A Parallel Virtual Machine for Executing Forward-Chaining Linear Logic Programs

Flavio Cruz[1,2], Ricardo Rocha[2], and Seth Copen Goldstein[1]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
{fmfernan, seth}@cs.cmu.edu
[2] CRACS & INESC TEC, Faculty of Sciences, University Of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc, ricroc}@dcc.fc.up.pt

Abstract. Linear Meld is a concurrent forward-chaining linear logic programming language where logical facts can be asserted and retracted in a structured way. The database of facts is partitioned by the nodes of a graph structure which leads to parallelism if nodes are executed simultaneously. Communication arises whenever nodes send facts to other nodes by fact derivation. We present an overview of the virtual machine that we implemented to run Linear Meld on multicores, including code organization, thread management, rule execution and database organization for efficient fact insertion, lookup and deletion. Although our virtual machine is a work-in-progress, our results already show that Linear Meld is not only capable of scaling graph and machine learning programs but it also exhibits some interesting performance results when compared against other programming languages.

Keywords: linear logic programming, virtual machine, implementation

1 Introduction

Logic programming is a declarative programming paradigm that has been used to advance the state of parallel programming. Since logic programs are declarative, they are much easier to parallelize than imperative programs. First, logic programs are easier to reason about since they are based on logical foundations. Second, logic programmers do not need to use low level programming constructs such as locks or semaphores to coordinate parallel execution, because logic systems hide such details from the programmer.

Logic programming languages split into two main fields: *forward-chaining* and *backwards-chaining* programming languages. Backwards-chaining logic programs are composed of a set of rules that can be activated by inputing a query. Given a query $q(\hat{x})$, an interpreter will work backwards by matching $q(\hat{x})$ against the head of a rule. If found, the interpreter will then try to match the body of the rule, recursively, until it finds the program axioms (rules without body). If the search procedure succeeds, the interpreter finds a valid substitution for the $\hat{x}$ variables. A popular backwards-chaining programming language is Prolog [4],

which has been a productive research language for executing logic programs in parallel. Researchers took advantage of Prolog's non-determinism to evaluate subgoals in parallel with models such as *or-parallelism* and *and-parallelism* [8].

In a forward-chaining logic programming language, we start with a database of facts (filled with the program's axioms) and a set of logical rules. Then, we use the facts of the database to fire the program's rules and derive new facts that are then added to the database. This process is repeated until the database reaches *quiescence* and no more information can be derived from the program. A popular forward-chaining programming language is Datalog [14].

In this paper, we present a new forward-chaining logic programming language called Linear Meld (LM) that is specially suited for concurrent programming over graphs. LM differs from Datalog-like languages because it integrates both classical logic and linear logic [6] into the language, allowing some facts to be retracted and asserted logically. Although most Datalog and Prolog-like programming languages allow some kind of state manipulation [11], those features are extra-logical, reducing the advantages brought forward by logic programming. In LM, since mutable state remains within the logical framework, we can reason logically about LM programs.

The roots of LM are the P2 system [12] and the original Meld [2,1]. P2 is a Datalog-like language that maps a computer network to a graph, where each computer node performs local computations and communicates with neighbors. Meld is itself inspired by the P2 system but adapted to the concept of massively distributed systems made of modular robots with a dynamic topology. LM also follows the same graph model of computation, but, instead, applies it to parallelize graph-based problems such as graph algorithms, search algorithms and machine learning algorithms. LM programs are naturally concurrent since the graph of nodes can be partitioned to be executed by different workers.

To realize LM, we have implemented a compiler and a virtual machine that executes LM programs on multicore machines[3]. We have implemented several parallel algorithms, including: belief propagation [7], belief propagation with residual splash [7], PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, among others.

As a forward-chaining linear logic programming language, LM shares similarities with Constraint Handling Rules (CHR) [3,10]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints (which can be seen as facts). Constraints can be consumed or generated during the application of rules. Some optimization ideas used in LM such as join optimizations and using different data structures for indexing facts are inspired by research done in CHR [9].

This paper describes the current implementation of our virtual machine and is organized as follows. First, we briefly introduce the LM language. Then, we present an overview of the virtual machine, including code organization, thread management, rule execution and database organization. Finally, we present preliminary results and outline some conclusions.

[3] Source code is available at `http://github.com/flavioc/meld`.

2 The LM Language

Linear Meld (LM) is a logic programming language that offers a declarative and structured way to manage state. A program consists of a database of facts and a set of derivation rules. The database includes persistent and linear facts. Persistent facts cannot be deleted, while linear facts can be asserted and retracted.

The dynamic (or operational) semantics of LM are identical to Datalog. Initially, we populate the database with the *program's axioms* (initial facts) and then determine which derivation rules can be applied using the current database. Once a rule is applied, we derive new facts, which are then added to the database. If a rule uses linear facts, they are retracted from the database. The program stops when *quiescence* is achieved, that is, when rules no longer apply.

