Declarative Coordination of Graph-based Parallel Programs *

Flavio Cruz Ricardo Rocha
CRACS & INESC TEC and Faculty of Sciences
University of Porto
Rua Campo Alegre 1021/1055
4169-007 Porto, Portugal
flavioc,ricroc@dcc.fc.up.pt

Seth Copen Goldstein
Carnegie Mellon University
Pittsburgh, PA 15213, USA
seth@cs.cmu.edu

Abstract

Declarative programming has been hailed as a promising approach to parallel programming since it makes it easier to reason about programs while hiding the implementation details of parallelism from the programmer. However, its advantage is also its disadvantage as it leaves the programmer with no straightforward way to optimize programs for performance. In this paper, we introduce Coordinated Linear Meld (CLM), a concurrent forward-chaining linear logic programming language, with a declarative way to coordinate the execution of parallel programs allowing the programmer to specify arbitrary scheduling and data partitioning policies. Our approach allows the programmer to write graph-based declarative programs and then optionally to use coordination to fine-tune parallel performance. In this paper we specify the set of coordination facts, discuss their implementation in a parallel virtual machine, and show—through example—how they can be used to optimize parallel execution. We compare the execution of CLM programs against the original uncoordinated Linear Meld and several other frameworks.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel Programming; D.3.4 [*PROCESSORS*]: Run-time environments

General Terms Design, Languages, Performance

Keywords Parallel Programming, Linear Logic

1. Introduction

Parallel programming is hard mainly because manipulating shared data may result in undesirable race conditions. Typically, such issues are handled with low level constructs such as locks, semaphores and/or condition variables, all of which require a fair amount of effort to get right. Declarative programming has been hailed as an alternative solution to this issue, since the problem of implementing the details of parallelism is moved from the programmer to the compiler and runtime environment. The programmer writes code without having to deal with parallel programming constructs and the compiler automatically parallelizes the program in order to take advantage of multiple threads of execution. This programming paradigm has been adopted with huge success in domain specific languages such as SQL and MapReduce [13]. Although general declarative languages have yet to be as successful, the future looks promising for this particular approach.

The main problem with declarative programming is that it gives the programmer little or no control over how execution is scheduled or how data is laid out, making it hard to improve efficiency. This introduces performance issues because even if the runtime system is able to reasonably parallelize the program using a general algorithm, there is a lack of specific information that a compiler cannot easily deduce. Such information could make execution better in terms of run time, memory usage, or scalability.

In this paper, we introduce Coordinated Linear Meld (CLM), a data-centric declarative language that extends the Linear Meld (LM) language [11, 12] with coordination to give the programmer control over scheduling and data placement. LM is a programming language designed for programs that operate on graphs and with support for structured manipulation of mutable state through the use of linear logic [16]. A program is composed of logical rules and computation happens through inference. LM's model of computation is to assign, in the abstract, a thread of control to each node in the graph being operated on. The computation on each node can proceed independently from all others, but nodes can communicate with each other. Thus, the logical rules may be applied - in parallel and asynchronously - to each node in the graph.

The CLM language features two kinds of coordination primitives that can be used in the same way as any other primitive, i.e., they are specified with a similar syntax and semantics as the rest of the language, modulo some restrictions discussed in Section 4. These coordination primitives can be used to improve program execution based on the state of the program and the state and structure of the underlying machine. The first kind of coordination primitives are called *sensing facts* and are used to sense information about the system the program is running on, e.g., scheduling and node placement on threads. The second kind of coordination primitives are called *action facts* and can be used to apply scheduling and partitioning decisions during execution. Coordination facts allow the programmer to write logical rules that depend on the current state of the program and then prioritize node computation or node placement in different threads.

CLM is a declarative programming language that gives the programmer the ability to control execution while staying declarative and without resorting to meta-language constructs. The declarative

* This work is partially funded by the North Portugal Regional Operational Programme (ON.2 O Novo Norte) and by the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF) and the Portuguese Foundation for Science and Technology (FCT), within projects NORTE-07-0124-FEDER-000059 and UID/EEA/50014/2013.

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851153

coordination in CLM exposes the state of the underlying runtime system in order to allow the programmer to write rules that understand how the program is executed on a concrete machine. In this way, non-deterministic transitions of the parallel machine are modeled using linear logic rules, which allows the implementation of logical rules that optimize scheduling and partitioning. Furthermore, coordination in CLM is first-class which allows programmers to easily reason about and integrate how scheduling and partitioning interact with their algorithm. The result is that programmers can optimize their programs while still retaining the conciseness and clarity of expression of their original unoptimized programs.

Our contributions are four-fold: (i) we introduce a small set of coordination primitives and explain their interaction with the underlying programming model and parallel machine; (ii) we show how coordination is implemented in a logic-based parallel virtual machine that runs CLM programs; (iii) we present several linear logic programs and describe how coordination allows small modifications to lead to better scheduling and partitioning; (iv) we measure and compare the performance of CLM programs against their uncoordinated versions and against other state-of the art systems such as Ligra [37] and GraphLab [23]. We find that CLM allows the writing of higher-level, declarative, graph-based programs that are more concise than competitive systems and yet yields competitive scalable parallel performance. The introduction of coordination facts enables the programmer to fine-tune programs without losing the positive aspects of declarative programming.

2. Related Work

Declarative Languages Programming paradigms such as *logic* and *functional* programming have been extensively exploited for their implicit parallelism. In logic languages such as Prolog, researchers took advantage of the non-determinism in proof-search to evaluate subgoals in parallel. In functional languages, the stateless computation allows multiple expressions to safely evaluate in parallel. This has been explored in several languages such as NESL [6], Id [28], and more recently Data Parallel Haskell [7]. NESL often obtains good performance, but is limited to nested lists. CLM limits the application space to graphs, but also gives the programmer a declarative way to coordinate computation.

Linear Logic and Logic Programming Linear logic has deep connections with concurrency and programming.

Linear logic has been used as a basis for logic-based programming languages [25], including forward-chaining and backwards-chaining programming languages. Lolli [18], for instance, is a programming language based on a fragment of intuitionistic linear logic that proves goals by lazily managing the context of linear resources during top-down proof search. For concurrent programming, linear logic has also been used to model interacting computational agents using the formulas-as-agents equivalence [35]. It has also been shown that concurrent models based on linear logic have connections with the π-calculus, a powerful model for concurrent computation [30].

Data Centric Languages Recently, there has been increasing interest in declarative data-centric languages. MapReduce [13], for instance, is a popular programming model that is optimized for large clusters. Intrinsic to its popularity is the simplicity of its scheduling and data sharing model. In order to facilitate the writing of programs over large datasets, SQL-like languages such as PigLatin [29] have been developed. PigLatin builds on top of MapReduce and allows the programmer to write complex data-flow graphs, raising the abstraction and ease of programmability of MapReduce programs. An alternative to PigLatin/MapReduce is Dryad [20] that allows programmers to design arbitrary computation patterns using DAG abstractions. It combines computational vertices with communication channels (edges) that are automatically scheduled to run on multiple computers/cores.

Graph Of Nodes The idea of partitioning computation along the nodes of the graph is not new and has been realized before in several parallel programming models. For instance, in the *partitioned global address space* (PGAS) model, there is a global address space that is partitioned among processes in order to increase locality and each process contains one or more activities that operate on local data. The PGAS model has been realized in X10 [8], a programming language that builds on top of object oriented programming. In logic programming, the concept of locations was first proposed in P2 [22], a system designed for declarative networking, where the nodes of the network graph represent real machines. P2 introduced a compiler that localizes rules so that computation is performed locally. But crucially, each node in the graph was a separate and concrete processor.

The original LM language was inspired by Meld [3], a Datalog-like language for programming distributed ensembles of modular robots. Meld also introduced the idea of sensing and action facts in order to sense and act on the outside world, respectively.

