

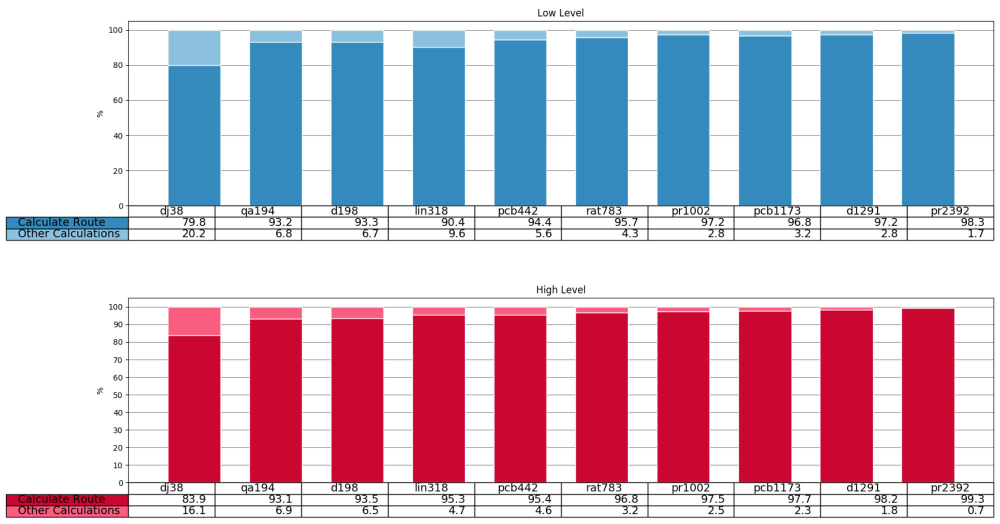
Table 2 shows the overall execution times for both parallel ACO implementations considering the different problems and setups.

	GeForce RTX 2080 Ti							
	1024		2048		4096		8192	
Problem	LL	*Musket*	LL	*Musket*	LL	*Musket*	LL	*Musket*
dji38	0.103	0.186	0.106	0.189	0.107	0.192	0.158	0.199
cat194	0.890	0.782	0.896	0.797	1.022	0.938	1.742	1.789
d198	0.906	0.852	0.914	0.861	1.037	1.001	1.784	1.804
lin318	2.217	2.103	2.250	2.103	2.726	2.665	6.175	6.261
pcb442	3.711	3.464	3.783	3.474	5.022	4.961	14.318	14.224
rat783	13.365	12.870	13.766	13.621	24.651	25.191	57.208	56.407
pr1002	25.409	25.529	26.307	26.116	43.194	43.657	91.539	93.51
pcb1173	34.680	34.201	39.201	40.23	70.605	72.321	151.416	158.965
d1291	45.37	46.744	48.832	51.306	87.808	91.888	182.741	197.17
pr2392	251.412	252.648	292.913	304.26	428.534	462.256	899.670	969.704

Table 2 Execution times comparison: low-level(LL) vs. *Musket*

Another factor that affects the execution times is the setup regarding the number of blocks and block size. As CUDA does not accept more that 1024 threads per block for most architectures and some kernels used the block size equal to the number of cities, some balancing was necessary. Using more blocks with less threads each, enables CUDA to run the algorithm but it also adds some overhead. Adaptations to solve this matter are easily done in the low-level program which generates a program with a better configuration of numbers of blocks and threads, which fits better to the problem, compared to high-level approaches. Of course the kernel instructions can be changed in order to optimize the execution time, but for comparison purposes only the numbers of blocks and threads were changed. In order to investigate further the runtime differences between both implementations, the runtime of single kernels was isolated and investigated separately.

The most important and time consuming step in ACO is the tour construction. Performed many times during the execution, it is affected by the colony size and also the graph size. Therefore, special attention was given to the time spent by each parallel implementation on creating routes. Figures 3 shows for the example of 1024 ants the proportional amount of time spent in each kernel by the *Musket* implementation and the low-level implementation. Obviously, even for the smallest map the route-construction kernel requires for both programs by far most of the runtime. Therefore, we firstly investigated the calculations of the route.



Low Level

	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Calculate Route	79.8	93.2	93.3	90.4	94.4	95.7	97.2	96.8	97.2	98.3
Other Calculations	20.2	6.8	6.7	9.6	5.6	4.3	2.8	3.2	2.8	1.7

High Level

	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Calculate Route	83.9	93.1	93.5	95.3	95.4	96.8	97.5	97.7	98.2	99.3
Other Calculations	16.1	6.9	6.5	4.7	4.6	3.2	2.5	2.3	1.8	0.7

Fig. 3 Proportional execution times of route Calculation

In order to compare the two implementations regarding the tour construction step, we have investigated in the average time spent in the tour construction per iteration as shown in Figure 4. The graphs show similar results to the total execution times mentioned previously and it is no wonder since the tour construction is the reason for a great part of the general execution times shown before.

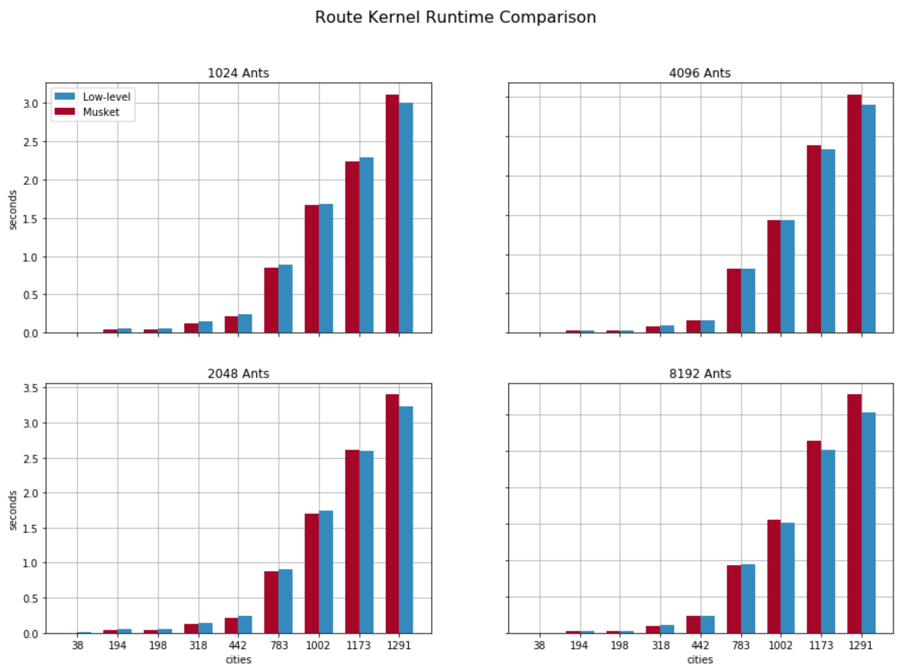


Fig. 4 Execution times comparison for the tour construction kernel

The kernels responsible for executing the other steps of the algorithm have almost equal execution times for both implementations. Furthermore, they represent a small, almost irrelevant, part of the whole execution time. Therefore no deeper analysis becomes necessary.

As an overall result, plenty of similarities between the execution times of both implementations investigated in this work can be observed. They show how the use of *Musket* can simplify the development of parallel programs, as the use of general purpose skeletons provided out of the box suffices to develop a parallel version of ACO in fewer lines of code and on a much lower complexity level when compared to the low-level CUDA implementation without impairing the performance.

7 Conclusion

The use of a high-level parallelization approach can be of great help for programmers aiming to run swarm intelligence algorithms on high-performance hardware, such as graphical processing units. In this work we have evaluated *Musket* as a approach for the parallelization of the Ant Colony Optimization algorithm in order to identify the pros and cons of using such a tool regarding the development aspect and also the performance aspect when compared to a low-level implementation.

Considering the development aspect, in its actual state, the skeletons embedded in *Musket* provide enough resources for the development of a parallel ACO. Furthermore, the experiments have shown that *Musket* offered some advantages in the terms of simplicity, requiring less skills to develop a high performance parallel version of ACO. Not only less lines of code were necessary, but it is also much simpler to program without having the concerns that regard the parallel aspects of programming a CUDA-based version of the code, such as data initialization, data transfers and the allocation of blocks and threads.

In terms of runtime, the ACO version implemented using *Musket* presented similar execution times compared to the low-level CUDA based implementation. Because of experimental reasons, the comparison was made using the same code for every test setup, without algorithm enhancements or adaptations for the problem instance. Results showed that the low-level version scaled better when the colony size increased and more resources were needed.

The ACO version used in this work was idealized to be a simple implementation for a single GPU environment. Many optimizations could be introduced in order to enhance the performance of the algorithm. Also, if the goal is to run in a new environment such as multiple GPUs or multiple computational nodes with multiple GPUs, more complex changes are necessary, which can be tricky even for experienced programmers. In this aspect, *Musket* has the advantage that the same program can be used to generate code for different architectures once there is a code generator for it.

Future works include the evaluation on different hardware, such as multiple GPUs in one computational node and also a cluster environment with many nodes and many GPUs per node. Furthermore, we want to enhance *Musket* to provide metaheuristic-specific skeletons in order to make better use of the hardware and reduce even more the execution times for such problems.

References

1. El-Ghazali Talbi. *Metaheuristics.* John Wiley & Sons, Inc., Hoboken, NJ, USA, jun 2009.
2. Nikolaos Ath Kallioras, Konstantinos Kepaptsoglou, and Nikos D. Lagaros. Transit stop inspection and maintenance scheduling: A GPU accelerated metaheuristics approach. *Transportation Research Part C: Emerging Technologies*, 55:246–260, 2015.
3. Marco Dorigo and Mauro Birattari. Ant Colony Optimization. (December), 2006.
4. Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. *Proceedings of the ACM Symposium on Applied Computing*, Part F147772:1534–1543, 2019.
5. Fabian Wrede, Christoph Rieger, and Herbert Kuchen. Generation of high-performance code based on a domain-specific language for algorithmic skeletons. *The Journal of Supercomputing*, (0123456789), 2019.
6. Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.
7. Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic, 1999.
8. Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Accelerating ant colony optimisation for the travelling salesman problem on the GPU. *International Journal of Parallel, Emergent and Distributed Systems*, 29(4):401–420, 2014.

9. José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.
10. Breno A.M. Menezes, Herbert Kuchen, Hugo A. Amorim Neto, and Fernando B. De Lima Neto. Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem. *2019 IEEE Congress on Evolutionary Computation, CEC 2019 - Proceedings*, pages 3094–3101, 2019.
11. Breno Augusto De Melo Menezes, Luis Filipe De Araujo Pessoa, Herbert Kuchen, and Fernando Buarque De Lima Neto. Parallelization strategies for GPU- ased ant colony optimization applied to TSP. *Advances in Parallel Computing*, 36:321–330, 2020.
12. Breno Augusto De Melo Menezes and Nina Herrmann. Musket repository. `https://github.com/wwu-pi/musket_dsl`, 2020. Last Change: 04.05.2020.
13. The Eclipse Foundation. Xtext documentation. `https://eclipse.org/Xtext/documentation/`, 2020.
14. Breno Augusto De Melo Menezes and Nina Herrmann. Programs and results. `https://github.com/wwu-pi/HLPP2020_ACO_Programs`, 2020. Last Change: 04.05.2020.
15. University of Waterloo. National traveling salesman problems. `http://www.math.uwaterloo.ca/tsp/world/countries.html`. Accessed: 14.03.2018.
16. Heidelberg University. Discrete and combinatorial optimization. `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/`. Accessed: 14.03.2018.

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration

Ryuta Tsunashima · Ryohei Kobayashi ·
Norihisa Fujita · Taisuke Boku ·
Seyong Lee · Jeffrey S. Vetter ·
Hitoshi Murai · Masahiro Nakao ·
Mitsuhisa Sato

Abstract Attached accelerators have been frequently used in recent High Performance Computing (HPC) systems because of their high performance/power ratio. In particular, the Graphics Processing Unit (GPU) is the most popular accelerator owing to its high peak FLOPS performance and high memory bandwidth supported by HBM2, etc. However, the performance of GPU depends highly on a large degree of SIMD parallelism and has difficulty sustaining a high performance on programs with frequent branch operations or a partially low degree of parallelism.

By contrast, a Field Programmable Gate Array (FPGA) has received attention as a different type of accelerator than GPU as a fully reconfigurable processor fitting the target applications. The high performance of FPGA is mainly provided by a pipelined operation and optimized circuit suitable for any operation even with frequent conditional branches. We have been focusing on the flexibility of FPGA to compensate for the weakness of GPU. We believe that the coupling of GPU with FPGA can result in one of the most powerful accelerating platforms available.

However, the program coding of GPU and FPGA coupling can be quite difficult for application users. Traditionally, CUDA by NVIDIA has been the most popular programming language with the largest share of GPUs used in HPC, whereas a hardware description language such as Verilog HDL has been used in FPGA programming. OpenCL coding has recently become available

Tsunashima, Ryuta · Kobayashi, Ryohei · Fujita, Norihisa · Boku, Taisuke
Graduate School of Science and Technology, University of Tsukuba, Ibaraki 305-8577, Japan
E-mail: tsunashima@hpcs.cs.tsukuba.ac.jp

Kobayashi, Ryohei · Fujita, Norihisa · Boku, Taisuke
Center for Computational Sciences (CCS), University of Tsukuba, Ibaraki 305-8577, Japan

Lee, Seyong · Vetter, Jeffrey S.
Oak Ridge National Laboratory (ORNL), P.O. Box 2008, Oak Ridge, TN 37831, USA

Murai, Hitoshi · Nakao, Masahiro · Sato, Mitsuhisa
Riken Center for Computational Science (R-CCS), RIKEN, Hyogo 650-0047, Japan

even on high-end FPGAs. Moreover, several recent studies have also enabled the OpenACC coding for use in FPGA. In this study, we provide a unified programming system based on OpenACC for a platform equipped with both GPU and FPGA aiming at the next-generation accelerated supercomputer framework. Our programming environment is called Multi-Hybrid OpenACC Translator (MHOAT), and in this paper, we describe the basic concept and prototype system of MHOAT based on an evaluation on the amount of coding required and the performance of a hybrid multi-device accelerated system.

Keywords GPU · FPGA · OpenACC · API · Multi-hybrid acceleration · MHOAT

1 Introduction

One of the most important issues on large-scale supercomputers targeting High Performance Computing (HPC) applications is how to reduce the power consumption while sustaining a high performance. General purpose CPUs such as multi-core Intel Xeon provide high programming flexibility and computational efficiency; however, the performance in terms of power consumption (FLOPS/Watt) is barely increased owing to limited semiconductor technology, and it is difficult to lower the circuit signal voltage owing to signal noise. Accelerators, by contrast, are an attractive solution with an extremely high peak performance with a relatively low power consumption. For instance, an NVIDIA Tesla V100 (Volta) GPU provides approximately 7 TFLOPS in 64bit double precision with a peak performance of under 300 W of power. In fact, seven out of the top-10 systems in a recent top-500 list from November 2019 are equipped with GPUs as accelerators[1].

The most important technology in driving GPU for HPC applications is the programmability. Because GPU was developed originally for graphics processing, the coding of this device is difficult for application users despite its high floating point operation performance. However, after OpenCL compilers were made available for GPUs, the barrier was lowered and NVIDIA started providing a CUDA programming environment for easier programming in the C/C++ language base. PGI began providing a Fortran compiler for CUDA to help with a large amount of dusty-deck Fortran coding for scientific computing.

Although the difference between the peak and sustained performances when applying GPU is significant, the performance is maintained at greater than 50% on the High-Performance Linpack (HPL) benchmark, which is used for the TOP500 list, and the use of GPU is extremely popular, specifically on the top-10 ranked ultra-large-scale systems. Because such systems consume significant power, it is extremely important to introduce high FLOPS/Watt devices such as GPUs. However, GPU is not a perfect solution even with high programmability. The extremely high performance of GPU is provided by a high degree of horizontal parallelism under a single instruction stream (in a SIMD manner, or SIMT manner in NVIDIA GPUs) to reduce the power consumption for operation control. Therefore, GPU must basically be applied for

Table 1 Acceleration devices

Device	GPU	FPGA
Parallelism	SIMD	Pipeline
Memory	Strong (large HBM2)	Weak (DDR and small HBM)
Conditional Branch (true/false part run sequentially)	Weak	Strong (true/false cases run in parallel)
Lower Parallelism	Weak (most of core goes to rest)	Strong
Inter-node Communication	Weak (by CPU interconnect)	Strong (own optical link)

simple data parallel applications. Some codes used to achieve frequent conditional branches, partially a low degree of parallelism, or a frequent inter-node communication, are unsuitable for GPU.

On the other hand, FPGA has been attractive as a new type of accelerator for HPC applications, and is a fully reconfigurable processor allowing a rewriting of the inner circuit according to the application. The benefits of this device are as follows:

- It is optimizable for computational applications.
- It is vertically parallelizable in a pipeline manner.
- The optimized performance based on the power consumption can exceed that of GPU.
- Recent high-end FPGAs enable a direct communication between FPGA devices using high-speed optical links.

However, FPGA also has several disadvantages for HPC use:

- The Hardware Description Language (HDL) is difficult to program for general application users.
- A long compilation time (usually from several to more than 10 hours) prevents the productivity of the codes.
- Hardware resources (logic elements and memory devices) are limited because they cannot be reused in different locations within a large code unlike with an ordinary CPU or GPU.

The recent high-end FPGA for HPC provides much larger hardware resources and memory capacity, which decreases some of the weaknesses described above. Herein, we focus on the complimentary characteristics of both devices, GPU and FPGA (Table 1). Thus, we have an idea to couple these devices together on a computational node even for use in a large-scale cluster system, achieving a type of 360-degree system that would allow each device to compensate the other for various types of HPC computing. We have been studying this concept, which we call Multi Hybrid Accelerated Supercomputing.

