International Journal of Parallel Programming, Vol. 19, No. 6, 1990

Full Prolog and Scheduling Or-Parallelism in Muse

Khayri A. M. Ali[1] and Roland Karlsson[1]

Received December 1990; Revised May 1991

Muse is a simple and efficient approach to Or-parallel implementation of the full Prolog language. It is based on having *mu*ltiple *se*quential Prolog engines, each with its local address space, and some shared memory space. It is currently implemented on a number of bus-based and switch-based multiprocessors. The sequential SICStus Prolog system has been adapted to Or-parallel implementation with very low extra overhead in comparison with other approaches. The Muse performanhce results are very encouraging in absolute and relative terms.

The Muse execution model and its performance results on two different multiprocessor machines for a parallel version of Prolog, named Commit Prolog, have been presented in previous papers. This paper discusses supporting the full Prolog language and describes mechanisms being developed for scheduling Or-parallelism in Muse. It also presents performance results of the Muse implementation on Sequent Symmetry after supporting full Prolog. The results show that the extra overhead associated with supporting the full Prolog language is negligible.

KEY WORDS: Or-parallelism; full Prolog; multiprocessors; experimental results; scheduling.

1. INTRODUCTION

A variety of approaches toward exploitation of parallelism in Prolog programs are under current investigation. Many of these deal with efficient implementation of Prolog on multiprocessor machines by exploiting either Or-parallelism[1-8] or Independent And-parallelism[9-11] or a combination of both[12-15]. The Muse approach is one of those that exploit only Or-parallelism.[1] Execution of a Prolog program forms a search tree.

[1] Swedish Institute of Computer Science, SICS, Box 1263, S-164 28 Kista, Sweden.

Or-parallel execution of a Prolog program means exploring branches of a Prolog search tree in parallel. In the Muse approach (as in other Or-parallel Prolog approaches, e.g., Aurora[7] and PEPSys[2]), Or-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). [In this paper we try to be consistent with the Aurora terminology.[7]] This paper describes the basic mechanisms used for exploring branches of a Prolog search tree by the Muse workers. It also describes mechanisms for maintaining the sequential semantics of cut, findal and side-effect constructs.

The Muse approach is based on having several sequential Prolog engines, each with its local address space, and some shared memory space. It is currently implemented on a bus-based shared memory machine TP881V, from Tadpole Technology, with 4 (88100) processors, a bus-based machine with local/shared memory with 7 (68020) processors constructed at SICS, a bus-based shared memory S81, Sequent Symmetry, with 16 (i386) processors, and switch-based shared memory machines, BBN Butterfly I (GP1000) and II (TC2000), with 96 (68020) and 45 (88100) processors respectively. The sequential SICStus Prolog,[16] a fast, portable system, has been adapted to Or-parallel implementation. The extra overhead associated with this adaptation is very low in comparison with the other approaches. It is around 3% for TP881V, and 5% for the constructed prototype and Sequent Symmetry. The performance results of Muse on the BBN Butterfly machines will be reported by Shyam Mudambi at Brandeis University who has ported the Muse system into the BBN Butterfly machines. Mudambi preliminary results are very promising.[17] The Muse execution model and its performance results on the constructed prototype and Sequent Symmetry machines for a parallel version of Prolog, named Commit Prolog, have been presented in previous papers.[1,18] Commit Prolog is a Prolog language with cavalier commit[2] instead of cut, asynchronous (parallel) side-effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. Cut and sequential side-effect semantics could be obtained on Commit Prolog by annotating Prolog programs according to some rules.[1] In this paper, we discuss supporting the full Prolog language without such annotation. Some parts of this paper has been presented in Ref. 20.

The paper is organized as follows. Section 2 briefly describes the Muse execution model. This helps for understanding the principles for scheduling work in Muse. Section 3 discusses principles for scheduling work in Muse.

[2] Cavalier commit prunes branches both to the left and right of the committing branch, and is not guaranteed to prevent side-effects from occurring on the pruned branches.[19]

Section 4 discusses principles for scheduling work in related approaches. Section 5 presents and discusses the basic mechanisms for supporting scheduling work in Muse. Section 6 discusses implementation of cut, findall, and sequential side-effect constructs. Section 7 presents some performance results of the Muse system. Section 8 discusses our plans for the continued development of Muse. Section 9 concludes the paper.