Each fact is a predicate on a tuple of *values*, where the type of the predicate prescribes the types of the arguments. LM rules are type-checked using the predicate declarations in the header of the program. LM has a simple type system that includes types such as *node*, *int*, *float*, *string*, *bool*. Recursive types such as *list X* and *pair X; Y* are also allowed. Each rule in LM has a defined priority that is inferred from its position in the source file. Rules at the beginning of the file have higher priority. We consider all the new facts that have been not used yet to create a set of *candidate rules*. The set of candidate rules is then applied (by priority) and updated as new facts are derived.

2.1 Example

We now present an example LM program in Fig. 1 that implements the key update operation for a binary tree represented as a key/value dictionary. We first declare all the predicates (lines 1-4), which represent the kinds of facts we are going to use. Predicate `left/2` and `right/2` are persistent while `value/3` and `replace/3` are linear. The `value/3` predicate assigns a key/value pair to a binary tree node and the `replace/3` predicate represents an update operation that updates the key in the second argument to the value in the third argument.

The algorithm uses three rules for the three cases of updating a key's value: the first rule performs the update (lines 6-7); the second rule recursively picks the left branch for the update operation (lines 9-10); and the third rule picks the right branch (lines 12-13). The axioms of this program are presented in lines 15-22 and they describe the initial binary tree configuration, including keys and values. By having the `update(@3, 6, 7)` axiom instantiated at the root node @3, we intend to change the value of key 6 to 7. Note that when writing rules or axioms, persistent facts are preceded with a `!`.

Figure 2 represents the trace of the algorithm. Note that the program database is partitioned by the tree nodes using the first argument of each fact. In Fig. 2a we present the database filled with the program's axioms. Next, we follow the right branch using rule 3 since $6 > 3$ (Fig. 2b). We then use the same rule again in Fig. 2c where we finally reach the key 6. Here, we apply rule 1 and `value(@6, 6, 6)` is updated to `value(@6, 6, 7)`.

```
type left(node, node).
type right(node, node).
type linear value(node, int, int).
type linear replace(node, int, int).

replace(A, K, New), value(A, K, Old)
   -o value(A, K, New). // we found our key

replace(A, RKey, RValue), value(A, Key, Value), !left(A, B), RKey < Key
   -o value(A, Key, Value), replace(B, RKey, RValue). // go left

replace(A, RKey, RValue), value(A, Key, Value), !right(A, B), RKey > Key
   -o value(A, Key, Value), replace(B, RKey, RValue). // go right

// binary tree configuration
value(@3, 3, 3). value(@1, 1, 1). value(@0, 0, 0).
value(@2, 2, 2). value(@5, 5, 5). value(@4, 4, 4).
value(@6, 6, 6).
!left(@1, @0). !left(@3, @1). !left(@5, @4).
!right(@1, @2). !right(@3, @5). !right(@5, @6).

replace(@3, 6, 7). // replace value of key 6 to 7
```

Fig. 1: Binary tree dictionary: replacing a key's value.

