Declarative coordination of parallel declarative programs

Abstract

Declarative programming has been hailed as a promising approach to parallel programming since it hides the implementation details of parallelism away from the programmer. However, its advantage has also been its downfall as it leaves the programmer with no straightforward way to optimize programs for performance. In this paper, we introduce Coordinated Linear Meld, a concurrent forward-chaining linear logic programming language, with a declarative way to coordinate the execution of the program allowing the programmer to change computation scheduling and data layout. Our approach allows the programmer to write declarative parallel programs and then optionally use coordination to fine tune programs, while keeping the program fully declarative and amenable to correctness proofs.

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

Writing parallel programs in sequential languages is hard because manipulating shared state using multiple threads may result in data race conditions. Such issues are handled with low level constructs such as locks and condition variables, requiring a fair amount of effort to get right. Declarative programming has been hailed as a solution to this issue, since the problem of implementing the details of parallelism is moved from the programmer to the compiler and runtime. 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. The programming paradigm has been adopted with huge success in domain specific languages such as SQL and MapReduce [Dean and Ghemawat 2008]. Although general declarative languages have yet to be as successful, the future looks promising for this particular approach.

The problem with declarative programming is that it leaves little to no programmer control over how execution is scheduled or how data is laid out, making it hard to achieve necessary 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 about the program that a compiler cannot easily deduce. Such information could make execution better in terms of run time time, memory usage, and scalability.

In this paper, we introduce Coordinated Linear Meld (CLM), a declarative language that extends the Linear Meld (LM) language [Cruz et al. 2014a,b] with coordination facts that give programmer control over scheduling and data placement. LM is a linear logic programming language designed for programs that operate on graphs. The use of linear logic [Girard 1987] supports structured manipulation of mutable state. In LM, computation is divided so that each node of the graph computes independently and is allowed to "communicate" with other nodes. Both computation and communication happen through the derivation of logical rules (which make up the program).

The CLM language features coordination primitives that can be used in the same way as any other primitive, in other words, they are specificied with the same syntax and semantics as the rest of the programming language. These coordination primitives can be used to improve program execution based on the state of the program and the underlying machine. The first kind of coordination facts 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 facts are *action facts* that when deleted in rules are used to apply a scheduling operation 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 place nodes in different threads.

To the best of our knowledge, this is the first time that a declarative language allows control over execution while staying declarative and without resorting to meta-language constructs. This is crucial to our goal of being able to prove programs correct since proofs can be constructed even in the presence of coordination facts. After briefly discussing related work, we present an overview of the base language, with an example. In Section 4, we introduce the current set of coordination facts, followed by a description of the changes required to implement the desired coordination mechanisms. In Section 6 we present several applications and show coordination can improve programs without destroying clarity or provability.

2. Related Work

There has been an increasing interest in declarative languages. MapReduce [Dean and Ghemawat 2008], for instance, is a popular data-centric 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 [Olston et al. 2008] have been developed. PigLatin builds on top of MapReduce and allows the programmer to write complex dataflow graphs, raising the abstraction and easy of programmability of MapReduce programs. An alternative to PigLatin/MapReduce is Dryad [Isard et al. 2007] that allows programmers to design arbitrary computation patterns using the DAG abstraction. It combines computational vertices with communication channels (edges). Dryad programs are automatically scheduled to run on multiple computers/cores and data is partitioned automatically during runtime.

Many programming languages follow the so-called *coordination paradigm* [Papadopoulos and Arbab 1998], 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 [Ahuja et al. 1986] is arguably the most famous coordination model. Linda implements a data-driven coordination model and features a *tuple space* that can be manipulated using the following coordination directives: `out(t)` to write a tuple `t` into the tuple space; `in(t)` to read a tuple using the template `t`; `rd(t)` to retrieve a copy of the tuple `t` from the tuple space; and `eval(p)` to add a process `p` in the tuple space and execute it in parallel. Linda is be implemented on top of many popular languages by simply creating a communication and storage mechanism for the tuple space and then adding the directives as a language library.

Another early coordination language is Delirium [Lucco and Sharp 1990]. Unlike Linda which is embbeded into another language, Delirium actually embeds operators written in other languages inside the Delirium language. The advantages of Delirium are improved abstraction and easier debugging because sequential operators are isolated from the coordination language.

Linda and Delirium are 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. But even more importantly, CLM differs from those languages because coordination acts on data instead of processing units. The abstraction is then raised by considering data and algorithmic aspects of the program instead of focusing how processing units are used. Furthermore, the LM language is both a coordination language and a computation language and there is no distinction between the two components.