2. MUSE EXECUTION MODEL—AN OVERVIEW

This section briefly describes the Muse execution model presented in Refs. 1 and 18. We assume herein that the reader is familiar with Warren's Abstract Machine (WAM).[21]

A node on a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared(private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each shared node is accessible only to workers within the subtree rooted by the node. Private nodes are only accessible to the worker that created them. Another distinction is that a node can be either *parallel* or *sequential*. Alternatives from a parallel node could be executed in parallel whereas alternatives from a sequential node can only be executed one at time, from left to right. A node is either *live* (having unexplored alternatives) or *dead* (no alternative to explore).

A major problem introduced by Or-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. The Muse execution model is based on having a number of sequential Prolog engines, each with its own local address space, and some global address space shared by all engines. Each sequential Prolog engine is a worker with its own stacks. [The assumed worker's stacks are: *a choicepoint stack, an environment stack, a term stack, and a trail.* The first two correspond to the WAM local stack and the second two correspond to the WAM heap and trail respectively.[21]] The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks.

This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the differing parts between the two workers states. This reduces copying overhead. Nodes are shared between workers to allow dynamic load balancing which reduces the frequency of copying.

To illustrate how workers share nodes in Muse, let us take a simple system having two workers P and Q. A worker Q runs out of work on its branch and a worker P has excess load (i.e., live nodes). Assume also that Q is positioned at a node, N, which is common to P and Q (see Fig. 1a). P allows Q to share its nodes by creating a data structure called *shared frame* in a shared-memory space for each private node, pointing to each shared frame from the corresponding choicepoint frame, and then copy to Q all choicepoint frames corresponding to nodes younger than N. [In the standard implementation of Prolog, a choicepoint frame is created in a choicepoint stack when a nondeterministic predicate is invoked.] Each shared frame basically contains information describing unexplored work at the node, workers which are at and below the node, and the node lock. At this moment, the choicepoint stacks of P and Q are identical and each node is associated with a shared frame which is referenced from the corresponding choicepoint frames.

In order to allow Q to execute alternatives from these shared nodes, Q has to get a copy of P's current state. Q gets only parts of the WAM stack that correspond to nodes younger than N along with P's modifications in the uncopied parts. These modifications are bindings made by P after creating the node N to variables created before N. These modifications are known from P's trail part created after N. Notice that after copying and sharing, P and Q have identical states and both share all P's nodes (see Fig. 1b). Now, they can work together exploring alternatives in the shared nodes by using the normal backtracking mechanism of Prolog. Reducing sharing and copying overheads in Muse model are described in Ref. 1.

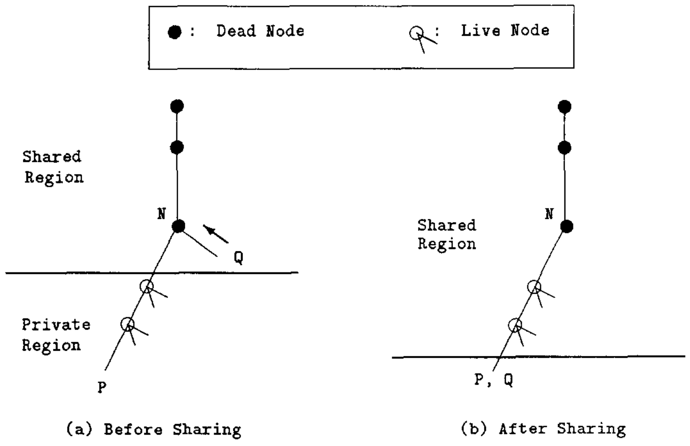

Fig. 1. Sharing Nodes.

3. PRINCIPLES FOR SCHEDULING WORK IN MUSE

Each worker can be in either engine mode or in scheduler mode. The worker enters the scheduler mode when it enters the shared part of the tree, or when it executes side-effects or *findall*. In the scheduler mode, the worker establishes the necessary coordination with other workers. The worker enters the engine mode when it leaves the scheduler mode. In the engine mode, the worker works exactly as a sequential Prolog engine on private nodes, but is also able to respond to interrupt signals from other workers.