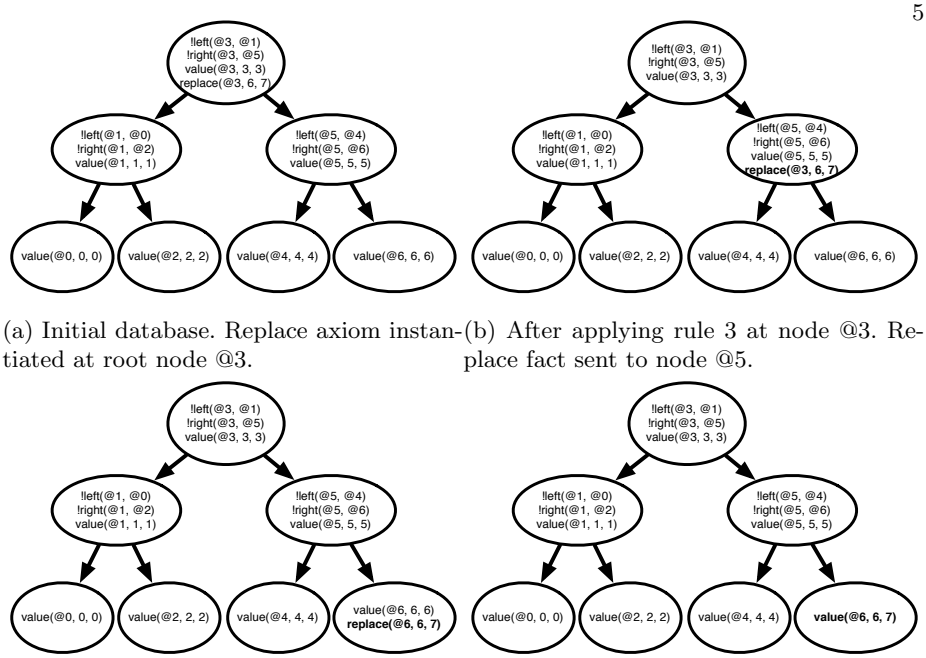
2.2 Syntax

Table 1 shows the abstract syntax for rules in LM. An LM program $Prog$ consists of a set of derivation rules Σ and a database D. Each derivation rule R can be written as $BE \multimap HE$ where BE is the body of a rule and HE is the head. Rules without bodies are allowed in LM and they are called *axioms*. Rules without heads are specified using 1 as the rule head. The body of a rule, BE, may contain linear (L) and persistent (P) *fact expressions* and constraints (C). Fact expressions are template facts that instantiate variables from facts in the database. Such variables are declared using either $\forall_x.R$ or $\exists_x.BE$. If using $\forall_x.R$ variables can also be used in the head of the rule. Constraints are boolean expressions that must be true in order for the rule to be fired. Constraints use variables from fact expressions and are built using a small functional language that includes mathematical operations, boolean operations, external functions and literal values. The head of a rule, HE, contains linear (L) and persistent (P) *fact templates* which are uninstantiated facts to derive new facts. The head can also have *comprehensions* (CE) and *aggregates* (AE). Head expressions may use the variables instantiated in the body.

Comprehensions Sometimes we need to consume a linear fact and then immediately generate several facts depending on the contents of the database. To solve this particular need, we created the concept of comprehensions, which are subrules that are applied with all possible combinations of facts from the database. In a comprehension $\{\ \widehat{x};\ BE;\ SH\ \}$, $\widehat{x}$ is a list of variables, BE is the body of the comprehension and SH is the head. The body BE is used to generate all possible combinations for the head SH, according to the facts in the database.

The following example illustrates a simple program that uses comprehensions:

```
!edge(@1, @2).
!edge(@1, @3).
```



(a) Initial database. Replace axiom instantiated at root node @3.

(b) After applying rule 3 at node @3. Replace fact sent to node @5.

(c) After applying rule 3 at node @5. Replace fact reaches node @6.

(d) After applying rule 1 at node @6. Value of key 6 has changed to 7.

Fig. 2: An execution trace for the binary tree dictionary algorithm.

```
iterate(@1).
iterate(A) -o {B | !edge(A, B) | perform(B)}.
```

When the rule is fired, we consume `iterate(@1)` and then generate the comprehension. Here, we iterate through all the `edge/2` facts that match `!edge(@1, B)`, which are: `!edge(@1, @2)` and `!edge(@1, @3)`. For each fact, we derive `perform(B)`, namely: `perform(@2)` and `perform(@3)`.

Aggregates Another useful feature in logic programs is the ability to reduce several facts into a single fact. In LM we have aggregates (AE), a special kind of sub-rule that works very similarly to comprehensions. In the abstract syntax $[\ A\ \Rightarrow\ y;\ \widehat{x};\ BE;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in BE, SH_1 and SH_2 and y is the variable in the body BE that represents the values to be aggregated using A. Like comprehensions, we use $\widehat{x}$ to try all the combinations of BE, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y and derive SH_2 only once using y. As an example, consider the following program:

```
price(@1, 3).
price(@1, 4).
price(@1, 5).
count-prices(@1).
count-prices(A) -o [sum => P | . | price(A, P) | 1 | total(A, P)].
```

Program	$Prog$	$::= \Sigma, D$
Set Of Rules	Σ	$::= \cdot \mid \Sigma, R$
Database	D	$::= \Gamma; \Delta$
Rule	R	$::= BE \multimap HE \mid \forall_x.R$
Body Expression	BE	$::= L \mid P \mid C \mid BE, BE \mid \exists_x.BE \mid 1$
Head Expression	HE	$::= L \mid P \mid HE, HE \mid CE \mid AE \mid 1$
Linear Fact	L	$::= l(\hat{x})$
Persistent Fact	P	$::= !p(\hat{x})$
Constraint	C	$::= c(\hat{x})$
Comprehension	CE	$::= \{\ \widehat{x};\ BE;\ SH\ \}$
Aggregate	AE	$::= [\ A\ \Rightarrow\ y;\ \widehat{x};\ BE;\ SH_1;\ SH_2\ ]$
Aggregate Operation	A	$::= \texttt{min} \mid \texttt{max} \mid \texttt{sum} \mid \texttt{count}$
Sub-Head	SH	$::= L \mid P \mid SH, SH \mid 1$
Known Linear Facts	Δ	$::= \cdot \mid \Delta, l(\hat{t})$
Known Persistent Facts	Γ	$::= \cdot \mid \Gamma, !p(\hat{t})$

Table 1: Abstract syntax of LM.

