Thread-Aware Logic Programming for Data-Driven Parallel Programs

Flavio Cruz[†‡], Ricardo Rocha[‡], Seth Copen Goldstein[†]
[†]*Carnegie Mellon University, Pittsburgh, PA 15213*
(*e-mail:* {fmfernan,seth}@cs.cmu.edu)
[‡]*CRACS & INESC TEC and Faculty of Sciences, University Of Porto*
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
(*e-mail:* {flavioc,ricroc}@dcc.fc.up.pt)

submitted 29 April 2015; accepted 5 June 2015

Abstract

Declarative programming in the style of functional and logic programming has been hailed as an alternative parallel programming style where computer programs are automatically parallelized without programmer control. Although this approach removes many pitfalls of explicit parallel programming, it hides important information about the underlying parallel architecture that could be used to improve the scalability and efficiency of programs. In this paper, we present a novel programming model that allows the programmer to reason about thread state in data-driven declarative programs. This abstraction has been implemented on top of Linear Meld, a linear logic programming language that is designed for writing graph-based programs. We present several programs that show the flavor of our new programming model, including graph algorithms and a machine learning algorithm. Our goal is to show that it is possible to take advantage of architectural details without losing the key advantages of logic programming.

KEYWORDS: Parallel Programming, Declarative Programming, Coordination

1 Introduction

Parallelism in functional (Blelloch 1996; Chakravarty et al. 2007) and logic (Gupta et al. 2001) programming languages has been exploited through the use of *implicit parallelism*, meaning that parallelism is not explicitely controlled by the programmer but by the underlying runtime system. Although these approaches remove many pitfalls of explicit parallel programming, they hide important information about the underlying parallel architecture that could be used to improve the scalability and efficiency of programs, thus leaving very little opportunity for the programmer to optimize the parallel aspects of the program. This is unfortunate since some programs are computed more efficiently if information about the parallel architecture (e.g., the number of cores) is exposed to the code being executed. It would be far better if the programmer could reason declaratively not only about the problem at hand but also about the underlying parallel architecture.

Linear Meld (LM) is a linear logic programming language especially suited for the parallel implementation of graph-based algorithms (Cruz et al. 2014). LM offers a concise and expressive framework that has been applied to a wide range of graph-based problems and machine learning algorithms, including: belief propagation with and without residual splash (Gonzalez et al. 2009), PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, minimax, and many others.

In this paper, we present an extension to LM that allows the programmer to reason about the underlying parallel architecture. The extension introduces the graph of threads, that map to the concrete threads executing on the system, as computable entities of the language. It is then possible to derive logical facts about the threads and to write inference rules that reason about the state of the threads along with the state of the program. This novel programming model allows the writing of declarative code that is both data-driven and architecture-driven. As we will see throughout the paper, this allows us to easily optimize programs that could not have been improved otherwise.

2 Language

Linear Meld (LM) is a logic programming language in the style of Datalog (Ramakrishnan and Ullman 1993) that offers a declarative and structured way to manage mutable state. LM is based on linear logic (Girard 1995), a logic system where truth is ephemeral and can be consumed when used to prove propositions. Like Datalog, LM is a *forward-chaining* logic programming language since computation is driven by a set of inference rules that are used to update a database of logical facts that represent the state of the program. Unlike Datalog, logical facts can be asserted and retracted freely, therefore inference rules can also retract facts. Moreover, each LM rule has a pre-defined priority that is inferred from its position in the source code which forces higher priority rules to be applied first. The program stops when *quiescence* is achieved, i.e., when inference rules no longer apply.

An inference rule of the form `a(M, N), b(M) -o c(N)` has the following meaning: if the database has facts `a(M, N)` and `b(M)` then an instance of both is retracted and a copy of `c(N)` is asserted. `A(M, N), b(M)` form the *body* of the rule, `c(N)` the head while `-o` expresses a *linear implication*. In addition to *linear facts*,

```
type left(node, node).
type right(node, node).
type found(node, int, string).
type not-found(node, int).
type linear value(node, int, string).
type linear lookup(node, int).

// We found the key we want
lookup(A, K),
value(A, K, Value)
   -o value(A, K, Value),
      !found(A, K, Value).

// The key must be to the left
lookup(A, K),
value(A, NKey, Value),
K < NKey
!left(A, B)
   -o value(A, NKey, Value),
      lookup(B, K).

// The key must be to the right
lookup(A, K),
value(A, NKey, Value),
K > NKey,
!right(A, B)
   -o value(A, NKey, Value),
      lookup(B, K).

// The key cannot be found
lookup(A, K)
   -o !not-found(root, K).

// Initial axioms (...)
lookup(root, 6).
```