Galois [32] is a parallel programming model optimized for graphs, trees and sets. A Galois parallel algorithm is viewed as a parallel application of an *operator* over an irregular data structure which generate *activities* on the data structure. Such operators may, for instance, be applied to a graph's node in order to change its data or change the structure of its neighborhood, allowing for data structure changes. Nodes with computation are called *active elements* and the set of nodes required to apply an operator is called the *neighborhood*. An *ordering* dictates the order in which operators are applied to active elements and required neighborhood. From the point of view of the programmer, the active elements are represented in a work-list, while operators can be implemented on top of iterators of the work-list.

Ligra [37] is a lightweight framework for large scale graph processing that exploits the fact that most huge graph datasets available today can be made to fit in the main memory of commodity servers. Ligra is a simple framework that exposes two main interfaces: `EdgeMap` and `VertexMap`. The former applies a function to a subset of edges of the graph, while the latter applies a function to a subset of vertices. The functions passed as arguments are applied to either a single edge or a single vertex and the user must ensure that the function can be executed in parallel.

Another interesting system is GraphLab [23], a C++ framework for developing graph-based parallel machine learning algorithms. GraphLab allows nodes to have read/write access to different scopes through different concurrent access models in order to balance performance and data consistency. GraphLab also provides different schedulers that dictate the order in which nodes are computed, allowing the programmer to optimize the program. Later in this paper, we will show how one GraphLab scheduler can be implemented in CLM through the use of coordination facts. Another, more restrictive, graph-based system is Pregel [24], where graph algorithms must be executed as a sequence of iterations of computation and message passing.

Galois, Ligra, GraphLab, and Pregel are not declarative programming languages, but they tend to provide better performance than CLM at the cost of a steeper learning curve. In particular, Pregel and Ligra are much more suited for processing large scale graphs than CLM. However, we argue that CLM programs are more amenable to reasoning than the programs written for the systems above due to CLM's logic programming foundations.

Coordination Languages and Systems Many programming languages and systems follow the so-called *coordination paradigm* [31], a form of distributed programming that divides execution in two parts: *computation*, where the actual computation is performed, and *coordination*, which deals with communication and cooperation between processing units. This paradigm attempts to clearly distinguish between these two parts by providing abstractions for coordination in an attempt to provide architecture and system-independent forms of communication.

Linda [1] is considered a pioneer coordination model and implements a data-driven coordination model featuring a *tuple space* that is manipulated using input/output operations. Linda is limited in the sense that the programmer can only coordinate the scheduling of processing units, while the placement of data is left to the implementation.

The Reo [2] system allows composition and coordination of concurrent processes using channels. Reo has connections with intuitionistic temporal linear logic [9], a powerful logic that can naturally model coordination patterns. Another programming model with connections with temporal linear logic is Timed Concurrent Constraint Programming (TCCP) [21, 36], a framework directed at modeling reactive systems which interprets computation as deduction in a fragment of temporal linear logic. The original LM language already allows the kind of coordination seen in both Reo and TCCP since it is possible to compose arbitrary, but less structured, communication patterns between nodes of LM graphs. The goal of CLM is to allow a kind of *meta-coordination* that acts on the real computation units, realized as computation threads, with the goal of improving efficiency.

Galois [32] allows the use of custom scheduling strategies for coordinating parallel execution. First, there is *compile-time coordination*, where the scheduling is computed during compilation. Secondly, there is *runtime coordination*, where the order of non-conflicting activities is computed during execution and computation proceeds in rounds. In a third strategy, *just-in-time coordination*, the order of activities is defined by the underlying data structure where the operator is applied (for instance, computing on a graph may depend on its topology). Nguyen et al. [27] expanded the concept of runtime coordination with the introduction of a flexible approach to specify scheduling policies for Galois programs. This approach was motivated by the fact that some algorithms run faster using different scheduling strategies. The scheduling language specifies 3 basic main scheduler types: First-In First-Out, Last-In First-Out and Ordered-By-Metric. These schedulers can then be composed and synthesized without requiring users to write complex concurrent code. When compared to Galois, CLM allows a more general approach to coordination by allowing the specification of arbitrarily complex coordination patterns using the provided facts.

Elixir [33] is a domain specific language that builds on top of Nguyen's work and allows easy specification of scheduling strategies. The main idea behind Elixir is that the user should be able to specify how operator application is scheduled and the framework will compile this high level specification to low level code using the provided scheduling specification. One of the motivating examples is the Single Source Shortest Path program that can be specified using multiple scheduling specifications, generating different well-known shortest path algorithms such as the Dijkstra or Bellman-Ford algorithm.

Halide [34] is a language and compiler for image processing pipelines with the goal of optimizing parallelism, locality and recomputation. Halide decouples the algorithm definition from its execution strategy, allowing the compiler to find which execution strategy may be the best optimization for locality and parallelism. The language permits the specification of the scheduling strategy, allowing the programmer to decide the order of computations, what intermediate results need to be stored, how to split the data among processing units, and how to use vectorization and the well-known sliding window mechanism. The compiler is able to use stochastic search to automatically find good schedules for Halide pipelines that are sometimes better than hand-written code.

Sequoia [15] and Legion [5] are programming languages designed for coordinating computation on hardware with deep complex memory hierarchies. Programs are written as sets of tasks and the programmer controls how tasks are laid out in memory. Legion supports both regular and irregular algorithms, while Sequoia requires extensions [4] in order to support irregular algorithms.

In contrast to the previous systems, CLM stands alone in making coordination (both scheduling and partitioning) a first-class programming construct and semantically equivalent to computation. Furthermore, CLM distinguishes itself by supporting data-driven dynamic coordination, particularly for irregular data structures. Elixir and Galois do not support coordination for data partitioning and, in Elixir, the coordination specification is separated from computation, limiting the programmability of coordination. Compared to CLM, Halide is targeted for regular applications and therefore only supports compile time coordination.

3. The LM Programming Language

LM [11] is a concurrent programming language designed for programs that operates on graphs. CLM is an extension of LM that introduces coordination facts to allow declarative partitioning and scheduling. In order to understand how CLM works, we review the basic ideas of LM.

LM programs consist of a set of *rules* and a *database of facts*. Rules have the form `a(N), b(N, M) -o c(M, N)` and can be read as follows: if fact `a(N)` and fact `b(N, M)` exist in the database then `c(M, N)` is added to the database. The expression `a(N), b(N, M)` is called the *body* of the rule and `c(M, N)` is the *head* of the rule. A fact is a predicate, e.g., `a`, `b` or `c`, and its associated tuple of values, e.g., the concrete values of `N` and `M`. Since LM uses linear logic as its foundation, we distinguish between *linear* and *persistent facts*. Linear facts are deleted during the process of deriving a rule, while persistent facts are not. Program execution starts by adding the *axioms* (the initial facts) to the database. Next, rules are recursively applied and the database is updated by adding new facts and deleting facts used during rule derivation. When no more rules are applicable, the program terminates.

LM was designed for writing programs that operate on graphs. To achieve concurrency, LM partitions the database by using the first argument of each fact. The first argument has type *node* and represents a node in the graph being operated on. For example, the fact `f(@1, 2)` is stored in node `@1`, while fact `p(@2)` is stored in node `@2`. LM restricts the body of every rule to facts with the same node so that nodes can derive rules independently. The head of the rule may refer to any node as long as that node is referred to somewhere in the body. This allows *communication* between nodes during rule derivation, since a node may *send* a fact to another node. Rule restrictions in turn make LM implicitly parallel because nodes are able to compute independently. This makes LM non-deterministic since nodes can be picked to run in any order, affecting which rules are applied and which facts are deleted or derived.

To make these ideas concrete, Fig. 1 presents a program to solve the single source shortest path (SSSP) problem. Later in the paper, we add coordination facts to improve the execution of this program.