The platform used by our concept is based on a complex computational node with CPUs, GPUs, and FPGAs connected by an intra-node PCIe switch[2]. Recent CPUs are equipped with a number of PCIe lanes (gen3 or gen4), where multiple devices can be easily connected. However, we need additional FPGAs beside of multiple GPUs on a node. Thus, we may need external PLX switches to enlarge the number of PCIe lanes sufficiently to support multiple GPUs and FPGAs together. Of course, another set of PCIe lanes is required for an interconnection network such as InfiniBand.

Although the hardware construction is relatively easy thanks to the generality of PCIe to connect all devices, the program coding is quite difficult for application users. The most popular GPU devices in current HPC systems include the NVIDIA Tesla series, such as a V100 (Volta architecture). CUDA is the common programming environment for these GPUs. However, OpenACC[3] has recently been focused on as an easier programming framework than CUDA because it introduces directive-based and incremental coding for an original sequential code such as OpenMP on a general CPU.

The programming environment for FPGA has also evolved. It started with HDL such as VHDL or Verilog HDL. Recently, however, major FPGA vendors such as Intel and Xilinx have provided a High Level Synthesis (HLS) environment including OpenCL or C/C++ languages. For users familiar with accelerated parallel programming, the OpenCL solution is quite welcoming to porting their code to a "relatively" high level programming language other than HDL. However, even OpenCL is difficult for users who are familiar with very traditional OpenMP + MPI style parallel programming. To support a higher level of programming, OpenACC compiler for FPGA has been studied[4].

Herein, we introduce a common basic environment for programming on both devices, OpenACC, to simplify the coding on this complicated platform of Multi Hybrid Accelerated Supercomputing. The CPU controls both devices and offloads the heavy computations to them according to the characteristics of the calculations. Because OpenACC allows a user to synchronize the data contents on the host memory and accelerating device memory (GPU or FPGA), the user can manage the data duplication and localization over multiple device memories on CPU, GPU, and FPGA.

This research covers the comprehensive programming interface using only OpenACC for conducting cooperative computations on GPU and FPGA based on our proposed concept. One of the important components of our research is the compiler of OpenACC used for the FPGA. We apply OpenARC compiler developed by Oak Ridge National Laboratory, which is a source-to-source compiler applied to generate OpenCL code from OpenACC to run on FPGA. Although OpenARC compiler can support other devices such as NVIDIA GPU, we use another commercial OpenACC compiler for the GPU for several reasons (discussed later in this paper). The research issues herein include the following:

- How to describe the target offloading devices.
- How to generate multiple codes to be compiled individually by different compilers for the GPU and FPGA.
- How to finalize the generation of a single object executable code to manage three devices, i.e., CPU, GPU, and FPGA, in a single process

2 Related Studies

In [5], an astrophysical simulation was implemented for FPGA designed to enhance the inter-node GPU-to-GPU communication as well as partial computation offloading to the FPGA. The dedicated FPGA system is called PCI Express Adaptive Communication Hub version 2 (PEACH2). The use of FPGA for a PCIe hub switch with a programmable communication feature to support GPU acceleration is also called a Tightly Coupled Accelerators (TCA) method[6]. PEACH2 extends the communication link from the intra-node GPU-FPGA to the external intra-node communication between them. The FPGA works as an DMA controller between remote GPUs based on memory address mapping of GPU global memory to the PCIe address space, and transfers the data using the PCIe protocol. The core computation of the target application of a Locally Essential Tree (LET) algorithm for a fundamental astrophysical gravity simulation is efficiently operated through a combination of FPGA computations and communications over an external PCIe link between FPGAs over the nodes. The prototype system of PEACH2 achieves a 7.2x improvement in the computing time than CPU for an N-body simulation based on the LET algorithm.

In [7], FPGA is introduced to offload another astrophysical simulation using the Authentic Radiation Transfer (ART) method, which is an essential physical phenomenon on the early stage universe generation in the space. The Accelerated Radiative transfer on Grids using Oct-Tree (ARGOT) program includes two astrophysical simulations, i.e., the ARGOT method and the ART method. With the ART method, the parallel computational elements are too small and without enough SIMD-style parallelism to be handled by GPU, particularly for small sized problems where the performance of GPU is as low as that of CPU. The pipelined computation by FPGA can be efficiently achieved and the simulation speed is not affected by the problem size with up to a 6.9x times faster performance than that of GPU computations.

We proposed a new concept of accelerated computing called Accelerators in Switch (AiS)[8], which is a computing model that aggressively applies FPGA for both computations and communication. This approach is expected to enable an ideal high-performance parallel computing system to be developed by combining the high-speed computing of GPU. In most applications, it is difficult to outdo the absolute high performance of GPU, although there are some weaker computational aspects of the GPU architecture. This solution realizes a complimentary system using GPU and FPGA to compensate one another. Specifically, multi-physics applications such as in [7] include multiple phenomena of simulations with different properties, and thus can be shared between the GPU and FPGA. However, the programming of a coupled GPU and FPGA introduces the following difficulties for users:

- The programming languages for development are different on each accelerator.
- Poor coding and performance portability occur owing to an incompatibility in multiple languages.

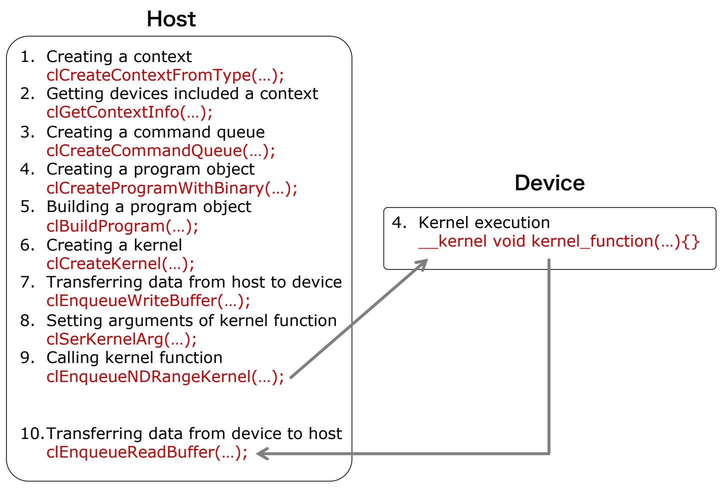

Host
1. Creating a context
clCreateContextFromType(…);
2. Getting devices included a context
clGetContextInfo(…);
3. Creating a command queue
clCreateCommandQueue(…);
4. Creating a program object
clCreateProgramWithBinary(…);
5. Building a program object
clBuildProgram(…);
6. Creating a kernel
clCreateKernel(…);
7. Transferring data from host to device
clEnqueueWriteBuffer(…);
8. Setting arguments of kernel function
clSerKernelArg(…);
9. Calling kernel function
clEnqueueNDRangeKernel(…);
10. Transferring data from device to host
clEnqueueReadBuffer(…);
Device
4. Kernel execution
__kernel void kernel_function(…){}

Fig. 1 Flow of OpenCL programming

– The performance of multiple devices must be balanced for their efficient use.
– A number of key performance optimization issues such as memory allocation and communication between devices must be overcome.

3 Existing programming environment

3.1 OpenCL

In general, FPGA programming is performed using HDL, which is a language applied to describe a logic circuit design for defining the components and behavior of such a circuit. Although the code looks like an ordinary program, each element of the code specifies the logic element, operational condition, and connectivity between logic elements. Most importantly, because each component is compiled in the corresponding hardware logic, these components work simultaneously with a potentially high degree of parallelism. Therefore, the description is quite complicated with large number of code lines and difficult to debug because the relationship and activated timing on the processor clock are not user friendly. The optimization is also difficult because the processor frequency is determined through the compilation and any unbalanced or overly long stage of logic element easily causes a bottleneck, enlarging the pipeline stage and degrading the active clock frequency. In conclusion, the code productivity of FPGA when applying HDL is quite low.

As a significant progress in FPGA programming, the recent FPGA development environment provides OpenCL coding instead of HDL. OpenCL is a programming language targeting heterogeneous computing platforms with accelerating elements. The parallel execution is conceptually involved in the description used to fit various types of accelerator architectures such as GPU or many-core processor. OpenCL supports API features to be shared for these devices, and the abstraction level is relatively low compared with ordinary CPU languages, and thus a high level of optimization is possible for users.

Regarding the recent high-end FPGAs, HLS environment with OpenCL is available using OpenCL alone or through a mixture with HDL. The grammar of OpenCL is based on C language and is much easier to introduce even for ordinary application users. It is also highly expected to reduce the coding cost when using FPGA compared with HDL coding. Figure 1 shows the flow of OpenCL programming. As like as CUDA, there are two types of program, i.e., the device code and the host code. The host code is a program used to control the accelerating devices such as GPU or FPGA on the CPU. The device code is a program running on the device. In OpenCL, these parts are separated into individual blocks and compiled individually. The host code is compiled by ordinary C compilers such as GCC, whereas the device code is compiled by special compilers for the target devices. For instance, AMD recommends using OpenCL for their GPUs, and device compiler tuning has been developed along with a CUDA compiler by NVIDIA. Although NVIDIA also provides the environment for OpenCL coding on their GPUs, they highly recommend using CUDA instead, and numerous codes have been developed using CUDA rather than OpenCL. In conclusion, it is difficult to achieve a uniform compilation environment based on OpenCL when combining GPU and FPGA.

3.2 Combined GPU and FPGA compilation environment

It is basically possible to combine GPU and FPGA devices in a single program by coupling heterogeneous language environments, using CUDA for GPU and OpenCL for FPGA, as examples. Because both CUDA and OpenCL share the same style of coding with the host and device codes, it is theoretically possible to share one host code in two different types of devices. We confirmed that the NVIDIA CUDA compiler for GPU and Intel FPGA SDK for OpenCL have no problem in terms of linkage, such as symbol conflicts[9].

However, such a programming environment is difficult to apply for users, although it works perfectly with high-level optimization. This is true not only for grammatical differences but also for differences in the basic architecture and optimization techniques used to exploit the potential parallelism, such as a kernel parallelism level specification and kernel argument specification. These differences might confuse programmers, inducing programming bugs and a poor optimization. In addition, the hardware resources on FPGA are limited and thus FPGA is unsuitable for extremely high parallel data level processing unlike with GPU. Therefore, a comprehensive programming interface enabling these accelerator features to be exploited is necessary.

3.3 OpenACC

OpenACC is another API for the programming of an accelerator, and is a language extension standard of directives, supporting C, C++, and Fortran as popular languages used in scientific applications. Such a directive-based programming is well accepted in the parallel computing community. Its advantages are as follows:

– Programming is easier than with CUDA and OpenCL thanks to a high level of abstraction.
– No distinction exists between the host and device codes.
– Hardware acceleration is enabled by only writing directives.
– High productivity with incremental improvement allowing directives to the applied in any part of the code.

To provide a higher level of abstraction and ease of coding in a unified API, we focus on OpenACC programming for both GPU and FPGA in our Multi Hybrid Accelerated Supercomputing approach. To understand our concept and design, we briefly introduce the basic framework of OpenACC code.

Figure 2 shows a code example in C. First, all of the initial data exist in the CPU memory. It is therefore necessary to transfer the target data to the target device through the following directive:

```
#pragma acc data
```

The `copyin()` clause specifies the data transfer from the host to the accelerator, and the `copyout()` clause specifies the opposite direction of a data transfer. The data transfer timing is automatically determined as the beginning of an offloaded block for a copyin() and the end of the block for a copyout().

The code block to be offloaded to the target device is then specified by the following directive:

```
#pragma acc kernels
```

In addition, OpenACC programs can even execute a computation on an accelerator using only this directive, although an unspecified processing depends on the compiler, particularly with the possibility of a large data transfer cost owing to a conservative data movement and timing between the host and device. Thus, it is important to describe the data directive explicitly with `#pragma acc data`.

The following directive is used to specify the parallel computations in a loop:

```
#pragma acc loop
```

An `independent` clause specifies that there is no dependency in the loop and suggests the compiler optimize the code such as through software pipelining.

```
1. Defining transfer data
   #pragma acc data copyin(in1[0:N], in2[0:N])
   copyout(out[0:N])
   {
2. Defining a target of kernel execution
   #pragma acc kernels
3. Defining for each loop
   #pragma acc loop independent
       for (i = 0; i < N; i++) {
           out[i] = in1[i] * in2[i];
       }
   }
```

Fig. 2 Example code of OpenACC

3.4 OpenACC compilers for different accelerators

There are several compilers ready for OpenACC already on the market, including PGI compiler[10] and GCC. In particular, the PGI compiler is the recommended compiler for NVIDIA GPUs with high level optimization because PGI is a group company of NVIDIA. By contrast, the OpenACC compiler for FPGA is still in the research stage and is unavailable commercially. OpenARC[4] is a research compiler for various accelerators including GPU and FPGA. This compiler has been open to research regarding FPGA utilization based on easy OpenACC coding. OpenARC for FPGA transfers the OpenACC target code to OpenCL to be processed by Intel SDK, allowing OpenCL to be applied to Intel FPGA. Because the tool provides HLS, it can convert a C-based `for` statement into a pipeline circuit that can process the streams. In the current implementation of OpenARC, C language is available for the target code, although the translated output is in C++. For this study, the Center for Computational Sciences (CCS) at the University of Tsukuba collaborates with Future Technology Group at Oak Ridge National Laboratory (ORNL) to apply OpenARC compiler to Multi Hybrid Accelerated Supercomputing. We also collaborated with the Riken Center for Computational Science (R-CCS) to apply their Omni OpenACC compiler for the frontend processing of our multi-hybrid language environment.

OpenARC can output codes individually for multiple target accelerators including GPU, many-core CPU or FPGA. For the GPU compilation, the target device is limited to the NVIDIA GPU and the compiler generates a CUDA code as the output. For FPGA, OpenCL code is generated for Intel SDK as previously described. However, the current OpenARC does not support multiple target devices *simultaneously* during a process, but supports multiple devices individually, and thus we cannot apply it directly to the handling of the GPU and FPGA together.

Therefore, we apply the OpenARC compiler as a backend compiler only for the FPGA part in our multi-hybrid programming environment. We cannot apply the same compiler for the GPU part because the codes generated for the FPGA and GPU share several symbols and functions but with different behaviors, and thus we cannot mix the two generated codes into a single process using the OpenARC compiler only. Therefore, we apply another OpenACC compiler, i.e., the PGI compiler, for the backend compiler of the GPU part.

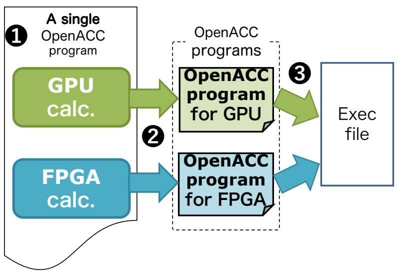


Fig. 3 Suggesting programming environment overview

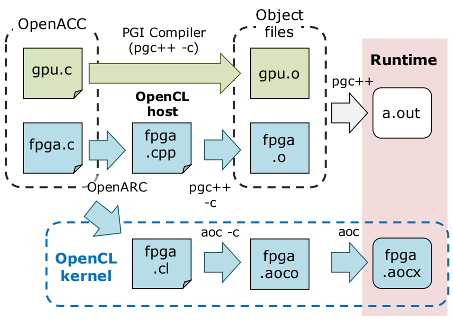


Fig. 4 Flow of backend compiling and linking

Although it is theoretically possible to separate the device offloading for each accelerator by hand and apply the appropriate compiler for each part, doing so is quite complicated and a burden for application users. For much easier coding to support users, multi-hybrid coding toward efficient 360-degree accelerated computing is introduced.

4 Unified programming environment for cooperative computations

4.1 Overview

To realize a programming environment for a multi-hybrid accelerator system, we propose the language processing framework shown in Figure 3. The target code is written using OpenACC in which it is explicitly specified which device should be applied for specified offloading parts. These parts are then detected and split into individual files allowing each accelerator to compile using the appropriate existing compiler. The file should be split automatically as the source-to-source translation from OpenACC to OpenACC. Finally, each back-end compiler generates a binary code to be executed on each device as a kernel function, and the binaries are assembled into an executable binary including the host code.

Figure 4 shows the flow of the backend compiler. Each part contains an offload description by the OpenACC directives and is compiled separately by the PGI compiler for GPU and the OpenARC compiler for FPGA. Finally, the following binary objects are linked into a single executable file; (1) GPU-related binaries by the PGI compiler such as user code itself, kernel invocation, data transfer, and other miscellaneous jobs, (2) FPGA-related host code compiled by Intel FPGA SDK for OpenCL[11] after translation by the OpenARC compiler, and (3) the peripheral libraries for GPU and FPGA are linked into a single executable file. The OpenCL kernel code generated by the OpenARC compiler is compiled into the FPGA programming bitstream file by the Intel FPGA SDK for OpenCL Offline Compiler `aoc`.

Our translator system operating as an OpenACC-OpenACC source converter is developed based on the Omni Compiler Infrastructure (Omni)[12] researched through collaboration between R-CCS and CCS. Omni is a compiler infrastructure based on a source-to-source translation for Fortran and C for use in large-scale parallel computers, supporting OpenACC, XcalableMP[13] and XcalableACC[14] languages. The OpenACC feature supports the generation of CUDA or PZCL (custom OpenCL for PEZY-SC)[15], although FPGA-ready OpenCL code generation is not supported. Moreover, the GPU code generation optimization is unsuitable for recent GPUs such as the Volta architecture by NVIDIA. For these reasons, we apply an Omni compiler as a parser of the source code to help with the translation of OpenACC-to-OpenACC as a part of our new environment for multi-hybrid acceleration.