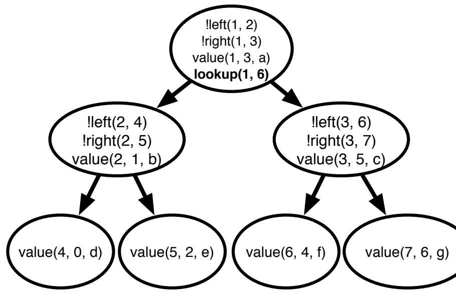
Fig. 1. LM program for performing lookups in a BST dictionary.

rules in LM can also use *persistent facts* which are never retracted. Such facts are preceded by a `!`, e.g., `!left` in line 18 of Fig. 1. Facts are instantiations of *predicates* and they map predicate arguments (which are typed) to concrete values. The type system of LM includes scalar types such as *node*, *int*, *float*, *string*, *bool*. Recursive types are also included, e.g., *list X*, *pair X; Y* and *array X*.

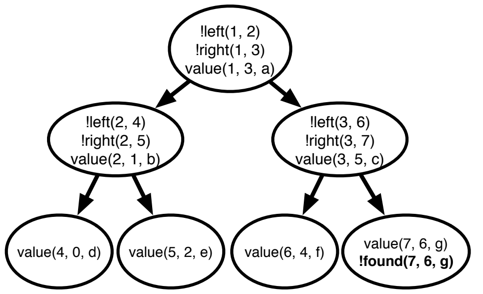
In Fig. 1, we present an LM program that implements the lookup operation in a binary search tree (BST) dictionary represented as key/value pairs. Lines 1-6 declare the predicates that are going to be used in the program, which includes four persistent predicates (`left`, `right`, `found` and `not-found`) and two linear predicates (`value` and `lookup`). The predicate `value` assigns a key/value pair to a tree node and `lookup` represents an lookup operation for a given key. Each node of the tree contains a key and a value. The nodes are connected using `left` and `right` branches, which take the search property of the tree into account.

The algorithm uses four rules for the three cases when looking up a given key `K` at a given node `A`. The first rule finds the key, meaning that the current node has the key we want (lines 8-12). The second rule detects that the key must be located in the left branch of tree (lines 14-20), while the third rule detects that the key must be in the right branch of the tree (lines 22-28). Finally, the fourth rule, which is inferred if nothing else matches, represents the case where the node has no branches or one of the branches available cannot possible contain the required key (lines 30-32). At the end, we have the initial axioms which includes the `left` and `right` axioms (not shown) and a `lookup(root, 6)` axiom for looking up the value with key 6.

Figure 2 illustrates the initial and final states of the example program. Note that we have partitioned the database by the first argument of each fact. In Fig. 2(a), we present the initial database filled with the program's axioms. Execution follows through the right branch twice using rule 3 since 6 > 3 and 6 > 5. We finally reach node 7 and apply rule 1, which derives fact `!found(7, 6, g)`. The final state of the program is show in Fig. 2(b).



(a) Initial state



(b) Final state, after reaching node 7

Fig. 2. Initial and final states of the BST dictionary program.

3 Semantics

An LM program consists of rules which manipulate facts. The facts can be viewed as a graph where the first argument, of type `node`, specifies the node of the graph where the fact belongs to. The remaining arguments of a fact, if any, describe the properties of the node including, if the argument is of type `node`, the edges in the graph. Rule inference is done concurrently by independently deriving rules on different nodes of the graph. Further, rules may only manipulate facts belonging to the same node. However, head expressions may refer to other nodes, as long as they are instantiated in the body.

