Declarative coordination of parallel declarative programs

Abstract

Declarative programming is good for parallel programming but makes it hard for the programmer to fine tune scheduling and data layout of the program. some programs require more information in order to execute better. with our approach we permit the programmer to write declarative programs but then optimize them later using sensing facts and action facts that understand and change how the programs are executed.

Linear Meld is a concurrent forward-chaining linear logic programming language where logical facts can be asserted and retracted in a structured way. In Linear Meld, a program is seen as a database of logical facts and a set of derivation rules. The database of facts is partitioned by the nodes of a graph structure which leads to

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

General Terms Design, Languages, Performance

Keywords Parallel Programming, Linear Logic, Virtual Machine, Implementation

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 away from the programmer to the compiler and runtime of the language. 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 more general declarative languages have yet to be as successful, the future looks promising for this particular approach.

We argue, however, that declarative programming leaves little to no programmer control over how execution is scheduled or how data is laid out. This introduces some 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 does not easily understand. In turn, such information would make execution better in terms of run time time, memory usage and scalability.

In this paper, we extend Linear Meld (LM), a declarative language by [Cruz et al. 2014a,b], with coordination facts that give programmer control over scheduling and data placement. LM is a linear logic programming language intended for programs over graph structures and with support for structured manipulation of mutable state using linear logic [Girard 1987]. In LM, the nodes of the graph perform computation through derivation of rules and communicate with other nodes when logical facts are derived.

The operational semantics of LM allow for plenty of non-determinism since nodes execute independently and can be executed with any specific order. Since LM uses mutable state, the order in which nodes are executed may be crucial in terms of performance. We extended the language with coordination facts that can be used as a regular fact in the program logic to improve program execution. The first kind of coordination facts are called *sensing facts* and are used to sense information about data placement (where is the node being executed). This allows the programmer to write logical rules that depend on the current state of the program. The second kind of coordination facts are *action facts* that, when deriving, will be consumed to apply a scheduling operation during execution. This allows the programmer to prioritize node computation or place nodes in different threads. Both kinds of facts are *linear facts* since they are consumed and/or change.

To the best of our knowledge, this is the first time that a declarative language allows control over execution without a significant change in the language itself and without using non-declarative operations. This proves specially important since programs can proven to be correct even in the presence of coordination facts. Our other two main contributions are as follows: a novel way to coordinate declarative programs through user control that introduce measurable performance improvements; several application where we add programmer control to declarative programs in order to make them perform better.

2. Related Work

3. Base Language

Linear Meld (LM) is a forward-chaining linear logic programming language [Cruz et al. 2014a] with roots on Datalog [Ramakrishnan and Ullman 1993]. Programs are written as a set of rules of the form `a(X), b(Y) -o c(Z)` and are read as following: if `a(X), b(Y)` (the body of the rule) is satisfied using some facts `a(X)` and `b(Y)` then `c(Z)` (the head of the rule) is true. The facts used during rule derivation are placed in a *database of facts*. Logical facts are composed of a predicate and a tuple of concrete values, representing the arguments. Since LM uses linear logic as its foundation, rule derivations may delete facts used in the body of the rule. 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 with new and deleted facts. When nore more rules are applicable, the program terminates.

LM has been designed for writing graph-based programs. LM accomplishes this by partitioning the logical facts by the first argument (the *node*) and by restricting the body of every rule to only refer to the same first argument. This allows rules to be executed using only facts of a single node. Although the body of rules is restricted, the head of the rule may refer to any other node as long as the node variable is refered somewhere in the body of the rule. This allows derivations of facts in other nodes. The whole database is then composed of smaller databases (the nodes) and rule derivations allows smaller databases to derive facts in other databases, creating edges in the graph.