The SSSP program starts with the declaration of the predicates (lines 1-3). Predicates specify the facts used in the program. The first predicate, `edge`, is a persistent predicate that describes the relationship between the nodes of the graph, where the third argument represents the weight of the edge. The `route` modifier informs the compiler that the `edge` predicate determines the structure of the graph, which, in this case, does not change. The

```
1   type route edge(node, node, int).
2   type linear shortest(node, int, list int).
3   type linear relax(node, int, list int).
4
5   !edge(@1, @2, 3). !edge(@1, @3, 1).
6   !edge(@3, @2, 1). !edge(@3, @4, 5).
7   !edge(@2, @4, 1).
8   shortest(A, +00, []).
9   relax(@1, 0, [@1]).
10
11  shortest(A, D1, P1), relax(A, D2, P2), D1 > D2
12     -o shortest(A, D2, P2),
13        {B, W | !edge(A, B, W) |
14           relax(B, D2 + W, P2 ++ [B])}.
15
16  shortest(A, D1, P1), relax(A, D2, P2), D1 <= D2
17     -o shortest(A, D1, P1).
```

Figure 1: *Single Source Shortest Path program code.*

predicates `shortest` and `relax` are specified as linear facts and thus are deleted when deriving new facts. The algorithm computes the shortest distance from node `@1` to all other nodes in the graph. Every node has a `shortest` fact that can be improved with new `relax` facts. Lines 5-9 declare the axioms of the program: `edge` facts describe the graph; `shortest(A, +00, [])` is the initial shortest distance (infinity) for all nodes; and `relax(@1, 0, [@1])` starts the algorithm by setting the distance from `@1` to `@1` to be 0.

The first rule of the program (lines 11-14) reads as follows: if the current `shortest` path `P1` with distance `D1` is larger than a new path `relax` with distance `D2`, then replace the current shortest path with `D2`, delete the new `relax` path and propagate new paths to the neighbors (lines 13-14) using a *comprehension*. A comprehension in LM has three components separated by the `|` symbol: the list of variables introduced in the scope of the comprehension, the body of the comprehension and the head of the comprehension. While the body and head can be understood as a sub-rule, the comprehension construct is special since it is applied as often as the database allows and for all possible combinations. In this particular case, for each `edge` fact available at node `A`, a new `relax` fact is derived at each neighbor `B`. The new `relax` fact indicates that a new path with distance `D2 + W` is available, which is the path `P2` extended with the edge to `B`. For example, in Fig. 2(a) we apply rule 1 on node `@1` where two new `relax` facts are derived at node `@2` and `@3`. Fig. 2(b) is the result after applying the same rule, but at node `@2`.

The second rule of the program (lines 16-17) is read as following: if the current shortest path `D1` is shorter than the new path `D2` then delete the new `relax` fact and keep the current shortest path.

There are many opportunities for concurrency in the SSSP program. For instance, after applying rule 1 in Fig. 2(a), it is possible to apply rules in either node `@2` or node `@3`. This depends largely on implementation factors such as node partitioning and the number of threads in the system. Still, it is easy to prove that no matter what schedule is used, the final result, as presented in Fig. 2(c), is achieved.

4. Coordination

The SSSP program is concise and declarative but its performance depends on the order in which nodes are executed. If nodes with greater distances are prioritized over other nodes, the program will generate more `relax` facts since it will take longer to reach the shortest distances. From Fig. 2, it is clear that the best scheduling is the following: `@1`, `@3`, `@2` and then `@4`, where only four `relax` facts are generated. If we had decided to process nodes in order `@1`, `@2`, `@4`, `@3`, `@4`, `@2`, then six `relax` facts would have been generated. The optimal solution for SSSP is to schedule the node with the shortest distance, which is essentially the Dijkstra shortest path algorithm [14]. Note how it is possible to change the nature of the algorithm by simply changing the order of node computation, but still retain the declarative nature of the program.

CLM extends LM with *coordination facts* which allow the programmer to change how the runtime schedules nodes and how it partitions the nodes among threads of execution. Coordination facts can be used in either the body of the rule, the head of the rule, or both. This allows scheduling and partition decisions to be made based on the state of the program and the state of the underlying machine. In this fashion, we keep the language declarative because we reason logically about the state of execution, without the need to introduce extra-logical operators into the language that would introduce significant issues when proving properties about programs. It can be proven that, in fact, programs that employ only action facts will obtain the same results as the programs without any coordination [10] because even though scheduling and partitioning is done differently, it does not change the results of the program because the proof of correctness for the uncoordinated program already takes into account all possible schedulings. For programs which use sensing facts and complex scheduling policies, a proof needs to take into account the semantics of coordination and how the virtual machine transitions from one state to another using built-in linear logical rules [10].

Coordination facts are classified into *sensing* and *action* facts. A sensing fact represents a part of the state of the underlying virtual machine and can be used in the body of rules or comprehensions similarly to how a normal fact is used with two exceptions. First, sensing facts cannot be deleted nor derived more than once. This maintains the consistency between the database of facts and the state of the virtual machine. Second, sensing facts are not restricted by the first argument. Note that when sensing facts are used in a rule, they need to be re-derived automatically[1].

Action facts are used in the head of rules or comprehensions in order to apply an action on the virtual machine, i.e., to change the state of the virtual machine. Semantically, action facts are consumed by a built-in logical rule that may or may not consume a sensing fact, forcing the virtual machine to change its state accordingly. Action facts can also be used in the body of rules, however, a rule with an action fact in its body would never be applied because the built-in rule would be used first.

4.1 Scheduling Facts

To support different scheduling strategies, we introduce the concept of *node priority* by assigning a priority value to every node in the program and by introducing coordination facts that manipulate these priority values. Initially, all nodes have a default priority of $-\infty$, meaning that, theoretically, nodes can be picked in any order. In practice, our implementation uses a FIFO approach since older nodes tend to have a higher number of unexamined facts, from which to derive subsequent new facts.

The sensing fact `priority(node A, float P)` is used to retrieve the current priority `P` of node `A`. The current priority, `P`, will be the node's *default priority* unless it has been changed by the `set-priority` action fact to a *temporary priority*. Once a node is processed by a thread of control, its priority is reset to its default priority. In the virtual machine, each node is either *running* (deriving rules) or *idle* (waiting to be selected by a thread of control). This information is represented using linear facts and corresponding logical rules which update the `priority` fact to the default priority.

The following list presents the scheduling action facts available in CLM:

[1] In fact, all coordination facts are linear and the system creates the necessary code to re-derive them without requiring programmer interaction.

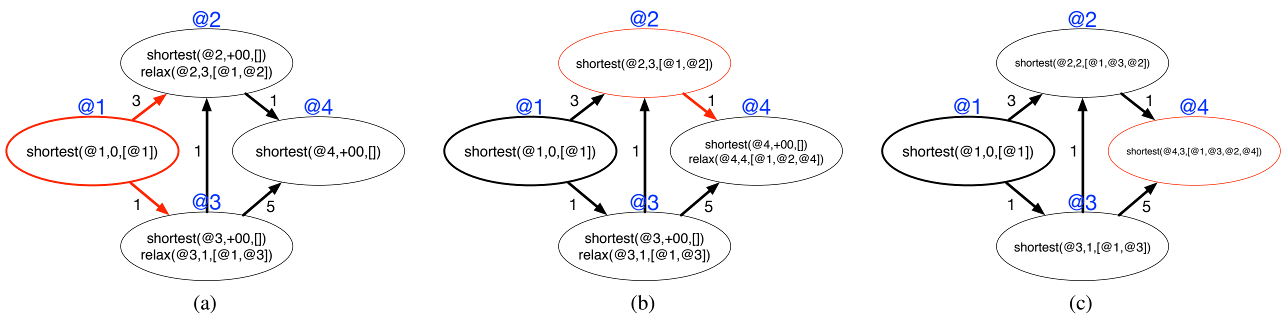


Figure 2: *Graphical representation of the SSSP program: (a) represents the program after propagating initial distance at node* `@1`*, followed by (b) where the first rule is applied in node* `@2` *and by (c), where all the shortest paths have been computed (final program state).*

- `set-default-priority(node A, float F)`: sets the default priority of node `A` to `F`.
- `set-priority(node A, float F)`: sets the temporary priority of node `A` to `F` if `F` is *better* than the current priority (either default or temporary). The programmer can decide if priorities are to be ordered in ascending or descending order.
- `add-priority(node A, float F)`: increases, temporarily, the priority of node `A` by `F`.
- `remove-priority(node A)`: removes the temporary priority from node `A`.
- `schedule-next(node A)`: changes the temporary priority of node `A` to be $+\infty$.