In LM's implementation, nodes are processed in parallel by *threads* which are realized as computing cores in a shared memory setting. The graph of nodes is initially partitioned among threads but nodes can be stolen from other nodes for work balancing. Nodes are either *active* or *inactive*. Active nodes contain new logical facts that have not been processed yet (i.e., may have applicable inference rules) and inactive nodes have been processed (i.e., no inference rules are applicable at the moment). In the case where all the nodes assigned to a given thread are inactive then the thread is allowed to steal active nodes from another thread. When every node becomes inactive, all the threads will go idle and the program terminates.

LM already provides special predicates, called *coordination predicates*, that allow the programmer to manage scheduling and data partitioning in order to improve performance. Coordination predicates fall into two groups: *scheduling predicates* and *partitioning predicates*. Scheduling and partitioning predicates are also classified into *action predicates* and *sensing predicates*. The sensing predicates are used to obtain information from the runtime system in the body of the rule. Action facts in the head of the rule manipulate the runtime system.

Scheduling predicates change how nodes are selected from the work queue. Every node can have a *priority value* which is used by the threads to choose the nodes from the work queue. We have action predicates to change the priority of a node, such as `set-priority`, `remove-priority`, and `schedule-next`, that puts a node on the head of the work queue, and sensing predicates, such as `priority(node A, float P)`, which indicates that node `A` has priority `P`. Note that sensing facts do not need to follow the body constraints mentioned earlier.

Partitioning predicates change how nodes are placed in threads. In terms of action predicates, we have a `set-thread` predicate, that changes the current thread of a given node, and `set-static`, which disallows work stealing. The sensing predicates allow the programmer to reason about the placement of the node, including knowing the thread where the node is currently placed. We also have the predicate `just-moved`, which is derived after the use of `set-thread`.

4 Thread Local Facts

As we have seen, the first argument of every fact must be typed as `node`. This enforces locality of facts to a particular node. While this restriction is the foundation for implicit parallelism in LM, it restricts how much information is known between nodes. The existence of coordination predicates brings some awareness about the underlying parallel system, including node scheduling and placement, however it is limited in the sense that the programmer is not able to reason directly about the state of the thread but only about the state of the node.

In order to remove this limitation, we introduce the concept of *thread facts*, which are logical facts stored at the thread level, meaning that, each thread is now an entity with its own logical facts. We extend the type system to include the type `thread`, which is the type of the first argument of *thread predicates*, indicating that the predicate is related and is to be stored in a specific thread. We can view the thread facts as forming a separate graph from the data graph, a graph of the processing elements which are operating on the data graph.

The introduction of thread facts increases the expressiveness of the system in the sense that it is now possible to write inference rules that reason about the state of the threads. This creates optimization opportunies since we can now write algorithms with global information stored in the thread, while keeping the LM language fully declarative. Moreover, threads are now allowed to explicitly communicate with each other, and in conjunction with coordination predicates, enable the writing of complex scheduling policies.

We discriminate between two new types of inference rules. The first type is the *thread rule* and has the form `a(T), b(T) -o c(T)`, and can be read as: if thread `T` has fact `a(T)` and `b(T)` then derive fact `c(T)`. The second type is the *mixed rule* and has the form `a(T), d(N) -o e(N)` and can be read as: if thread `T` is executing node `N` and has the fact `a(T)` and node `N` has the fact `d(N)` then derive `e(N)` at node `N`. Thread rules reason solely at the thread level, while mixed rules allow reasoning about both thread and node facts. Logically, the mixed rule uses an extra fact `running(T, N)`, which indicates that thread `T` is currently executing node `N`. The `running` fact is implicitly retracted and asserted every time the thread selects a different node for execution. This makes our implementation efficient since a thread does not need to look for nodes that match mixed rules and it is then the scheduling of the program that drives the matching of such rules. Figure 3 represents a schematic view of the graph data structure of a program with two threads: thread $T1$ is executing node 1 and $T2$ is executing node 4. Both threads have access to all the facts in the thread itself and to the corresponding node facts.

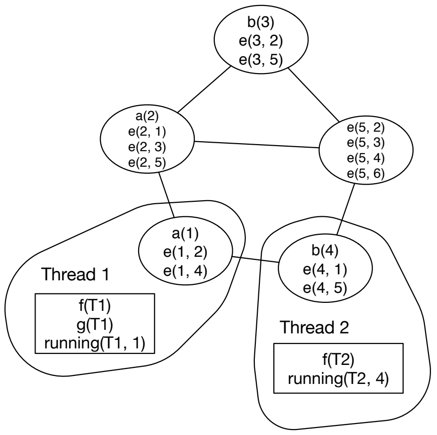


Fig. 3. A program being executed with two threads. Note that each thread has a `running` fact that stores the node currently being executed.