Forward-chaining logic programming came into its own with the Datalog language [Ullman 1990]. Traditionally used in deductive databases, it is now being increasingly used in different fields such as distributed networking [Loo et al. 2006], sensor nets [Chu et al. 2007] and cloud computing [Alvaro et al. 2010]. The LM language was inspired by Meld [Ashley-Rollman et al. 2009], a Datalog-like language for programming distributed emsembles of modular robots. Meld introduced the idea of sensing and action facts in order to sense and act on the outside world, respectively. LCM uses sensing facts to sense the underlying system that the program is running on and action facs act on that system.

3. Base Language

Coordinated Linear Meld (CLM) is a forward-chaining linear logic programming language based on LM [Cruz et al. 2014a]. Programs consist of a set of *rules* and a *database of facts*. Rules such as `a(X), b(Y) -o c(X, Y)` can be read as follows: if fact `a(X)` and fact `b(Y)` exist in the database then `c(X, Y)` is added to the database. The expression `a(X), b(Y)` is called the *body* of the rule and `c(X, Y)` 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 `X` and `Y`, substitued for the arguments. Since LM uses linear logic as its foundation, we distinguish between *linear* and *persistent facts*. Linear facts are deleted during the process of deriving rule, while persistent facts are not.

Program execution starts by adding the *axioms* (the initial facts) of the program to the database. Next, the rules are recursively applied and the database is updated by adding new facts or deleting facts used during rule derivation. When no more rules are applicable, the program terminates.

CLM has been designed for writing programs that operate on graphs. CLM partitions the database by using the first argument of each fact. The first argument has type *node* and represents a node in the graph. To achieve concurrency, our implementation partitions the database by the first argument, e.g., the fact `f(@1, 2)` is stored in node `@1`, while fact `p(@2)` is stored in node `@2`. CLM restricts the body of every rule to facts with the same node so that nodes can derive rules independently. Although the body is restricted, the head of the rule may refer to any node as long as that node is refered 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 CLM implicitly parallel because nodes are able to compute independently. This makes CLM non-deterministic since nodes can be picked to run in any order, affecting which rules are applied and which facts are deleted or derived.

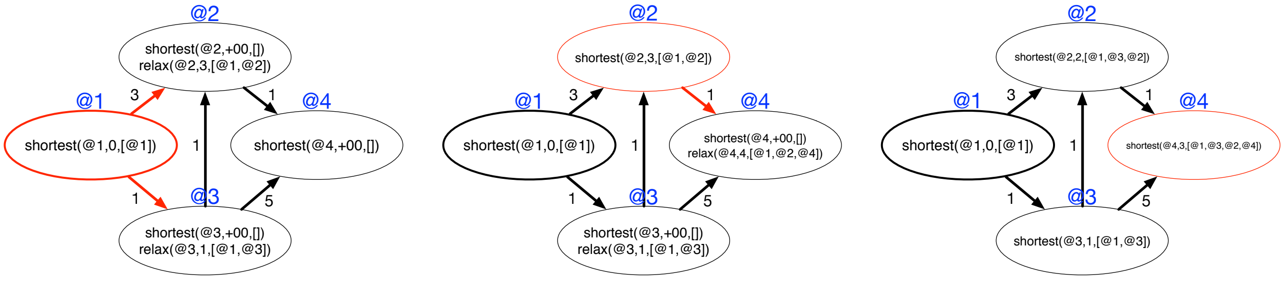
To make these ideas concrete, we present a simple program: the single source shortest path program (SSSP). Later in the paper, we add coordination facts to improve the execution of the program.

The SSSP program in Fig. 1 starts (lines 1-3) with the declaration of the predicates. 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). The 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 is 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 initial distance from `@1` to `@1` to be 0.

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

Figure 1: Single Source Shortest Path program code.

The first rule of the program (lines 11-14) reads as following: 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 14-15) using a *comprehension*. The comprehension iterates over the edges of node `A` and derives a new `relax` fact for each node `B` with the distance `D2 + W`, where `W` is the weight of the edge. For example, in Fig. 2 (a) we apply rule 1 in 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`.



(a) (b) (c)

Figure 2: Graphical representation of the SSSP program. Figure (a) represents the program after propagating initial distance at node `@1`, followed by Figure (b) where the first rule is applied in node `@3`. Figure (c) represents the state of the final program, where all the shortest paths have been computed.

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 old 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 either apply rules in either node `@2` or node `@3`. This decision depends largely on implementation factors such as node partitioning and number of threads in the system. Still, it is easy to prove that no matter the scheduling used, the final result present in Fig. 2 (c) is achieved.

4. Coordination

The SSSP program present in Fig. 1 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 4 `relax` facts are generated. If we had decided to process nodes in order `@1`, `@2`, `@4`, `@3`, `@4`, `@2`, then 6 `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 [Dijkstra 1959]. 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.

Coordination facts allow the programmer to change how the run time 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 on 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.