4.2 Partitioning facts

CLM also provides several coordination facts to influence node partitioning among the running threads. We introduce the type *thread* to refer to threads that are running on separate processors. In terms of action facts, we have the following:

- `set-thread(node A, thread T)`: places node `A` in thread `T` until the program terminates or a `set-moving(A)` fact is derived.
- `set-affinity(node A, node B)`: places node `B` in the thread of node `A` until the program terminates or a `set-moving(B)` fact is derived.
- `set-moving(node A)`: allows node `A` to move freely between threads.

In terms of sensing facts, we have the following:

- `thread-id(node A, thread T)`: linear fact that maps node `A` to thread `T` which `A` belongs to. Action fact `set-thread` implicitly updates fact `thread-id`.
- `is-moving(node A)`: fact available at node `A` if `A` is allowed to move between threads.
- `is-static(node A, thread T)`: fact available at node `A` if `A` is not allowed to move between threads and is currently placed in thread `T`.

5. Implementation

The implementation of CLM extends the original LM implementation with an updated compiler and virtual machine (VM) to support coordination. The implementation of LM is described in [12] and supports parallel execution and scheduling of programs using supporting data structures to manage the database of facts.

5.1 Compilation

We updated the original LM compiler to translate each rule to a C++ procedure. This made the implementation more efficient and competitive with other systems. The resulting C++ procedures are compiled using a C++ compiler and then linked together with the virtual machine library. Each rule procedure loops over all possible combinations of the rule, retrieving facts from the database, performing join operations and then consuming and deriving facts. The compiler implements join optimizations (to support efficient rule filtering) and fact updates (to reduce allocations).

Coordination directives are compiled in two different ways, depending on whether they appear in the body or in the head of the rule. Coordination facts in the body are compiled into VM API calls that inspect the state of the virtual machine. For example, the `priority` fact inspects the target node and retrieves the current priority. Coordination facts in the head of the rule are also implemented as VM API calls, but they perform some action, instead of being added to the database as facts. Semantically, action facts are like any other fact. However, since they are immediately used by the machine, there is no need to store them in the database, therefore avoiding unnecessary allocations and deallocations. For optimization purposes, we implemented *coordination coalescing* so that facts such as `set-priority` and `add-priority` are buffered before being applied. We first fully process the node by deriving candidate rules using newly available facts and then apply any scheduling decisions inferred by the rules. This allows us to reduce inter-thread communication and contention.

5.2 Execution

The virtual machine is implemented in C++11 and uses the threading system from the standard library to implement multi-threading.

Threads To support coordination, each thread has two pairs of queues: a pair of doubly linked lists known as the *standard queue* and a pair of *min/max* heaps known as the *priority queue*. The standard queue contains nodes without priorities and supports push into tail, remove node from the head, remove arbitrary node, and remove first half of nodes. The priority queue contains nodes with priorities and is implemented as a binary heap array. It supports the following operations: push into the heap, remove the *min* node, remove an arbitrary node, remove half of the nodes (horizontal split of the binary heap tree), and priority update. Operations for removing half of the queue are implemented in order to support node stealing, while operations to remove arbitrary nodes or update priority allows threads to change the priority of nodes. The four local queues are also partitioned into the *movable queue* (1 priority queue and 1 standard queue) for nodes that may move between threads and the *static queue* (1 priority queue and 1 standard queue) for nodes that cannot be moved.

Nodes The implementation of CLM retains the node and thread data structure of LM. A node data structure is represented as a collection of facts (per predicate) and an indexing structure that keeps track of the available facts and potential candidate rules. Furthermore, each thread, represented by a thread data structure,

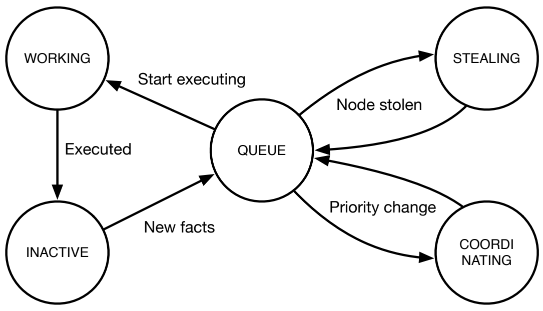


Figure 3: *The node state machine as represented by the state variable. During the lifetime of a program, each node goes through different states as specified by the state machine.*

is responsible for executing a subset of *active nodes*. A node is active if it has unexamined facts. After a node is processed, it becomes inactive until a new fact is derived for it, i.e., added to its database. When a new fact is derived for an inactive node, the node is activated and placed on the appropriate queue of the thread responsible for processing that node. Threads do useful work by processing active nodes from their queues. Whenever a thread does not have nodes to process, it attempts to steal nodes from a random thread. If unsuccessful, the thread becomes idle and waits for program termination while periodically attempting to steal nodes.

In order to implement coordination, we added a `state` variable to the node data structure. The state machine in Fig. 3 represents the valid state transitions of a node:

- **working**: the node is executing.
- **inactive**: the node is inactive, i.e., it has no new facts and is not in any queue for processing.
- **queue**: the node is active with new facts and is in some queue waiting to be processed.
- **stealing**: the node has just been stolen and is in the process of being moved to another thread.
- **coordinating**: the node is being scheduled or moved to another thread.

We also added a `static_owner` field that points to the thread where the node is currently placed and from which it cannot be moved. If `static_owner` is `NULL` then the node can be stolen from a thread's queue.

Each node is protected by a spin-lock that allows threads to update `state` and other node attributes such as: incoming facts, `owner` thread and `static_owner`.

Whenever a node has an empty database and no references from other nodes, it is deleted from the graph. The virtual machine uses reference counting to detect such cases.

Computation The main loop of each thread proceeds as follows. First, one active node, called the *current node*, is fetched from one of the thread's queues. Next, the candidate rules of the current node are computed and applied, resulting in new facts for the current node and other nodes. Finally, The current node becomes inactive and the loop is repeated again, until all active nodes are processed.

Nodes are picked by first inspecting the two priority queues in order to fetch the highest priority node, and then the two standard queues, in cases where priorities are not being used. The thread continues processing facts for the current node until there are no new facts generated for it. If all queues of the thread are empty, then the thread will try to steal work (i.e., active nodes in the movable queue) from another thread.

To minimize inter-thread communication, node priorities are implemented at the thread level. Thus, when a thread picks the highest priority node from the priority queue, it is only the highest priority with respect to the set of nodes owned by the thread and not the highest priority node in the whole program.

As an optimization, the `next` and `prev` pointers of the standard queue are part of the node structure in order to save space. These pointers are also used as the index and current priority for the priority queue, respectively.

Communication Threads synchronize with each other using mutual exclusion. We use a spin-lock in each queue to protect queue operations. For coordination, given threads T_1 and T_2, if T_1 needs to perform coordination operations to a node in T_2, it needs to synchronize with T_2 during priority updates in order to move the node in T_2's queues. Likewise, when using `set-thread` or `set-affinity`, the target thread's queues also need to be locked when moving from T_1 to T_2.

Coordination overhead We measured the impact of our implementation changes on unmodified LM programs and our experiments indicate that there is little to no impact on the overall performance of these programs.

