Towards Parallel and Distributed Programming using Bottom-Up Logic Programming

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

Michael Ashley-Rollman Seth Copen Goldstein
Carnegie Mellon University, Pittsburgh, PA 15213
{mpa,seth}@cs.cmu.edu

Abstract

In the last few years there as been a steady increase of processing power. Multicore processors are now increasingly prevalent and networks of commodity computers present an opportunity to solve bigger problems. Programming in these kinds of systems is not as simple as it should be since optimizing programs to run on different types of distributed systems is notoriously hard. We propose a logic programming language based on Datalog that is suitable for developing general purpose programs for both parallel and distributed systems. The language uses the notion of link restricted rules for parallelizing computation and *action facts* for doing input/output. We developed a compiler and a runtime system with several execution strategies, from multithreaded execution to distributed execution using MPI. We also developed several techniques based on XY-stratification to reduce memory usage. Finally, we present several application cases, from graph problems to machine learning algorithms.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms

Keywords

1. Introduction

The last decade has seen a priority shift for processor companies. If clock frequency was once the main metric for performance, today computing power is measured in number of cores in a single chip. For software developers and computer scientists, once focused in developing sequential programs, newer hardware usually meant faster programs without any change to the source code. Today, the free lunch is over. Multicore processors are now forcing the development of new software methodologies that take advantage of increasing processing power. However, parallel programming is difficult, usually because programs are written in imperative stateful programming languages that make use of relatively low level synchronization primitives such as locks, mutexes and barriers. This tends to make the task of managing multithreaded execution quite intricate and error-prone, resulting in race hazards and deadlocks. In the future, *many-core* processors will make this task even more daunting.

On the other hand, advances in network speed and bandwidth are making distributed computing more appealing. For instance, *cloud computing* is a new emerging paradigm that wants to make every computer connected to the Internet connected to the same pool of computing power, where data can be retrieved and computation can be done. For high performance computing, the *computer cluster* is a well established paradigm that uses fast local area networks to improve performance and solve problems that would take a long time with a single computer.

Developments in parallel and distributed programming have originated several programming models. At the end of the spectrum are lower-level programming abstractions such as *message passing* (e.g., MPI [11]) and *shared memory* (e.g., Pthreads and OpenMP [8]). While such abstractions are very expressive and enable the programmer to write very performant code, they tend to be very hard to use and debug, due to synchronization problems, making it difficult to prove the program's correctness.

An approach that attempts to address the correctness challenges, is to use declarative languages such as *logic programming languages* or *functional programming languages*. In logic languages such as Prolog, researchers took advantage of the non-determinism of proof-search to evaluate subgoals in parallel with models such as *or-parallelism* [1] and *and-parallelism* [16]. In functional languages, the stateless nature of computation allows multiple expressions to evaluate safely in parallel. This has been explored in several languages such as NESL [7] and Id [15].

Recently, there has been an increasing interest in declarative and data-centric languages. MapReduce [10], for instance, is a popular data-centric programming model that is optimized for large clusters. The scheduling and data sharing model is very simple: in the *map phase*, data in each node is transformed and in the *reduce phase*, nodes communicate to compute the final result.

A declarative approach that is becoming popular is Datalog [17], a bottom-up logic programming language. Traditionally used in deductive databases, Datalog is being increasingly used in different fields such as: distributed networking [14], sensor nets [9] and cloud computing [2].

In the context of the Claytronics project [12], a massive distributed system we've been using a Datalog variant called Meld [3, 4]. Meld is specially suited for writing programs for distributed systems where the network topology can change dynamically. Such systems are called *ensembles* and they include programmable sensor networks and modular robotic systems.

We now have been adapting Meld to implement general parallel algorithms in multicore machines and clusters. We extended the language with lists and reduced the need to do deletion. We are using the concept of *action facts* to model output.

We have implement a new compiler and a virtual machine that is able to execute on multicore machines and distributed networks. For multicore machines we have implemented different scheduling schemes, including static division of work and work stealing. For distributed networks and clusters we are using MPI with a static division of work. Finally, the runtime system is able to take advantage of multicores when doing distributed computation.