In order to fully understand how the language works, we present a very simple algorithm: 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 kinds of facts used in the program. The first predicate, `edge`, specifies edges between nodes of the graph, where the third argument represents the weight of the edge. The `route` modifier informs the compiler about the structure of the program graph. The predicates `shortest` and `relax` are specified as linear facts and thus are deleted when used in rules. The idea of the algorithm is to compute 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 with the initial distance from `@1` to `@1`.

```
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 relax(A, D1, P1), shortest(A, D2, P2),
12 D1 < D2
13    -o shortest(A, D1, P1),
14       {B, W | !edge(A, B, W) |
15          relax(B, D1 + W, P1 ++ [B])}.
16
17 relax(A, D1, P1), shortest(A, D2, P2),
18 D1 >= D2
19    -o path(A, D2, P2).
```

Figure 1: Single Source Shortest Path program code.

The first rule of the program (lines 11-15) replaces the current path in `shortest` with a shorter one in `relax`. The rule deletes both `relax` and `shortest` facts and derives a new `shortest` fact. In lines 14-15, we have a *comprehension* where the program iterates over the edges of node `A` and derives a new `relax` fact at `B` with the new shorter distance plus the weight of the edge. For instance, 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`.

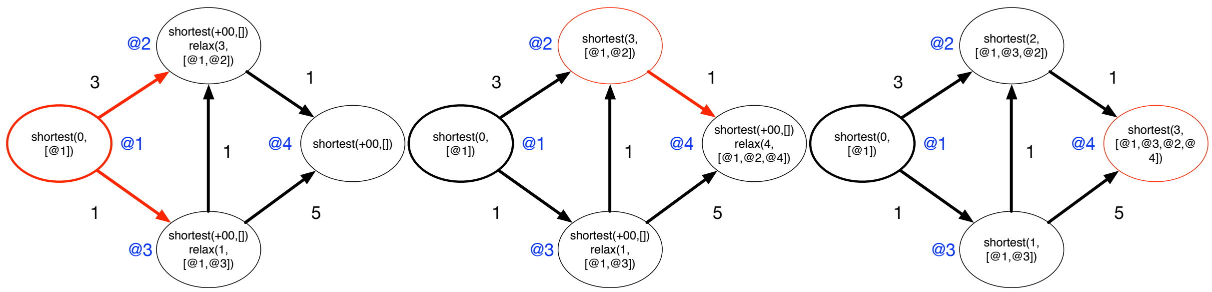
The second rule of the program (lines 17-19) deals with the case where the new distance in `relax` is not better than the current one, so it is thrown away and the current distance is kept.

There 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`. In our implementation, we partition the graph into subgraphs that are processed by multiple threads of execution. Eventually, it is no longer to possible to apply rules and the final result present in Fig. 2 (c) is achieved.

4. Coordination

Since LM uses linear logic and supports deletion of facts, the order in which nodes are scheduled can impact the performance and even the results of the program.

The SSSP program present in Fig. 1 is concise and declarative but its performance may depend in the order in which



(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.

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 computational ordering 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. Therefore, 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.

We introduce the concept of *coordination facts* that allow the programmer to change how the run time schedules nodes and partitions the nodes among threads of execution. Coordination facts can be used in the body or head of the rules to allow the programmer to reason and change how execution is to be done. In this fashion, we keep the language declaratively because we reason logically about the state of execution, without the need to introduce extra-logical operators into the language that would reduce problems when attempting to prove properties about programs.

There are two kinds of coordination facts. The first kind of coordination facts are called *sensing facts* and are used to sense information about data placement (where is the node being executed) or scheduled (what is the priority of the node). The second kind of coordination facts are *action facts* that, when derived, are consumed to apply a coordination operation during execution. These logical rules that use coordination are implemented efficiently in the run time system and are hidden from the programmer.

4.1 Scheduling Facts

We can use action facts to change the order in which nodes are evaluated by adding *priorities*. Node priority works at the thread level so that each thread can make a local decision about which node of the graph to run next. Note that, by default, nodes are picked arbitrarily (a FIFO approach is used).

We have two kinds of priorities: a *temporary priority* and a *default priority*. A default priority is used indefinitely, while a temporary priority is only used once, so that when node with a temporary priority P is picked to run, the priority will default back to D, the default priority. 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:

- `type linear set-priority(node, float)`: This sets the temporary priority of a node. If the node already has a priority, we only change the priority if the new one is higher priority. The programmer can decide if priorities are to be ordered in ascending or descending order.
- `type linear action add-priority(node, float)`: Increments, temporarily, the current priority of the node.
- `type linear schedule-next(node)`: The work will fetch the highest priority node's priority P from its set of nodes and set the action's argument node's priority as $P + 1.0$. If using the priorities in ascending order, we pick the lowest priority and subtract 1.0.
- `type linear set-default-priority(node, float)`: Sets the default priority of the node.
- `type linear action stop-program(node)`: Immediately stops the execution of the whole program.