To show how the new extension works, we are going to take the BST example shown in Fig. 1 and improve it by using thread facts. We assume that there is a BST and a sequence of *n* lookup operations for different keys in the BST (which may or may not be repeated). A single lookup has worst-case time complexity $O(h)$ where h is the height of the BST, therefore n lookups can take $O(h \times n)$ time. In order to improve the execution time of the program, we can cache the search operations so that repeated lookup operations become faster. Instead of traversing the entire height of the BST, we look in the cache and send the lookup operation immediately to the node where the key is located. Without thread facts, we might have cached the results at the root node, however this is not a scalable approach as it would introduce a bottleneck.

Figure 4 shows the updated BST code with a thread cache. We just added two more predicates, `cache` and `cache-size`, that are facts placed in the thread and represent cached keys and the total size of the cache, respectively. We also added three new rules that handle the following cases: (i) a key is found and is also in the cache; (ii) a key is found but is not in the cache; and (iii) a key is in the cache, therefore a `lookup` fact is derived in the target node. Note that it is easy to extend the cache mechanism to use an LRU type approach in order to limit the size of the cache.

```
type linear cache(thread, node, int).
type linear cache-size(thread, int).

// Key exists and is also in cache
lookup(A, K),
value(A, K, Value),
cache(T, A, K)
   -o value(A, K, Value),
      !found(A, K, Value),
      cache(T, A, K).

// Key exists and is not in cache
lookup(A, K),
value(A, K, Value),
cache-size(T, Total)
   -o value(A, K, Value),
      !found(A, K, Value),
      cache-size(T, Total + 1),
      cache(T, A, K).

// Cached by the thread
lookup(A, K),
cache(T, TargetNode, K)
   -o lookup(TargetNode, K),
      cache(T, TargetNode, K).

// Remaining rules (...)
```

Fig. 4. LM program for performing lookups in a BST with a thread cache.

In order to understand how well the new program performs, we have experimented with a binary tree with 17 levels and 100000 lookup operations. In our experimental setup, we used a machine with 4 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 code using GCC 4.8.3 (g++) with the flags `-O3 -std=c++11 -fno-rtti -march=x86-64`[1].

The scalability results shown in Fig. 5 are presented by comparing the run time of different versions against the run time of the regular version (without thread facts) using 1 thread. The results show that caching brings improved scalability and reduced run time due to pruned paths that would need to be searched without a cache. For example, when using a single thread, the cached version achieves a 2-fold speedup over the regular version and when using 16 threads, it achieves a 16-fold speedup over the regular version using 1 thread.

This program shows that it is possible to use architectural details in a declarative style to improve the scalability of programs. With a few extra facts stored at the thread level and a few extra logical rules, we were able to significantly improve the linear logic program, while remaining fully declarative.

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

5 Further Applications

5.1 Graph Reachability

Consider the problem of checking if a set of nodes S in a graph G is reachable from an arbitrary node N. An obvious solution to this problem is to start at N, gather all the neighbor nodes into a list and then recursively visit all those reachable nodes, until S is covered. This reduces to a problem of performing a breadth or depth-first search on graph G. However, this solution is sequential and does not have much concurrency. An alternative solution to the problem is to recursively propagate the search to all neighbors and aggregate the results in the node where the search started.

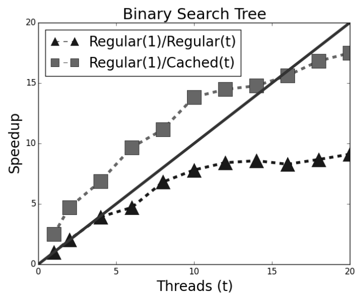


Fig. 5. Scalability results comparing the regular and cached BST programs. The dataset is a tree with 262000 nodes and 100000 lookup operations distributed among 12000 nodes.