The rest of the paper is organized as follows. In the next section we describe the Meld language and the modifications we made to make it more suitable for parallel programming. Next, we present how the runtime system distributes the computation across processors and machines and the different scheduling schemes. In Section 5 we evaluate the language modifications with several applications from graph theory and machine learning. Finally, we end the paper outlining some conclusions and further work.

2. The Meld Language

Meld is a bottom-up logic programming language based on Datalog. Like Datalog, it consists of a database of *facts* and a set of *production rules* for generating new facts. Each fact is an association between a *predicate* and a tuple of values. A predicate can be seen as a relation or table in which the tuples are its elements. Production rules have the form $\mathtt{p} : -\mathtt{q}_1, \mathtt{q}_2, ..., \mathtt{q}_n.$, meaning that "if $\mathtt{q}_1$ and $\mathtt{q}_2$ and ... and $\mathtt{q}_n$ are true then $\mathtt{p}$ is true".

2.1 Evaluation

When a Meld program starts executing, the set of initial axioms in the form p. are instantiated and added to the database. With these new facts, rules are then fired and new facts are generated and added to the database, until a *quiescent* state is reached, where no more facts can be generated. In Meld, we can also have *action facts*, which are syntactically similar to regular facts but cause some side effect. In the context of modular robotics, action facts activate the robot's actuators in order to produce movement or control devices. For parallel programming we use the concept of action facts for writing results to files. Note that action facts are not added to the database.

2.2 Rules

A Meld program contains a set of rules. A rule has two main components: the *head*, which indicates the facts to be generated; and the *body*, which contains the pre-requisites to generate the head facts. The body may contain the following as pre-requisites: *subgoals*, expression constraints or/and variable assignments. The head can only contain subgoals. Variables are limited to the scope of the rule and must be defined in the body facts or through the body variable assignments. Variables that only appear on the head are forbidden, since each instantiated fact must have only *ground facts* and thus all arguments must be instantiated. The abstract syntax for the language is presented in Fig. 1.

Whenever all body pre-requisites are satisfied, the head subgoals are instantiated as facts and then they are added to the program database (except when action facts are derived). To satisfy all pre-requisites, the body subgoals must be matched against the program database. Both constraints and subgoals match when consistent substitution is found for the body's variables such that one or more facts in the database are matched. Constraint expressions are boolean expressions that use variables from subgoals (and thus database facts) and from variable assignments. Allowed expressions in constraints include arithmetic, logical and set-based operations.

Structural Facts	$\Gamma ::=$	$\cdot \| \Gamma, f(\hat{t})$
Sensing Facts	$\Theta ::=$	$\cdot \| \Theta, f(\hat{t})$
Accumulated Actions	$\Psi ::=$	$\cdot \| \Psi, a(\hat{t})$
Set of Rules	$\Sigma ::=$	$\cdot \| \Sigma, R$
Actions	$A ::=$	$a(\hat{x})$
Facts	$F ::=$	$f(\hat{x})$
Constraints	$C ::=$	$c(\hat{x})$
Assignments	$A ::=$	$e(\hat{x})$
External Functions	$X ::=$	$f(\hat{x})$
Expression	$E ::=$	$E \wedge E \| F \| C \| A \| X$
Rule	$R ::=$	$E \Rightarrow F \| E \Rightarrow A \| agg(F, g, y) \Rightarrow F$

Figure 1. Abstract syntax for Meld programs

Each predicate used in the program must be explicitly typed. Each field is either of a basic type or a structured type. Basic types include the integer type, floating point numbers and the *node address* (see 2.4). A structured type includes lists of basic types. Syntax-wise, lists have a Prolog-like syntax.

2.3 Aggregates

In contrast to Datalog, Meld does not have negation, but has *set aggregates* [1]. The purpose of an aggregate is to define a type of predicate that combines the facts of that predicate into a single fact. The definition of an aggregate includes the field of the predicate and the type of operation to be done on the collection of facts.