LM provides the sensing fact `type linear priority(node, node, float)`, where the second argument is the target node and the third argument is the node priority. 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 consumed, they are re-derived automatically, except if the programmer explicitly tells the compiler otherwise. Logically, `set-priority` and `set-default-priority` update the value of `priority` facts, but this done seamlessly by the runtime system.

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:

- `type linear set-cpu(node, int)`: Moves the node to a thread of execution.
- `type linear set-affinity(node, node)`: Places the first node in the same thread as the second node.
- `type linear set-moving(node)`: Allows the node to move freely between threads.
- `type linear set-static(node)`: Forces the node to stay in the same thread indefinitely.

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

- `type linear cpu-id(node, node, int)`: The third argument indicates the thread where the node of the second argument is currently running.
- `type linear moving(node, node)`: Fact available if the node in the second argument is allowed to move between threads.
- `type linear static(node, node)`: Fact available if the node in the second argument is not allowed to move between threads.

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 resume 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 we have enough facts to satisfy the rule. 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 allow rule filtering invalid combinations) and fact updates are implemented by the compiler. The latter transforms fact derivations of the same predicate to a simple update operation, avoiding a deallocation operation followed by an allocation of the new fact.

5.2 Node partitioning

The compiler builds a representation of the graph by inspecting axioms of the program. Next, it orders the nodes of the graph using a breadth-first topological sort. This node ordering is written to the byte-code file and then used to initially partition nodes across available threads.

5.2.1 Coordination directives

Coordination directives are compiled in two different ways, depending 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 returning something. While these are still facts from the point of view of the language, they are consumed immediatelly in order to apply the operation, instead of being derived like any other regular fact. This is more efficient since we do not need to create a new fact.

5.3 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 nodes. Nodes are either inactive, with no new facts, or active, where they will be placed in the corresponding thread's queue. Threads do useful work by processing the nodes in their queues. Whenever a thread becomes idle, it attempts to steal nodes from a random thread into its own queues. If unsuccessful, the thread synchronizes with the other threads to become inactive.

5.3.1 Nodes

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

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. Tries are trees where facts are indexed by common prefix arguments.
- *Doubly Linked Lists* are used to store linear facts. We use a double linked list because it is a very efficient way to add and remove facts.

- *Hash Tables* are used to improve lookup when linked lists are too long and when we need to do search 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 being executed.

idle : the node is not active, 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 is about to move to another thread.

coordinating : the node is moving from one queue to another (i.e. changing priority).

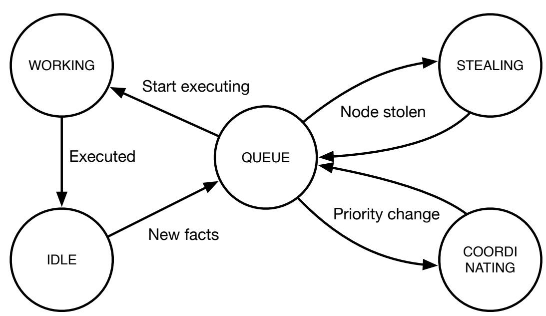


Figure 3: The node state machine. During the lifetime of a program, each node goes through different states as specified by the state machine.

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

5.3.2 Threads

Threads pop active nodes from their queues and execute new candidate rules. In order to allow node priorities, we use 2 queues per thread: a doubly linked list known as the *standard queue* and a min/max heap known as the *priority queue*. The regular queue contains nodes without priorities and implements the operations `push_tail(node)` (push to the tail), `pop_head()` (remove node from the head), `remove(node)` (remove an arbitrary node) and `remove_half()` (removes first half). The priority queue contains nodes with priorities and is implemented as a binary heap array. It supports the operations `push(node,priority)` (add new node to the heap), `pop_min()` (remove the best node), `remove(node)` (remove an arbitrary node), `pop_half_min()` (remove the best half) and `move(node,new_prior` (update priority). Operations such as `remove_half()` are supported in order to support node stealing, while operations `remove(node)` or `move(node,new_priority)` 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.