By applying the rule, we consume `count-prices(@1)` and derive the aggregate which consumes all the `price(@1, P)` facts. These are summed and `total(@1, 12)` is derived. LM provides several aggregate operations, including the *minimum*, *maximum*, *sum*, and *count*.

2.3 Concurrency

LM is at its core a concurrent programming language. The database of facts can be seen as a graph data structure where each node contains a fraction of the database. To accomplish this, we force the first argument of each predicate to be typed as a *node*. We then restrict the derivation rules to only manipulate facts belonging to a single node. However, the expressions in the head may refer to other nodes, as long as those nodes are instantiated in the body of the rule.

Due to the restrictions on LM rules, nodes are able to run rules independently without using other node's facts. Node computation follows a *don't care* or *committed choice* non-determinism since any node can be picked to run as long as it contains enough facts to fire a derivation rule. Facts coming from other nodes will arrive in order of derivation but may be considered partially and there is no particular order among the neighborhood. To improve concurrency, the programmer is encouraged to write rules that take advantage of the non-deterministic nature of execution.

3 The Virtual Machine

We developed a compiler that compiles LM programs to byte-code and a multi-threaded virtual machine (VM) using POSIX threads to run the byte-code. The

goal of our system is to keep the threads as busy as possible and to reduce inter-thread communication.

The load balancing aspect of the system is performed by our work scheduler that is based on a simple work stealing algorithm. Initially, the system will partition the application graph of N nodes into P subgraphs (the number of threads) and then each thread will work on their own subgraph. During execution, threads can steal nodes of other threads to keep themselves busy.

Reduction of inter-thread communication is achieved by first ordering the node addresses present in the code in such a way that closer nodes are clustered together and then partitioning them to threads. During compilation, we take note of predicates that are used in communication rules (arguments with type *node*) and then build a graph of nodes from the program's axioms. The nodes of the graph are then ordered by using a breadth-first search algorithm that changes the nodes of addresses to the domain $[0, n[$, where n is the number of nodes. Once the VM starts, we simply partition the range $[0, n[$.

Multicore When the VM starts, it reads the byte-code file and starts all threads. As a first step, all threads will grab their own nodes and assign the `owner` property of each. Because only one thread is allowed to do computation on a node at any given time, the owner property defines the thread with such permission. Next, each thread fills up its *work queue* with the initial nodes. This queue maintains the nodes that have new facts to be processed. When a node sends a fact to another node, we need to check if the target node is owned by the same thread. If that is not the case, then we have a point of synchronization and we may need to make the target thread active.

The main thread loop is shown in Fig. 3, where the thread inspects its work queue for active nodes. Procedure `process_node()` takes a node with new candidate rules and executes them. If the work queue is empty, the thread attempts to steal one node from another thread before becoming idle. Starting from a random thread, it cycles through all the threads to find one active node. Eventually, there will be no more work to do and the threads will go idle. There is a global atomic counter, a global boolean flag and one boolean flag for each thread that are used to detect termination. Once a thread goes idle, it decrements the global counter and changes its flag to idle. If the counter reaches 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.

Byte-Code A byte-code file contains meta-data about the program's predicates, initial nodes, partitioning information, and code for each rule. Each VM thread has 32 registers that are used during rule execution. Registers can store facts, integers, floats, node addresses and pointers to runtime data structures (lists and structures). When registers store facts, we can reference fields in the fact through the register.

Consider a rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y)` and a database with `!a(1,2)`, `!a(2,3)`, `b(1,3)`, `b(5,3)`, `c(1,2)`, `c(1,3)`, `c(5,3)`. Rule execution

```
void work_loop(thread_id tid):
   while (true):
      current_node = NULL;
      if(has_work(tid)):
         current_node = pop_work(tid); // take node from the queue
      else:
         target_thread = random(NUM_THREADS);
         for (i = 0; i < NUM_THREADS && current_node == NULL; ++i): // need to steal a node
            target_thread = (target_thread + 1) % NUM_THREADS;
            current_node = steal_node_from_thread(target_thread)
      if(current_node == NULL):
         become_idle(tid);
         if(!synchronize_termination(tid)):
            return;
         become_active(tid);
      else:
         process_node(current_node, tid);
```

Fig. 3: Thread work loop.

proceeds in a series of recursive loops, as follows: the first loop retrieves an iterator for the persistent facts of `!a/2` and moves the first valid fact, `!a(1,2)`, to register 0; the inner loop retrieves linear facts that match `b(1,Z)` (from the *join constraint*) and moves `b(1,3)` to register 1; in the final loop we move `c(1,2)` to register 2 and the body of the rule is successfully matched. Next, we derive `d(2)`, where 2 comes from register 0. Fig. 4 shows the byte-code for this example.

```
PERSISTENT ITERATE a MATCHING TO reg 0
  LINEAR ITERATE b MATCHING TO reg 1
      (match).0=0.0
    LINEAR ITERATE c MATCHING TO reg 2
        (match).0=0.0
        (match).1=0.1
      ALLOC d TO reg 3
      MVFIELDFIELD 0.1 TO 3.0
      ADDLINEAR reg 3
      REMOVE reg 2
      REMOVE reg 1
      TRY NEXT
    NEXT
  NEXT
RETURN
```