Consider a predicate $\mathtt{p}(\mathtt{f}_1, ..., \mathtt{f}_{m-1}, \mathtt{agg}\ \mathtt{f}_m, \mathtt{f}_{m+1}, ..., \mathtt{fm} + \mathtt{n})$, where f_m is the aggregate field associated with the operation agg. For each combination of values $\mathtt{f}_1, ..., \mathtt{f}_{m-1}$ there will be a set of facts matching the same combination. To calculate the aggregated fact of each collection, we take all the $\mathtt{f}_m$ fields and apply the agg operation. The fields between $\mathtt{f}_{m+1}$ and $\mathtt{f}_{m+n}$ for the aggregated fact are chosen depending on the operation.

Meld allows several aggregate operations, namely: max, to compute the maximum value; min, to compute the minimum value; sum, to sum all the values of the corresponding fields; and first, to select the first fact field.

2.4 Localization

Another difference between Meld and Datalog, is that in the former, the first field of each predicate indicates must have the type node address. In rules, the first argument of each subgoal is called the *home variable* and refers to the node where the fact will be stored, so that each node has its own database.

This convention originated in the context of declarative networking [14], in the P2 system. For modular robotics, this makes it trivial to distribute data across the ensemble. In the context of this paper, each fact is a *structural fact* and can be seen as part of the data structure representing a node in the graph. Moreover, when each fact is associated with a certain node, it helps us distribute and parallelize computation and also improve data locality (see 3.1).

While each fact must be associated with some node, a rule may contain subgoals that refer to different nodes. In order to simplify rule activation, the rules are transformed so that each subgoal refers to the same node. The process of *localization* is fundamental in the distribution of computation.

Before localization, each rule can be either a *local rule* or a *link-restricted rule*. A local rule does not need to be transformed since only facts from a single node are needed. A link-restricted rule is a rule where subgoals may refer to different nodes. This type of rule must be transformed so that the right subparts of the rule are

[1] It has been shown in [18] that set aggregates can be implemented using negation, however we implement aggregates directly, without using source code transformation.

matched on the right nodes (by matching the facts of that node only) and the instantiation of the head subgoals into facts is done on the subgoal the head subgoals refer to. To accomplish this, the nodes involved in the computation of the rule must communicate between each other so that the rule is fired when everything has been known to match. For this, we introduce a special class of facts called the *link facts*. These facts represent the edges in the graph and force the nodes to do direct communication with only their direct neighbors.

For an example, consider the following block of Meld code, where a rule refers to three different nodes, A, B and C.

```
fact2(A, 2) :-
   fact1(A, A1), A1 > B1,
   edge(A, B), fact1(B, B1), B1 > C1,
   edge(A, D), fact1(D, D1),
   edge(B, C), fact1(C, C1).
```

To localize this rule, we first build a tree representing the connection paths between nodes by picking an arbitrary root node, in this case, the node A, and then by adding edges using link facts. The resulting tree is represented in Fig. 2. Note how the constraints were placed in the first node where all the variables used in the constraint were available. This optimization aims to reduce communication by testing constraints as soon as possible.

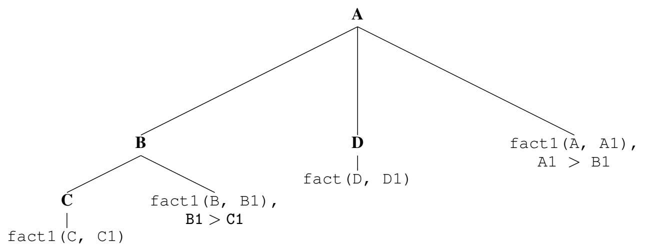

Figure 2. Localization tree

Once connection paths are known, localization transforms the original rule into several *send rules*, that are characterized by having the same node in the body and a different node in the head. Once a send rule body matches, all the subgoals in the head are instantiated and then sent to the corresponding node, where the matching of the original rule can continue, until the complete rule matches. The following code shows the final result.

```
__edge(X, Y) :- edge(Y, X).

fact2(A, 2) :-
   fact1(A, A1), A1 > B1,
   __remote1(A, B1), __remote3(A).

__remote1(A, B1) :-
   __edge(B, A), fact1(B, B1), B1 > C1,
   __remote2(B, C1).
__remote2(B, C1) :-
   __edge(C, B), fact1(C, C1).
__remote3(A) :-
   __edge(D, A), fact1(D, D1).
```