There are two kinds of coordination facts: *sensing* and *action* facts. Sensing facts are used to sense information about the underlying runtime system, including node placement and node scheduling. Action facts are used to apply a coordination operations on the runtime system.

4.1 Scheduling Facts

In order to allow 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 such priority values. By default, nodes do not have any priority and can be picked in any order. In our implementation, we use a FIFO approach because this tends to work better for most programs since older nodes tend to a higher number of unexamined facts.

We have two kinds of priorities: a *temporary priority* and a *default priority*. A temporary priority momentarily changes the default priority D of a node, so that once the node is done, the priority will default back to D. Initially, all nodes have a default priority of 0.

The following list presents the action facts available to manipulate the scheduling decisions of the system:

- `set-priority(node A, float F)`: This sets the temporary priority of `A` to `F`. If `A` has a priority `F'`, we only change the priority if `F` is higher. 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`.
- `schedule-next(node A)`: Changes the temporary priority of node `A` to be 1 unit higher than the current highest priority.
- `set-default-priority(node, float)`: Sets the default priority of the node.
- `stop-program(node A)`: Immediately stops the execution of the whole program.

LM provides the sensing fact `priority(node A, node B, float P)` in order to sense the priority `P` of node `B` (at node `A`). Sensing facts are only used in the body of the rules in order to fetch information from the runtime system. Note that when sensing facts are used to prove new facts, they are re-derived automatically. [1]

4.1.1 Partitioning facts

We provide several coordination facts for dealing with node partitioning among the available threads executing the program. In terms of action facts, we have the following:

- `set-cpu(node A, int C)`: Moves node `A` to thread `C`.
- `set-affinity(node A, node B)`: Places node `B` in the thread of node `B`.
- `set-moving(node A)`: Allows node `A` to move freely between threads.
- `set-static(node A)`: Forces node `A` to stay in the same thread indefinitely.

For sensing facts, we have the following set of coordination facts:

- `cpu-id(node A, node B, int C)`: Store at node `A` which thread, `C`, `B` is actually running on.
- `moving(node A, node B)`: Fact available at node `A` if `B` is allowed to move between threads.
- `static(node A, node B)`: Fact available at node `A` if `B` is not allowed to move between threads.

[1] The observant reader will know that all the coordination facts are linear and XXX Logically, `set-priority` and `set-default-priority` update the value of `priority` facts, but this done automatically by the runtime system.

5. Implementation

The implementation of LM includes a compiler and a virtual machine that runs byte-code. The virtual machine uses 32 registers for operations and executes procedures to iterate over the database in order to match facts. The implementation of the original machine is described in [Cruz et al. 2014b]. Here, we review the most important details of that paper, while focusing on the changes for efficiently implementing coordination.

5.1 Compilation

The compiler translates each rule to a procedure and a list of facts that need to exist to satisfy the rule. This procedure is executed by the virtual machine whenever all the facts mentioned in the body exist. The procedure loops over all possible combinations of the rule, retrieving facts from the database, performing join operations and then consuming and deriving facts. Optimizations such as join optimizations (to support efficient rule filtering) and fact updates are implemented by the compiler.

5.1.1 Coordination directives

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 special instructions that inspect the state of the virtual machine. For example, the directive `node-priority` will inspect the target node, retrieve the current priority and assign the priority to a register. Coordination facts in the head of the rule are also implemented as special instructions of the virtual machine, but they perform some action, instead of being added to the database as facts. Semantically, action facts are like any other. However, since they are immediatelly used by the machine, there is no need to store them in the database, avoiding unnecessary allocations and deallocations.

5.2 Execution

The virtual machine is implemented in C++11 and uses the threading system from the standard library to implement multithreading. Each thread 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. Inactive nodes have no work to do. When a new fact is derived for an inactive node, the node is made active and placed on the appropriate queue of the appropriate thread. Threads do useful work by processing active nodes from their queues. Whenever a thread becomes idle, it attempts to steal nodes from a random thread. If unsucessful, the thread becomes idle and waits for program termination while periodically attempting to steal nodes.

5.2.1 Nodes

A node is represented as a collection of facts (per predicate) and an indexing structure that keeps track of the available facts and potential candidate rules. We need a separate indexing structure per node since rules run local to each node.

Facts need to be stored efficiently because the virtual machine instructions perform searches on the database by fixing arguments of a predicate to concrete values. Each predicate is stored using one of the following data structures:

- *Tries* are used exclusively to store persistent facts.
- *Doubly Linked Lists* are used to store linear facts. We use a double linked list because we need to efficiently add and remove facts.
- *Hash Tables* are used to improve lookup when linked lists are too long and when we need to do searches filtered

by a fixed argument. The virtual machine decides which arguments are best to be indexed and then uses a hash table indexed by the appropriate argument.

In order to facilitate the implementation of coordination, we added a state variable for each node. 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 waiting in some queue 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 moving from one queue to another queue.