The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead. The main sources of overhead in the Muse model are (1) copying a part of worker state, (2) making local nodes shareable, and (3) grabbing a piece of work from a shared node. Our scheduling work strategies, that attempt to minimize the overhead, are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the number of shared work between the workers and allow each worker to release work from the bottom-most node on its branch (dispatching on the bottom-most) by using backtracking with almost no extra overhead. Dispatching on the bottom-most allows also less speculative work than dispatching on the topmost—work is taken from the topmost live node on a branch (see Ref. 22). (Speculative work is defined as work which is within the scope of a cut and therefore may never be done by the sequential implementation.)
- When a worker runs out of work from its branch it will try to share work with the nearest worker which has maximum load. The load is measured by the number of local unexplored alternatives, and nearness is determined from positions of workers on the search tree. This strategy attempts to maximize the shared work and minimize sharing overhead.
- Workers which cannot find any work in the system will try to distribute themselves over the tree and stay at positions where sharing of new work is expected to be with low overhead.
- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position on the tree before interrupting the busy worker for requesting sharing. This allows a busy worker to concentrate on its task [A task is a continuous piece of work executed by a worker.] and to respond only to interrupts that have to be handled by it.

4. SCHEDULING WORK IN RELATED APPROACHES

In this section we discuss scheduling work strategies developed for the related Or-parallel Prolog approaches that are based on several sequential engines with constant access time of variables and nonconstant task-switching time.(23) Kabu Wake,(6) Aurora,(7) and ORBIT,(8) are examples of these approaches.

Four separate schedulers are being developed for Aurora: the Argonne scheduler,(19) the Manchester scheduler,(24) the Wavefront scheduler,(25) and the Bristol scheduler.(26) The basic strategy of the first three schedulers is that dispatching on topmost and sharing one parallel node on each branch at a time. That is, all the three schedulers attempt to maintain at most one, live, shareable node on their current branch. The advantage of this strategy is that the size of shared region is minimized and the size of tasks is kept as large as possible. One disadvantage is that finding a task always involves a general search in the tree, leading to relatively high task switching costs for fine (and medium) granularity programs.(27) Another disadvantage is more speculative work.

The Argonne scheduler uses local information that is maintained in each node to indicate whether there is work available below the node. Workers use this local information to migrate towards parts of the tree where work is available. The advantage of this scheduler is its simplicity of the design. It gives better performance for coarse granularity programs.

The Manchester scheduler tries to match workers with the nearest available task, where *nearness* is measured by the number of bindings to be updated between the worker's current position and the available work. By using this strategy it is hoped to keep the task switching overheads to a minimum. The Manchester scheduler tries also to distribute idle workers evenly over the tree. The performance results of this scheduler shows improvements over the Argonne scheduler for fine and medium granularity programs.

The Wavefront scheduler maintains a data structure known as the wavefront which links all topmost live nodes together. Workers traverse and extend the wavefront when they are looking for work. The available experimental results indicate a small performance improvements for fine granularity programs over the Manchester scheduler, and a little worse performance for coarse granularity programs in comparison with the first two schedulers.

The Bristol scheduler is based on the Muse (or BC-machine(28)) principles; dispatching on bottom-most and sharing several nodes on each branch at a time. Bristol scheduler shows some performance improvements over the other three schedulers for the fine and medium granularity programs. The overall performance results of the Bristol scheduler is similar to that of the Manchester scheduler.

In spite of the fact that the scheduling work principles of the Bristol scheduler is based on the Muse principles, there exist differences between the Muse scheduler and the Bristol scheduler. In the Bristol scheduler, a busy worker with maximum load will be matched to any idle worker and not to the nearest idle worker as that in Muse. Another difference is that idle workers in the Bristol scheduler stay at the bottom-most detected dead node of their branches, whereas in Muse idle workers distribute themselves over the tree (on the current branches) and stay at positions that are expected to be better ones. A third difference is the measure used for estimating a worker's load. In Bristol scheduler, a worker load is measured by the number of live parallel nodes on its branch. It is impractical, for efficiency reasons, to keep this load exact. The estimated load by the Bristol scheduler is an overestimate of the real load. This leads to busy workers that do not have excess load to be asked for sharing exactly as in Muse (see Section 5.3.2). The estimate of a worker's load used in Mure is the number of unexplored alternatives of private live parallel nodes. The Muse measure of load gives better estimate than the one used by the Bristol scheduler, because the number of alternatives of each live node is not always equal for most programs. We believe that better estimate of the load allows better decisions to be made. Finally, all mechanisms presented in this paper are completely different from those used by the Bristol scheduler.