2.5 Pipelined Evaluation

Evaluation of Datalog programs (including programs with recursive rules) can be done using several techniques, one of the most well-known being the *semi-naive fixpoint* evaluation [5, 6]. However, these techniques are more appropriate for a centralized evaluation and are expensive in a distributed environment, because they require global synchronization. To this end, we use the *pipelined semi-naive* evaluation from P2 [14], which relaxes the semi-naive fixpoint by making the concept of iteration local to a node.

```
type end(node).
type route edge(node, node, int).
type path(node, min int, list node).

r1: path(A, W, [A, B]) :-
      edge(A, B, W),
      end(B).

r2: path(A, D + W, [A | P]) :-
      edge(A, B, W),
      path(B, D, P).

r3: write_int(A, W), write_list(A, P) :-
      terminated(A),
      path(A, W, P).

r4: edge(@0, @1, 1). edge(@1, @2, 1).
    edge(@2, @3, 3). edge(@0, @2, 4).
    edge(@2, @0, 2). edge(@1, @4, 2).

r5: end(@4).
```

Figure 3. Shortest Path in Meld

Whenever a new fact is generated by the node or sent by a neighbor node, the fact is added to local queue which contains all new facts. As execution proceeds, a fact pulled out of the queue. Then, all rules that use the fact in their body are selected as candidate rules. For each candidate, the rest of its rule body is matched against the facts in the database. If the rule is proved, the subgoals in the head of the rule are instantiated and added to the local queue as new facts.

2.6 Example: Shortest-Path

For an example, consider the program in Fig. 3 that computes the shortest distance and the corresponding path between any node to node 4. Each edge in the graph is represented by the edge relation in r4, and includes the source node, the destination node and the distance of the connection. Note how node addresses are prepended with the symbol @.

The path predicate is a transitive closure subject to an aggregated field, which forces the selection of the fact with the least distance. In r1, we define the path for each neighbor node of node 4, by declaring that a node A has an edge to node B and node B is the selected final node. In r2 we complete the transitive closure, declaring that the shortest path between A and a node B is the sum of the distance between A and B and the shortest distance between B and node 4.

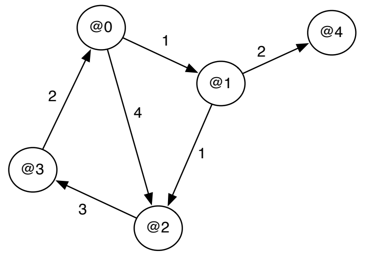

Figure 4. Shortest path graph.

Using the localization method, we know that node 4 will send a fact representing a match of rule r1 to its neighbors, in this case, node 1. Node 1 then instantiates a new path fact `path(@1, 2, [@1, @4])`. At this moment, node 1 could generate the aggregated `path` fact representing the least path to node 4 or could wait until all the `path` are generated.

In a distributed setting, it is difficult to assess if all facts of an aggregate collection are already on the database (we present techniques to avoid this in XXX). However, we use a technique borrowed from P2 called *deletion* that allows Meld to retract facts derived from invalid facts such as aggregates based on partial information or changes in the environment (in the case of modular robotics). Deletion is presented in detail in the next section.

In this case of this graph (Fig. 4), node 1 can generate the least path safely and send it to node 0. Node 0 receives this fact and generates a new aggregate: `path(@0,3,[@0,@1,@4])`. Node 0 then sends a new fact to node 3, node 3 generates a new aggregate and generates a new fact that is sent to node 2. Node 2 then generates its least path aggregate: `path(@2,8,[@2,@3,@0,@1,@4])` and fires rule r2 for neighbors node 0 and node 1. However these new paths are not minimal and thus nothing is generated, resulting in a fixpoint in the graph. If, however, a better path was found, the everything derived from the old path would have to be deleted and the new path added to the database.