5.3.3 Communication

Threads synchronize with each other using mutual exclusion. We use a spinlock in each queue to protect queue operations and one spinlock to protect the thread's state. 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 on 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 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.

5.3.4 Termination

There is a global atomic counter, a global boolean flag and one boolean flag (active/idle) per thread for detecting termination. Once a thread goes idle, it decrements the global counter and changes its flag to idle. If the counter goes to zero, the global flag is set to idle. Since every thread will be busy-waiting and checking the global flag, they will detect the change and exit the program.

5.3.5 Node collection

Whenever a node has an empty database and no references from other nodes, it is deleted from the graph. The virtual machine detects such cases by keeping a count of fact references to every node. Once the count drops to zero and the database is empty, the node is immediatelly deleted in order to reduce memory usage.

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 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` [1].

6.1 Single Source Shortest Path

The Single Source Shortest Path (SSSP) program was already presented in Section 3. We are going to update the code in Fig. 1 so that nodes with the shortest distances are selected to run before others nodes. The coordinated version of the same algorithm is presented in Fig. 4 and it uses one coordination fact, namely `set-priority` (line 15). 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 is not optimal since threads do not share a global view of priorities, many rule derivations are going to be avoided.

```
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 relax(A, D1, P1), shortest(A, D2, P2),
11 D1 < D2
12    -o shortest(A, D1, P1),
13       {B, W | !edge(A, B, W) |
14          relax(B, D1 + W, P1 ++ [B]),
15          set-priority(B, float(D1 + W))}.
16
17 relax(A, D1, P1), shortest(A, D2, P2),
18 D1 >= D2
19    -o shortest(A, D2, P2).
```

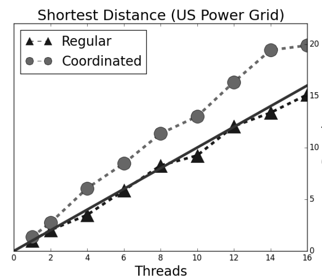
Figure 4: Shortest Path Program.

The most interesting property of the SSSP program presented in Fig. 4 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 5 shows experimental results for a variant of the SSSP program that computes the SSSP for many of the nodes of the graph. There are some situations where unecessary facts are propagated because although the shortest distance is selected, sub-optimal 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.

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

(a)



(b)

Figure 5: 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 version and coordinated version. The speedup values are computed using the execution time of the regular version for 1 thread.

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 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. The tree levels are built in a way that each player plays in turns and the first level represents the moves made by player 1. 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 it can no longer be expanded since the state represents a possible end of the game. 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 eventually deleted once they have nore more facts or no more references to them. The latter happens when

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

the leaves scores are computed or when a node fully minimizes or maximizes the children scores. When the program ends, only the root node is have facts in its database.

The LM code in Fig. 6 implements the initial expansion of the tree. The first three rules (lines 1-10) deal with the cases where no children nodes are created and the last three rules (12-29) deal with cases where new nodes may be created. Thes rule in lines 12-25 generate new nodes using the `exists` language construct, which creates a new node address `B`, where a new `play` fact is created with the new board. A `parent(B,A)` fact is derived to allow node `B` to communicate with its parent.

Since our implementation uses a FIFO ordering for nodes, meaning that when children are expanded, we first expand each children and then the children of the children. The default ordering of the system works well enough for most practical purposes. We note however, that this is not optimal for this program since the complete tree needs to be expanded before computing the scores at the leaves. This introduces memory issues since, in the worst case, the space complexity of the program are $\mathcal{O}(n)$, 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
6    -o maximize(A, N, -00, 0).
7
8  expand(A, Board, [], N, P, Depth),
9  N > 0, NextPlayer <> RootPlayer
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 6: 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. In Fig. 7, we have two threads (squares) and the first one has expanded the tree on the left side. When a second thread comes up, it steals the first unexpanded node and expands the tree from that position. 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.