6. Applications

To better understand how coordination facts are used, we present some programs that take advantage of them. In our experimental setup, we used a machine with a 32 Core AMD Opteron(tm) Processor 6274 HE @ 1400 MHz with 32 GBytes of RAM memory running the Linux Kernel 3.18.6-100.fc20.x86_64. We compiled our virtual machine using GCC 4.8.3 (g++) with the flags `-O3 -std=c++11 -march=x86-64` [2].

For comparison purposes, we wrote sequential versions of the CLM programs in the C++ programming language. We also compare some programs against the Ligra [37] and GraphLab frameworks [23]. In the plots shown next, the left axis represents the run time of programs using a logarithmic scale and the right axis represents the speedup of the `B` version using the `A` baseline which is calculated as `A(1)/B(t)`.

6.1 Single Source Shortest Path

We start by adding coordination to the SSSP program described before in Fig. 1. The coordinated version of the SSSP (Fig. 4) uses a global program directive to order priorities in ascending order (line 5) and the coordination fact `set-priority` (line 14).

When run on one thread, the algorithm behaves like Dijkstra's shortest path algorithm. When using multiple threads, each thread will pick the shortest distance from their subset of nodes. While this does not yield the optimal program with relation to 1 thread, it allows for parallel execution and locally avoids unnecessary work. The result scales well and it is close to Dijkstra's algorithm.

The most interesting property of the SSSP program presented in Fig. 4 is that the code remains declarative and provably correct, although it applies rules using a smarter ordering. Since the proof of correctness considers that, eventually, the shortest path is computed at all nodes of the graph, the use of `set-priority` does not change the proof at all.

Figure 5 shows experimental results for the SSSP when run with 3 different graphs[3]. The C++ version uses the Dijkstra algorithm to compute the distances from the source nodes, while the Ligra version uses the BellmanFord program provided in the Ligra source code. To make the comparison fairer, the Ligra execution time also includes the time required for loading the graph into memory (both CLM and C++ execution times also include loading times).

[2] Implementation, example programs and program proofs available in `http://github.com/flavioc/meld`

[3] Datasets retrieved from `http://snap.stanford.edu/data/`

```
type route edge(node, node, int).
type linear shortest(node, int, list int).
type linear relax(node, int, list int).

priority @order asc.

shortest(A, +00, []).
relax(@1, 0, [@1]).

shortest(A, D1, P1), relax(A, D2, P2), D1 > D2
   -o shortest(A, D2, P2),
      {B, W | !edge(A, B, W) |
         relax(B, D2 + W, P2 ++ [B]),
         set-priority(B, float(D2 + W))}.

shortest(A, D1, P1), relax(A, D2, P2), D1 <= D2
   -o shortest(A, D1, P1).
```

Figure 4: *Coordination code for the SSSP program. Underlined portions are the only changes from Fig. 1.*

The coordinated version of SSSP produces between 40% to 80% fewer facts than the regular version due to pruning of redundant distances. Consequently, coordinated code performs better than the regular version for all the three datasets. For instance, in the Orkut dataset, the coordinated version sees a 12-fold speedup for 32 threads, while the regular version enjoys only a 6-fold speedup. In the coordinated version, there are some situations where unnecessary facts are propagated because the shortest distance that is selected locally may not be the shortest distance globally. Thus, sub-optimal distances may be propagated because many SSSP distances are computed at the same time. Fortunately, this is not an issue even in datasets such as the US Powergrid dataset, where 4941 source nodes are used.

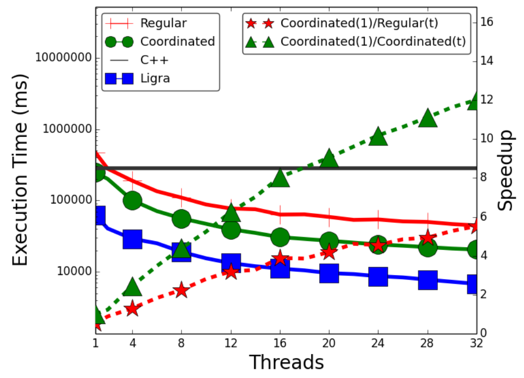
Overall, CLM performs well against C++ and needs only 4 threads to beat the C++ version. When compared to Ligra, CLM is not as competitive since Ligra is 3 (Orkut) and 10 (Live Journal) times faster (on average) than CLM. We analyzed the behavior of Ligra with these datasets and found that Ligra has the best cache behavior (fewer hits and fewer misses). However, Ligra does not perform well when computing the shortest distance from multiple sources (such as in US Powergrid). We found Ligra primitives unsuitable for simultaneous computation of shortest distances and it would have been better to write a program from scratch using a simple task-parallel approach (also note that Ligra does not store the computed shortest distances, but CLM does).

6.2 MiniMax

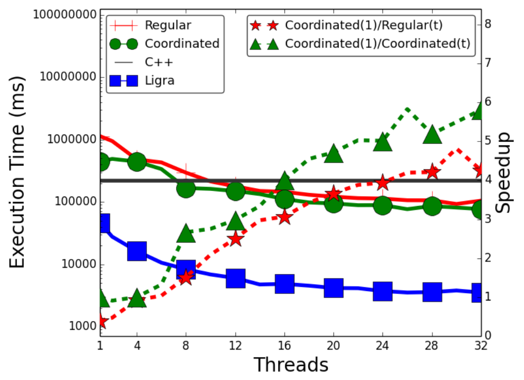
The MiniMax algorithm is a decision rule algorithm for minimizing the possible loss for a worst case (maximum loss) scenario in a zero sum game for 2 (or more) players who play in turns. With this program, we show that CLM also enables coordination of programs with dynamic graphs using both scheduling and partitioning.

The algorithm builds a game tree, where each tree node represents a game state and the children represent the possible game moves that can be made by either player 1 or player 2. An evaluation function is used to compute the score of the board for each leaf of the tree. A node is a leaf when the game state can no longer be expanded. Finally, the algorithm recursively minimizes or maximizes the scores of each node. To select the best move for player 1, the algorithm picks the move maximized at the root node.

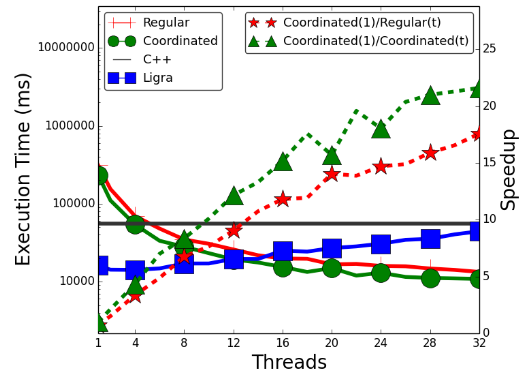
In CLM, the program starts with a root node (with the initial game state) which is expanded with the available moves at each level. The graph of the program is dynamic since nodes are created and then deleted once they are no longer needed. The latter happens when the leaf scores are computed or when a node fully minimizes or maximizes the children scores. When the program ends, only the root node has facts in its database.



(a) Orkut network. SSSP computed for 2 nodes on a graph with 3072441 nodes and 117185083 edges.



(b) Live Journal network. SSSP computed for 2 nodes on a graph with 4847571 nodes and 68993773 edges.



(c) US Powergrid network. SSSP computed for all nodes on a graph with 4941 nodes and 13188 edges (around 25 million shortest distances computed).