After the fixpoint is reached, a new fact, `terminated` is generated for each node and rule r3 is activated, which forces the writing of the minimum distance and the shortest path to the terminal. The `terminated` fact is called a *sensing fact*, since it is not part of the node data structure, but is associated with the state of execution.

2.7 Deletion

Deletion is mechanism that allows Meld to delete all facts derived from invalid facts such as aggregates based on partial (and incomplete) information. In modular robotics, deletion is used to delete everything derived from old world information and thus keep the database of facts consistent to the robot's environment.

Deletion works by considering a deleted fact and matching the rules in exactly the same way as derivations are done to determine which facts depend on the deleted one. Care must be taken when deleting facts that were derived by different rules. To accomplish this, we use a reference counting scheme similar to garbage collection techniques. We keep track of the number of derivations of the fact and, optionally, the the depth of derivation of the fact.

The reference count is incremented or decremented whenever the fact is derived or deleted, respectively. When the reference count reaches zero, we delete the fact and anything derived from it. The derivation depth is used for facts that have cyclic derivations so that no infinite cyclic derivations are left with no start. This mechanism is throughly explained in [4].

2.8 Aggregates and Recursive Rules

2.8.1 XY-Stratification

[18]

2.8.2 Aggregate modifiers

Aggregate modifiers

3. Parallel Execution

Distributing the execution of Meld programs is relatively easy since our compiler automatically transforms the original program into fully distributed code. Due to the language semantics, messaging and placement of data is pre-defined which helps distribute execution.

In our execution model, we have the set of nodes in the graph and a set of *workers*. A worker is an independent unit of processing and it is usually either a *thread* or a *process*. A worker can process new facts at most from one node at the same time and a node can be owned at most by one worker. This restriction simplifies parallelization by not allowing several works to manipulate the database of a node at the same time.

The main task of our runtime system is then to balance the load between workers, so that we can maximize the speedups. However, we need to make a distinction between *monotonic* facts (non-aggregates) and *non-monotonic facts* (aggregates). The former maximizes distribution, while the latter requires some form of synchronization in order to minimize deletion of facts. Note that doing computation with the wrong fact and the fact that workers compute in a non-deterministic manner, it makes the completion process highly non-deterministic, since many facts can be wrongly derived and then deleted and recomputed, everything depending on how the order of processing of workers.

In order to make evaluation deterministic for any set of workers, the distributed evaluation procedure of Meld programs is as follows:

- Workers process all monotonic facts (i.e., non-aggregates) and no aggregates are fired, except if they can be generated *safely*;
- When a worker does not have more work, it enters into the *idle state*.
- Once the graph reaches a quiescent state, that is, when all workers are in the idle state, all workers synchronize through a *termination barrier*;
- Upon synchronization, aggregates are then generated using the facts of each aggregate collection;
- If any new fact was derived (or a deletion is to be done), they are added to the corresponding queues, and we start a new *computation round*;
- If no new facts were derived, the computation is marked as complete and finishes.

Parallelization is maximized when the number of global synchronizations required to compute the whole program is minimized. Making the generation of aggregated facts a local decision at the node level thus contributes to increased asynchronicity between nodes and increased throughput of the whole system.

In the remaining subsections we describe several optimizations and scheduling strategies of our runtime system.

3.1 Topology Ordering

For certain scheduling schemes, in order to distribute computation across workers it is important to increase locality of communication, so that a node makes most of its communication to neighbor nodes that are being handled by the same worker. This means that the data travels in the same worker, which can potentially increase performance. If a worker is a process, this means that less inter-process communication or network communication will be done, and, in the case of a thread, it may mean more data locality and cache hits.

Our compiler is able to know how many nodes are in the graph and, in most cases, all the edges of the graph. This is detected by analyzing the node address constants (that are prepended by the symbol @) and the axioms of the program. After parsing and type-checking the program code, the compiler then optimizes the topology by building an internal representation of the graph. In this phase, each node address a is mapped using a function $M(x)$ to a normalized node address n. Function $M(x)$ is bijective and the domain is the set of all nodes described in the source code. The image of M is $[0, N[$, where N is the number of nodes in the graph. The byte-code of a Meld program includes all the pairs $(x, M(x))$ so that the runtime system can put this information to use.