4.2 MHOAT: meta-compiler

We developed a sort of meta-compiler over a compiler toolkit and a back-end compiler, as described in the previous subsections. This meta-compiler is named as Multi-Hybrid OpenACC Translator (MHOAT). Currently, MHOAT supports C language only owing to the code parsing of the Omni C compiler, which is not ready for OpenACC Fortran.

In MHOAT, the main feature is to recognize the user description to map the offloaded part to one of the target accelerators, and separate that part to the target OpenACC code for the device. The current OpenACC specification implies a feature to determine the target device; however, this feature is used for code optimization and applies a different set of parameters, and thus is not suitable for our purpose. Instead, we originally introduce a new pragma to specify the target device as follows.

```
#pragma accomn ondevice(DEVICE)
```

"`accomn`" is a special pragma under the pragma family of "`omn`," which is originally introduced for the unique extension of OpenMP pragma in the Omni compiler suite. We define the new pragma `accomn` according to this tradition of Omni, particularly for an OpenACC extension. Listing 1 shows an example of the use case of this pragma. The `ondevice(DEVICE)` clause specifies the target device provided by the input value DEVICE used to assign that offloaded part to the pragma. Currently, we support only `GPU` or `FPGA` as

```
void fuga() {
#pragma accomn ondevice(FPGA)
{
#pragma acc data copy(a, b)
{
#pragma acc kernels loop independent
        for(i=0; i < N; i++)
                a[i] = a[i] + b[i];
}
}
}
```

Listing 1 accomn ondevice usage

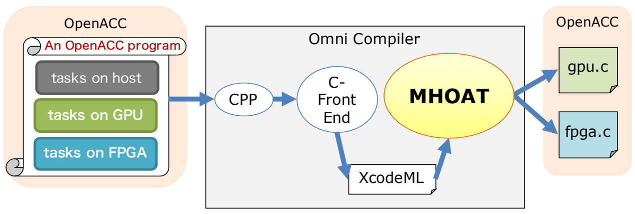


Fig. 5 MHOAT processing flow

a DEVICE. We designed MHOAT for easy extension of this target for future use.

Under the restriction caused by the current implementation, this pragma must be described in any function except the main() function. This is because the current implementation requires binding a unique symbol for each offloaded part written in an individual function. It does not affect the coding much, but our future work will be to relax this restriction. Some other detailed features of OpenACC are not fully supported owing to the prototype implementation of MHOAT. The purpose of the current version of MHOAT is to demonstrate the fact that the functionality of our concept fits the practical applications under the minimum extension of the standard OpenACC code.

Figure 5 shows the processing flow of MHOAT. In the Omni toolkit, the included header part and macro functions in C are first treated by the C Preprocessor (CPP). The code is then parsed and translated into a special internal object form called XcodeML, which is a common object in the Omni toolkit. Finally, the XcodeML code is handled by MHOAT to analyze and separate the code into the appropriate partial source for GPU and FPGA compilation. As a result, MHOAT generates two sets of files for the GPU to be handled by the PGI compiler and for the FPGA to be handled by the OpenARC compiler. Inside the OpenARC compiler, the code is translated into OpenCL to be handled by Intel FPGA SDK for OpenCL. In addition, a non-offloaded code including the main() function is also treated by the PGI compiler to be included in the GPU part to avoid any confusion caused by the C++ parsing of OpenARC.

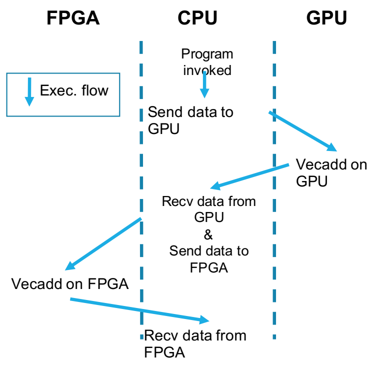


Fig. 6 Flow of the test program

5 Verification of MHOAT functionality

We confirmed the functionality of MHOAT using a test code with GPU and FPGA multi-hybrid acceleration based on OpenACC and our extended pragma, using the backend compilers of PGI and OpenARC. We carefully separate the original code into several split files to verify the functionality of the code translation by MHOAT. The target code verified herein is a toy program with the process flow shown in Figure 6.

The program consists of three code blocks in which the CPU, GPU, and FPGA execute individually. Current MHOAT can handle single FPGA and GPU for each on a computation node. First, the host initializes the data on CPU memory and transfers the data to the GPU using the `data` pragma of OpenACC. Then, the GPU performs a vector add computation with two vector data and generates a resulting vector. This is transferred to the CPU memory and transferred to the FPGA memory sequentially because there is no direct mapping or synchronization feature of the different devices in the current version of MHOAT. Next, another vector add operation is conducted on the FPGA, and finally the resulting vector is sent back to the CPU memory. The following code is the actual program we verified in this study.

Although the program in Listing 2 is a simple toy code without any scientific meaning, it is a sort of collection of typical processing in HPC applications and we can confirm the correctness of the code execution, data handling, and kernel invocation for both the GPU and FPGA to lead a complicated and practical multi-hybrid accelerated code. The correctness of the execution result is confirmed through both GPU+FPGA and CPU execution.

We applied our Cygnus[17] cluster system at CCS, University of Tsukuba for verification of MHOAT. MHOAT is a meta-compiler for a single process, for which only a single node of Cygnus is used. The specifications of Cygnus are shown in Table 2, and a block diagram of the computational nodes is shown in Figure 7. As shown here, Cygnus is one of the worlds most advanced clusters, in which each computation node is equipped with multiple FPGA cards and

```
#include <stdio.h>
#include <stdlib.h>
#include ”acc_ondevice.h”
#include ”each_block_global_decl.h”
int main(int argc, char** argv) {
    int workers = 16;
    int gangs = 256;
    int size = workers * gangs;
    float* A = (float*) malloc(size * sizeof(float));
    float* B = (float*) malloc(size * sizeof(float));
    float* D = (float*) malloc(size * sizeof(float));
    float* E = (float*) malloc(size * sizeof(float));
    int i, error=0;
    for (i = 0; i < size; i++) {
        A[i] = (float) i;
        B[i] = (float) i * 100;
    }
    funcGPU(A, B, D, size);
    for (i = 0; i < size; i++) {
        if (D[i] != (float) i + (float) i * 100) error++;
    }
    printf(”errorGPU:%d\n”, error);
    funcFPGA(D, E, size);
    for (i = 0; i < size; i++) {
        if (E[i] != (float) i + (float) i * 100 + (float) i) error++;
    }
    printf(”errorFPGA:%d\n”, error);
    return 0;
}
void funcGPU(float* a, float* b, float* d, int size) {
#pragma accomn ondevice(GPU)
{
    int j;
#pragma acc data copyin(a[0:size], b[0:size]) copyout(d[0:size])
{
#pragma acc kernels loop independent gang worker(16)
        for (j = 0; j < size; j++) {
            d[j] = a[j] + b[j];
        }
} } }
void funcFPGA(float *a, float *b, int size) {
#pragma accomn ondevice(FPGA)
{
    int j;
#pragma acc data copyin(a[0:size]) copyout(b[0:size])
{
#pragma acc kernels
{
#pragma acc loop independent
    for (j = 0; j < size; j++) {
        b[j] = a[j] + (float)j;
    }
 } } } }
```

Listing 2 The target program code

multiple GPU cards. In fact, there are two FPGA cards and four GPU cards all connected by the PCIe gen3 interface as well as a CPU and InfiniBand interconnection. Through this verification, we apply only a single FPGA and a single GPU in a computational node under current limitation of MHOAT.

For the target code, MHOAT generates two codes, one for the FPGA (Listing 3) and one for the GPU (Listing 4). Unnecessary comments are omitted. In Listing 3, it is shown that only the lines for an `#include` directive statement and the function describing `#pragma accomn ondevice(FPGA)` in Listing 2 are generated. Here, statements such as `# 1 "each_block.c"` are line markers added by CPP, and are used by MHOAT for a code analysis. After processing, although they remain for further use, they are mostly ignored by the following compilers. By contrast, as shown in Listing 4, the main() function and the accelerated part (function) remain for the GPU using `#pragma accomn ondevice(GPU)` because the host part is treated by the PGI compiler.

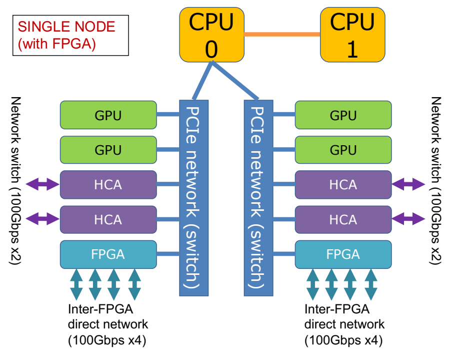

SINGLE NODE (with FPGA)
CPU 0
CPU 1
GPU
GPU
HCA
HCA
FPGA
PCIe network (switch)
PCIe network (switch)
GPU
GPU
HCA
HCA
FPGA
Network switch (100Gbps x2)
Network switch (100Gbps x2)
Inter-FPGA direct network (100Gbps x4)
Inter-FPGA direct network (100Gbps x4)

Fig. 7 GPUFPGA node in Cygnus

Table 2 Verification environment

CPU	Intel Xeon Gold 6126 (12C / 2.6GHz) x2
GPU	NVIDIA Tesla V100 (32GiB HBM2 PCIe 3.0 x16) x4
FPGA	Intel Stratix 10 GX 2800 (BittWare 520N[18] PCIe Gen3 x16) x2
OS	CentOS 7.3
GPU compiler	PGI Compiler 19.1
FPGA compiler	OpenARC V0.17 (Oct, 2019)
OpenCL compiler	Intel FPGA SDK for OpenCL 19.1.0.240

These codes are compiled by the backend compiler, as shown in Figure 4. As a result, a single executable file is generated as an ordinary executable binary.

6 Evaluation of code volume and execution time

After verification of the code generation and execution, we evaluated the ease of the code production and the actual performance of simple code on a single node of the Cygnus cluster. Herein, we verify the coding amount of a traditional mixture of CUDA and OpenCL for GPU/FPGA coupling and a MHOAT supported OpenACC unified code. The execution environment is as follows: OpenARC V0.21 (February 13, 2020), Intel FPGA SDK for OpenCL 19.4.0.64, CUDA 10.2, and PGI Compiler 19.10.

This is again a type of toy program because our purpose is the comparison of code amount between traditional approaches and our methods, but more complicated and similar to a typical HPC code (but still without scientific meaning). We first apply a matrix-matrix multiplication on the GPU, followed by a conjugate gradient (CG) method for a linear equation solver on the FPGA. The target matrix was taken from the Matrix Market [16]. Here, the resulting data of the GPU is referred to by the FPGA. We adjusted

```
#include <stdio.h>
#include <stdlib.h>
#include "acc_ondevice.h"
#include "each_block_global_decl.h"
# 1 "each_block.c"
# 2 "each_block.c" 2
# 3 "each_block.c" 2
# 4 "each_block.c" 2
# 5 "each_block.c" 2
# 57 "each_block.c"
void funcFPGA(float * a, float * b, int size)
{
# 61 "each_block.c"
;
{
int j;
# 64 "each_block.c"
;
#pragma acc data copyin ( a [ 0 : size ] ) copyout ( b [ 0 : size ] )
{
# 66 "each_block.c"
;
#pragma acc kernels
{
# 68 "each_block.c"
;
#pragma acc loop independent
# 68 "each_block.c"
for(j = (0); j < size; j++) {
{
# 70 "each_block.c"
(*(b + j)) = ((*(a + j)) + ((float)(j)));
}
}
}
}
}
}
```

Listing 3 computation of FPGA

the same matrix size of the matrix-multiplication on the GPU and CG on the FPGA as 10974 10974, and with 219,812 non-zero elements on the CG method, and the number of iterations of the CG method is stopped at 1000.

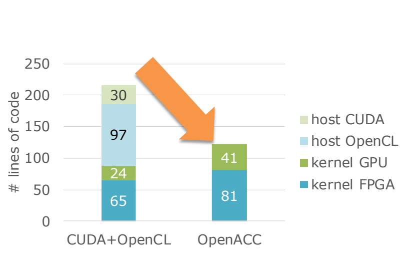
First, we compare the amount of code description. Figure 8(a) shows the number of lines of CUDA+OpenCL versus that of OpenACC-only. The first method corresponds to a traditional CUDA (GPU) and OpenCL (FPGA) coupling, whereas the latter case is by MHOAT. In the CUDA+OpenCL case, we measured the host code for the CUDA and OpenCL separately according to their operation. In OpenACC-only, only the kernel-GPU and kernel-FPGA are referenced because there is almost no host code description in OpenACC in this target code. All comments and blank lines are omitted. Here, we can see that our MHOAT with OpenACC-only coding reduces the total line count by approximately 44% compared with the original CUDA+OpenCL mixture coding. In particular, the traditional method requires a number of host code lines even with such a small and simple program. In addition to this advantage proof, we examined the number of characters instead of lines. The complexity of a line may be extremely large in several cases, and thus we focus on the actual number of characters. The result is shown in Figure 8(b). Here, we can see that the number of characters is reduced by approximately 64%. The reduction in the host code greatly contributes to OpenACC-only manner. With the CUDA+OpenCL method, many specific notations regarding the memory

```
#include <stdio.h>
#include <stdlib.h>
#include "acc_ondevice.h"
#include "each_block_global_decl.h"
int main(int argc, char * * argv)
{
int workers = 16;
int gangs = 256;
int size = workers * gangs;
float * A = (float * )(malloc(size * (sizeof(float))));
float * B = (float * )(malloc(size * (sizeof(float))));
float * D = (float * )(malloc(size * (sizeof(float))));
float * E = (float * )(malloc(size * (sizeof(float))));
int i;
int error = 0;
for(i = (0); i < size; i++) {
{
(*(A + i)) = ((float)(i));
(*(B + i)) = (((float)(i)) * (100));
}
}
funcGPU(A, B, D, size);
# 26 "each_block.c"
for(i = (0); i < size; i++) {
{
{
if((*(D + i)) != (((float)(i)) + (((float)(i)) * (100)))) {
error++;
}}}}
printf("errorGPU:%d\n", error);
funcFPGA(D, E, size);
for(i = (0); i < size; i++) {
{
{
if((*(E + i)) != ((((float)(i)) + (((float)(i)) * (100))) + ((float)(i)))) {
error++;
}}}}
printf("errorFPGA:%d\n", error);
return 0;
}
void funcGPU(float * a, float * b, float * d, int size)
{
;
{
int j;
;
#pragma acc data copyin ( a [ 0 : size ] , b [ 0 : size ] ) copyout ( d [ 0 : size ] )
{
;
#pragma acc kernels loop independent gang worker ( 16 )
for(j = (0); j < size; j++) {
{
(*(d + j)) = ((*(a + j)) + (*(b + j)));
}}}}}
```

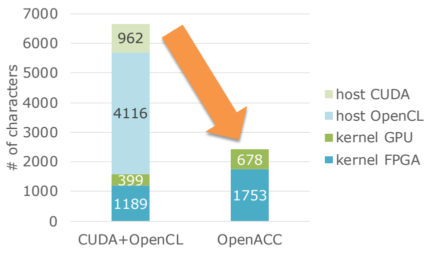
Listing 4 computation of GPU and host processing

and thread are required, which are not fundamental for computational programming, and cause a large overhead for application users.

Next, we measured the execution time of both methods, the results of which are shown in Figure 9. Unfortunately, the total execution time with the GPU and FPGA when applying the MHOAT method is approximately 22% larger than that of a traditional method. Here, the execution time using GPU with OpenACC is approximately 1.7x larger than that of GPU with CUDA, whereas the difference between OpenCL and OpenACC on FPGA is only 7% (OpenACC coding is slightly larger with a 1.07x increase in the computation time). We are investigating such a large performance degradation on GPU. The offloading instruction on OpenACC is highly abstract, and the default setting of the OpenACC kernel execution depends highly on the compiler. In this example, we found that the shared memory of the CUDA device is properly used in the CUDA coding, whereas that of the OpenACC compiler is not.



(a) Number of lines



(b) Number of characters

Fig. 8 Programming cost comparison

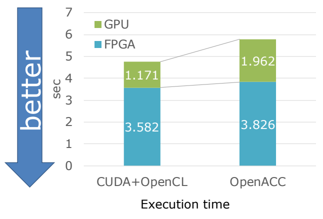


Fig. 9 Execution time

We think this evaluation is extremely preliminary and unfair for both methods. Please remind that the performance degradation is caused by CUDA to OpenACC rewriting, not by MHOAT itself. On the other hand, the OpenARC compiler for FPGA seems to obtain a good performance, and the development of OpenARC is still on going, and thus we can expect a higher performance in the near future.

7 Conclusion

In this paper, we proposed a new platform and programming methodology using Multi Hybrid Accelerated Supercomputing with multiple accelerators, and the coupling of GPU and FPGA on a single computational node. The accelerator characteristics of both devices are quite different, and there is a large room for compensation with each other toward a highly sustained performance without any bottleneck on a single program under limited power consumption. To support OpenACC-only coding and a unified code to support both devices, we developed a prototype meta-compiler called MHOAT. MHOAT allows users to describe an OpenACC directive-base acceleration to specify the offloaded device on each accelerated part by an extended pragma on OpenACC. A traditional method exists for programming both devices by hand using CUDA and OpenCL for GPU and FPGA, respectively; however, the coding with a unified framework of OpenACC is much easier for users in terms of both simplicity and an abstraction of the code.

Through a preliminary evaluation, we first confirmed the correctness of MHOAT, followed by the amount of code generated and the execution performance. Using a simple code and compilation, we confirmed the correctness of MHOAT with proper offloading on two different devices. Based on an examination of the amount of code lines and characters, the numbers of lines and characters were reduced to 44% and 64%, respectively. Along with the amount of code generated, it is also important to avoid the complexity for a mixture of programming by users. We demonstrated the ease of MHOAT coding through a unified programming style even on multi-hybrid accelerators.