Unfortunately, there are still some issues with this solution. First, as the search process goes on, there is no sharing of the nodes that were already found, since only the start node actually stores which nodes have been found. It would be prohibitly expensive to share such information between nodes. Second, once the search has reached all the required nodes, the search process will not stop, exploring unneeded nodes. Fortunately, we can use thread facts to solve both these issues. The search process is still done concurrently as before, but the search state is now stored in each thread, allowing the thread to store partial results and coordinate with other threads. The code for this *coordinated version* is shown in Fig. 6. The axioms (not shown) represent the `search` facts, each containing the `Id` of the search and the list of nodes to reach.

Lines 1-4 start the search process by assigning a thread `Owner` to search `Id` using the persistent fact `!thread-list` which contains the list of all available threads in the system (the total number of threads is given by `@threads`). In line 3, a fact `thread-search` is created for all threads using a *comprehension*, which is a construct made of 3 parts that, for the variables in the first part (`T2`), iterates over the facts in the second part (`!thread(T, T2)`) and derives the facts in the third part (`thread-search(T2, Id, ToReach, Owner)`). We use predicate `do-search` to propagate the search through the graph and a predicate `visited` to mark nodes already processed for a specific search. The two rules in lines 14-27 propagate the search process to the neighbor nodes and check if the current node is part of the list of nodes we want to reach.

An interesting property of this version is that each owner thread responsible for a search keeps track of the remaining nodes that need to be reached. In line 18, we derive `remove-thread-search` in order to inform owner threads about new reachable nodes. Once an owner thread detects that all nodes have been reached (lines 32-34), all the other threads will know that and update their search state

```
1  search(A, Id, ToReach),
2  !thread-list(T, L), Owner = nth(L, Id % @threads)            // Allocate search to a thread
3     -o {T2 | !thread(T, T2) | thread-search(T2, Id, ToReach, Owner)},
4        do-search(A, Id).
5
6  thread-search(T, Id, [], Owner),                                      // Nothing left to find
7  do-search(A, Id)
8     -o thread-search(T, Id, [], Owner).
9
10 do-search(A, Id),
11 visited(A, Id)                                                          // Already visited
12    -o visited(A, Id).
13
14 do-search(A, Id),
15 thread-search(T, Id, ToReach, Owner),
16 !value(A, Val), Val in ToReach                                           // New node found
17    -o thread-search(T, Id, remove(ToReach, Val), Owner),
18       remove-thread-search(Owner, Id, Val),                  // Tell owner thread about it
19       {B | !edge(A, B) | do-search(B, Id)},
20       visited(A, Id).
21
22 do-search(A, Id),
23 thread-search(T, Id, ToReach, Owner),
24 !value(A, Val), ~ Val in ToReach                                  // Node is not on the list
25    -o thread-search(T, Id, ToReach, Owner),
26       visited(A, Id),
27       {B | !edge(A, B) | do-search(B, Id)}.
28
29 remove-thread-search(T, Id, Val), thread-search(T, Id, ToReach, Owner)
30    -o thread-search(T, Id, remove(ToReach, Val), Owner),
31       check-results(T, Id).
32 check-results(T, Id), thread-search(T, Id, [], Owner)
33    -o thread-search(A, Id, [], Owner),
34       {B | !other-thread(T, B) | signal-thread(B, Id)}.
35  check-results(T, Id), thread-search(T, Id, ToReach, Owner), ToReach <> []
36    -o thread-search(T, Id, ToReach, Owner).
37 signal-thread(T, Id), thread-search(T, Id, ToReach, Owner)     // Thread knows search is done
38    -o thread-search(T, Id, [], Owner).
```

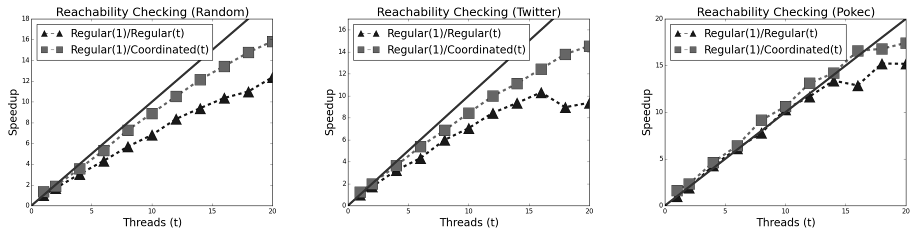
Fig. 6. Coordinated version of the reachability checking program.

accordingly (lines 37-38). When every thread knows that all nodes were reached, they will consume `do-search` facts (lines 6-8), effectively pruning the search space.