Fig. 4: Byte-code for rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y).`

In case of failure, we jump to the previous outer loop in order to try the next candidate fact. If a rule matches and the head is derived, we backtrack to the inner most *valid loop*, i.e., the first inner loop that uses linear facts or, if there are no linear facts involved, to the previous inner loop. We need to jump to a valid loop because we may have loops with linear facts that are now invalid. In our example, we would jump to the loop of `b(X,Z)` and not `c(X,Y)`, since `b(1,3)` was consumed.

The compiler re-orders the fact expressions used in the body in order to make execution more efficient. For example, it forces the join constraints in rules to appear at the beginning so that matching will fail sooner rather than later. It also does the same for constraints. Note that for every loop, the compiler adds a *match object*, which contains information about which arguments need to match, so that runtime matching is efficient.

Our compiler also detects cases where we re-derive a linear fact with new arguments. For example, as shown in Fig. 5, the rule `a(N) -o a(N+1)` will compile to code that reuses the old `a(N)` fact. We use a `flags` field to mark updated nodes (presented next).

```
LINEAR ITERATE a MATCHING TO reg 0
  MVFIELDREG 0.0 TO reg 1
  MVINTREG INT 1 TO reg 2
  reg 1 INT PLUS reg 2 TO reg 3
  MVREGFIELD reg 3 TO 0.0
  UPDATE reg 0
  TRY NEXT
RETURN
```

Fig. 5: Byte-code for rule `a(N) -o a(N+1)`.

Database Data Structures We said before that LM rules are constrained by the first argument. Because nodes can be execute independently, our database is indexed by the node address and each sub-database does not need to deal with synchronization issues since at any given point, only one thread will be using the database. Note that the first argument of each fact is not stored.

The database must be implemented efficiently because during matching of rules we need to restrict the facts using a given match object, which fixes arguments of the target predicate to instantiated values. Each sub-database is implemented using three kinds of data structures:

- *Tries* are used exclusively to store persistent facts. Tries are trees where facts are indexed by the common arguments.
- *Doubly Linked Lists* are used to store linear facts. We use a double linked list because it is very efficient 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 (see "Indexing") and then uses a hash table indexed by the appropriate argument. If we need to go through all the facts, we just iterate through all the facts in the table. For collisions, we use the above doubly linked list data structure.

Figure 6 shows an example for a hash table data structure with 3 linear facts indexed by the second argument and stored as doubly linked list in bucket 2. Each linear fact contains the regular list pointers, a `flags` field and the fact arguments. Those are all stored continuously to improve data locality. One use of the `flags` field is to mark that a fact is already being used. For example, consider the rule body `a(A,B), a(C,D) -o ...`. When we first pick a fact for `a(A, B)` from the hash table, we mark it as being used in order to ensure that, when we retrieve facts for `a(C, D)`, the first one cannot be used since that would violate linearity.

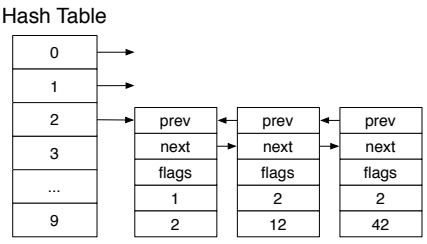
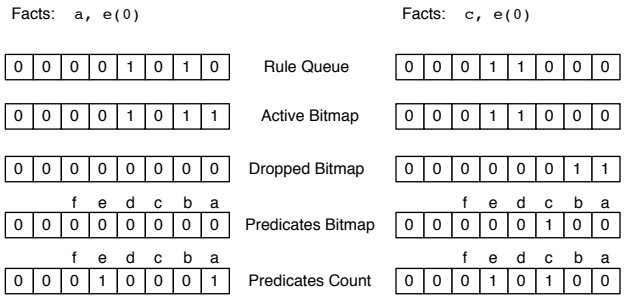


Fig. 6: Hash table data structure for storing predicate `a(int,int)`.

Rule Engine The rule engine decides which rules may need to be executed while taking into account rule priorities. There are 5 main data structures for scheduling rule execution; `Rule Queue` is the bitmap representing the rules that will be run; `Active Bitmap` contains the rules that can be fired since there are enough facts to activate the rule's body; `Dropped Bitmap` contains the rules that must be dropped from `Rule Queue`; `Predicates Bitmap` marks the newly derived facts; and `Predicates Count` counts the number of facts per predicate. To understand how our engine works, consider the program in Fig. 7a.

```
a, e(1) -o b.
a -o c.
b -o d.
e(0) -o f.
c -o e(1).
```

(a) Program



(b) Before and after applying the 2nd rule.