The performance comparison between CUDA+OpenCL and MHOAT OpenACC-only on simple matrix-matrix multiplications and the CG method shows the disadvantage of MHOAT in its current implementation including the target code itself. The performance of MHOAT is 22% lower than a traditional approach, particularly with approximately a 1.7x longer execution on the GPU side. The result is still under investigation, but we believe our method will open a new way to achieving multi-hybrid programming for the next-generation combined accelerator environment.

Our future study will focus on the performance improvement of MHOAT, particularly an optimization methodology to apply OpenACC for both devices. We are preparing a real application, such as an ARGOT program[7] rather than a toy code, to demonstrate the practicality and total performance when applying GPU+FPGA coding and its execution. Another issue is a parallelized code with multiple nodes combined with MPI programming. This theoretically causes no problem, and we will create such examples. Finally, we will apply this new compilation method to our original PGAS-based parallel-language XcalableACC in the future.

Acknowledgements This research is partially supported by the "Communication-Computation Unified Supercomputing" project under MEXT's "Next Generation Supercomputer R&D" program and through collaborative research among CCS, R-CCS, and ORNL. This research (in part) used computational resources from Cygnus provided through the Multidisciplinary Cooperative Research Program at the Center for Computational Sciences, University of Tsukuba, titled "HPC application and system software development on FPGA-GPU combined platform" and "Development of parallel language XcalableMP 2.0 with high performance portability".

References

1. June 2019 — TOP500 Supercomputer Sites, <`https://www.top500.org/lists/2019/06/`>
2. Kobayashi, R., Fujita, N., Yamaguchi, Y., Nakamichi, A., & Boku, T., "GPU-FPGA Heterogeneous Computing with OpenCL-Enabled Direct Memory Access", 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2019.
3. homepage — OpenACC, <`https://www.openacc.org`>
4. Lee, S., Lambert, J., Kim, J., Vetter, J. S., & Malony, A. D., "OpenACC to FPGA: A Directive-Based High-Level Programming Framework for High-Performance Reconfigurable Computing", CS18, (2018).

5. Tsuruta, C., Miki, Y., Kuhara, T., Amano, H., & Umemura, M., "Off-loading let generation to peach2: A switching hub for high performance GPU clusters", ACM SIGARCH Computer Architecture News, 43, 4: 3-8 (2016).
6. Hanawa, T., Fujii, H., Fujita, N., Odajima, T., Matsumoto, K., & Boku, T., "Evaluation of FFT for GPU cluster using tightly coupled accelerators architecture", In: Cluster Computing (CLUSTER), 2015 IEEE International Conference on. IEEE, p. 635-641 (2015).
7. Fujita, N., Kobayashi, R., Boku, T., Oobata, Y., Yamaguchi Y., Yoshikawa, K., Abe, M., Umemura, M., "Accelerating Space Radiative Transfer on FPGA using OpenCL", Proc. of HEART2018 (Int. Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies), Toronto (2018).
8. Tsuruta, C., Kaneda, T., Nishikawa, N., & Amano, H., "Accelerator-in-switch: A framework for tightly coupled switching hub and an accelerator with FPGA", 2017 27th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2017.
9. Kobayashi, R., Fujita, N., Yamaguchi Y., Boku, T., Yoshikawa, K., Abe, M., Umemura, M., "Accelerating Radiative Transfer Simulation withGPU-FPGA Cooperative Computation", Proc. of ASAP2020, (2020).
10. PGI Compilers & Tools, <`https://www.pgroup.com/`>
11. Intel:Intel FPGA SDK for OpenCL, <`https://www.intel.co.jp/content/www/jp/ja/software/programmable/sdk-for-opencl/overview.html`>
12. Omni Compiler, <`http://omni-compiler.org/`>
13. Nakao, M., Murai, H., Boku, T., Sato, M., "Performance Evaluation for Omni XcalableMP Compiler on Many-core Cluster System based on Knights Landing", IXPUG Workshop Asia 2018, Tokyo, Japan (2018).
14. Nakao M., Odajima, T., Murai, H., Tabuchi, A., Fujita, N., Hanawa, T., Boku, T., Sato, M., "Evaluation of XcalableACC with Tightly Coupled Accelerators/InfiniBand Hybrid Communication on Accelerated Cluster", International Journal of High Performance Computing Applications (2019).
15. Akihiro Tabuchi, Yasuyuki Kimura, Sunao Torii, Video Matsufuru, Tadashi Ishikawa, Taisuke Boku, Mitsuhisa Sato, "Design and Preliminary Evaluation of Omni OpenACC Compiler for Massive MIMD Processor PEZY-SC", Proc. of IWOMP2016 (International Workshop on OpenMP (LNCS 9903: OpenMP: Memory, Devices, and Tasks), pp.293-305, Nara, Oct. 2016.
16. Matrix BCSSTK17 <`https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc2/bcsstk17.html`>
17. Supercomputers - Center for Computational Sciences, <`https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#Cygnus`>
18. 520N - BittWare FPGA Acceleration, <`https://www.bittware.com/fpga/520n/`>

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

Restoration of Legacy Parallelism in C and C++ Applications

Vladimir Janjic · Christopher Brown ·
Adam D. Barwell ·

Abstract *Parallel patterns* are a high-level programming paradigm that enables non-experts in parallelism to develop structured parallel programs that are maintainable, adaptive, and portable whilst achieving good performance on a variety of parallel systems. However, there still exists a large base of *legacy-parallel code* developed using ad-hoc methods and incorporating low-level parallel/concurrency libraries such as *pthreads* without any parallel patterns in the fundamental design. This code would benefit from being restructured and rewritten into pattern-based code. However, the process of rewriting the code is laborious and error-prone, due to typical concurrency and pthreading code being closely intertwined throughout the business logic of the program. In this paper, we present a new software restoration methodology, to transform legacy-parallel programs implemented using e.g. *pthreads* into structured patterned equivalents. We demonstrate our restoration technique on a number of benchmarks, allowing the introduction of patterned parallelism in the resulting code; we record improvements in cyclomatic complexity and speedups.

Keywords Parallel patterns, restoration, pthreads, program transformation, code analysis

1 Introduction

Parallel patterns are a well-established high-level parallel programming model for producing portable, maintainable, adaptive, and efficient parallel code. They have been endorsed by some of the biggest IT companies, such as Intel and Microsoft, who have developed their own parallel pattern libraries (Intel

V. Janjic
School of Science and Engineering, University of Dundee, UK.
E-mail: vjanjic001@dundee.ac.uk

C. Brown, A. Barwell
School of Computer Science, University of St Andrews, UK.
E-mail: cmb21,adb23@st-andrews.ac.uk

TBB [1], Microsoft PPL, etc.) A standard way to use these libraries is to start with a sequential code base, identifying in it the portions of code that are amenable to parallelisation, together with the exact parallel pattern to be applied. Then instantiating the identified pattern at the identified location in the code, after possibly restructuring the code to accommodate the parallelism.

Sequential code gives the cleanest starting point for introduction of parallel patterns. There exists, however, a large base of *legacy* code that was parallelised using lower-level, mostly ad-hoc parallelisation methods and libraries, such as *pthreads* [10]. This code is usually very hard to read and understand, is tailored to a specific parallelisation, and optimised for a specific architecture, effectively preventing alternative (and possibly better) parallelisations and limiting portability and adaptivity of the code. An even bigger problem, from the software engineering perspective, is the maintainability of the legacy-parallel code: commonly, the programmer who wrote it is the only one who can understand and maintain the code. This is due to both complexity of low-level threading libraries and the need for custom-built data structures, synchronisation mechanisms, and sometimes even thread/task scheduling implemented in the code. The benefits of using parallel patterns lie in a clear separation between sequential and parallel parts of the code and a high-level description of the underlying parallelism, making the patterned applications much easier to maintain, change, and adapt to new architectures. Common examples include *farms* and *pipelines*. In a farm, a single computational worker is applied to a set of independent inputs. The parallelism arises from applying the worker to different input elements in parallel. In a parallel pipeline, a sequence of functions, $f_1, f_2, ..., f_m$ are applied to a stream of independent inputs, $x_1, ..., x_n$ where the output of f_i becomes the input to f_{i+1}; the parallelism arises from executing $f_{i+1}(f_i(...f_1(x_k)...))$ in parallel with $f_i(f_{i-1}(...f_1(x_{k+1})...))$.

In this paper, we present a new methodology for the restoration of legacy-parallel code into an equivalent *patterned* form, through application of a number of identified program transformations; the ultimate goal of which is to provide a semi-automatic way of converting legacy-parallel code into an equivalent patterned code, therefore increasing its maintainability, adaptivity, and portability whilst either improving or maintaining performance. This paper makes the following specific research contributions:

1. we present a novel software restoration methodology for converting legacy-parallel applications into their structured (patterned) parallel equivalents;
2. we present a new set of *restoration* transformations that attempt to *systematically*, *i*) *eliminate* pthread operations from legacy C/C++ programs; *ii*) perform *code repair*, fixing any bugs introduced in *i*; and, *iii*) *reshape* code in preparation for parallel pattern introduction;
3. we evaluate these transformations on a set of benchmarks, demonstrating that removal of parallelism can allow us to manually derive structured parallel code that is comparable to the original legacy-parallel version in terms of performance, while being more portable, adaptive, and maintainable.

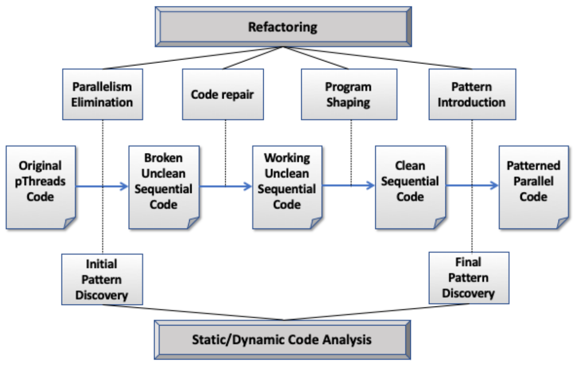


Fig. 1: Software Restoration Process

2 Software Restoration

In this section, we propose a new *Software Restoration* methodology for improving the structure of legacy-parallel C++ code by applying a series of incremental program analysis and transformation steps to rewrite the code into its patterned equivalent. Software restoration is based on program transformation and code analysis and aims to:

1. *discover* the instances of common patterns in legacy-parallel code;
2. *eliminate undesirable* legacy parallel primitives from the same code; and
3. *replace* the removed parallel primitives with instances of parallel patterns.

The input to the Software Restoration process is a legacy-parallel C/C++ code that is based on some low-level parallelism library, such as pthreads, and the output is the semantically-equivalent code based on parallel patterns. In this way, we obtain well-structured code based on a higher level of parallel abstraction, which is significantly more maintainable and adaptive while still preserving good performance of the original, highly-tuned parallel version. In this paper, we will focus on the TBB library as our target code.

The Software Restoration methodology consists of a number of steps, each applying a class of code transformations, some of which are driven by the pattern discovery code analysis. The whole process is depicted in Figure 1. In the below description, we will focus on the code transformation steps. We will use a synthetic, but representative, parallel pipeline as a running example in order to demonstrate the transformation. Listing 1 presents aspects of the original parallel code with pthreads that are pertinent to this demonstration.

Listing 1: Original Simple Pipeline Code

```
int main(int argc, char *argv[]) {
  ...
  // create the workers, then wait for them to finish
  pthread_create(&workerid[0], &attr, Stage1, (void *)&stage_queues[0]);
  pthread_create(&workerid[1], &attr, Stage2, (void *)&stage_queues[1]);
```

```
 6    pthread_create(&workerid[2], &attr, Stage3, (void *)&stage_queues[2]);
 7
 8    for (i = 0; i < NRSTAGES; i++)
 9      pthread_join(workerid[i], NULL);
10
11    ...
12  }
13
14  // Second stage reads an element from the input queue, adds 1 to it,
15  // and writes it to the output queue.
16  void *Stage2(void *arg) {
17    int my_input, my_output;
18
19    pipeline_stage_queues_t *myQueues = (pipeline_stage_queues_t *)arg;
20    queue_t *myOutputQueue = myQueues->outputQueue;
21    queue_t *myInputQueue = myQueues->inputQueue;
22
23    do {
24      my_input = read_from_queue(myInputQueue);
25      if (my_input > 0)
26        my_output = my_input + 1;
27      else // 0 is a terminating token. Pass on if received.
28        my_output = 0;
29      add_to_queue(myOutputQueue, my_output);
30    } while (my_input>0);
31
32    return NULL;
33  }
34
35  void add_to_queue(queue_t *queue, int elem)
36  {
37    pthread_mutex_lock(&queue->queue_lock);
38    // If the queue is full, wait until something reads from it before adding a new element
39    if (queue->nr_elements == queue->capacity)
40      pthread_cond_wait(&queue->queue_cond_read,&queue->queue_lock);
41    queue->elements[queue->addTo] = elem;
42    queue->addTo = (queue->addTo + 1) % queue->capacity;
43    queue->nr_elements++;
44    pthread_cond_signal(&queue->queue_cond_write);
45    pthread_mutex_unlock(&queue->queue_lock);
46  }
```

In the above `main` function (Lines 1–12), a pipeline of three stages is created using three threads. The stages are connected by queues such that the first stage has an output queue, and stages two and three have both an input and an output queue. After creation, the `main` function waits for the threads to finish their work (Lines 8–9) before continuing. In Lines 14–32, we show the function for the middle stage of the pipeline, which reads an integer from the input queue, increments it by one, then puts it into the output queue. The first and third stages have a similar structure, where the first stage acts as a source of integers for the second stage, and the third stage doubles its inputs before adding them to the final output queue.

All the relevant synchronisation code for the queues can be found in two functions: `add_to_queue` and `read_from_queue`. Only `add_to_queue` (Lines 35–46) is shown here, since `read_from_queue` is similar. Both functions use one mutex lock and two conditional variables. The conditional variables are used for synchronisation when threads are waiting to insert an element into a full queue or for reading from an empty queue (e.g. at the start of the program). When a thread needs to add to the queue, it first acquires the queue lock and checks if the queue is full (Lines 39–40). When the queue is full, the thread releases the lock and waits for a signal that some other thread has consumed an element of this queue (`queue->queue_cond_read` conditional variable at line 40). After this conditional variable is signalled, the thread adds the element to the queue, updating the queue counter and pointer in the process (Lines 42–44). Finally, the thread signals that an element has been added to the queue (`queue->queue_cond_write` conditional variable in Line 44) and releases the queue lock (Line 45) before returning.

Parallelism Elimination. The initial code analysis step, *Initial Pattern Discovery*, analyses the original pthreaded code and discovers those parts of it, if any, that correspond to instances of parallel patterns. In our example, this stage identifies the pipeline created in Lines 4–6, with the pipeline stages being the functions: `Stage1`, `Stage2`, and `Stage3`. Following pattern discovery, the first code transformation step is applied, where pthread operations and primitives are either removed or transformed so as to eliminate parallelism. In Listing 1, this impacts the `main` and `add_to_queue` functions; Listing 2 gives the results of parallelism elimination on both functions.

Listing 2: Simple Pipeline Code with Parallelism Removed

```c
int main(int argc, char *argv[]) {
  ...
  // Calls to pthread_create are converted to function calls.
  Stage1((void *)&stage_queues[0]);
  Stage2((void *)&stage_queues[1]);
  Stage3((void *)&stage_queues[2]);

  // The loop containing pthread_join is removed.
  ...
}

void add_to_queue(queue_t *queue, int elem) {
  // All mutex and conditional variable operations are removed.
  queue->elements[queue->addTo] = elem;
  queue->addTo = (queue->addTo + 1) % queue->capacity;
  queue->nr_elements++;
}
```

Whilst all pthread operations have been removed or transformed, and the program is now sequential, the Parallelism Elimination stage *does not guarantee that a program's semantics are preserved.* Accordingly, as in our running example, errors may be introduced. Here, `Stage1` contains a `do`-loop that adds items to its output queue. Since the second stage, which reads from that queue, is no longer consuming those elements concurrently, and the queue is smaller than the total number of elements produced, the second stage will now consume and process only a subset of its inputs in the original pthreaded version after `Stage1` returns. Ultimately, the semantics and output of the program produced by the Parallelism Elimination stage is not the same as the original pthreaded program; the code must therefore be *repaired.*

Code Repair. As observed in the previous step, Parallelism Elimination might result in code that is broken and, hence, not semantically equivalent to the original legacy-parallel code. Our example is just one of many instances in which merely removing pthread constructs actually breaks the code (see Section 4 for more examples). The next step in Software Restoration is, therefore, to repair the potentially broken code produced by Parallelism Elimination. Due to the potential complexity of this repair stage, multiple transformations may need to be applied.

In order to repair the broken pipeline in our running example it is necessary to stop the first stage from overflowing its output queue. This can be achieved by merging the loops found in `Stage1`, `Stage2`, and `Stage3`, thereby resulting in loop where the operations in stages two and three are applied to each integer produced by stage one in the same iteration that produces it. The result of this process can be found below in Listing 3.

Listing 3: Simple Pipeline Code after Code Repair

```c
void Pipe(void* a1, void* a2, void* a3) {
  // STAGE ONE
  int my_output_1, i_1 = MAXDATA;

  pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
  queue_t *myOutputQueue_1 = myQueues_1->outputQueue;

  // STAGE TWO
  int my_input_2, my_output_2;
  ...
  do {
    // STAGE ONE
    if (i_1 >= -1) { ... }

    // STAGE TWO
    my_input_2 = read_from_queue(myInputQueue_2);
    if (my_input_2 >= 0) {
      if (my_input_2 > 0)
        my_output_2 = my_input_2 + 1;
      else
        my_output_2 = 0;
      add_to_queue(myOutputQueue_2, my_output_2);
    }

    // STAGE THREE
    my_input_3 = read_from_queue(myInputQueue_3);
    if (my_input_3 >= 0) { ... }
  } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
}
```

Here, the calls to `Stage1`, `Stage2`, and `Stage3` in `main` are first *lifted* into a new function, `Pipe`. Each of those calls are then *unfolded* in order to expose the `do`-loops that they contain. These loops are then *merged*, allowing all three stages to be executed within a single iteration. This avoids the first stage overflowing its output queue, and consequently, results in a program that is sequential but semantically equivalent to the original pthreaded program.