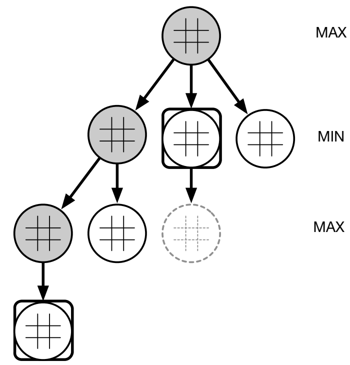
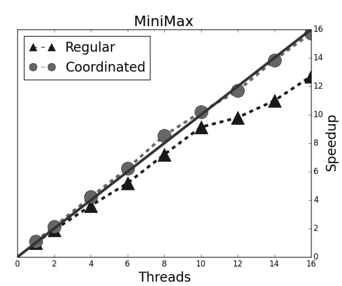


Figure 7: Expanding the MiniMax tree using coordination. By prioritizing deeper nodes, threads are forced to expand the tree using a depth-first approach, which is superior since there is no need to expand the whole tree before computing the node scores.

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 versio nneeds around 14GB. There is also some correlation between memory usage and number of threads. This is explained by the need for more data structures for handling multiple threads, specially concerning the memory allocator used by the virtual machine. In terms of run time, our experimental results show linear scalability for the coordinated version and a 20% run time reduction when using 16 threads.

6.3 Heat Transfer

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 [Gonzalez et al. 2009] (shown in the next section) and Heat Transfer (HT).

In HT, we have a graph of nodes that exchange heat values between them. 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 if we had to compute heat values using iterations.

Fig. 9 shows the HT logical rules that send new heat values to neighbor nodes. We added the first rule 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 node 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. Note that the LM compiler will merge the three rules in Fig. 9 into a single one that handles the 3 branches.

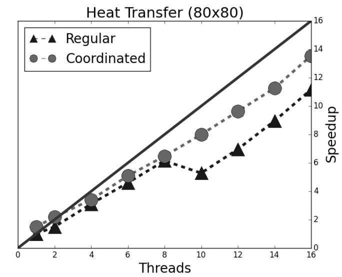
```
new-heat(A, New, Old),
Delta = fabs(New - Old),
Delta > epsilon * 2.0
   -o {B, W | !edge(A, B, W) |
         new-neighbor-heat(B, A, New),
         update(B), add-priority(B, Delta)}.

new-heat(A, New, Old),
Delta = fabs(New - Old)
Delta < epsilon * 2.0,
Delta > epsilon
   -o {B, W | !edge(A, B, W) |
         new-neighbor-heat(B, A, New), update(B)}.

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

Figure 9: Coordination code for the Heat Transfer program. We added one rule that covers cases where Δ is at least 2ϵ in order to increase the priority of neighbors nodes. 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. Speedup values for the coordinated and uncoordinated version are computed using the run time of the uncoordinated version using 1 thread.

6.4 Splash 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.

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 re-

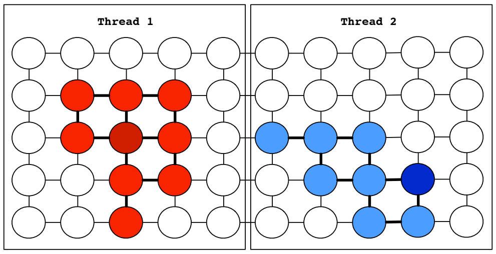


Figure 11: Creating splash trees for belief propagation. Each threads picks the highest priority node and creates a tree from that node. The belief values are updated in two phases: first from the leaves to the root and then from the root to the leaves.

mains the same. We add new logical ruels that coordinate the creation and execution of the splash trees:

Tree building : Each node has a `waiting` fact that is used to start the tree building process. When the highest priority is picked, a token is created that will navigate through the tree. Note that in the rule located in lines 14-21, newly added nodes to the tree must have a positive priority (due to new belief updates) and must be residing in the same thread. We want the tree to be kept in the same thread in order to maximize parallelism with 1 tree per thread.

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 in order 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  // continue tree
6  waiting(A), token(A, All, Next) -o token(A, All, Next).
7  // start tree
8  waiting(A), @priority(A, A, P), P > 0.0
9     -o token(A, [A], [A]).
10 // end tree building
11 token(A, All, Next), length(All) > maxnodes
12    -o first-phase(A, All, reverse(All)).
13 // expand tree
14 token(A, All, [A | Next])
15    -o [collect => L | Side | !edge(A, L, Side),
16          0 = count(All, L),
17          0 = count(Next, L),
18          @priority(A, L, P), P > 0.0,
19          @cpu-id(A, L, Id1),
20          @cpu-id(A, A, Id2), Id1 = Id2 |
21          send-token(A, All, Next ++ L)].
22
23 send-token(A, All, [])
24    -o first-phase(A, All, reverse(All)).
25 send-token(A, All, [B | Next])
26    -o schedule-next(B),
27       token(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), waiting(A), update(A).
41 second-phase(A, [A], Back)
42    -o update(A), waiting(Back),
43       waiting(A), set-priority(A, 0.0).
44 second-phase(A, [A, B | Next], Back)
45    -o update(A), waiting(Back), schedule-next(B),
46       second-phase(B, [B | Next], A).
```