Fig. 7: Program and rule engine data structures.

We take the least significant rule from the `Rule Queue` bitmap, which is the candidate rule with the higher priority, and then run it. In our example, since we have facts `a` and `e(0)`, we will execute the second rule `a -o c`. Because the derivation is successful, we will consume `a` and derive `c`. We thus mark the `c` predicate in the `Predicates Bitmap` and the first and second rules in `Dropped Bitmap` since such rules are no longer applicable (`a` is gone). To update the `Rule Queue`, we remove the bits marked in `Dropped Bitmap` and add the active rules marked in `Active Bitmap` that are affected by predicates in `Predicates Bitmap`. The engine thus schedules the fourth and fifth rules to run (Fig. 7b).

Note that every node in the program has the same set of data structures present in Fig. 7. We use 32 bits integers to implement bitmaps and an array of 16 bits integers to count facts, resulting in $32 + 2P$ bytes per node, where P is the number of predicates.

We do a small optimization to reduce the number of derivations of persistent facts. We divide the program rules into two sets: *persistent rules* and *non persistent rules*. Persistent rules are rules where only persistent facts are involved. We compile such rules incrementally, that is, we attempt to fire all rules where a persistent fact is used. This is called the *pipelined semi-naive* evaluation and it originated in the P2 system [12]. This evaluation method avoids excessing rederivations of the same fact. The order of derivation does not matter for those rules, since only persistent facts are used.

Thread Interaction Whenever a new fact is derived through rule derivation, we need to update the data structures for the corresponding node. This is trivial if the thread that owns the node derived the fact also. However, if a thread $T1$ derives a fact for a node owned by another thread $T2$, then we may have problems because $T2$ may be also updating the same data structures. We added a lock and a boolean flag to each node to protect the access to its data structures. When a node starts to execute, we activate the flag and lock the node. When another thread tries to use the data structures, if first checks the flag and if not activated, it locks the node and performs the required updates. If the flag is activated, it stores the new fact in a list to be processed before the node is executed.

Indexing To improve fact lookup, the VM employs a fully dynamic mechanism to decide which argument may be optimal to index. The algorithm is performed in the beginning of the execution and empirically tries to assess the argument of each predicate that more equally spreads the database across the values of the argument. A single thread performs the algorithm for all predicates.

The indexing algorithm is performed in three main steps. First, it gathers statistics of lookup data by keeping a counter for each predicate's argument. Every time a fact search is performed where arguments are fixed to a value, the counter of such arguments is incremented. This phase is performed during rule execution for a small fraction of the nodes in the program.

The second step of the algorithm then decides the candidate arguments of each predicate. If a predicate was not searched with any fixed arguments, then it will be not indexed. If only one argument was fixed, then such argument is set as the indexing argument. Otherwise, the top 2 arguments are selected for the third phase, where *entropy statistics* are collected dynamically.

During the third phase, each candidate argument has an entropy score. Before a node is executed, the facts of the target predicate are used in the following formula applied for the two arguments:

$$Entropy(A, F) = -\sum_{v \in values(F,A)} \frac{count(F, A = v)}{total(F)} \log_2 \frac{count(F, A = v)}{total(F)}$$

Where A is the target argument, F is the multi-set of linear facts for the target predicate, $values(F, A)$ is set of values of the argument A, $count(F, A = v)$ counts the number of linear facts where argument A is equal to v and $total(F)$ counts the number of linear facts in F. The entropy value is a good metric because it tells us how much information is needed to describe an argument. If more information is needed, then that must be the best argument to index.

For one of the arguments to score, $Entropy(A, F)$ multiplied by the number of times it has been used for lookup must be larger than the other argument.

The argument with the best score is selected and then a global variable called `indexing_epoch` is updated. In order to convert the node's linked lists into hash tables, each node also has a local variable called `indexing_epoch` that is compared to the global variable in order to rebuild the node database according to the new indexing information.

Our VM also dynamically resizes the hash table if necessary. When the hash table becomes too dense, it is resized to the double. When it becomes too sparse, it is reduced in half or simply transformed back into a doubly linked list. This is done once in a while, before a node executes.

We have seen very good results with this scheme. For example, for the all-pairs shortest paths program, we obtained a 2 to 5-fold improvement in sequential execution time. The overhead of dynamic indexing is negligible since programs run almost as fast as if the indices have been added from the start.

4 Preliminary Results

This section presents preliminary results for our VM. First, we present scalability results in order to show that LM programs can take advantage of multicore architectures. Next, we present a comparison with similar programs written in other programming languages in order to show evidence that our VM is viable.

For our experimental setup, we used a machine with two 16 (32) Core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of RAM memory and running the Linux kernel 3.8.3-1.fc17.x86_64. We compiled our VM using GCC 4.7.2 (g++) with the flags `-O3 -std=c+0x -march=x86-64`. We ran all experiments 3 times and then averaged the execution time.