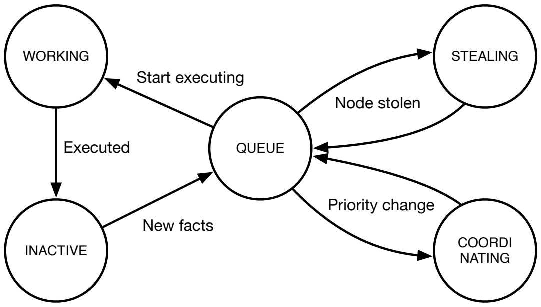


Figure 3: The node state machine as represented by the state variable. During the lifetime of a program, each node goes through different states as specified by the state machine.

Each node is protected by a main spinlock that allows threads to change node attributes: incoming facts, owner thread, state variable and locality information. A database spinlock is locked whenever the node enters into the **working** state. If a node sends facts to another node placed in another thread, the current thread first attempts to lock the database lock in order to update the indexing structures of the target node, otherwise it adds the facts to the list of incoming facts that are later processed by the owner thread.

5.2.2 Threads

Threads pop active nodes from their queues and execute new candidate rules. In order to reduce inter-thread communication, node priorities are implemented at the thread level. This means that, when a thread picks the highest priority node, it is actually the highest priority from the set of nodes owned by the thread and not the highest priority node in the whole program. To allow node priorities, we use 2 two pairs of queues per thread: a pair of doubly linked lists known as the *standard queue* and a pair of min/max heap known as the *priority queue*.

The regular 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), 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 the priority of nodes allows threads to change the priority of nodes.

The `next` and `prev` pointers of the regular queue are part of the node structure in order to save space. These pointers are also used as the index in the priority queue and current priority, respectively.

Both the regular and priority queue are actually implemented as 2 queues each. The *static queue* contains nodes that are local to a thread (and thus cannot be stolen) and the *dynamic queue* maybe accessed by other threads during node stealing.

5.2.3 Communication

Threads synchronize with each other using mutual exclusion. We use a spinlock in each queue to protect queue operations. Given threads T_1 and T_2, we enumerate the most important synchronization intensive places in the virtual machine:

New facts: When a node executes in T_1 and derives facts to a node in T_2, T_1 first buffers the facts and then sends them to the target node. Here, it checks if the node is currently **idle** and then synchronizes with T_2 to add the node to the T_2's queue.

Thread activation: If T_2 is inactive when adding facts to a node in T_2, T_1 also synchronizes with T_2 to change T_2's state to *active*.

Node stealing: T_1 synchronizes with T_2 when it attempts to steal nodes from T_2 by removing half of the nodes from one of T_2's queues.

Coordination: If T_1 needs to perform coordination operations to a node in T_2, it may need to sychronize with T_2 during priority updates in order to move the node in T_2's queues.

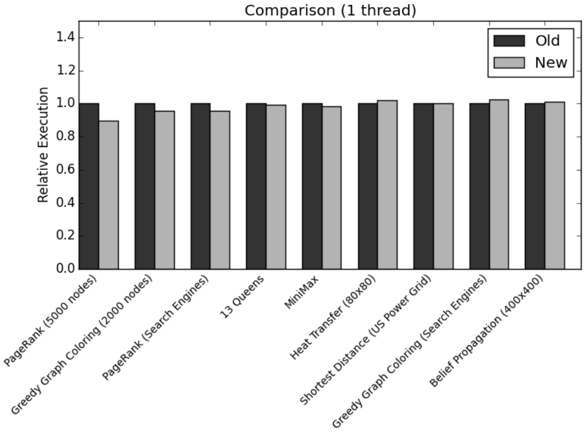
5.2.4 Garbage collection

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.

5.2.5 Coordination overhead

In the paper [Cruz et al. 2014b], the LM virtual machine was measured and compared against implementations of similar programs but written in other languages. For instance, the LBP program was found to be around 2 times slower than GraphLab, while N Queens program was found to be around 10 to 15 slower than a sequential C implementation. Some of those programs were compared against a Python implementations and LM faired fairly well. In this section, we measure the impact of the coordination mechanisms we have implemented for the new virtual machine.

Fig. 4 shows the comparison between the original virtual machine and the new virtual machine with coordination mechanisms. All the programs benchmarked do not use any kind of coordination. As hoped for, there is no degration of performance by adding coordination mechanisms. Since we also performed general performance improvements, programs such as PageRank and Greedy Graph Coloring programs now perform better than before.



(a)

Figure 4: Measuring the overhead of coordination mechanisms. The **Old** bars represent the performance of the original virtual machine, while **New** is the relative performance of the new version that includes coordination mechanisms.

6. Applications

To better understand how coordination facts are used, we next present some programs that take advantage of them.