Figure 5: *Experimental results for SSSP: execution time and speedup.*

The code in Fig. 6 shows the tree expansion process. The first three rules (lines 1-10) deal with the case where no children nodes are created and the last three rules (12-29) deal with the cases that create new nodes. In particular, the two rules in lines 12-26 generate new nodes using the `exists` language construct, which creates a child node `B`. We link `B` with its parent (`parent(B, A)`) and kick-start the expansion of that node `B` by adding a `play` fact.

```
1   expand(A, Board, [], 0, P, Depth)
2     -o leaf(A, Board).
3
4   expand(A, Board, [], N, P, Depth),
5   N > 0, P = player1
6     -o maximize(A, N, -00, 0).
7
8   expand(A, Board, [], N, P, Depth),
9   N > 0, P = player2
10    -o minimize(A, N, +00, 0).
11
12  expand(A, Board, [0 | Xs], N, P, Depth),
13  Depth >= 5
14    -o exists B. (set-affinity(A, B),
15         set-default-priority(B, float(Depth + 1)),
16         play(B, Board ++ [P | Xs], next(P), Depth + 1),
17         expand(A, Board ++ [0], Xs, N + 1, P, Depth),
18         parent(B, A)).
19
20  expand(A, Board, [0 | Xs], N, P, Depth),
21  Depth < 5
22    -o exists B. (
23         set-default-priority(B, float(Depth + 1)),
24         play(B, Board ++ [P | Xs], next(P), Depth + 1),
25         expand(A, Board ++ [0], Xs, N + 1, P, Depth),
26         parent(B, A)).
27
28  expand(A, Board, [C | Xs], N, P, Depth) C <> 0
29    -o expand(A, Board ++ [C], Xs, N, P, Depth).
```

Figure 6: *Coordination code for the MiniMax program.*

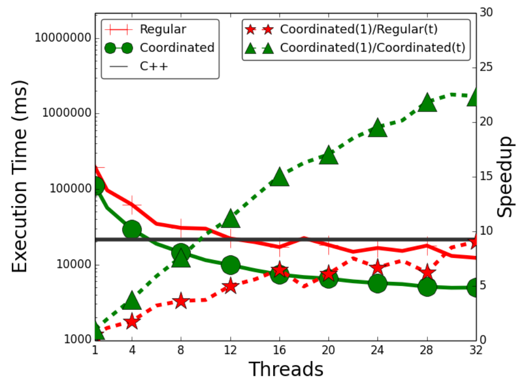


Figure 7: *Experimental results for MiniMax: execution time and speedup.*

As noted in Sect. 4.1, the default scheduler is breadth-first, which in this case leads to the complete expansion of the tree before computing the scores at any of the leaves. This results in an impractical program which uses $\mathcal{O}(n)$ memory, where n is the number of nodes in the tree.

With coordination, we set the priority of a node to be its depth (lines 15 and 23) so that the tree is expanded in a depth-first fashion, leading to a memory complexity of $\mathcal{O}(dt)$, where d is the depth of the tree and t is the number of threads. Since threads prioritize deeper nodes, the scores of the first leaves are immediately computed and then sent to the parent node. At this point, the leaves are deleted and reused for other nodes in the tree, resulting in minimal memory usage.

We also take advantage of memory locality by using `set-affinity` (line 14), so that nodes generated beyond a certain level are not stolen by other threads. While this is not critical for performance in shared memory systems where node stealing is fairly efficient, we expect that such coordination decisions to be critical in distributed systems.

```
new-heat(A, New, Old),
Delta = fabs(New - Old),
Delta > epsilon
   -o {B | !edge(A, B) |
         new-neighbor-heat(B, A, New),
         update(B), add-priority(B, Delta)}.

new-heat(A, New, Old)
fabs(New - Old) <= epsilon
   -o {B | !edge(A, B) |
         new-neighbor-heat(B, A, New)}.
```

Figure 8: *Coordination code for the Heat Transfer program.*

In Table 1 we compare the memory usage of the regular and coordinated MiniMax program versions. The **Average** columns show the average memory used by the program, while the **Final** columns show the final memory used by the program. The coordinated version uses, on average, significantly less memory (at most 44KB for 32 threads) than the regular version (around 222.7MB). The scalability results, presented in Fig. 7, show that the coordinated version running on 32 threads is more than 4 times faster than the sequential C++ program, while the regular version is only slightly faster than the same C++ program. Note that the C++ program uses a simple recursive function to compute the MiniMax score.

Threads	**Average**		**Final**	
	Regular	Coord	Regular	Coord
1	13.5GB	62KB	30KB	31KB
2	6.7GB	54KB	60KB	61KB
4	2.4GB	52KB	119KB	121KB
8	1545.6MB	50KB	236KB	240KB
16	624.2MB	47KB	472KB	479KB
24	400.5MB	45KB	707KB	717KB
32	222.7MB	44KB	942KB	955KB

Table 1: *Memory statistics for the MiniMax program.*

6.3 Heat Transfer

In the Heat Transfer (HT) program, we have a graph where heat values are exchanged between nodes. The program stops when the heat values of all the nodes of the graph converge to their true solution, that is, when the difference δ in heat between step i and step i_1 is smaller than a small value ϵ, as follows: $\delta = |H_i - H_{i-1}| \leq \epsilon$. The algorithm works asynchronously, i.e., heat values are updated using information as it arrives from neighboring nodes in a sequence of steps $0, \cdots, i$. This increases concurrency since nodes do not need to synchronize between steps since not all nodes require the same number of steps.

Fig. 8 shows the HT rules that send new heat values to neighbor nodes. In the first rule we added `add-priority` to increase the priority of the neighbor nodes if the current node has a large δ. The idea is to prioritize the computation of nodes (using `update`) that have a neighbor that changed significantly. Multiple `add-priority` facts will increase the priority of a node so that nodes with many neighbors with large deltas will have more priority.

To improve locality, we split the second rule to avoid sending small δ values if the target node is in another thread (Fig. 9). This new rule increases the scalability of the program, but comes at the price of increased errors in the heat values since heat values are computed using less accurate information. It is possible to write more complicated rules where nodes could accumulate incoming heat values and then compute and propagate new heat values when appropriate. This would also increase locality but without increasing the statistical error.

In Fig. 10 we present the results for the **Regular** (no coordination), the **Coordinated** (with `add-priority`), and the **Local** ver-

```
new-heat(A, New, Old)
fabs(New - Old) <= epsilon
thread-id(A, C)
   -o {B, D | !edge(A, B), thread-id(B, D), D = C
            | new-neighbor-heat(B, A, New),
              thread-id(B, D)},
      thread-id(A, C).
```

Figure 9: *To improve locality, we add an extra constraint to the second rule to avoid sending small δ values if the target node is in another thread.*

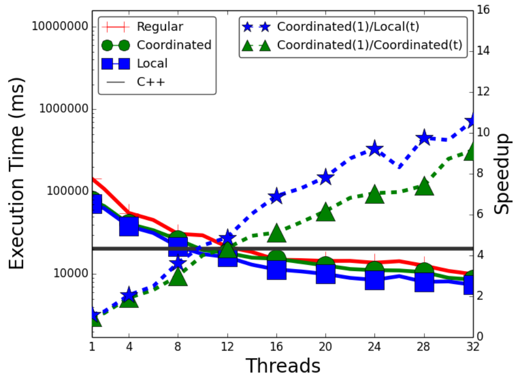


Figure 10: *Experimental results for Heat Transfer: execution time and speedup.*

sion (using `thread-id`). The dataset used is a square grid of size 120x120 with an inner square which is initialized to be very hot. When comparing the **Coordinated** version with the **Regular** version, for 1 thread there is a 50% reduction in run time, while for 32 threads there is, on average, a 25% reduction. The **Local** version sees even further run time reductions, with a 35% reduction for 32 threads. When compared to the C++ version, the **Local** version using 32 threads is 3 times faster than the sequential C++ code.

As we have seen for this program, the improvements from coordination are smaller when using more threads. This is expected since a large number of threads will perform more computation since each thread only picks the highest priority node from a smaller subset of nodes. It would be possible to add another kind of coordination fact, for instance, `set-global-priority`, that would define a priority which could be selected from all the executing threads. Since coordination is a first class construct in the language, CLM can be easily extended with new facts that allow the programmer to take advantage of different scheduling and partitioning semantics. Furthermore, this is achieved without modifications to the core semantics of the language.