Scalability Results For this section, we run each program using 1, 2, 4, 6, 8, 10, 12, 14 and 16 threads and compared the runtime against the execution of the sequential version of the VM. We used the following programs:

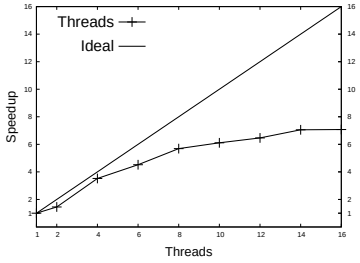
- Greedy Graph Coloring (GGC) colors nodes in a graph so that no two adjacent nodes have the same color. We start with a small number of colors and then we expand the number of colors when we cannot color the graph.
- PageRank implements a PageRank algorithm without synchronization between iterations. Every time a node sends a new rank to its neighbors and the change was significant, the neighbors are scheduled to recompute their ranks.
- N-Queens, the classic puzzle for a 13x13 board.
- Belief Propagation, a machine learning algorithm to denoise images.

Figure 8 presents the speedup results for the GGC program using 2 different datasets. In Fig. 8a we show the speedup for a search engine graph of 12,000 webpages[4]. Since this dataset follows the power law, that is, there is a small number of pages with a lots of links (1% of the nodes have 75% of the edges), the speedup is slightly worse than the benchmark shown in Fig. 8b, where we use a random dataset of 2,000 nodes with an uniform distribution of edges.

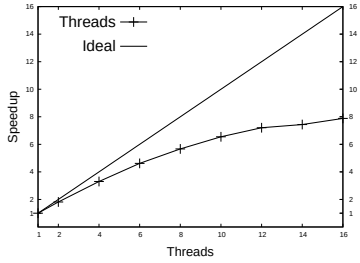
The PageRank results are shown in Fig. 9. We used the same search engine dataset as before and a new random dataset with 5,000 nodes and 500,000 edges. Although the search engine graph (Fig. 9a) has half the edges (around 250,000), it scales better than the random graph (Fig. 9b), meaning that the PageRank program depends on the number of nodes to be more scalable.

The results for the N-Queens program are shown in Fig. 10a. The program is not regular since computation starts at the top of the grid and then rolls down, until only the last row be doing computation. Because the number of valid states for the nodes in the upper rows is much less than the nodes in the lower rows, this may potentially lead to load balancing problems. The results show that our system is able to scale well due to work stealing.

[4] Available from `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

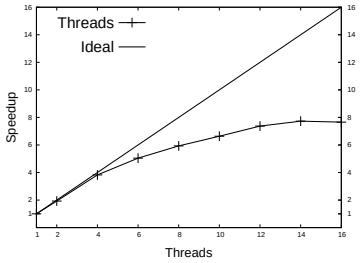


(a) Using a graph of web pages with around 12,000 nodes and 292,000 edges.

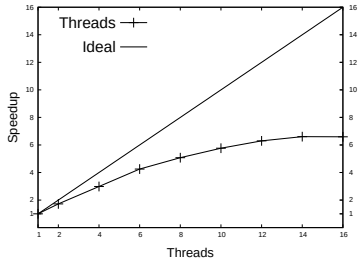


(b) Using a random graph with 2,000 nodes and 600,000 edges.

Fig. 8: Experimental results for the GGC algorithm.



(a) Using a graph of web pages collected from a search engine (around 12,000 nodes and 292,000 edges)



(b) Using a random graph with 5,000 nodes and 500,000 edges.

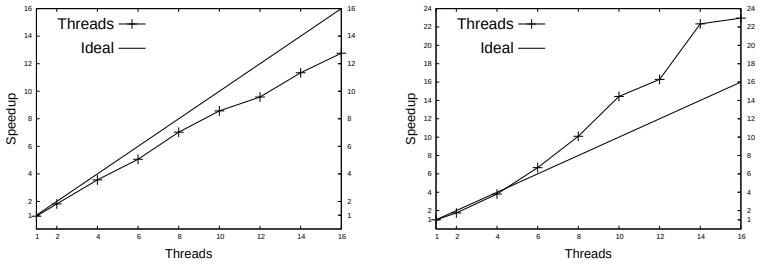
Fig. 9: Experimental results for the asynchronous PageRank algorithm.

Finally, we shown the results for the Belief Propagation (BP) program in Fig. 10b. BP is a regular and asynchronous program and benefits (as expected) from having multiple threads executing since the belief values of each node will converge faster. The super-linear results prove this assertion.

Absolute Execution Time As we have seen, our VM scales reasonably well, but how does it compare in terms of absolute execution time with other competing systems? We next present such comparison for the execution time using one thread.