In our experimental setup, we used a machine with four AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores) and 64 GB of DDR-2 667MHz (16x4GB) RAM, running GNU/Linux (kernel 3.15.10-201 64 bits). We compiled our virtual machine using GCC 4.8.3 (g++) with the flags `-O3 -std=c++11 -fno-rtti -march=x86-64` [2].

6.1 Single Source Shortest Path

In Section 3, we said that the SSSP program has an optimal scheduling if the nodes with the shortest distances are selected to run before other nodes. The coordinated version of the SSSP program is presented in Fig. 5 and it uses one coordination fact, namely `set-priority` (line 14). Note that we also use a global program directive to order priorities in ascending order (line 5).

When using a single thread, the algorithm behaves like Dijkstra's shortest path algorithm [Dijkstra 1959]. However, when using multiple threads, each thread will pick the smallest 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 avoid unnecessary work. XXX

```
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   priority @order asc.
6
7   shortest(A, +00, []).
8   relax(@1, 0, [@1]).
9
10  shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)
11     -o shortest(A, D2, P2),
12        {B, W | !edge(A, B, W) |
13           relax(B, D2 + W, P2 ++ [B]),
14           set-priority(B, float(D2 + W))}.
15
16  shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)
17     -o shortest(A, D1, P1).
```

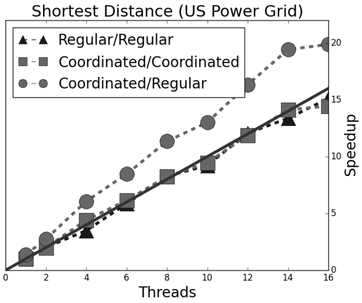
Figure 5: Shortest Path Program.

The most interesting property of the SSSP program presented in Fig. 5 is that it remains provably correct, although it applies rules using smarter ordering. The derivation of `set-priority` does not change the behavior of the logical rules and the code remains declarative.

Figure 6 shows experimental results a SSSP program that computes the SSSP for 20% of the nodes of the graph. There are some situations where unecessary facts are propagated because although the shortest distance is selected, suboptimal distances may be propagated because many SSSP distances are computed at the same time. However, we see a reduction of over 50% in the number of derived facts when using the coordinated version **C** over the regular version **R**. The results also show that the cordinated version using 16 threads is 1.3 times faster than the regular version using 16 threads. In turn, this makes the coordinated version 20 times faster than the regular version using 1 thread. Note that the coordinated program is as scalable as the regular program.

# T	R	C
1	333K	206K
2	300K	210K
4	316K	208K
8	328K	211K
16	343K	212K

(a)



(b)

Figure 6: Experimental results for the SSSP program using US powergrid network. On the left we show the number of facts derived for the regular **R** and coordinated version **C** using a variable number of threads **# T**. On the right, we have the scalability of the regular and coordinated version. The speedup values are computed using the execution time for 1 thread.

6.2 MiniMax

The MiniMax algorithm is a decision rule algorithm for minimizing the possible loss for a worst case (maximum

[2] Implementation available in http://github.com/.../meld

loss) scenario in a zero sum game for 2 (or more) players that play in turns [Edwards 1954].

The algorithm proceeds by building 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 `f(State)` 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 LM, the program starts with a root node (with the initial game state) that 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.

The code in Fig. 7 shows part of the program. 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 for creating new nodes. The rule in lines 12-25 generates new nodes using the `exists` language construct, which creates a new node address `B`, where a new `play` fact is created with the a board. A `parent(B,A)` fact is derived to allow node `B` to communicate with its parent.

As noted before in Section 4.1, the runtime schedules nodes by XXX, in a FIFO ordering for nodes, meaning that when children are expanded, we first expand the children and then the children of the children. In this case, the defined ordering is not optimal for since the complete tree needs to be expanded before computing the scores at the leaves. The memory complexity of the program is $\mathcal{O}(n)$, where n is the number of nodes in the tree.

```
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-static(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. (set-default-priority(B, float(Depth + 1)),
23        play(B, Board ++ [P | Xs], next(P), Depth + 1),
24        expand(A, Board ++ [0], Xs, N + 1, P, Depth),
25        parent(B, A)).
26
27 expand(A, Board, [C | Xs], N, P, Depth)
28 C <> 0
29   -o expand(A, Board ++ [C], Xs, N, P, Depth).
```

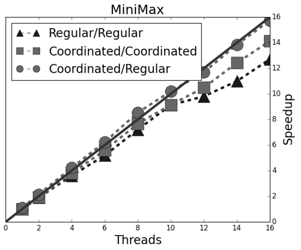
Figure 7: MiniMax: checking if the game has ended and expanding the tree

With coordination, we set the priority nodes to be the same as the depth so that the tree is expanded in a depth-first fashion (lines 15 and 22). The memory complexity of the program then becomes $\mathcal{O}(dt)$, where d is the depth of the tree and t is the number of threads. As an example, consider a system with 2 threads, T_1 and T_2, where T_1 first expands the root node and then the first child. Since T_2 is idle, it steals half of the root's children nodes and starts expanding one of the nodes in a depth-first fashion. Since threads prioritize deeper nodes, the scores of the first leaves are immediatelly 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-static` (line 14), so that nodes after 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 to be critical in distributed systems.