Experimental results for the graph reachability program are shown in Fig. 7. The Random dataset is a dense graph, which makes it one of the best performing datasets. In this dataset, the coordinated version reaches a 16-fold speedup with 20 threads versus a 12-fold speedup for the regular version. Twitter and Pokec are datasets generated from real data[2] and the coordinated version is able to scale well up to 20 threads, while the regular version shows some issues once it uses more than 16 threads. This is because the number of searches is relatively small and the nodes where the searches started have some contention since those nodes accumulate the search results. This behavior is not seen in the coordinated version because searches are equally distributed among the threads.

The Pokec dataset is noteworthy because the coordinated version is almost twice as fast as the regular version when using only 1 thread. We think that such run time improvement happens because the number of searches is small, which makes it easier to perform the joins in lines 14-15.

[2] Twitter and Pokec datasets retrieved from `http://snap.stanford.edu/data/`



(a) Random: 50,000 nodes, 1,000,000 edges and 20 searches. (b) Twitter: 81,306 nodes, 1,768,149 edges and 50 searches. (c) Pokec: 1,632,803 nodes, 30,622,564 edges and 5 searches.

Fig. 7. Scalability results for the graph reachability program using different datasets and number of searches. All searches in the datasets target around 2-5% of all nodes in the graph.

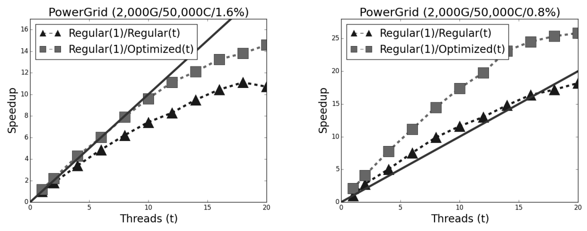
This graph reachability program shows how to introduce complex coordination policies between threads by reasoning about the state of each thread. In addition, the use of linear logic programming makes it easier to prove properties of the program since computation is done by applying controlled changes to the state.

5.2 *PowerGrid Problem*

Consider a powergrid with C consumers and G generators. We are interested in connecting each consumer to a single generator, but each generator has a limited capacity and the consumer draws a certain amount of power from the generator. A valid powergrid is built in such a way that all consumers are served by a generator and that no generator is being overdrawn by too many consumers. Although consumers and generators may be connected through a complex network, in this section we analyze the case where any consumer can be served by any generator.

A straightforward distributed implementation for the powergrid problem requires that each consumer is able to connect to a any generator. Once a generator receives a connection request, it may or may not accept it. If the generator has no power available for the new consumer, it will disconnect from it and the consumer must select another generator. If a generator initiates too many disconnections, then it disconnects all its consumers in order to restart the process. This randomized algorithm works but can take a long time to converge, depending on the amount of power available in the generators.

The issue with the straightforward distributed implementation is that it lacks a global view of the problem or requires a more complicated synchronization algorithm between consumers and generators. As we have seen before, thread local facts are an excellent mechanism to introduce a global view of the problem without complicating the original algorithm written in a declarative style. In our solution, we partition the set of generators G among the threads in the system. With this partitioning, each thread assumes the ownership of its generators and is able to process consumers with a global view over its set of generators. The thread can then immediately assign the consumers to its generators when possible, otherwise it uses the regular distributed algorithm.



(a) Normal power (101.6%). (b) Reduced power (100.8%).

Fig. 8. Scalability results for the powergrid program.

Figure 8 shows the scalability results for the powergrid program when using 2000 generators and 50000 consumers. We varied the total capacity of the generators in relation to the power required by all consumers. In Fig. 8(a) the power of the generators is 101.6% of the required by the consumers. And in Fig. 8(b), we reduced the gap to only 100.8% in order to see how the two programs would behave in a reduced power situation. As shown in the figures, the optimized version performs the best when the power is reduced and is almost twice as fast than then regular version when using just 1 thread.