We have two methods for defining the function $M(x)$:

- *Randomized*: the mapping is done randomly.
- *Breadth-First*: the mapping is built by picking an arbitrary node, n_{zero} and setting $M(n_{zero}) = 0$, then we select all neighbors of n_{zero} and start defining their mappings in increasing order, 1, ..., $N - 1$, and adding its neighbors for later processing in a breadth first fashion.

The breadth-first method is used with the intent of clustering closer nodes in an ordered fashion. While not optimal, using a breadth-first approach is very efficient and has good results for irregular graphs. If we have a static division of work between N workers, where each worker is responsible to process a pre-defined set of nodes, we can efficiently slice the domain of function $M(x)$ and divide it between the N workers.

For an example, consider the graph in Fig. 5. The node addresses represented are the ones included in the source code. Using a breadth-first method starting by node 1, we get the following order: 1, 2, 3, 7, 5, 6 and 4. If we had to do a static division of worker for 2 workers, worker 1 would get 1, 2, 3, 7 and worker 2 would get 5, 6 and 4. Note on Fig. 5 that only 3 edges exist between the nodes of worker 1 and worker 2. This greatly reduces communication between workers and improves parallel efficiency.

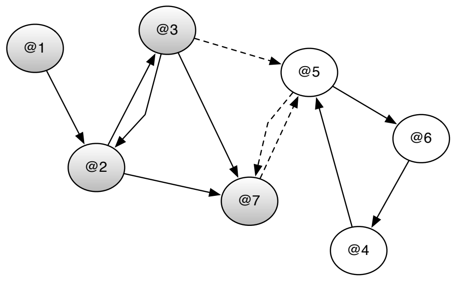

Figure 5. Topology using a breadth-first method.

We intend to explore other methods for defining $M(x)$ in the future, such as the METIS partitioning method [13].

3.2 Scheduling

We implemented several *scheduling schemes* to distribute computation across workers. Most of these methods are dependent on the nature of the worker in order to maximize efficiency. We present results for several programs in Section 5 using the schedulers described here to assess how the scheduling scheme improves or degrades the speedup.

We previously said that each node had a queue of new facts to process. However, this can be relaxed in order to account for the existence of workers. We have been experiencing with two main different types of queue organizations for parallel and distributed computation:

3.2.1 Local Queues

In the local queues organization, each node has a local queue of facts to process and there is one or multiple queues of *active nodes* to handle. An active node is a node that contains new facts to process and must be handled by some worker.

In Fig. 6 we present a schematic of local queues. For each local queue we have the new facts which are represented the `predicate`, the `tuple` and the `count`. The `count` field is an integer and is used to distinguish between derivations and deletions.

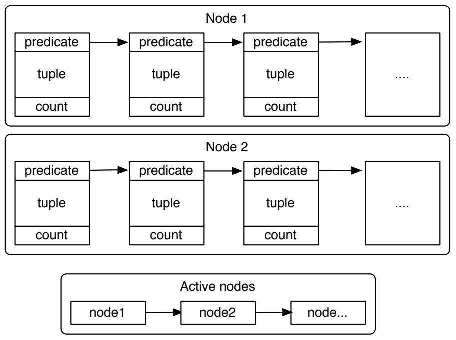

Figure 6. An example of local queues.

3.2.2 Global Queues

In the global queues organization, there is one or more queues with facts to process and the nodes have no queues themselves. Figure 7 present a global queue with the fields for each element of the queue. Note that now each queue element also has a `node` field, which corresponds to the node the fact is related.

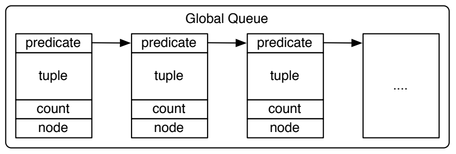

Figure 7. An example of a global queue.

3.3 Multithreaded Scheduling

Our virtual machine supports multithreaded execution using Pthreads. The workers in all the following four scheduling schemes are realized as threads.