# T	R	C
1	11.80GB	0.50MB
2	12.19GB	1.45MB
4	13.82GB	2.35MB
8	14.87GB	4.36MB
16	13.79GB	8.1MB

(a)



(b)

Figure 8: Memory usage and scalability of the regular and coordinated versions of MiniMax.

In Fig. 8 we compare the memory usage and scalability of the coordinated MiniMax against the regular MiniMax. The coordinated version sees a huge reduction of memory usage, needing at most 8MB for 16 threads, while the regular version needs almost 14GB. Note that as the number of threads goes up, memory usage also goes up. This is an artifact of our parallel memory allocator that allocates large chunks of memory beforehand. In terms of scalability, our experimental results show a 14-fold speedup for the coordinated version against a 12-fold for the regular version when using 16 threads. When comparing the two versions directly, there is a 20% run time reduction when using 16 threads in the coordinated version.

6.3 Heat Transfer

In the Heat Transfer algorithm (HT), we have a graph where heat values are exchanged between nodes. The goal of the program is to exchange heat to a point where new heat value $\delta = |H_i - H_{i-1}| \leq \epsilon$ for all nodes of the graph. The algorithm works asynchronously since heat values can be updated by using new partial information coming from neighbor nodes. This increases parallelism since nodes do not need to synchronize between iterations.

Fig. 9 shows the HT logical 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 heat values 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 multiple large deltas will have more priority. To further improve scalability, we added the third rule to ignore to not send small δ values if the target node is in another thread. Note that the LM compiler will merge the three rules in Fig. 9 into a single one that handles the 3 branches.

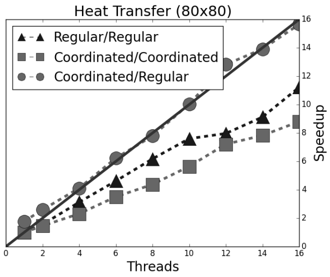
```
new-heat(A, New, Old),
Delta = fabs(New - Old),
Delta > epsilon
   -o {B, W | !edge(A, B, W) |
         new-neighbor-heat(B, A, New),
         update(B), add-priority(B, Delta)}.

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

new-heat(A, New, Old)
fabs(New - Old) <= epsilon,
cpu-id(A, B, C1),
cpu-id(A, A, C2),
C1 <> C2
   -o 1. // nothing is derived
```

Figure 9: Coordination code for the Heat Transfer program. We updated the first rule to increase the priority of neighbor nodes and added the third rule to not send small δ if the neighbor is in another thread. The code logic remains exactly the same as before, however bigger changes in heat values are now propagated faster.

Fig. 10 presents the scalability results for the uncoordinated and coordinated version. The dataset used is a square grid with an inner square with high heat nodes. With 1 thread, there's a 33% reduction in run time, while for 16 threads, there's a 18% reduction.



(a)

Figure 10: Experimental results for the HT program. The coordinated version is, on average, 30% faster than the regular version although it has a slightly worse scalability due to a reduction in work available.

6.4 Splash Belief Propagation

Randomized and approximation algorithms can obtain significant benefits from coordination directives because although the final program results will not be exact, they follow important statistical properties and can be computed faster. Examples of such programs are Loopy Belief Propagation. Loopy Belief Propagation [Murphy et al. 1999] (LBP) is an approximate inference algorithm used in graphical models with cycles. In its essence, LBP is a sum-product message passing algorithm where nodes exchange messages with their immediate neighbors and apply some computations to the messages received.

LBP is an algorithm that maps very well to the graph based model of LM. 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 [Gonzalez et al. 2009] developed an optimal algorithm to compute this algorithm by first building a tree and then updating 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 other nodes in the tree must have a non-zero priority. Note that the priorities are updated whenever a neighbor updates their belief. These *splash trees* are built iteratively until we reach convergence. In Fig. 11 we represent two threads creating two different splash trees. It is encouraged to create several splash trees concurrently as long as we have threads to create them.

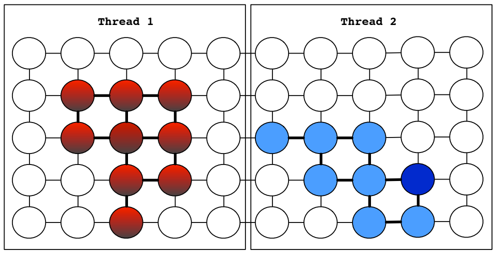


Figure 11: Creating splash trees for belief propagation. Each threads picks the highest priority node and creates a tree from that node. The belief values are updated in two phases: first from the leaves to the root and then from the root to the leaves.