Program Shaping. Despite correcting those errors introduced during Parallelism Removal, the code produced by the Code Repair stage may still contain artefacts from the original legacy parallelisation. In our running example, such artefacts include the queues between the stages. In other examples artefacts can include custom-built representations of flat data structures, such as arrays, perhaps introduced for chunking purposes. These artefacts are redundant and could hinder alternative (and possibly better) parallelisations of the code. The next step is, therefore, to eliminate residual artefacts of legacy parallelism, and to improve structure where such improvements make the code more amenable to the introduction of patterned parallelism. As in Code Repair, due to the potential complexity of this task, multiple transformations may need to be applied. Each Program Shaping refactoring results in a program that is semantically equivalent to the one it transforms. In our running example, we remove the now redundant queues in between the stages, the result of which can be found in Listing 4.

Listing 4: Clean Sequential Simple Pipeline Code

```
 1  struct PipeStruct {
 2    // Input to pipeline
 3    int* i_1;
 4    // Output of pipeline
 5    queue_t* myOutputQueue_3;
 6    // Inter-stage temporary variables, used in loop-condition.
 7    int my_output_1;
 8    int my_output_2;
 9  };
10
11  PipeStruct S1(PipeStruct arg) { ... }
12
13  PipeStruct S2(PipeStruct arg) {
14    if (arg.my_output_1 >= 0) {
15      if (arg.my_output_1 > 0) arg.my_output_2 = arg.my_output_1 + 1;
16      else arg.my_output_2 = 0;
17    }
18    return arg;
19  }
20
21  PipeStruct S3(PipeStruct arg) { ... }
22
23  void Pipe(void* a1, void* a2, void* a3) {
24    // STAGE ONE
25    int my_output_1, i_1 = MAXDATA;
26    ...
27    do {
28      PipeStruct arg = PipeStruct {&i_1, myOutputQueue_3, &my_output_1, &my_output_2};
29      PipeStruct r = S3(S2(S1(arg)));
30      my_output_1 = r.my_output_1;
31      my_output_2 = r.my_output_2;
32    } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
33  }
```

Here, calls to `add_to_queue` and `read_from_queue` are first *unfolded*, allowing the specific read and write statements that represent the passing of data between stages to be matched and ultimately simplified to a single assignment statement between stages. The stages are then *lifted* into the functions `S1` (Line 11), `S2` (Lines 13–19), and `S3` (Line 21), and a `struct` (Lines 1–9) generated to ensure that each stage is a single-input single-output function. The function composition on Line 32 is now in a form where pattern-based parallelism can be simply introduced, perhaps again by refactoring.

Pattern Introduction. After the final pattern discovery analysis is performed and the final patterns to be introduced are identified, together with the locations in the code where this will be done, the final step is to introduce instances of parallel patterns into the now-clean sequential code. The parts of the sequential code are replaced by calls to the functions from the high-level pattern libraries such as *Intel TBB* [1] or *OpenMP* [14]. This results in final, patterned parallel code that is semantically equivalent to the starting legacy-parallel code, but with much cleaner structure and simpler, higher-level code that allows easier maintainability, adaptivity and portability.

3 Restoration Transformations

We propose a series of program transformations to facilitate the restoration of C programs that have been previously parallelised using low-level pthread parallelism techniques. The following transformations are grouped according to the stages in Section 2 in which they are used. In addition to the following, standard transformations may also facilitate the restoration process. For instance, the transformation to *unfold* a function definition [9] is used in both Code Repair and Shaping stages; e.g. in the former, it allows loops to be merged, and in the latter, it allows the elimination of intermediate queues. The *extract method* [17] transformation can be similarly used to lift a pipeline into a self-contained function, or to lift its individual stages (back) into separate functions.

3.1 Parallelism Elimination

Parallelism Elimination comprises a single composite transformation that either removes or transforms pthread operations. As noted in Section 2, Parallelism Elimination, and by extension this transformation, does *not* guarantee that the result of the transformation will be semantically equivalent to the transformed program. Parallelism Elimination effects the following transformations.

- Removes `#include <pthread>`.
- Removes all pthread operations aside from calls to both `pthread_join` and `pthread_create`.
- Removes all variable declarations whose types are defined as part of the pthread library, excepting `pthread_t` declarations.
- Declarations in the form `pthread_t t;` are transformed into `void* t;`.
- Calls to `pthread_create` of the form,

```
pthread_create(t,a,f,x)
```

  are transformed into the form:

```
t = f(x);
```

  Recall that Parallelism Elimination converts the type of `pthread_t` variables to `void*` variables of the same name(s), and that `pthread_create` requires that `f` returns a value of type `void*`.

- Calls to `pthread_join` are transformed according to whether the second argument is `NULL`. When the second argument is *not* `NULL`, e.g. `pthread_join (t,x)`, the join operation is transformed into the form `x = t`. Otherwise, when the second argument is `NULL`, the call to `pthread_create` is removed.
- In cases where a call to `pthread_join` or `pthread_create` forms the right-hand-side of an assignment statement, e.g.

```
r = pthread_join(t,x);
```

  in addition to the transformation of the pthread operation, an assignment statement is inserted where the variable being assigned, `r` in the above example, is assigned the value of a successful call to the original pthread operation, here `pthread_join` and `0`. In the above example, the code resulting from the transformation is:

```
r = 0;
x = t;
```

- Any `for`-loop whose body contains no statements following the removal of a pthread operation will itself be removed.
- Any `if`-statement with a branch whose body contains no statements following the removal of a pthread operation will be transformed to have only the other branch, or itself removed, if no such branch exists. For instance, given the `for`-loop from the synthetic pipeline example in Listing 1,

```
8  for (i = 0; i < NRSTAGES; i++)
9    pthread_join(workerid[i], NULL);
```

  since the second argument to `pthread_join` is `NULL`, the join operation result is itself a statement, and the body of the `for`-loop contains no other operations, this `for`-loop is removed.

3.2 Code Repair

In addition to *unfolding* and *extract method* refactorings, the merging of loops is a key transformation of the Code Repair stage when restoring pipelines. In order to avoid the overheads involved with thread creation, individual stages of a pipeline may loop until a termination token or condition is met. Merging the loops across pipeline stages from which pthreads have been eliminated using the transformations in Section 3.1 can be necessary to avoid overflowing any buffers or queues in between pipeline stages, thus restoring the original semantic behaviour of the code. Here, we describe only the merging of `do`-loops, but a similar approach can be used to merge, e.g., `for`-loops.

Merge `do`-loops. A sequence of n `do`-loops, in the same compound statement can be merged such that the result is a single loop containing the bodies of the original loops in the same order that they appeared in the original source code. We note that any statements that appear in between loops in the original code, must be commutative with respect to any preceding loops; i.e. it must be possible to swap the ordering of the statements and preceding loops without changing the behaviour of the program.

To illustrate this transformation we use the below code, derived from the synthetic simple pipeline example in Section 2.

Listing 5: Intermediate Code Repair Stage for Simple Pipeline Example

```
void Pipe(void* a1, void* a2, void* a3) {
  // STAGE ONE
  int my_output_1, i_1 = MAXDATA;
  ...
  do {
    ...
  } while(i_1>=0);

  // STAGE TWO
  int my_input_2, my_output_2;
  ...
  do {
    my_input_2 = read_from_queue(myInputQueue_2);
    ...
  } while (my_input_2>0);

  // STAGE THREE
  int my_input_3, my_output_3;
  ...
  do {
    my_input_3 = read_from_queue(myInputQueue_3);
    ...
  } while (my_input_3>0);
}
```

This represents the example following the Parallelism Elimination stage (Listing 2), and where the calls to `Stage1`, `Stage2`, and `Stage3` have been lifted into the function `Pipe` using *extract method* and then *unfolded*. Since the statements in between the above loops consist solely of declarations and assignment statements and can be safely executed prior to the first and second loops, it is possible to merge these loops.

Listing 6: Following Merging of loops in Listing 5)

```
void Pipe(void* a1, void* a2, void* a3) {
  int my_output_1, i_1 = MAXDATA;
  ...
  do {
    // STAGE ONE
    if (i_1 >= -1) {
      ...
    }

    // STAGE TWO
    my_input_2 = read_from_queue(myInputQueue_2);
    if (my_input_2 >= 0) {
      ...
    }

    // STAGE THREE
    my_input_3 = read_from_queue(myInputQueue_3);
    if (my_input_3 >= 0) {
      ...
    }
  } while (i_1 >=0 || my_input_2 > 0 || my_input_3 > 0);
}
```

The bounding condition of the merged loop is formed of the disjunction of the conditions of the original loops. Similarly, the body of the merged loop comprises the bodies of the original loops wrapped in `if`-statements. The condition of one of these `if`-statements is the *weakened* condition of the respective original `do`-loop; e.g. the condition `my_input_2>0` above is weakened to `my_input_2>=0`. This weakening is necessary, since the body of a `do`-loop is executed before the bounding condition is checked. Moreover, because the loop body is guaranteed to execute once, it is possible that a variable used in the bounding condition may be declared outside of the loop, but only initialised within it. For example, in the second and third stages above, neither `my_input_2` nor `my_input_3` are initialised before the assignment *inside* the body of their respective loops (Lines 15 & 23, Listing 5). In order to merge these loops such that the second and third stages will execute, we move the aforementioned assignment statements outside of the introduced `if`-statement around the loop body (Lines 12 & 18, Listing 6). Such assignment statements can only be lifted out of the body if they themselves depend upon variables already assigned outside of the loop. Variables are renamed in the bodies of the loops as necessary.

3.3 Program Shaping.

Program Shaping represents the broadest stage in the process and presents the programmer with the largest range of choices in terms of transformations that may be effected. In addition to *unfolding* definitions and creating new functions via *extract method*, other standard transformations may be applied, e.g. *dead-code elimination* [23], in order to improve or simplify the structure of the code. In order to remove aspects of the code that represent optimisations enacted for the legacy parallelisation, both existing and novel transformations may be necessary. Novel transformations may include the unchunking of data, the removal intermediate, and now redundant, queues between stages, and a tupling (and potential localisation) of arguments to present transformations that introduce algebraic skeletons with a simple composition of single-parameter functions. In line with our running example, we propose transformations to remove intermediate queues, and to merge arguments.

Remove Intermediate Queues. In a pthreaded pipeline, passing the result of a stage to the next can involve intermediary queues. Once the parallelism from the pipeline has been eliminated and the code repaired, so too can these queues be removed. We remove these intermediate queues by inspecting, matching, and transforming *read*, *write*, and *update* operations pertaining to those queues. In our recurring example we begin this process following the Code Repair stage, and having *unfolded* `add_to_queue` and `read_from_queue` operations for *intermediate queues only*; note that the output queue operation on Line 35 has not been unfolded.

```
1 void Pipe(void* a1, void* a2, void* a3) {
 2    int my_output_1, i_1 = MAXDATA;
 3
 4    pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
 5    queue_t *myOutputQueue_1 = myQueues_1->outputQueue;
 6    ...
 7    do {
 8      // STAGE ONE
 9      if (i_1 >= 0) {
10        ...
11        myOutputQueue_1->elements[myOutputQueue_1->addTo] = my_output_1;
12        myOutputQueue_1->addTo = (myOutputQueue_1->addTo + 1) % myOutputQueue_1->capacity;
13        myOutputQueue_1->nr_elements++;
14      }
15
16      // STAGE TWO
17      my_input_2 = myInputQueue_2->elements[myInputQueue_2->readFrom];
18      myInputQueue_2->nr_elements--;
19      myInputQueue_2->readFrom = (myInputQueue_2->readFrom + 1) % myInputQueue_2->capacity;
20
21      if (my_input_2 >= 0) {
22        ...
23        myOutputQueue_2->elements[myOutputQueue_2->addTo] = my_output_2;
24        myOutputQueue_2->addTo = (myOutputQueue_2->addTo + 1) % myOutputQueue_2->capacity;
25        myOutputQueue_2->nr_elements++;
26      }
27
28      // STAGE THREE
29      my_input_3 = myInputQueue_3->elements[myInputQueue_3->readFrom];
30      myInputQueue_3->nr_elements--;
31      myInputQueue_3->readFrom = (myInputQueue_3->readFrom + 1) % myInputQueue_3->capacity;
32
33      if (my_input_3 >= 0) {
34        ...
35        add_to_queue(myOutputQueue_3, my_output_3);
36      }
37    } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
38  }
```

A variable is *read* from when that variable occurs in a statement and that variable is not being *updated*; e.g. `capacity` on Line 12 above. Similarly, a variable undergoes a *write* when it is being assigned to and is not being *updated*; e.g. `elements` in the first output queue is written to on Line 13. Finally, a variable is *updated* when it occurs in a statement that is both *reading* from and *writing* to that variable; e.g. `addTo` in Line 12 above. Basic increment operators, e.g. `nr_elements++` on Line 13, are similarly considered *updates* due to their semantics. In order to transform these *read*, *write*, and *update* operations, we pair the operations in the order that they appear in the code and according to the variables they *read*, *write*, or *update*, and transform those pairs according to their composition. If two queues are semantically the same but referred to by different variables then they themselves will be considered the same during pairing; e.g. `myOutputQueue_1` and `myInputQueue_2` refer to the same intermediate queue, thus `myOutputQueue_1->elements` and `myInputQueue_2 ->elements` are similarly considered to be the same variable for pairing. In the above example, two cases arise:

1. *Updates* to variables that do not occur elsewhere in the code pertain to queue housekeeping operations are therefore removed. In the above code, Lines 12, 13, 18, 19, 24, 25, 30, and 31 are all removed.

2. A *write* followed by a *read* is merged into a single assignment statement s.t. the RHS of the *read* is replaced with the RHS of the *write*, and where the original *write* statement is removed. For example, in the above code, the *write* to `elements` on Line 11 and the *read* from `elements` on Line 17 can be paired (due in part to the behaviour of the queue reading the element that has just been added). Since this represents passing `my_output_1` on Line 11 to `my_input_2` on Line 17, it is possible to remove Line 17 and transform Line 11 into the form `my_input_2 = my_output_1`.

An unpaired *read* that is part of an *update*, e.g. `capacity` on Line 12, or a paired *write*, e.g. `addTo` on Line 11, is removed or otherwise transformed along with the *update* or paired *write* statement. Similarly, an unpaired *read* that is part of a *paired read* statement, e.g. `readFrom` on Line 17, is also transformed according to the paired *read* statement. When applied, the above transformations result in the removal of the two intermediate queues.

```
void Pipe(void* a1, void* a2, void* a3) {
  int my_output_1, i_1 = MAXDATA;

  pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
  queue_t *myOutputQueue_1 = myQueues_1->outputQueue;
  ...
  do {
    // STAGE ONE
    if (i_1 >= 0) {
      ...
      my_input_2 = my_output_1;
    }
    // STAGE TWO
    if (my_input_2 >= 0) {
      ...
      my_input_3 = my_output_2;
    }
    // STAGE THREE
    if (my_input_3 >= 0) {
      ...
      add_to_queue(myOutputQueue_3, my_output_3);
    }
  } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
}
```

Merge Arguments. During restoration, the stages of a pipeline may each be represented by a single function. Since the majority of patterned pipeline implementations expect pipeline stages to take a single argument, i.e. usually the result of the preceding stage, it may be necessary to tuple the parameters of each stage. This can be done (semi-)automatically. A `struct` can be synthesised across each of the functions representing stages in the pipeline, with each function transformed to both return and take the synthesised `struct` as its argument. In our running pipeline example, following the removal of the intermediate queues, we have three stages represented by the functions `S1`, `S2`, and `S3`, respectively. `S1` takes a pointer to an integer and returns an integer value; `S2` takes and returns an integer value; and `S3` takes an integer value and a pointer to its output queue and returns nothing.

```
int S1(int* i_1) {
  int my_output_1;
  if (*i_1 >= 0) {
    my_output_1 = *i_1;
    *i_1 = *i_1-1;
  }
  return my_output_1;
}

int S2(int my_output_1) {
  ...
  return my_output_2
}

void S3(int my_output_2, queue_t* myOutputQueue_3) {
  ...
}

void Pipe(void* a1, void* a2, void* a3) {
  ...
  do {
    my_output_1 = S1(&i_1);
    my_output_2 = S2(my_output_1);
    S3(my_output_2, myOutputQueue_3);
  } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
}
```

Here, we observe that the result of the first two stages are used in the merged loop bounding condition and must therefore be propagated through the stages when converted to composition form. This is achieved by synthesising a new `struct`, `PipeStruct`, that contains all those variables used in the condition statement, and both the input and output to the pipeline.

```
struct PipeStruct {
  // INPUTS
  int* i_1;
  // OUTPUTS
  queue_t* myOutputQueue_3;
  // USED IN LOOP CONDITION
  int my_output_1;
  int my_output_2;
};

PipeStruct S1(PipeStruct arg) {
  ...
  return arg;
}

PipeStruct S2(PipeStruct arg) {
  ...
  return arg;
}

PipeStruct S3(PipeStruct arg) {
  ...
  return arg;
}

void Pipe(void* a1, void* a2, void* a3) {
  ...
  do {
    PipeStruct arg = PipeStruct {&i_1, myOutputQueue_3};
    PipeStruct r = S3(S2(S1(arg)));
    my_output_1 = r.my_output_1;
    my_output_2 = r.my_output_2;
  } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
}
```

Here, in order to introduce a TBB pipeline, we keep the input and output to the pipeline as pointers. The variables my_output_1 and my_output_2 are only used as part of the condition and are not outputs of the pipeline, as such they are stored by their value. In this particular example, it is possible to remove the second and third disjunctions and therefore remove my_ouptut_1 and my_ouptut_2 from PipeStruct, and thus further simplify the pipeline, but this is left to another Shaping transformation or sequence of transformations.