5.3 *Splash Belief Propagation*

Approximation algorithms can obtain significant benefits from using optimal evaluation strategies due to their inherent non-determinism. A good example is the Loopy Belief Propagation (LBP) program (Murphy et al. 1999). LBP is an approximate inference algorithm used in graphical models with cycles. LBP is a sum-product message passing algorithm where nodes exchange messages with their immediate neighbors and apply computations to the messages received.

The most basic evaluation strategy for LBP is to update the belief values of nodes in synchronous iterations. First, the beliefs of all nodes are computed and then sent to the neighbor nodes in rounds, requiring synchronization after each round. Another computation strategy is to compute the beliefs asynchronously, by using partial information at the node level. The asynchronous approach is an improvement over the synchronous version because it leads to faster convergence time. An improved evaluation strategy is the Splash Belief Propagation (SBP) program (Gonzalez et al. 2009), where belief values are computed by first building a tree and then updating the beliefs of each node twice, first from the leaves to the root and then the reverse. These *splash trees* are built by starting at a node whose belief changed the most in the last update. The trees must be built iteratively until convergence is achieved.

In an environment with *T* threads, it is then possible to build *T* splash trees concurrently. First, we partition the nodes into *T* regions and then assign each region to a thread. Each thread is then responsible for iteratively building splash trees on that region until convergence is reached. Figure 9 shows the LM implementation for the SBP program.

The programs starts in lines 3-7 by partitioning the nodes into regions using `set-thread` and by creating the first splash tree (line 7) using `start-tree(T)`. The

```
1  partitioning(T, @world / @threads).                      // Move @world/@threads nodes
2
3  !coord(A, X, Y), start(A)                                           // Moving this node
4     -o set-static(A), set-thread(A, grid(X, Y)).
5  just-moved(A), partitioning(T, Left)                     // Thread received another node
6     -o partitioning(T, Left - 1).
7  partitioning(T, 0) -o start-tree(T).
8
9  start-tree(T), priority(A, P), P > 0.0                                  // Tree building
10    -o priority(A, P), expand-tree(T, [A], []).
11 expand-tree(T, [A | All], Next)
12    -o thread-id(A, Id),
13       [collect => L | !edge(A, L), ~ L in All, ~ L in Next, priority(L, P), P > 0.0,
14          thread-id(L, Id2), Id1 = Id2 | priority(L, P), thread-id(L, Id2) |
15          new-tree(T, [A | All],
16             if len(All) + 1 >= maxnodes then [] else Next ++ L end)].
17
18 new-tree(T, [A | All], [])
19    -o schedule-next(A), first-phase(T, reverse([A | All]), [A | All]).
20 new-tree(T, All, [B | Next])
21    -o schedule-next(B), expand-tree(T, [B | All], Next).
22
23 first-phase(T, [A], [A]), running(T, A)                                   // First phase
24    -o running(T, A), update(A), remove-priority(A), start-tree(T).
25 first-phase(T, [A, B | Next], [A]), running(T, A)
26    -o running(T, A), update(A), schedule-next(B), second-phase(T, [B | Next]).
27 first-phase(T, All, [A, B | Next]), running(T, A)
28    -o running(T, A), update(A), schedule-next(B), first-phase(T, All, [B | Next]).
29
30 second-phase(T, [A]), running(T, A)                                      // Second phase
31    -o running(T, A), update(A), remove-priority(A), start-tree(T).
32 second-phase(T, [A, B | Next]), running(T, A)
33    -o running(T, A), update(A), schedule-next(B), second-phase(T, [B | Next]).
```

Fig. 9. LM implementation for the Splash Belief Propagation program.

algorithm is then divided in three main phases, named *tree building*, *first phase* and *second phase*, as described next:

Tree building Tree building starts after the rule in lines 9-10 is fired. Since the thread always picks the higher priority node, we start by adding that node to the list that represents the tree. In lines 13-16, we use an *aggregate* (Cruz et al. 2014) to gather all the neighbor nodes that have a positive priority (due to a new belief update) and are in the same thread. Nodes are collected into list `L` and appended to list `Next` (line 16).