The code for Splash Belief Propagation (SBP) in Fig. 12 presents the coordination code for LBP. Please note that we just appended the code in Fig. 12 to a working but unoptimized version of the algorithm, every other logical rule remains the same. We add new logical ruels that coordinate the creation and execution of the splash trees:

Tree building : Each node has a `inactive` fact that is used to start the tree building process. When the highest priority node is picked, a `tree` is created that will navigate through the tree. In lines 15-21, we use an *aggregate* [Cruz et al. 2014a] 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` (line 21) and appended to list `Next`.

First phase : In the third rule (lines 11-12), when the number of nodes of the tree reaches a certain limit, a `first-phase` is generated to update the beliefs of all nodes in the tree, starting from the leaves and ending at the root As the nodes are updated, an `update` fact is derived to update the belief values (line 35).

Second phase : In the second phase, the computation of beliefs is performed from the root to the leaves and the belief values are updated a second time (line 42).

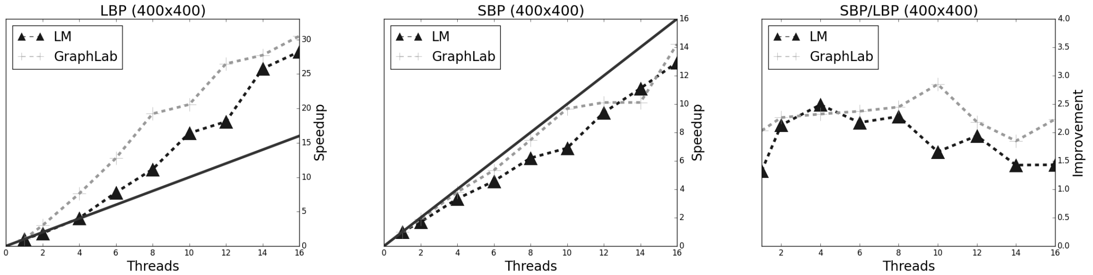
Note that the `set-static` and `set-cpu` action fact are used in line 2 in order to (1) force nodes to stay in the thread and (2) to partition nodes as a grid of threads. This setups a well defined area of nodes for threads to build splash trees on.

In this program, coordination assumes a far more important role than we have seen before. Coordination rules fully drive the behavior of the algorithm and although the final result of the algorithm is identical to the original algorithm (minus probabilitistic errors), SBP works in a very different way. A system that also implements SBP is GraphLab [Low et al. 2010], a C++ framework for writing machine algorithms. GraphLab provides the splash scheduler for these types of inferences as part of the framework. This particular scheduler is around 350 lines of complicated C++ code. With our coordination facts, it is possible to create fairly complicated coordination patterns with only 12 simple logical rules.

```
1  !coord(A, X, Y), start(A)
2     -o set-static(A), set-cpu(A, grid(X, Y)).
3
4  // TREE BUILDING
5  // expand tree by adding neighbor nodes
6  inactive(A), tree(A, All, Next) -o expand-tree(A, All, Next).
7  // start tree since we do not have one
8  inactive(A), @priority(A, A, P), P > 0.0
9     -o expand-tree(A, [A], [A]).
10 // end tree building
11 expand-tree(A, All, Next), length(All) >= maxnodes
12    -o first-phase(A, All, reverse(All)).
13 // expand tree
14 expand-tree(A, All, [A | Next]), length([A | Next]) < maxnodes
15    -o [collect => L | Side | !edge(A, L, Side),
16          0 = count(All, L), // L is not in All
17          0 = count(Next, L), // L is not in Next
18          @priority(A, L, P), P > 0.0,
19          @cpu-id(A, L, Id1),
20          @cpu-id(A, A, Id2), Id1 = Id2 |
21          send-tree(A, All, Next ++ L)].
22
23 send-tree(A, All, [])
24    -o first-phase(A, All, reverse(All)).
25 send-tree(A, All, [B | Next])
26    -o schedule-next(B),
27       tree(B, All ++ [B], [B | Next]).
28
29 // FIRST PHASE
30 first-phase(A, [A], [A]) -o second-phase(A, [], A).
31 first-phase(A, [A, B | Next], [A])
32    -o update(A), schedule-next(B),
33       second-phase(B, [B | Next], A).
34 first-phase(A, All, [A, B | Next])
35    -o update(A), schedule-next(B),
36       first-phase(B, All, [B | Next]).
37
38 // SECOND PHASE
39 second-phase(A, [], _)
40    -o set-priority(A, 0.0), inactive(A), update(A).
41 second-phase(A, [A], Back)
42    -o update(A), inactive(Back),
43       inactive(A), set-priority(A, 0.0).
44 second-phase(A, [A, B | Next], Back)
45    -o update(A), inactive(Back), schedule-next(B),
46       second-phase(B, [B | Next], A).
```

Figure 12: Coordination code for the Splash Belief Propagation program. LM needs 50 lines of rules to implement splash trees, while GraphLab needs a total of 350 lines of C++ to implement the same functionality. Note that when a linear fact is prefixed by @, it indicates that the fact is going to be re-derived.