In Fig. 11a we compare the LM's N-Queens version against 3 other versions: a straightforward sequential program implemented in C using backtracking; a sequential Python [15] implementation; and a Prolog implementation executed in YAP Prolog [5], an efficient implementation of Prolog. Numbers less than 1 mean that LM is faster and larger than 1 mean that LM is slower. We see that LM easily beats Python, but is 5 to 10 times slower than YAP and around 15 times slower than C. However, note that if we use at least 16 threads in LM, we can beat the sequential implementation written in C.

In Fig. 11b we compare LM's Belief Propagation program against a sequential C version, a Python version and a GraphLab version. GraphLab [13] is a parallel C++ library used to solve graph-based problems in machine learning. C and GraphLab perform about the same since both use C/C++. Python runs



(a) N-Queens program (13x13 board). (b) BP (400x400 image).

Fig. 10: Experimental Results for N-Queens and Belief Propagation.

Size	C	Python	YAP Prolog
10	16.92	**0,62**	5,42
11	21.59	**0.64**	6.47
12	10.32	**0.73**	7.61
13	14.35	**0.88**	10.38

(a) N-Queens problem.

Size	C	Python	GraphLab
10	**0.67**	**0,03**	**1.00**
50	*1.77*	**0.04**	*1.73*
200	*1.99*	**0.05**	*1.79*
400	*2.00*	**0.04**	*1.80*

(b) Belief Propagation program.

Fig. 11: Comparing absolute execution times (System Time / LM Time).

very slowly since it is a dynamic programming language and BP has many mathematical computations. We should note, however, that the LM version uses some external functions written in C++ in order to improve execution time, therefore the comparison is not totally fair.

We also compared the PageRank program against a similar GraphLab version and LM is around 4 to 6 times slower. Finally, our version of the all-pairs shortest distance algorithm is 50 times slower than a C sequential implementation of the Dijkstra algorithm, but it is almost twice as fast when compared to the same implementation in Python.

5 Conclusions

We have presented a parallel virtual machine for executing forward-chaining linear logic programs, with particular focus on parallel scheduling on multicores, rule execution and database organization for fast insertion, lookup, and deletion or linear facts. Our preliminary results show that the VM is able to scale the execution of some programs when run with up to 16 threads. Although this is still a work in progress, the VM fairs relatively well against other programming languages. Moreover, since LM programs are concurrent by default, we can easily get better performance from the start by executing them with multiple threads.

In the future, we want to improve our work stealing algorithm so that each thread steals nodes that are a better fit to the set of nodes owned by the thread (e.g. neighboring nodes). We also intend to take advantage of linear logic to perform whole-program optimizations, including computing program invariants and loop detection in rules.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme; by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and within projects SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281); and by the Qatar National Research Fund under grant NPRP 09-667-1-100. Flavio Cruz is funded by the FCT grant SFRH / BD / 51566 / 2011.

References

1. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A language for large ensembles of independently executing nodes. In: International Conference on Logic Programming (ICLP) (2009) 1
2. Ashley-Rollman, M.P., Rosa, M.D., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J.D.: Declarative programming for modular robots. In: Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007 (2007) 1
3. Betz, H., Frühwirth, T.: A linear-logic semantics for constraint handling rules. In: Principles and Practice of Constraint Programming - CP 2005, Lecture Notes in Computer Science, vol. 3709, pp. 137–151 (2005) 1
4. Colmerauer, A., Roussel, P.: The birth of prolog. In: The Second ACM SIGPLAN Conference on History of Programming Languages. pp. 37–52. New York, NY, USA (1993) 1
5. Costa, V.S., Damas, L., Rocha, R.: The yap prolog system. CoRR abs/1102.3896 (2011) 4
6. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987) 1
7. Gonzalez, J., Low, Y., Guestrin, C.: Residual splash for optimally parallelizing belief propagation. In: Artificial Intelligence and Statistics (AISTATS) (2009) 1
8. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: A survey. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(4), 472–602 (2001) 1
9. Holzbaur, C., de la Banda, M.J.G., Stuckey, P.J., Duck, G.J.: Optimizing compilation of constraint handling rules in hal. CoRR cs.PL/0408025 (2004) 1
10. Lam, E.S.L., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. CoRR abs/1006.3039 (2010) 1
11. Liu, M.: Extending datalog with declarative updates. In: International Conference on Database and Expert System Applications (DEXA). vol. 1873, pp. 752–763 (1998) 1
12. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M.: Declarative networking: Language, execution and optimization. In: International Conference on Management of Data (SIGMOD). pp. 97–108 (2006) 1, 3
13. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. In: Conference on Uncertainty in Artificial Intelligence (UAI). pp. 340–349 (2010) 4
14. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming 23, 125–149 (1993) 1
15. van Rossum, G.: Python reference manual. Report CS-R9525, Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands (Apr 1995), `http://www.python.org/doc/ref/ref-1.html` 4