Figure 12: Coordination code for the Splash Belief Propagation program. When a linear fact is prefixed by `@`, it indicates that the fact is going to be re-derived.

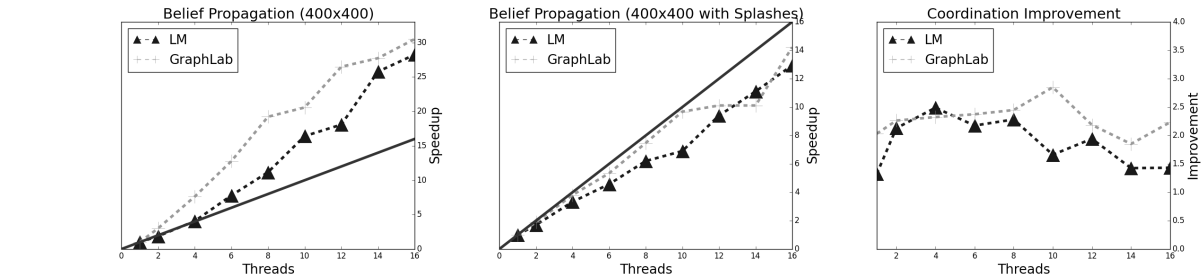
We measured the behavior of LBP and SBP for both LM and GraphLab. Fig. 13 shows that both systems have very similar behavior when using a variable number of threads. Note that GraphLab avoids message passing between nodes since new neighbor belief updates are changed immediatelly, resulting in slightly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

6.5 N Queens

7. Experimental Results and Discussion

7.1 Absolute runtime

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

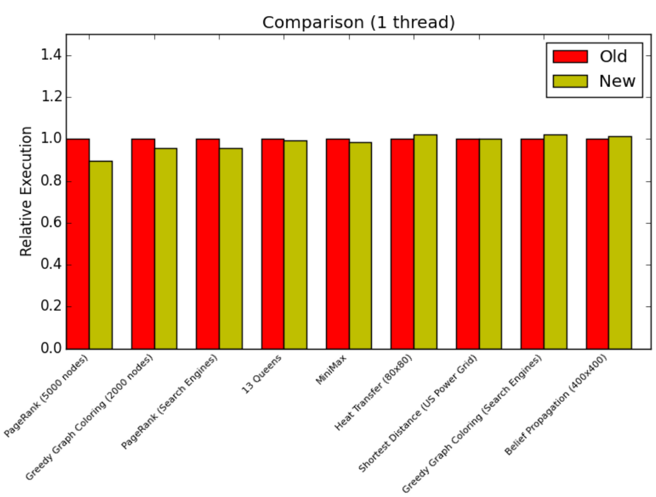


(a) (b) (c)

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.

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. 14



(a)

Figure 14: 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. Most programs show little to no degradation in performance.

8. Conclusions

9. Acknowledgements

References

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.

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.

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.

R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23: 125–149, 1993.