6.4 N Queens

The N-Queens puzzle is a program which places N queens on an NxN chessboard so that no pair of queens attack each other [19]. The challenge of finding all the distinct solutions is a well-known benchmark in designing parallel algorithms. While this problem does not have a straightforward implementation in CLM, it shows that CLM can be used in a wide range of applications and is not limited to programs that only map naturally to graphs.

The CLM solution considers the squares of the chess board as a graph of nodes which exchange valid configurations with each other. Initially every square in the first row of the board gets the empty state. Then, each square adds its own position to the state and sends the state down, to the next row. Once a square receives new configurations, it attempts to add its position to the configurations.

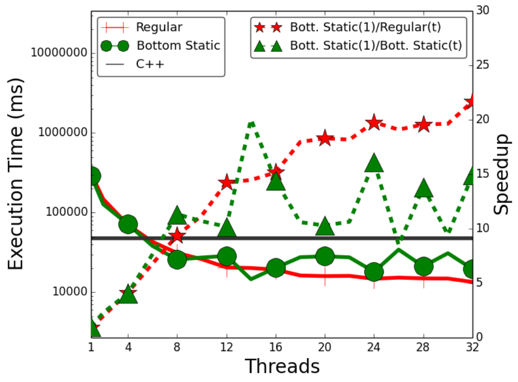


Figure 11: *Experimental results for 14 Queens: execution time and speedup.*

If valid, that configuration is then sent, recursively, to the next row, until all rows are traversed. At the end of the program, the squares at the bottom row will have all the valid configurations.

The N Queens program incrementally builds and shares lists representing valid board states that are transmitted from top to bottom. Since computation goes from the top row to the bottom row, not all placements of nodes to threads will perform equally. This is especially true because the bottom rows tend to perform the most work. A potential placement is to split the board vertically with axiom `set-thread(A, vertical(X, Y))` so each thread gets the same number of columns, where `X` and `Y` are the coordinates of a particular square. Since `set-thread` pins nodes to threads, it is expected that such a configuration will improve memory locality. Threads also need to manipulate board states that share a significant number of elements since each board state needs to be iterated over before being extended with a new position. To accomplish this, we may use the coordination fact `set-default-priority(A, X)` so that lower rows have a higher priority.

Experimental results are presented in Fig. 11. In these figures we use the configuration **Bottom Static** to represent the N Queens version using both `set-thread` and `set-default-priority`. It is clear that the **Bottom Static** configuration is worse for the most part when compared to **Regular**, however, it outperforms the **Regular** version when using 14 threads. We also see the best results for 13 threads in the 13 Queens program. This corresponds to the configuration where the columns of the chess board are perfectly partitioned among threads. Interestingly, this configuration also outperforms the **Regular** version with 32 threads with less than half the number of threads. Valgrind's CacheGrind tool also shows that the **Bottom Static** program version has the smallest number of cache misses. This indicates that it is helpful to match the problem to the number of available CPUs in the system in order to increase memory locality and also shows the power of having a correct data partitioning and scheduling policies through the use of coordination facts.

We experimented with three other configurations, namely, **Top** (upper rows have higher priority), **Top Static**, and **Bottom**, however, we did not see any improvements over the **Regular** version.

6.5 Splash Belief Propagation

Randomized and approximation algorithms can obtain significant benefits from coordination directives because their inherent non-determinism can be harnessed to evaluate rules in different orders. A good example is the Loopy Belief Propagation (LBP) program. LBP [26] is an approximate inference algorithm used in graphical models with cycles. In a nutshell, LBP is a sum-product message passing algorithm where nodes exchange messages with their neighbors and apply computations to the messages received.

LBP maps very well to the graph based model of CLM. In its original form, the belief values of nodes are computed by synchronous iterations. LBP offers more concurrency when belief values are computed asynchronously leading to faster convergence. For this, every node keeps track of all messages sent/received and recomputes the belief using partial information from neighbor nodes. It is then possible to prioritize the computation of beliefs when a neighbor's belief changes significantly.

The asynchronous approach proves to be a nice improvement over the synchronous version. Still, it is possible to do even better. Gonzalez et al. [17] developed an optimal algorithm, named Splash Belief Propagation (SBP), that first builds a tree and then updates the beliefs of each node twice, first from the leaves to the root and then from the root to the leaves. The root of this tree is the node with the highest priority (based on belief) while the rest of the tree must have a positive priority. Note that the priorities are updated when a neighbor updates its belief. These *splash trees* are built iteratively until we reach convergence.

The code in Fig. 12 presents the SBP coordination code for LBP. Please note that we just appended the code in Fig. 12 to a working but uncoordinated version of the algorithm, every other rule remains the same. We added new rules that coordinate the creation and execution of the splash trees:

Tree building : Each node has an `inactive` fact that is used to start the tree building process. When the highest priority node is picked, an `expand-tree` fact is created in order to create a new splash tree. In lines 18-24, we use an *aggregate* [11] to gather all the neighbor nodes that have a positive priority (due to a new belief update) and are in the same thread. Nodes are collected into list `L` and appended to list `Next` (line 24).

First phase : When the number of nodes in the tree reaches a certain limit, a `first-phase` (lines 13-14) is generated to update the beliefs of all nodes in the tree. As the nodes are updated, starting from the leaves and ending at the root, an `update` fact is derived to update the belief values (line 36).

Second phase : The computation of beliefs is performed from the root to the leaves and the belief values are updated a second time (line 49).

The `set-thread` action fact is used in line 2 to (1) force nodes to stay in the thread and (2) partition nodes as a grid of threads. This sets up areas of nodes for threads to build splash trees on.

In this program, coordination assumes a far more important role than we have seen before. Coordination rules drive the behavior of the algorithm and while the result of the algorithm is statistically identical to the original algorithm, SBP works very differently than LBP. SBP is also implemented in GraphLab [23], a C++ framework for writing machine algorithms. GraphLab provides the **splash** scheduler as part of its framework. It includes 350 lines of C++ code. With our coordination facts, it is possible to create the necessary scheduling with only 50 lines of code.

We measured the behavior of LBP and SBP for both CLM and GraphLab. Figure 13 and 14 show that both systems have very similar behavior when using a variable number of threads. In terms of absolute performance, CLM's BP program is, on average, 1.4 times slower than GraphLab, although the CLM program code is much easier to understand. The reason for this low slowdown ration is because LBP/SBP spends most of run time performing mathematical computations, which are optimized when compiling CLM rules into C++ code. For SBP, the overhead ratio of CLM over GraphLab is two-fold since CLM now performs more fact derivations and manipulations by building splash trees. Still, CLM is as scalable as GraphLab.

```
1  !coord(A, X, Y), start(A)
2     -o set-thread(A, grid(X, Y)).
3
4  // Keep expanding the tree.
5  inactive(A), tree(A, All, Next)
6     -o expand-tree(A, All, Next).
7  // Start tree by picking node with the highest
8  // priority.
9  inactive(A), priority(A, P), P > 0.0
10    -o expand-tree(A, [A], [A]), priority(A, P).
11
12 // Tree has finished expanding: start first phase.
13 expand-tree(A, All, Next), len(All) >= max
14    -o first-phase(A, All, reverse(All)).
15 // Expand tree.
16 expand-tree(A, All, [A | Next]), len(Next) < max-1,
17 thread-id(A, Id1)
18    -o [collect => L | Side | !edge(A, L, Side),
19          0 = count(All, L), // L is not in All
20          0 = count(Next, L), // L is not in Next
21          priority(L, P), P > 0.0,
22          thread-id(L, Id2), Id1 = Id2 |
23          priority(L, P), thread-id(L, Id2) |
24          send-tree(A, All, Next ++ L)],
25       thread-id(A, Id1).
26
27 send-tree(A, All, [])
28    -o first-phase(A, All, reverse(All)).
29 send-tree(A, All, [B | Next])
30    -o schedule-next(B),
31       tree(B, All ++ [B], [B | Next]).
32
33 // First phase: process nodes from leaves to root.
34 first-phase(A, [A], [A]) -o second-phase(A, [], A).
35 first-phase(A, [A, B | Next], [A])
36    -o update(A), schedule-next(B),
37       second-phase(B, [B | Next], A).
38 first-phase(A, All, [A, B | Next])
39    -o update(A), schedule-next(B),
40       first-phase(B, All, [B | Next]).
41
42 // Second phase: process nodes from root to leaves.
43 second-phase(A, [], _)
44    -o remove-priority(A), inactive(A), update(A).
45 second-phase(A, [A], Back)
46    -o update(A), inactive(Back),
47       inactive(A), remove-priority(A).
48 second-phase(A, [A, B | Next], Back)
49    -o update(A), inactive(Back), schedule-next(B),
50       second-phase(B, [B | Next], A).
```