4 Evaluation

In this section, we present an evaluation of our restoration methodology on a number of examples of pthreaded C and C++ applications taken from a variety of domains, including image convolution, nqueens, cholesky decomposition, blackscholes, pgpry, mandelbrot and matrix multiplication. For each benchmark we evaluate the effectiveness of our technique using standard metrics, such as McCabe's Cyclomatic Complexity [26], lines of code and difference in runtimes between the original pthread version and the restored TBB version, using the maximum number of available cores; these results are summarised in Table 1, which also labels if each benchmark is a standard task from implementation (F) or a pipeline, where each stage can also be farmed (P). All of our execution experiments are conducted on a server with Intel Xeon E5-2690 CPU with 28 cores, running at 2.6 GHz with 256 GB of RAM, with the Scientific Linux 6.2 operating system.

4.1 Image convolution

Image Convolution is a technique widely used in image processing applications for blurring, smoothing and edge detection. We consider an instance of the image convolution from video processing applications, where we are given a list of images that are rocessed by applying a filter. Applying a filter to an image consists of computing a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

$$out(i,j) = \sum_m \sum_n in(i-n, j-m) \times filt(n,m) \tag{1}$$

Listing 7: Original Convolution with PThreads

```
void add_to_queue(queue_t *queue, task_t elem)
{
  /* Same as in Listing 1 */
}

task_t read_from_queue(queue_t *queue)
{
  ...
}

```

```
11  void* stage1() {
12    ..
13    while(1) {
14      t = read_from_queue(tq1);
15      r = workerStage1(t); /* Reads in pixels from a file into an array */
16      add_to_queue(tq2, r);
17    }
18    return NULL;
19  }
20
21  void* stage2() {
22    ..
23    while(1) {
24      t = read_from_queue(tq2);
25      r = workerStage2(t); /* Applies transformation to each pixel in a received array */
26      add_to_queue(tq3, r);
27    }
28    return NULL;
29  }
30
31  int main (int argc, char **argv)
32  {
33    ...
34    /* Reading in the images in the task queue tq1 */
35    ...
36    /* Create the pipeline */
37    for (int i=0; i<nw1; i++)
38      pthread_create(&workers1[i], NULL, stage1, NULL);
39    for (int i=0; i<nw2; i++)
40      pthread_create(&workers2[i], NULL, stage2, NULL);
41    ...
42    /* Wait for threads to finish execution and output results to files */
43  }
```

For the convolution example, we start off with a pthreaded version in Listing 7, with a similar structure as the other pipelined examples in this paper, and outlined in Section 2. After setting up the task queue for the first stage of the pipeline (e.g. by reading a list of names of files with images), the example creates the pipeline in Lines 37–40, spawning a number of worker threads for each stage of the pipeline. The pipeline stages are shown at Lines 11 and 21, respectively; each stage has a similar structure: a non-terminating `while` loop that retrieves a task from the stage's input queue (`tq1` and `tq2` for `stage1` and `stage2`, respectively), computes the unit of work on the task item (Lines 15 and 25) and then places the result on an output queue (Lines 16 and 26). Functions `add_to_queue` and `read_from_queue` put a task in an output queue and read a task from an input queue, respectivelly, in a thread safe manner. The code for `add_to_queue` was shown in Listing 1.

The *first step* to restoration is to remove the threading code; this is a fairly straightforward process, but results in an executable that no-longer terminates. This is due to the fact that there is no termination condition of the `while` loops within the stages. A simple repair for this step is to add a termination token, `EOS`, which threads through the pipeline computation when no more tasks are on the original input queue amd terminates the stages when received (Listing 8).

Listing 8: Convolution, Repaired with a Termination Token

```
  if ((int)(task_t)t == EOS) {
    puttask(tq2, (task_t2 *)EOS);
    break;
  }
```

The next step is to perform *program shaping* which goes through various steps, including *unfolding* the various calls to `gettask` and `puttask` in the stages, *merging* the stages together, and finally *removing* the intermediate queue between the two stages (leaving the input and output queue; see Listing 9).

Listing 9: Stages merged, unfolded and intermediate queue removed

```
/* Unfolded gettask function, reutrning t1 as an input task to stage 1 */
. . .
r1 = workerStage1(t1);
r2 = workerStage2(r1);
/* Unfolded puttask function that puts r2 into queue tq2 */
tq2->elements[tq2->addTo] = r2;
tq2->addTo = (tq2->addTo + 1) % tq2->capacity;
tq2->nr_elements ++
```

The final step in the shaping process is to arrive at the code shown in Listing 10, where we remove the input and output queues completely, and transform the program into a simple function composition; the function composition has been *unfolded* into the original `for` loop (Line 37–40 from Listing 7), and the loops merged into a single loop.

Listing 10: Convolution Shaped

```
for (int i=0; i<NIMGS; i++) {
      workerStage2(workerStage1(i));
}
```

Finally, the fully shaped program from Listing 10 can be parallelised using a structured pattern approach. Here we use TBB, to define a pipeline, using C++ classes, as shown in Listing 11.

Listing 11: Convolution Restored with TBB

```
tbb::parallel_pipeline(
      ntoken,tbb::make_filter<void,task_t2*>(tbb::filter::serial, Stage1(NIMGS) )
      &  tbb::make_filter<task_t2*,int>(tbb::filter::parallel, Stage2() )
```

4.2 Discussion

Table 1 shows the summary of our results for all the benchmarks. For all benchmarks we see comparable results in the McCabe metrics, where the TBB version gives a better result, apart from Blackscholes, where the complexity is equal, and Matrix Multiplication, where the complexity actually increases. This is most likely because both of these benchmarks are simple farms, and the TBB logic actually introduces some complexity over simply calling `pthread_create` multiple times. The number of lines of code for the TBB version is mostly comparable, with most benchmarks showing a decrease in lines of code. Blackscholes shows a slight increase in LOC, most likely, again,

Benchmark		McCabe		Lines		Performance	
		Before	After	Before	After	Before	After
Blackscholes	F	29	29	366	393	38.5	39
Matrix Multiplication	F	9	15	176	146	909.4	896.5
Mandelbrot	F	12	11	145	142	2.21	2.28
NQueens	P	41	24	421	337	8.63	8.622
Cholesky Decomposition	P	31	19	321	226	16.97	17.08
PGPry	P	23	19	210	243	138.1	131
Image Convolution	P	71	29	714	280	12.85	5.2

Table 1: Metrics for each benchmark, where F = Farm, and P = Pipeline; performance times are in seconds on a 28-core machine.

due to the slight increase in code logic for TBB versus the pthread version. In terms of performance, again, the TBB versions are mostly comparable, with the exception of a few cases. For convolution, the TBB version performs 2.4x faster, due to the pthreading version introducing extra overheads in the locking code; Blackscholes also performs very slightly worse, by 0.5 seconds.

5 Related Work

The concept of a *systematic*, or *structured* approach to *software restoration* has, to our knowledge, been largely previously unexplored. A concept that is probably most related to *software restoration* is that of *reverse engineering*, which is a technique used to retrieve high-level requirements from existing sequential code [12,13]. Yu et al. [31] proposed a technique that attempts to use refactoring to try and recover requirements goal models from legacy code. However, the work only targets sequential code and only capture high-level information that is not useful for parallel restoration. Refactoring has roots in Burstall and Darlington's fold/unfold system [9], and has been applied to a wide range of applications as an approach to program transformation [27], with refactoring tools a feature of popular IDEs including, *i.a.*, Eclipse [16] and Visual Studio [28]. Previous work on parallelisation *via* refactoring has primarily focussed on the introduction and manipulation of parallel pattern libraries in C++ [8,22] and Erlang [7,6]. Another approach has been the automated introduction of annotations in the form of C++ attributes [30]. Dig proposed an approach to parallel loops in Java [15], but did not use high-level algorithmic skeletons. Aldinucci and Danelutto proposed an approach to convert between skeleton configurations and could be used to introduce parallelism, but where the sequential program must also be defined using (sequential) skeletons [2]. Thompson et al. [24] proposed an approach to refactor sequential Erlang programs into concurrent versions, using program slicing to guide the refactoring process. However, their approach was not focussed on parallel performance, and did not use restoration or parallel patterns. High-level parallel patterns, sometimes known as *algorithmic skeletons* offer high-level abstraction over low-level concurrency methods [4,18]. A range of pattern/skeleton implementations have been developed for a number of programming languages; these include: RPL [22]; Feldspar [5]; FastFlow [3]; Microsoft's Pattern Parallel Library [11]; and Intel's Threading Building Blocks (TBB) library [1]. Since patterns are well-defined, rewrites can be used to automatically explore the space of equivalent patterns, e.g. optimising for performance [25, 20] or generating optimised code as part of a DSL [19]. Moreover, since patterns are architecture-agnostic, patterns have been similarly implemented for multiple architectures [21, 29].

6 Conclusions

In this paper, we have introduced a software restoration methodology for converting legacy-parallel applications into structured parallel code using parallel patterns. This ensures portability, maintainability and adaptivity of parallel code while maintaining, and sometimes even increasing, performance. We also presented transformations to eliminate ad-hoc pthread parallelism from legacy-parallel code, transformations that repair the code from bugs introduced by the elimination step, and , shape the code in order to patternise it. Furthermore, we evaluated out software restoration methodology on a number of realistic benchmarks and use-cases, demonstrating benefit in terms of gained performance, increased adaptivity, portability and maintainability.

References

1. TBB (intel threading building blocks). In: Encyclopedia of Parallel Computing, p. 2029. Springer (2011)
2. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: PDCS, pp. 955–962 (1999)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280 (2017). DOI 10.1002/9781119332015.ch13
4. Asanovic, K., Bodík, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D.A., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.A.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
5. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar - an embedded language for digital signal processing. In: IFL, *Lecture Notes in Computer Science*, vol. 6647, pp. 121–136. Springer (2010)
6. Barwell, A.D., Brown, C., Hammond, K., Turek, W., Byrski, A.: Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in erlang. Computing and Informatics **35**(4), 792–818 (2016)
7. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming **42**(4), 564–582 (2014)
8. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.W.: Agricultural reform: More efficient farming using advanced parallel refactoring tools. In: PDP, pp. 36–43. IEEE Computer Society (2014)
9. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
10. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., USA (1997)

11. Campbell, C., Miller, A.: A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures, 1st edn. Microsoft Press (2011)
12. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3), 215–249 (1998)
13. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: ICSE, pp. 439–448. ACM (2000)
14. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 4655 (1998)
15. Dig, D.: A refactoring approach to parallelism. IEEE Software **28**(1), 17–22 (2011)
16. Foundation, E.: Eclipse - an Open Development Platform (2009). `http://www.eclipse.org`
17. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series. Addison-Wesley (1999)
18. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw., Pract. Exper. **40**(12), 1135–1160 (2010)
19. Gorlatch, S.: Domain-specific optimizations of composed parallel components. In: Domain-Specific Program Generation, *Lecture Notes in Computer Science*, vol. 3016, pp. 274–290. Springer (2003)
20. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: IPPS/SPDP, pp. 492–499. IEEE Computer Society (1999)
21. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: CGO, pp. 100–112. ACM (2018)
22. Janjic, V., Brown, C., Mackenzie, K., Hammond, K., Danelutto, M., Aldinucci, M., García, J.D.: RPL: A domain-specific language for designing and implementing parallel C++ applications. In: PDP, pp. 288–295. IEEE Computer Society (2016)
23. Kennedy, K.: A Survey of Data Flow Analysis Techniques, p. 554 (1981)
24. Li, H., Thompson, S.J.: Safe concurrency introduction through slicing. In: PEPM, pp. 103–113. ACM (2015)
25. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: Euro-Par, *Lecture Notes in Computer Science*, vol. 3149, pp. 644–653. Springer (2004)
26. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976)
27. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. **30**(2), 126–139 (2004)
28. Microsoft: Visual Studio IDE (2019). `https://visualstudio.microsoft.com/vs/`
29. Reyes, R., Lomüller, V.: SYCL: single-source C++ accelerator programming. In: PARCO, *Advances in Parallel Computing*, vol. 27, pp. 673–682. IOS Press (2015)
30. del Rio Astorga, D., Dolz, M.F., Sánchez, L.M., García, J.D., Danelutto, M., Torquati, M.: Finding parallel patterns through static analysis in C++ applications. IJHPCA **32**(6) (2018)
31. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: RE, pp. 363–372. IEEE Computer Society (2005)

13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

Fortress Abstractions in X10 Framework

Anshu S. Anand · Karthik Sayani · R. K. Shyamasundar

Abstract Fortress provides a nice set of abstractions used widely in scientific computing. The use of such abstractions enhances the productivity of programmers/users. Also, in scientific computations, boilerplate code has extensive usage. Keeping this in view, we embed Fortress abstractions in an X10 environment so that we can get better productivity without losing performance. In this paper, we transform Fortress into X10 through a transcompilation system. We describe compilation strategies for a few important constructs and discuss the performance of the generated X10 code with respect to the original Fortress code. The translated X10 code outperforms the original Fortress code with a maximum of 206x speedup achieved in the best case. The system also supports the multiresolution language approach that simplifies parallel programming by allowing domain scientists to write programs in the Fortress syntax that is closer to the mathematical notation. The translated X10 code, which can further be compiled to either C++ or Java, implicitly assures performance and may further be optimized for performance by utilizing the low-level features of X10 (or C++/Java).

Keywords Transpiler · Fortress · X10 · Abstraction · Xtext · Xtend

The authors' names are listed in alphabetical order.

Anshu S. Anand
Indian Institute of Information Technology Allahabad
E-mail: anshusanand2001@gmail.com

Karthik Sayani
Indian Institute of Information Technology Vadodara
E-mail: kartik.sayani.3196@gmail.com

R. K. Shyamasundar
Indian Institute of Technology Bombay
E-mail: shyamasundar@gmail.com

1 Introduction

The accelerating pace of advances in Computational Science has challenged the scientific community to use advanced computing capabilities to understand and solve complex problems from various scientific disciplines like numerical simulations, model fitting, and data analysis. These problems involve large computations that are usually executed on multi-node computing clusters with each node comprised of multi-core processors and accelerators that specialize in number crunching and parallel computing. To exploit the available hardware, several high-performance languages have been developed, which enable programmers to run an application on thousands of threads over hundreds of computing units. For an experienced computer programmer, it is a matter of a few days to learn and get familiar with the features of these languages. However, all these languages have their own sophisticated syntaxes and constructs. This leads to two challenges:

- Firstly, there is much boilerplate code involved in writing a parallel program. Programmers themselves have to spawn and manage the thread pool and are also responsible for synchronization and joining of these threads. For example, for a simple FOR loop construct, the body of which is required to be executed in parallel, a programmer has to invoke a certain number of threads, write code to issue tasks to each thread, and also write separate code for invoking these threads on several processors. The situation gets further complicated when atomic blocks and several computing devices with multiple cores are involved.
- Secondly, for a person not familiar with the syntax of the language, it becomes increasingly difficult to understand how the problem is being solved by a computer. This is important because the end-users of the output of the solutions to these problems are they themselves. It is therefore required that there be some way to get the syntax as close as possible to mathematical notations, which are universally understood.

With these requirements in mind, Guy Steele and his team at Sun Microsystems Inc. began working on Project Fortress in 2002. Fortress [1] was developed as a part of the High Productivity Computing Systems (HPCS) program, along with Chapel [12] and X10 [10,11], for development of computing systems with a focus on productivity and performance. However, even as Chapel and X10 have evolved as full-fledged programming languages, the development of Fortress language, unfortunately, ceased in the year 2012 due to various reasons. However, Fortress did introduce some exciting syntax and constructs. The goal of the developers was to build a language that was as close as possible to mathematical notation, was efficient, scalable, and at the same time, was as simple as possible to facilitate productivity in high-performance software development.

Motivation

The aim of this project is fueled by and carries forward the original goals of Fortress, i.e., to build a high-performance language that is close to mathematical notation, scalable, and simplifies writing parallel code. Since domain/computational scientists are by far the largest user community of HPC, Fortress's proximity to mathematical notation reduces the scope for errors in translating the mathematical equations into programs, while also simplifying debugging, thereby improving their productivity significantly. Thus, providing continued support to such a language becomes absolutely necessary. However, Project Fortress being no longer under development, and in its current state only having a working interpreter, all the features provided by Fortress become unreachable to the entire scientific computing community. In this paper, we try to address this problem using the X10 programming language.

X10 is an object-oriented and statically-typed language that has found wide popularity in the HPC community. Both X10 and Fortress unite in the view of the abstract programming model that they follow: Partitioned Global Addressing Space (PGAS), wherein the global addressing space is logically divided into several partitions such that each partition is local to some processing element (called Places in X10). This approach is particularly helpful in exploiting data locality in multi-core, multi-node clusters.

Since both Fortress and X10 work on very similar execution models and are both compiled and ran on JVM, it opens up an opportunity to build a source-to-source compiler (transcompiler/transpiler) for Fortress code to be converted into X10. Thus, in this paper, we present a transpiler that translates Fortress programs into X10, enabling users to still write programs in the Fortress syntax that is similar to mathematical notation, and at the same time, get better performance using X10's infrastructure. Our work also supports Multiresolution langauge philosophy [2] that simplifies parallel programming by allowing the domain scientists to use the high-level specification for convenience and productivity, and at the same time, provide fine-grained control to the HPC programmers using low-level representations for performance. Since the X10 compiler has two backends: C++ and Java, X10 itself can be compiled to both C++ and Java, and therefore provides more flexibility to the HPC programmers for fine-grained optimizations of the original Fortress code.