3.3.1 Global Static Division

The *Global Static Division (GSD)* is a scheduling strategy where each thread has a global queue and a pre-defined set of nodes to handle. Every fact that is added to the global queue must be from a node the thread actually owns.

How is node ownership defined? Using the topology ordering defined previously, we divide statically the N nodes in the graph across T threads. Because execution always refers to mapped node addresses, we must know what thread is responsible for a given node address. For a node with address n, we compute $min(n/(N/T), N-1)$ to know the corresponding thread number.

Threads use their global queues to pop a new fact and node and then trigger new derivations. A new fact for a node owned by the same thread is simply added to the thread's queue. A new fact for a node not owned by the executing thread is, instead of inserted directly into the corresponding thread's queue, put into a buffer, so that when this buffer reaches a certain size, we can push the whole buffer directly into the other thread's queue. This helps improve data locality by making threads not touching other's threads cache lines.

Threads enter into the idle state when their global queues are empty. While idle, threads check for new work and for termination.

3.3.2 Local Static Division

3.3.3 Local Dynamic Division

3.3.4 Local Dynamic Division without Ownership

3.4 Distributed Scheduling

3.4.1 Global Static Division

3.4.2 Local Static Division

3.5 Mixed Mode Scheduling

3.5.1 Mixed Global Static Division

3.5.2 Mixed Local Static Division

3.5.3 Mixed Dynamic Division

3.5.4 Mixed Dynamic Division without Ownership

4. Virtual Machine Details

In this section we present several details about our virtual machine that directly deal with parallel and distributed execution.

4.1 Byte Code

4.2 Memory

4.3 Queues

4.4 Lists

4.5 Detecting Parallel Termination

4.6 Detecting Distributed Termination

5. Evaluation

In this section we present several programs written Meld that show possible uses of the language.

5.1 All-Pairs Shortest Path

5.2 PageRank

5.3 Belief Propagation

5.4 Neural Networks

6. Conclusions and Further Work

bla bla linear logic state bla bla splash bp

Acknowledgments

References

[1] K. Ali. Or-parallel Execution of Prolog on a Multi-Sequential Machine. *International Journal of Parallel Programming*, 15(3):189–214, 1986.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys'10*, pages 223–236, 2010.

[3] M. P. Ashley-Rollman, M. De Rosa, S. S. Srinivasa, P. Pillai, S. C. Goldstein, and J. D. Campbell. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS '07*, October 2007.

[4] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In *Proceedings of the International Conference on Logic Programming (ICLP '09)*, July 2009.

[5] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *The Journal of Logic Programming*, 4(3):259 – 262, 1987. ISSN 0743-1066. doi: DOI:10.1016/0743-1066(87)90004-5. URL `http://www.sciencedirect.com/science/article/pii/0743106687900045`.

[6] F. Bancilhon. On knowledge base management systems: integrating artificial intelligence and d atabase technologies. chapter Naive evaluation of recursively defined relations, pages 165–178. Springer-Verlag New York, Inc., New York, NY, USA, 1986. ISBN 0-387-96382-0. URL `http://portal.acm.org/citation.cfm?id=8789.8804`.

[7] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, March 1996. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/227234.227246. URL `http://doi.acm.org/10.1145/227234.227246`.

[8] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027.

[9] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 175–188, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. doi: http://doi.acm.org/10.1145/1322263.1322281. URL `http://doi.acm.org/10.1145/1322263.1322281`.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`.

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[12] S. C. Goldstein, J. D. Campbell, and T. C. Mowry. Programmable matter. *IEEE Computer*, 38(6):99–101, June 2005.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998. ISSN 1064-8275. doi: http://dx.doi.org/10.1137/S1064827595287997. URL `http://dx.doi.org/10.1137/S1064827595287997`.

[14] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, and J. M. Hellerstein. Declarative networking: Language, execution and optimization. 2008.

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

[16] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-Parallel Scheme (DDAS). In *Joint International Conference and Symposium on Logic Programming*, pages 717–731. The MIT Press, 1992.

[17] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 071678162X.

[18] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the ldl++ approach. In *Deductive and Object-Oriented Databases*, pages 204–221, 1993.