Figure 12: *Coordination code for the SBP program.*

7. Costs of Coordination

Coordination support introduces overhead in two different ways. First, by manipulating the priority queues which require operations on a min heap. Second, by requiring more lock operations as we move nodes between queues and within queues.

As can be seen in Table 2, except for N Queens, most coordinated programs require significantly more queue operations. The data for this table comes from recording the number of queue operations and facts derived. Queue operations represent the number of normal queue operations executed (each costs one unit) plus the number of *percolate-up*/*percolate-down* operations executed for each manipulation of the priority queue. As expected, in all cases, coordination adds a significant number of total queue operations, but the resulting overhead is more than compensated for by an improved schedule and a reduction in number of facts produced. Alternatively, for MiniMax and N Queens, the number of facts derived is the same for all configurations, which means that the performance seen for those programs arises from reduced memory usage and improved memory locality.

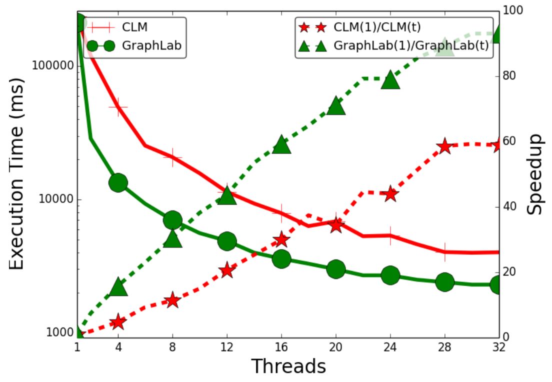


Figure 13: *Experimental results for LBP: comparing CLM and GraphLab.*

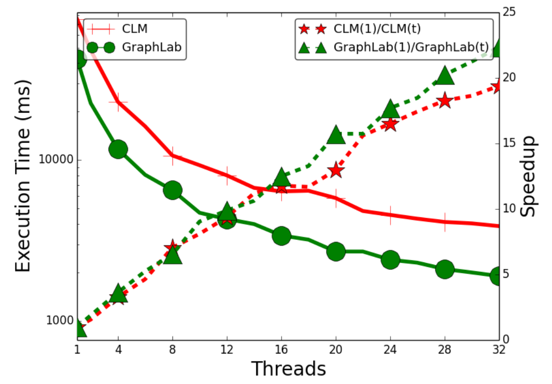


Figure 14: *Experimental results for SBP: comparing CLM and GraphLab.*

Program	**Queue Operations**			
	1	4	8	16
SSSP (Regular)	92.0M	384.9M	454.3M	411.1M
SSSP (Coordinated)	300.4M	875.7M	880.0M	993.8M
HT (Regular)	93.8M	141.4M	145.9M	146.5M
HT (Coordinated)	135.5M	179.1M	212.3M	220.0M
HT (Local)	135.5M	180.1M	198.7M	179.3M
LBP	15.7M	16.0M	11.3M	8.5M
SBP	97.2M	90.0M	84.3M	79.2M
MiniMax (Regular)	16.4M	19.0M	18.8M	20.4M
MiniMax (Coordinated)	52.0M	52.0M	52.0M	52.0M
N Queens (Regular)	10.5K	15.3K	15.2K	20.8K
N Queens (Bottom S.)	3.7K	5.7K	11.1K	18.3K

Table 2: *Total number of queue operations per program. The SSSP program uses the Live Journal dataset.*

8. Conclusions

We have presented a novel way of adding coordination to a declarative language without changing the nature of the language or introducing non-declarative constructs. We took advantage of the fact that CLM uses linear logic, which allows us to derive coordination facts that can be deleted in order to perform actions on the underlying runtime system. Sensing facts allow the programmer to reason about data locality and scheduling of computation in a data-driven fashion. Action facts are then used to affect how the program runs on the underlying system. Our experimental results show that coordination improves execution time, memory usage and scalability of programs. This allows programs to be first written without taking performance into account and then optimized through the judicious use of coordination facts.

Our work makes CLM the ideal framework for rapid development of irregular graph algorithms since CLM programs are short and declarative. CLM not only provides decent performance when compared to other competing frameworks, but it also allows the programmer to experiment with custom scheduling policies that can be later implemented in more efficient frameworks. As further work, we would like to explore CLM's coordination principles in distributed systems, where data locality is far more important than in shared memory systems.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We are also thankful to Vijay Saraswat and Tim Harris for their suggestions during the shepherding process which significantly improved our paper.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[2] F. ARBAB. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 6 2004.

[3] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In *International Conference on Logic Programming*, 2009.

[4] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. In *ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 13–24, New York, NY, USA, 2011.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 66:1–66:11, Los Alamitos, CA, USA, 2012.

[6] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.

[7] M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal nested data parallelism in haskell. In R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150, pages 524–534. 2001.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005.

[9] D. Clarke. Coordination: Reo, nets, and logic. In *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 226–256. 2008.

[10] F. Cruz. *Linear Logic and Coordination for Parallel Programming*. PhD thesis, Carnegie Mellon University, University of Porto, 2016.

[11] F. Cruz, R. Rocha, S. Goldstein, and F. Pfenning. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. *Journal of Theory and Practice of Logic Programming, 30th International Conference on Logic Programming, Special Issue*, pages 493–507, July 2014.

[12] F. Cruz, R. Rocha, and S. C. Goldstein. Design and Implementation of a Multithreaded Virtual Machine for Executing Linear Logic Programs. In O. Danvy, editor, *International Symposium on Principles and Practice of Declarative Programming*, pages 43–53, September 2014.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006.

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[17] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics*, 2009.

[18] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110:32–42, 1994.

[19] E. J. Hoffman, J. C. Loessi, and R. C. Moore. Construction for the solutions of the M queens problem. *Mathematics Magazine*, 42(2): 66–72, 1969.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems*, pages 59–72, 2007.

[21] R. Jagadeesan, G. Nadathur, and V. Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In S. Sarukkai and S. Sen, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 517–528. 2005.

[22] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, and J. M. Hellerstein. Declarative networking: Language, execution and optimization. In *International Conference on Management of Data*, pages 97–108, 2006.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence*, pages 340–349, 2010.

[24] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data*, pages 135–146, 2010.

[25] D. Miller. An overview of linear logic programming. In *in Computational Logic*, pages 1–5, 1985.

[26] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Conference on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.

[27] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 333–344, 2011.

[28] R. S. Nikhil. An overview of the parallel language id (a foundation for pH, a parallel dialect of haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[30] C. Palamidessi, V. Saraswat, B. Victor, and F. Valencia. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *IEEE Computer Society*, 2006.

[31] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, pages 329–400, 1998.

[32] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[33] D. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 375–394, 2012.

[34] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530, 2013.

[35] V. Saraswat and P. Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, 1992.

[36] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 272–285, New York, NY, USA, 1995.

[37] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, Feb. 2013.