The paper has been organized as follows: In the next section, we give a brief background of the Xtext framework and the Xtend language used to realize the transpiler. The architecture of the translation process from Fortress to X10 is described in detail in Section 3.2. We then present a case study of Buffon's Needle algorithm and give its Fortress implementation and the translated X10 code in Section 4. The results of run-time comparisons of benchmark applications in Fortress and the corresponding translated X10 code is discussed in Section 5.

2 Background

The Fortress-to-X10 Transpiler has been built using Xtext [6] which is a framework for development of programming languages especially parser-based external Domain-Specific Languages (DSL) [3,4]. Unlike internal DSLs which are written inside an existing host language in the form of an API, external DSLs are parsed independently of the host language and have their own syntax. The transpiler also employs the Xtend language [7] for code generation. We first give a brief overview of the Xtext framework and the Xtend language, and then discuss the similarities and differences in Fortress and X10 languages.

2.1 Xtext framework

Xtext provides complete infrastructure, including parser, linker, type-checker, compiler as well as editing support for Eclipse. It uses the LL(*) parser generator of ANTLR in the background, allowing it to cover a wide range of syntax.

In order to build a DSL, Xtext requires the DSL's grammar to be defined using the Xtext grammar language. Xtext grammar language itself is an EBNF-like DSL developed using Xtext [9]. Now, to derive the various language components, we need to execute the *Generate Xtext Artifacts* command. This generates the following:

1. a metamodel based on Eclipse Modelling Framework's (EMF) Ecore model [5]. From this Ecore model, a Java-API is generated that allows the AST to be accessed programmatically.
2. an ANTLR-based parser that generates the abstract syntax tree (AST) for textual DSL models.
3. a full-featured text editor with support for code highlighting, syntax coloring, content assist, code navigation, etc.

2.2 Xtend Language

Xtend is a statically typed template language for implementing generators, interpreters, and model transformations. Since each of these require access to AST, Xtend enables programmatic access to it using the Java-API mentioned in Section 2.1. Since Xtend is compiled to Java, it seamlessly integrates with all existing Java libraries. Like Xtext's grammar language, Xtend language too has been built using Xtext. It provides a rich set of language features like:

– lambda expressions
– template expressions
– active annotations
– type-based switch statements
– polymorphic method invocation

The meta models of Xtext DSLs are represented as Ecore models and since Xtend itself is built using Xtext, each Xtend program is also represented as an Ecore model. A detailed discussion of the relationship between Xtext, Xtend and Ecore can be found in [8].

2.3 Fortress and X10 languages

Here, we introduce Fortress and X10 languages through a comparison of a few important programming aspects:

1. **Programming model:** As mentioned in Section 1, both Fortress and X10 use the PGAS programming model which provides an abstraction of a single shared address space even though the address space is partitioned into regions based on the underlying NUMA architecture. X10, in addition, also supports asynchronous operations and control flow, which permits the creation of asynchronous tasks locally and globally, due to which it is said to be an Asynchronous PGAS (APGAS) language.

2. **Basic execution model:** The basic unit of execution in Fortress is a thread - implicit or explicit (launched using `spawn`), while in X10 it is known as an activity - a lightweight thread or a user-level thread that is much cheaper to create and manage than kernel-level threads.

3. **Memory abstraction:** Unlike X10 and Chapel, which provide flat memory abstractions, Fortress provides a hierarchical abstraction of the target architecture. This is realized using regions that map to an element of the systems hierarchy i.e., node, processor, core, or memory, thereby forming a hierarchical tree. Every thread, object, and array element has an associated region in Fortress, which can be queried using the function *region* provided by Fortress.
   X10, on the other hand, uses the notion of places that represent various computational units with local memory. Both Fortress and X10 allow computation to be placed near data using the same construct: `at`.

Fortress and X10 have many similar features. While Fortress uses `spawn-at`, `for-spawn`, and `at-atomic`, X10 uses `at-async`, `for-async`, and `at-atomic` for asynchronous remote tasks, nested parallelism, and remote transactions, respectively. Also, X10 provides distributed arrays for data distribution, while Fortress provides arrays, vectors, and matrices, that are assumed to be distributed across the machine [1]. *Tuples* in Fortress are also similar to *Points* in X10.
Fortress, in addition, supports several unique features that were aimed at improving the productivity of programmers. Here, we list a few:

1. Growable syntax [17], [18]
   Growable syntax allows growing a programming language using syntactic abstraction. Thus, the growable syntax of Fortress allows it to adapt to the

changing needs of users by providing support for adding new constructs in libraries by defining them in terms of existing constructs.

2. Dimensions and units [19]
   Many applications involve representing physical quantities that are usually expressed as raw numbers. By providing adequate support for units and dimensions, Fortress eliminates bugs that may arise due to mis-representation of different physical measurements. For example, addition/substraction/-comparison of quantities that are expressed in different units.

3. Function contracts [1]
   allows a user to express certain semantic properties that cannot be expressed through the static type system.

4. High-level combinators [16]
   It allows nested data structures to be generated through a set of primitives, called Generators of Generators (GoGs).

Further description of Fortress language follows in Section 3.2. We now describe the architecture of the Fortress to X10 transpiler for the translation of Fortress programs to X10.

3 Translation from Fortress to X10

In this section, we describe the challenges in compiling Fortress to X10 and describe the implementation of the Fortress-to-X10 transpiler along with an illustration of translation of a few key constructs.

3.1 Challenges

Targeting X10 for translation has raised many challenges in the design of the transpiler. Issues such as extensive usage of Left Recursion in the original Fortress grammar expressed in Parser Expression Grammar and inclusion of unimplemented features such as Dimensions, Coercion, Tests, and Properties embedded in the Fortress Grammar presented a major task of filtering and adapting the essential abstractions from the Fortress grammar while staying true to the objectives of the project. The resulting modified Fortress Grammar faced the following major challenges:

– **Multiplication Operator**:
  Originally, multiplication in Fortress is implied by juxtaposing two operands together. Juxtaposition itself is an overloaded operator that is given semantic actions at run-time, i.e., When the left operand is a function, number or a string, juxtaposition performs function application, multiplication and string concatenation, respectively.

Such an operator was possible to be defined in Parsing Expression Grammar (PEG) due to infinite look-ahead, which is not the case with LL(*) parser. And hence, multiplication operator '*' was used.

- **Parsing nested expressions**:
  Since Xtext doesn't support left-recursive parser rules, parsing of nested expressions is difficult due to their recursive nature. To get rid of left-recursion, the grammar needs to be left-factored. Operator precedence is handled by defining an order of delegation. An operator with higher precedence has its rule listed above other operators.

- **Dimensions and Units**:
  Since Fortress is being translated to X10, which doesn't provide support for dimensions and units, providing support for it in Fortress poses a challenge. This is because dimension-checking would require manipulation of dimensions according to a dimensional algebra. There are type systems that model units as types so that dimensional analysis is reduced to type checking [20]. However, due to the complex semantics of units, more powerful algorithms are required to perform type checking for unit correctness.

- **Return Statements**: Fortress, being a expression-oriented language, has every construct as an expression that returns some value. For example, in the following code:

```
factorial(n:ZZ64):ZZ64 =
    if n == 0
        then 1
    else
        n*factorial(n-1)
end
```

  the IF block is an expression that returns a value. We illustrate an example in Section 3.3, where we revisit the same code (but in a different context) and show how we address it.

3.2 An Architecture for translation from Fortress to X10

The Fortress to X10 transpiler, built using the Xtext framework and the Xtend language, takes as input a Fortress program and translates it into the corresponding X10 code. The architecture of Fortress-to-X10 transcompilation is shown in Figure 1.

To implement the Fortress to X10 transpiler, the Xtext project takes as input the grammar of Fortress language (obtained from the Fortress language specification [1]) specified in Xtext's grammar language. On generating the

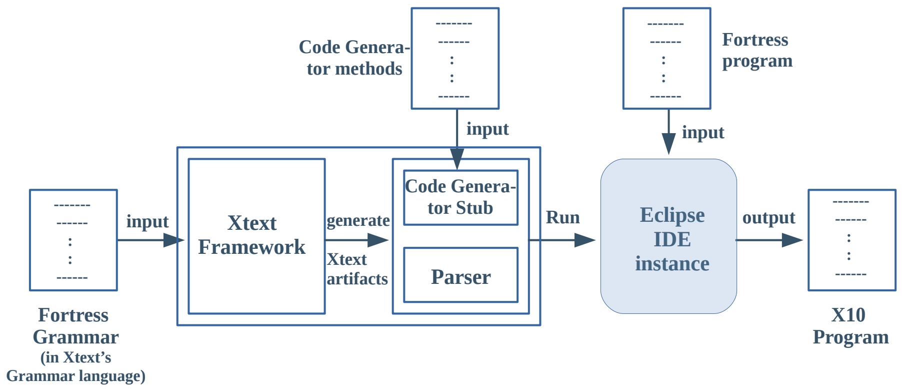


Fig. 1 Architecture of Fortress-to-X10 transcompilation

Xtext artifacts for the Fortress grammar, apart from the parser and other infrastructure being generated, a code generator stub is put into the runtime project. The code generator is then written using the Xtend language by defining methods to translate each element of the EMF's metamodel.

On executing the project, a new instance of Eclipse IDE is generated with support for functionalities like code completion, syntax highlighting, syntactic validation, linking errors, the outline view, find references, etc. This enables complete Eclipse support for programs written in the Fortress language.

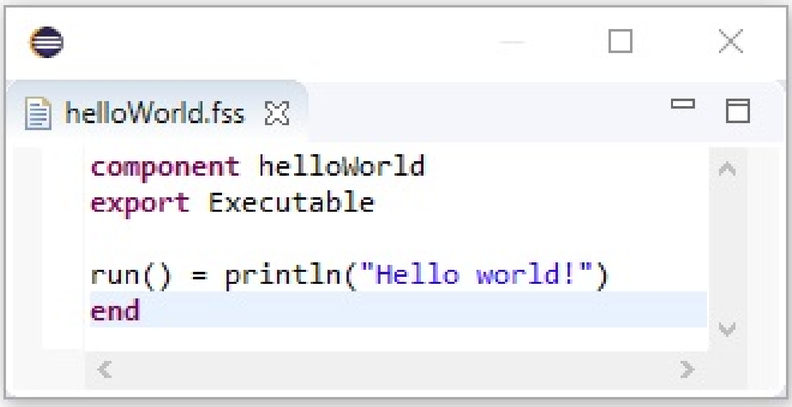
To translate Fortress programs to X10, the code generator invokes (Xtend) methods corresponding to each element of AST generated by the parser. Since the DSL Xtext editor is already integrated in the automatic building infrastructure of Eclipse, the generator will be automatically called when the source is written/modified in our DSL (Fortress, in our case). Thus this change is automatically reflected in the translated code.

We now illustrate a simple "Hello World" application that is translated from Fortress to X10. Figure 2 shows the application written in Fortress. Figure 3 shows a representation of the AST for the "Hello World" program, obtained using the *Sample Reflective Ecore Model Editor* tool.

The translated X10 code is shown in Listing 1.

Listing 1 Translated X10 code

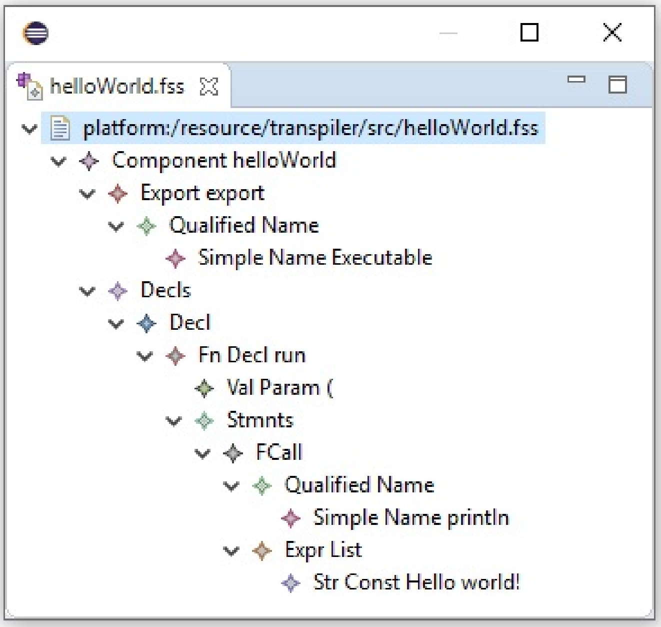
```
public class helloWorld{
        public static def main(args:Rail[String]){
                Console.OUT.println("Hello world!");
        }
}
```



```
component helloWorld
export Executable

run() = println("Hello world!")
end
```

Fig. 2 A simple hello world application in Fortress



```
platform:/resource/transpiler/src/helloWorld.fss
  Component helloWorld
    Export export
      Qualified Name
        Simple Name Executable
    Decls
      Decl
        Fn Decl run
          Val Param (
          Stmnts
            FCall
              Qualified Name
                Simple Name println
              Expr List
                Str Const Hello world!
```

Fig. 3 AST of the Fortress hello world application

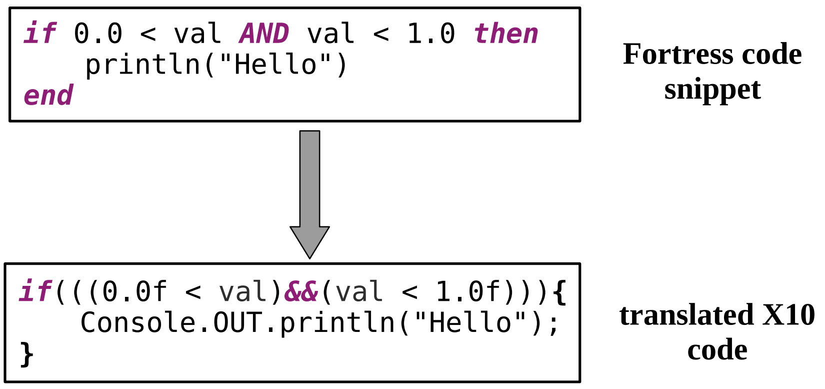
We now illustrate the translation of a few key constructs of Fortress into X10.

3.3 Illustration of translation of a few key constructs

For illustration, we show the translation of a few key constructs in Fortress to X10.

3.3.1 Translation of if-then block

Figure 4 shows a Fortress code snippet of an *if-then* block and the corresponding translated X10 code.



```
if 0.0 < val AND val < 1.0 then
    println("Hello")
end
```

```
if(((0.0f < val)&&(val < 1.0f))){
    Console.OUT.println("Hello");
}
```

Fig. 4 Fortress to X10 translation of `if-then` block

The Xtend method that realizes this translation is listed below:

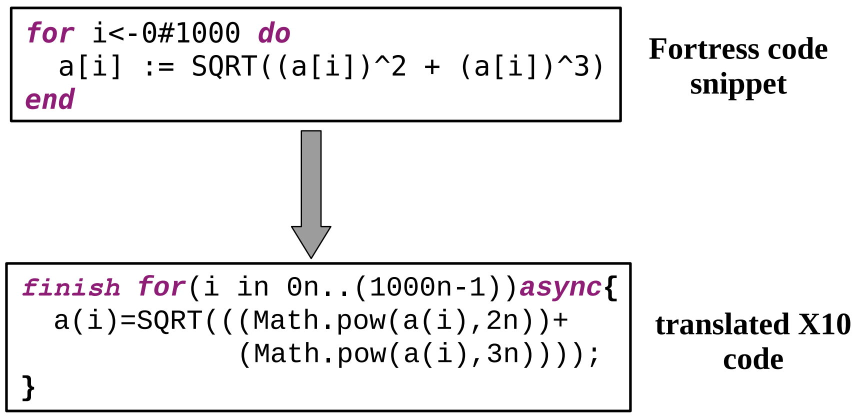
```
var s= '''if('''+d.cond.compile+''')'''+
     '''{''' + ”\n” + d.blocks.compile +
     ”\n”+'''}'''+”\n”
```

The Xtend method generates the if-then block in X10 syntax using simple string operations. As with Java, the string literals in Xtend (enclosed in ”' ”') can be concatenated with the '+' operator. The transpiler traverses through the AST and translates every element of *if* to the corresponding code in the X10 syntax using the above method.

3.3.2 Translation of for loop

Since *for* loops in Fortress are parallel by default, the equivalent translated X10 code can be implemented using *async* and *finish*. For each loop iteration, a new asynchronous activity in X10 is spawned using *async*. Figure 5 shows a Fortress code snippet and the corresponding translated X10 code.

Parallelism in *for* loops is specified by the generators used. Thus, if there is any dependency that limits parallel execution, the programmer should specify



```
for i<-0#1000 do
  a[i] := SQRT((a[i])^2 + (a[i])^3)
end
```

```
finish for(i in 0n..(1000n-1))async{
  a(i)=SQRT(((Math.pow(a(i),2n))+
             (Math.pow(a(i),3n))));
}
```

Fig. 5 Fortress to X10 translation of `for loop`

the use of sequential generator (eg. i $\leftarrow$ $seq(1 : n)$) that forces the iterations to be performed sequentially.