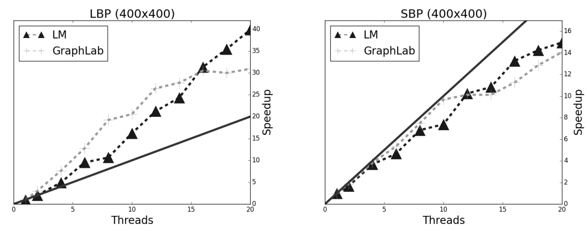
First phase When the number of nodes in the tree reaches a certain limit, a `first-phase` is generated to update the beliefs of all nodes in the tree (line 19). As the nodes are updated, starting from the leaves and ending at the root, an `update` fact is derived to update the belief values (lines 26 and 28).

Second phase The computation of beliefs is performed from the root to the leaves and the belief values are updated a second time (lines 31 and 33).

GraphLab (Low et al. 2010), a C++ framework for writing machine learning algorithms, provides the splash scheduler as part of its framework. To put LM's implementation in perspective, we thus measured the behavior of LBP and SBP for both LM and GraphLab. Figure 10 shows the results. We can observe that both systems have very similar behavior when using a variable number of threads, but for higher number of threads and, in particular, for more than 15 threads, LM shows better speedups than GraphLab. In terms of running times, LM is, on average, 1.4 times slower than GraphLab, although LM program code is more concise.

6 Concluding Remarks

Logic programming, as a declarative style of programming, has been a fertile research field in terms of parallel models (Gupta et al. 2001; Rocha et al. 2005). These approaches have been relatively successful in parallelizing regular Prolog programs, however the programmer has little control over the scheduling strategy and policies. An exception is the proposal by Casas et al. which exposes execution primitives for and-parallelism (Casas et al. 2007), allowing for different scheduling policies. Compared to LM, this approach offers a more fine grained control to parallelism but has limited support for reasoning about thread state.



(a) LBP (b) SBP

Fig. 10. Experimental results for the Loopy Belief Propagation (LBP) and Splash Belief Propagation (SBP) programs using the LM and GraphLab systems.

The new LM extension shares some similarities with the Linda coordination language (Ahuja et al. 1986). Linda implements a data-driven coordination model and features a *tuple space* that can be manipulated by a set of processes. Like LM, those processes communicate through the tuple space, by deriving tuples, which are akin to logical facts. The programmer is able to create arbitrary communication patterns by reading and writing to the tuple space. Unfortunately, Linda is implemented on top of other languages and, by itself, its not a declarative language.

Galois (Pingali et al. 2011; Nguyen and Pingali 2011) and Elixir (Prountzos et al. 2012) are parallel language models that allow the specification of different scheduling strategies in order to optimize programs. The programmer first writes the basic program and then a scheduler specification changes how threads prioritize computation. However, these specifications only reason about the data being computed and not about the parallel architecture.

In this work, we have extended the LM language with a declarative mechanism for reasoning about the underlying parallel architecture. LM programs can be first written in a data-driven fashion and then optimized by reasoning about the state of threads, thus enabling the implementation of more efficient evaluation strategies. This novel mechanism enables the programmer to take advantage of both implicit and explicit parallelism by using the principles of logic programming without introduction of extra language constructs.

Acknowledgments

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

References

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

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

Casas, A., Carro, M., and Hermenegildo, M. V. 2007. Towards high-level execution primitives for and-parallelism: Preliminary results.

Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. 2007. Data parallel haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming*. New York, NY, USA, 10–18.

Cruz, F., Rocha, R., Goldstein, S., and Pfenning, F. 2014. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue abs/1405.3556*, 493–507.

Girard, J.-Y. 1995. Linear logic: Its syntax and semantics. In *Advances in Linear Logic*. New York, NY, USA, 1–42.

Gonzalez, J., Low, Y., and Guestrin, C. 2009. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics*. Clearwater Beach, Florida.

Gupta, G., Pontelli, E., Ali, K. A. M., Carlsson, M., and Hermenegildo, M. V. 2001. Parallel execution of prolog programs: A survey. *ACM Transactions on Programming Languages and Systems 23,* 4, 472–602.

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

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

Nguyen, D. and Pingali, K. 2011. Synthesizing concurrent schedulers for irregular algorithms. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 333–344.

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

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

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

Rocha, R., Silva, F., and Costa, V. S. 2005. On applying or-parallelism and tabling to logic programs. *Journal of Theory and Practice of Logic Programming 5,* 1 & 2, 161–205.