We measured the behavior of LBP and SBP for both LM and GraphLab. Fig. 13 shows that both systems have very similar behavior when using a variable number of threads. Note that GraphLab avoids message passing between nodes since new neighbor belief updates are changed immediatelly, resulting in slightly better scalability for GraphLab.



(a) (b) (c)

Figure 13: Experimental results for LBP and SBP. Figure (a) shows the scalability of LBP for both LM and GraphLab and Figure (b) shows the scalability of SBP. Figure (c) presents the improvements seen in SBP against LBP, where SBP runs, on average, 1.5 to 2.5 times faster than LBP.

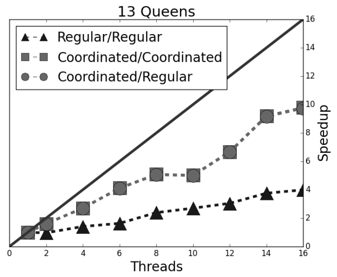
6.5 N Queens

The N-Queens puzzle is the problem of placing N chess queens on an NxN chessboard so that no pair of two queens attack each other [Hoffman et al. 1969]. The challenge of finding all the distinct solutions to this problem is a good benchmark in designing parallel algorithms.

The LM solution considers the chess board as a graph where nodes exchange valid configurations between each other. First, we start with the empty state in the first row of the chess board. 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. If valid, that configuration is then sent, recursively, to the next row, until all rows are traversed. At the end of the program, the nodes at the bottom row will have all the valid configurations.

Since computation goes from the top row to the bottom row, not all placements of nodes on threads may be equally performant. This is specially true because the bottom rows tend to perform the most work. The best placement is then to split the board vertically, so that, in optimal conditions, each thread gets the same number of columns. Our experiments show that, on shared memory, it does not matter much if we start with a bad placement since node stealing overcomes those issues by improving load balancing dynamically. But what if we want run the program on a distributed system, where moving nodes between workers can be expensive? We measured the performance of the program without node stealing in order to assess how important data placement would be for such systems. We used the default partition of the program as it is, which results in the board being split by rows and another configuration using the coordination action `set-cpu(A, vertical(X, Y))` that forces nodes to be moved to another thread using a vertical partitioning. Note that `X` and `Y` are the coordinates of the node in the chessboard.

Figure 14 shows the performance results of coordinating the placing of nodes on threads without work stealing. There is a noticiable performance improvement when using an optimal data placement strategy. Note however that in the plot we do not take into account the costs of moving nodes between workers. In a real system, the performance gap may not be as wide since in the worst case there may be 144 moves.



(a)

Figure 14: Experimental results for the 13 Queens program. The coordinated version is on average 2 times faster than the version with bad data placement.

7. 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 LM uses linear logic, which allows us to derive coordination facts that can be deleted in order to perform actions in the underlying runtime system. In the other hand, sensing facts allow the programmer to reason about locality and scheduling of computation in a data-driven fashion. We have presented several applications where a judicious use of coordination can lead to better performance behavior.

We would like to explore this coordination paradigm in distributed systems, where data locality is far more important than in shared memory systems. We also think that LM may be useful as a glue distributed/parallel language since it can easily model any computation pattern using graphs. Furthermore, we think that other declarative paradigms would benefit from using a similar approach. For instance, in functional programming, one may consider annotation functions that pick the right granularity for each problem.

References

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

P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems (EuroSys)*, pages 223–236, 2010.

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 (ICLP)*, 2009.

D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 175–188, 2007.

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 (ICLP 2014), Special Issue*, pages 493–507, July 2014a.

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, *Proceedings of the International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, pages 43–53, Canterbury, UK, September 2014b. ACM Press.

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

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

W. Edwards. The theory of decision making. *Psychological Bulletin*, 41:380–417, 1954.

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

J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS)*, 2009.

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.

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 (EuroSys)*, pages 59–72, 2007.

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 (SIGMOD)*, pages 97–108, 2006.

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 (UAI)*, pages 340–349, 2010.

S. Lucco and O. Sharp. Delirium: an embedding coordination language. In *Supercomputing '90., Proceedings of*, pages 515–524, Nov 1990.

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 (UAI)*, pages 467–475, 1999.

C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

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

J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. 1990.