The grammar rules for *for* loop in Fortress, specified in the Xtext's grammar language is listed below:

```
DelimitedExpr:
'for' gen=Generators dofront=DoFront 'end';

Generators:  binding=Binding
                 (',' clause+=GenClause)* ;

Binding: idtup=Qualified '<-' g=GenSource
   | idtup=Qualified '<-'
   seq='seq' '('g=GenSource')';

GenSource: Expr({GenSource.start=current}
                                   '#' end=Expr)?;

GenClause: binding=Binding
                | expr=Expr ;
```

A *for* loop consists of the 'for' keyword followed by a generator clause list. In Fortress, comma-separated generator clause lists are utilized to express parallel iterations (eg. for i$\leftarrow$ 1:m, j$\leftarrow$ 1:n do ..... end). Thus, the body of a 'for' loop is evaluated for every combination of values bound in the generator clause list (i, j in the above example), in parallel. The generator clause list begins with a binding that consists of one or more identifiers followed by the token '$\leftarrow$' and a generator source that is essentially a sub-expression that specifies the range of values for which the *for* loop is to be evaluated.

3.3.3 Translation of function contracts

Function Contract is a key feature of Fortress that allows a function to impose certain conditions on its execution. They enable us to express semantic properties that cannot be expressed through the static type system.

Fortress code

```
factorial(n: ZZ64):ZZ64
          requires {n>=0}
      ensures {result >= 0}=
    if n === 0
        then 1
    else
        n*factorial(n-1)
end
```

Translated X10 code

```
static def factorial(n:Long):Long{
    var result:Long;
    if((n >= 0)){
        if((n == 0)){
            result = 1;
        }
        else{
            result = (n*factorial((n-1)));
        }
    }else
        throw new Exception("CallerViolation");

    if((result >= 0)==false)
        throw new Exception("calleeViolation");

    return result;
}
```

Fig. 6 Fortress to X10 translation of `Function Contracts`

It allows three optional clauses in the function's declaration that are evaluated in the scope of the function body: `requires`, `ensures`, and an `invariant` clause. A brief description of these clauses is given below:

1. `requires`
   It specifies constraints (as a sequence of comma-separated boolean expressions) that the argument to a function must satisfy. The body of the func-

tion is evaluated only if these expressions evaluate to true, or else an exception (CallerViolation) is thrown.

2. `ensures`
   An `ensure` clause is evaluated after a `requires` clause. It consists of a sequence of ensures subclauses, each comprising of a boolean expression, followed by an optional `provided` subclause. The `provided` subclause consists of the keyword `provided` followed by a boolean expression. The boolean expression preceding `provided` is evaluated after the function body is evaluated, only if the expression following `provided` evaluates to true or in the absence of the `provided` subclause. If this expression (preceding `provided`) evaluates to *false*, an exception *CalleeViolation* is thrown.
3. `invariant`
   It specifies a sequence of expressions (of any type), enclosed by curly braces. These expressions are evaluated both before and after a function call. For each expression e, if the value of e evaluated before the function call is not equal to the value of e evaluated after the function call, an exception *CalleeViolation* is thrown.

Figure 6 shows an example of the usage of function contracts in Fortress and the corresponding translated code in X10. The code also highlights the ability of Fortress to succinctly express these semantic properties, which could be realized in X10 only with much extra boilerplate code, affecting the readability of code. Also, since the function has a return type (Long), the transpiler declares a variable *result* that captures the values of expressions returned by the `if-then` block, which is eventually returned at the end.

We now present a case study of the actual translation of a Fortress program to X10 code using our transpiler framework.

4 Case Study

For the case study, we consider an implementation of Buffons needle [13] which is a Monte-Carlo Simulation to estimate the value of π. Given a floor with equally spaced parallel lines distance d apart, it finds the probability that a needle of length l lands on any of the lines. This probability is then used to estimate the value of π.

4.1 Fortress code

The buffon's needle program implemented in Fortress is listed below:

```
component buffons
export Executable
run ( ) : ( ) = do
  needleLength : RR64 = 20.0
  numRows : RR64 = 10.0
  tableHeight : RR64 = needleLength *
                                   numRows
  var hits:RR64 = 0.0
  var n:RR64 = 0.0
  start:RR64 = nanoTime()
  println ( "Starting parallel Buffons")
  for i<-1#3000 do
    delta_X = random (2.0) - 1.0
    delta_Y = random (2.0) - 1.0
    rsq = delta_X^2 + delta_Y^2
    if 0.0 < rsq < 1.0 then
      y1 = tableHeight * random (1.0)
      y2 = y1 + needleLength *
                     (delta_Y / SQRT(rsq))
      (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
      temp1 : RR64 = y_L / needleLength
      temp2 : RR64 = y_H / needleLength
      if |/temp1\| = |\temp2/| then
        atomic do hits := hits + 1.0 end
      end
      atomic do n := n + 1.0 end
      end
    end
    probability = hits/n
    pi_est = 2.0/ probability
    println("hits =" || hits ||"n = "||n)
    println ("Buffons : estimated Pi= "
                                   || pi_est)
    finish = nanoTime ( ) - start
    println(finish)
  end
end
```

4.2 The translated X10 code

The X10 code of the Buffon's needle program translated using the Fortress-to-X10 transpiler is listed below:

```
import x10.io.Console;
import x10.lang.Math;
import x10.array.Array_1;
import x10.array.Array_2;
import x10.array.Array_3;
import x10.util.Random;
/*needs to import
```

```
*/
/*exports
export Executable
*/

public class buffons{

  public static def main(args:Rail[String])
  {
    val needleLength:Double = 20.0f as
                                   Double;
    val numRows:Double = 10.0f as Double;
    val tableHeight:Double =
        (needleLength*numRows) as Double;
    var hits:Double = 0.0f as Double;
    var n:Double = 0.0f as Double;
    val start:Double = nanoTime() as
                                 Double;
    Console.OUT.println("Starting
                      parallel Buffons");
    finish for(i in 1n..(3000n-1)) async{
      val delta_X = (random(2.0f)-1.0f);
      val delta_Y = (random(2.0f)-1.0f);
      val rsq = ((Math.pow(delta_X,2n))
                +(Math.pow(delta_Y,2n)));
      if(((0.0f < rsq)&&(rsq < 1.0f))){
        val y1 = (tableHeight
                      *random(1.0f));
        val y2 = (y1+(needleLength
                * (delta_Y/SQRT(rsq))));
        val y_L = min(y1,y2);
        val y_H = max(y1,y2);
        val temp1:Double =
            (y_L/needleLength) as Double;
        val temp2:Double =
            (y_H/needleLength) as Double;
        if((Math.ceil(temp1) ==
                    Math.floor(temp2))){
          atomic{
            hits = (hits+1.0f);
          }
        }
      }
      atomic{
        n = (n+1.0f);
      }
    }
  }
  val probability = (hits/n);
  val pi_est = (2.0f/probability);
  Console.OUT.println("hits =");
  Console.OUT.println(hits);
  Console.OUT.println("n = ");
  Console.OUT.println(n);
  Console.OUT.println("Buffons :
                        estimated Pi= ");
  Console.OUT.println(pi_est);
  val finish = (nanoTime()-start);
  Console.OUT.println(finish);
  }
}
```

The case study demonstrates the translation of Buffon's Needle from Fortress to X10 code using our transpiler framework. It demonstrates the following constructs/features of Fortress: component, export, run, for loop, and atomic. The codes show the conciseness of the Fortress code and how the syntax is closer to mathematical notation and hence more intuitive. An important distinction between the two languages is the implicit data parallelism in Fortress given by the parallel-by-default for construct.

5 Experimental Results

In this section, we discuss a few benchmark applications in Fortress and their performance with respect to the transformed X10 codes.

All the experiments were conducted on Ubuntu 16.04 LTS system with the Intel i7-4710HQ processor supported by 8 GB of DDR3 RAM. The Fortress codes have been given full JVM memory allowance to create as many implicit threads as needed to achieve the maximum performance. The X10 codes are all run with 4 Places and as many number of activities required by each place. All the run-times are averaged over 5 runs.

We first compare the execution times of the following three applications:

– Matrix Multiplication (*MM32*): employs Divide and Conquer strategy to multiply two matrices of size 32*32.
– Positive Feedback (*PosFeedback*): Given a set of entities, each having a positivity score and a bank of facts, evolving them by supplying random facts.
– Buffons Pi (*Buffons*) [13]: A Monte Carlo method using Buffons needle to approximate the value of Pi.

The results of these experiments are shown in Figure 7.

It is evident from the results that the translated X10 code outperforms the

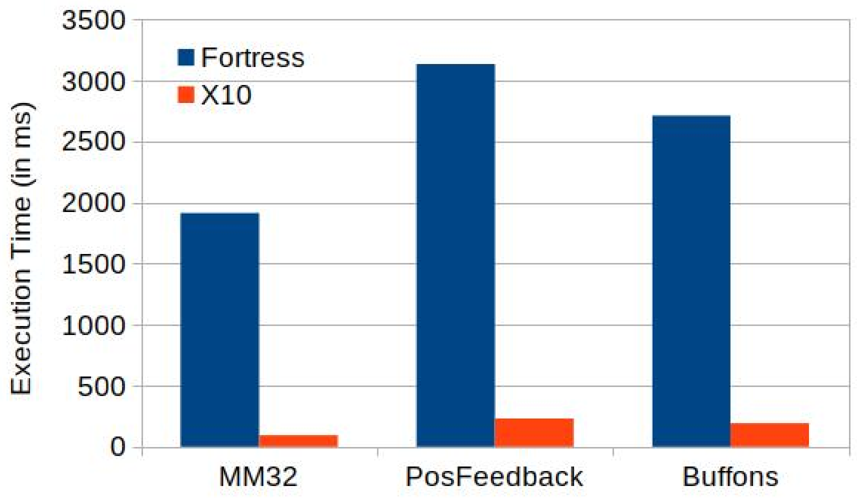


Fig. 7 Fortress Vs X10: Run-time Comparison

original Fortress code in terms of execution time, with an average speedup of 16x observed across the three experiments. While the speedup is 20.36x for *MM32*, it is 13.63x and 14.13x for *PosFeedback* and *Buffons*, respectively.

We also performed experiments with the following benchmark applications:

– Array Stream Benchmarks [15]: It is a synthetic benchmark program that measures memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The results for array stream benchmarks have been shown in Tables 1 and 2.

Table 1 Fortress Stream Benchmark Test

Int32	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	0.76380121485	10.4739294
	Sscale	0.73007764609	10.957738595
	Add	0.91273471055	13.147303221
	Triad	0.80712187147	14.867643195
	Sscale	0.79928385621	10.008959818
Int64	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	1.79794160235	8.899065453
	Sscale	1.47913110244	10.81716149
	Add	2.00083967138	11.994964086
	Triad	1.493870698	16.065647472
	Sscale	1.514487715	10.564628446
Float	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	1.782398159	8.976669955
	Sscale	1.84008156	8.695266747
	Add	1.756414426	13.664201139
	Triad	2.035071163	11.793199389

Table 2 X10 Stream Benchmark Test

Int32	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	8.42345860721409	4.748643267
	Sscale	9.524426067121496	4.199728122
	Add	13.528420739316202	4.435107479
	Triad	8.899139206154164	4.494816754
Long	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	16.486417432848583	4.852479341
	Sscale	16.41668567602583	4.873090804
	Add	21.6605982250443	5.540013196
	Triad	14.58383811756954	5.485524411
	Sscale	17.148535071815235	4.66512152
Float	**OpType**	**Rate(MB/s)**	**Avg. Time(s)**
	Copy	8.02382936905595	4.985150875
	Sscale	8.483019312473154	4.715302244
	Add	10.898170595394351	5.505511175
	Triad	7.578662969599845	5.277975833

In all of the experiments (Int, Int64/Long, and Float), an average speedup of more than 2x was observed for the translated X10 code. Also, in all cases, the highest speedup was observed for the *Triad* operation when compared to other operations.

– NAS Fast Fourier Transform (and Inverse) of 3D grids [14]: This application solves a 3-dimensional partial differential equation using the Fast Fourier Transform. The results for the NAS Fast Fourier Transform (FT) have been shown in Tables 3 and 4.

Table 3 Fortress NAS FT Benchmark

Class	**Grid**	**Iterations**	**Avg. Run Time**
Class T:	2x4x4	1	0.650441406 seconds
Class S:	64x64x64	1	503.49956373 seconds
Class W:	128x128x32	1	1040.398577308 seconds

Table 4 X10 NAS FT Benchmark

Class	**Grid**	**Iterations**	**Avg. Run Time**
Class T:	2x4x4	1	0.019727665 second
Class S:	64x64x64	6	2.653876066 seconds
Class W:	128x128x32	6	5.042912789 seconds

For NAS-FT, X10 clearly outperforms Fortress by a large margin. The highest speedup of 206x is achieved by the X10 code for Class W.

6 Discussion

The experimental results show an impressive performance of the translated X10 code with respect to the original Fortress code. The huge performance improvement was as expected because Fortress was an experimental and interpreted language, with an inefficient interpreter. However, by providing this tool for translating Fortress code to X10, we provide an opportunity for better engagement of domain scientists with the X10 community. It will enable the domain scientists to express their problems in a representation that is closer to the mathematical notation, thereby reducing the scope for errors and simplifying debugging. It will also significantly improve their productivity since they can get their implementation to work without worrying about low-level performance aspects. The 90/10 rule [2], which states that almost 90% of a programs execution time is spent within 10% of its code, also suggests better productivity for domain scientists by limiting their focus to problem-solving than on performance improvement. Tuning the program for performance, which requires a good understanding of low-level implementation details, could be realized by HPC programmers. As an example, the domain scientists may write an application using the Fortress arrays. The translated X10 code can then be optimized for performance by taking advantage of the rich support for arrays provided by X10. For example, while *Region* array in X10 could be used as the default choice due to the flexibility they offer, the array implementation could be changed to *Rail* for better performance in certain cases.*Rail* is an intrinsic one-dimensional fixed-size array that supports only up to three dimensions using row-major ordering, thus allowing efficient optimizations on indexing operations. Further, since X10 itself can be compiled to either C++ or Java, HPC programmers have more flexibility to optimize the original Fortress code in a language of their choice.

7 Conclusion

In order to leverage the novel syntactic features of Fortress, we have built a Fortress to X10 transpiler that enables a program written in Fortress syntax to be automatically translated into the corresponding X10 code. We have showcased the tremendous performance gain of X10 codes over the existing Fortress system. Thus, we have shown how the reduction in boilerplate code, readability achieved by Fortress and the impressive performance of X10 can be amalgamated together to form a powerful language for high performance and scientific computing. This also supports the Multiresolution language philosophy that will enable the domain scientists to write programs easily in the Fortress syntax that is close to the mathematical notation, without bothering about performance. The translated X10 code inherently improves performance and can further be optimized for performance by utilizing the low-level features of X10, Java or C++.

There are certain features from the original Fortress implementation that have not yet been implemented in the transpiler like growable syntax and high-level combinators [16]. The current transpiler only supports a single component i.e. a single class and also does not provide support for APIs, which is another name for Interfaces in Fortress. This is the immediate future goal of the transpiler, to provide the notion of multiple classes, APIs, and packages. There are several novel features of Fortress that are not present in the original implementation either, such as dimensions, units, and where clauses. Including these features will further simplify the language for a programmer and hence increase productivity. Another possible addition for the transpiler would be operator overloading and user-defined types and operators. This is an interesting addition that can be useful in various scenarios.

References

1. Sun MicroSystems Inc., The Fortress Language Specification (2008)
2. Multiresolution Languages for Portable yet Efficient Parallel Programming, Bradford L. Chamberlain, whitepaper (2007)
3. Fowler, M., Domain-Specific Languages, Addison-Wesley (2010)
4. Voelter, M., et al.: DSL Engineering - Designing. Implementing and Using Domain-Specific Languages, 1558 dslbook.org , ISBN: 978-1-4812-1858-0 (2013)
5. Steinberg, D., et al., EMF: Eclipse Modeling Framework 2.0, 2nd. Addison-Wesley Professional, ISBN: 0321331885 (2009)
6. Xtext homepage, http://www.eclipse.org/Xtext/
7. Xtend Language homepage, http://www.xtend-lang.org
8. Klaus Birken, Building Code Generators for DSLs Using a Partial Evaluator for the Xtend Language, In Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Technologies for Mastering Change - Volume 8802, Springer-Verlag New York, Inc., New York, NY, USA, 407-424 (2014)
9. Lorenzo Bettini, Implementing DSL with Xtext and Xtend, Second Edition (2nd ed.). Packt Publishing (2016)
10. Kemal Ebcioglu, Vijay Saraswat, Vivek Sarkar, X10: Programming for hierarchical parallelism and non-uniform data access, International Workshop on Language Runtimes, OOPSLA (2004)
11. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar, X10: an object-oriented approach to non-uniform cluster computing, SIGPLAN Not. 40, 519-538, (2005)
12. David Callahan, Bradford L. Chamberlain, Hans P. Zima, The Cascade High Productivity Language, In 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), pages 52-60, IEEE Computer Society, (2004)
13. Buffon, G., Essai darithmetique morale, Histoire Naturelle, Gnrale, et Par-ticulire, Supplment, 4 , 685 (1777)
14. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html
15. J. D. McCalpin, STREAM Benchmark, Link:http://www.cs.virginia.edu/stream/
16. Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki and Masato Takeichi, Generators-of-Generators Library with Optimization Capabilities in Fortress. Euro-Par (2) 2010: 26-37. https://doi.org/10.1007/978-3-642-15291-7_4
17. Sukyoung Ryu, Parsing Fortress syntax, In 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09), ACM, NY, USA, 76-84 (2009)
18. E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu, Growing a syntax, In Proceedings of Workshop on Foundations of Object-Oriented Languages, 2009.
19. Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele. 2004. Object-oriented units of measurement. In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 04), ACM, New York, NY, USA, 384403 (2004)
20. Lingxiao Jiang and Zhendong Su. 2006. Osprey: a practical type system for validating dimensional unit correctness of C programs. In 28th international conference on Software engineering (ICSE 06). ACM, New York, NY, USA, 262271 (2006)
21. Sascha Roloff, Frank Hannig and Jrgen Teich, Modeling and Simulation of Invasive Applications and Architectures, Springer Singapore, 2019.