In Kabu-Wake approach[6] and in ORBIT approach,[8] nodes cannot be shared among workers (i.e., processors), because these approaches are intended for nonshared-memory multiprocessors. Work at the topmost node is split into two parts and each of the two workers involved in copying takes a part of the work. This strategy does not allow dynamic load balancing between workers. Good results have been obtained only for coarse granularity programs.

Although Muse strategies seem more complex and more difficult to implement than those used in the above schedulers, we believe that they should lead to a more efficient way of matching work with workers.

5. SCHEDULING WORK ALGORITHM

The basic algorithm of the Muse scheduler for matching idle workers with available work is as follows. *(1) When a worker finishes a task, it attempts to get the nearest piece of available work on the current branch. (2) If none exists, it attempts to select a busy worker with excess work for sharing. (3) If none exists, it becomes idle and stays at a suitable position on the current branch.*

In the next three subsections, we are going to present and discuss supporting efficiently the above three parts of the basic algorithm. Data structure used for supporting this algorithm will be presented in the context of using them. In order to simplify the presentation, locking is not covered in detail.

5.1. Nearest Available Task

The nearest piece of available work is found in the bottom-most live node on the current branch. In order to support this operation efficiently, an efficient mechanism for checking whether the nodes in the current branch contain a live node and determining such a node is required. In the current Muse implementation, we have a simple representation of the tree (see Section 6.1) which allows a worker to access only nodes on its branch from its current position to the root of the tree. That is, a worker can only move on this part of the tree. Thus a worker should leave its position on the tree (i.e., backtrack) only when the new position is better than the old one, i.e., closer to a live node or to a busy worker with excess load.

Our mechanism assumes an extra field, *nearest-livenode*, on each shared node (i.e., in each shared frame associated with each shared node). This field contains either a reference to the nearest upper live node in the current branch or a *dummy value*. The latter means all upper nodes are dead.

When a worker finishes a task, it first checks the bottom-most shared node on the current branch. If the node is live, it just takes work from that node. If the node is dead, the worker checks the *nearest-livenode* field of the node. If *nearest-livenode* is not the *dummy value*, the worker checks a chain of shared nodes referred from *nearest-livenode* of the current node to determine the location of the nearest live node, keeping its position on the tree. (No locking is used in this operation.) If there is no such node, all nodes on the current branch get the *dummy value* in their *nearest-livenode* field. Then the worker backtracks to the nearest node with other workers and tries to select a busy worker with excess load as described in Section 5.2.

If work is found at node N, all shared nodes below N will get a reference to N, and the worker will attempt to position itself at N, as fast as possible, in order to take a piece of that work. The worker stops backtracking to N in one of the following situations:

1. The available work at node N is taken by the other workers. In this case, the worker repeats the procedure of determining the nearest live node described above by looking for nodes higher up than N on the current branch.

2. The worker is alone in a sequential node with available work. In this case, it takes a piece of work from the sequential node.
3. There is a pending cut that should be performed by that worker (see Section 6.3).

5.2. Matching Workers

The second part of the scheduling work algorithm is matching idle workers with busy workers having excess load. Matching each idle worker with the nearest busy worker with maximum load is expected to be a good heuristic. Such matching minimizes copying overhead by allowing less data to be copied and less frequency of copying. Supporting efficiently this heuristic is not an easy task, but we now present a mechanism that attempts to support a version of this heuristic.

The goal here is to match idle workers with busy workers having excess load in such a way that a worker with maximum load within a subtree will be assigned to the closest idle worker. The basic idea of the matching mechanism is as follows. When a worker *Q* becomes idle, it first determines a set of busy workers, within its subtree, which are not closer to any other idle worker. [When we say a worker subtree, we mean the subtree rooted by the current node of that worker.] Then, *Q* selects for sharing the worker *P* which has maximum load in this set.

If *Q* cannot find any worker with excess load in its subtree, it will determine a set of busy workers, outside its subtree, that are not closer to any idle worker. Then, *Q* backtracks to the nearest common node on the tree with workers in this set. After backtracking to the node, *Q* will have busy workers in its new subtree, and the same idea described above for selecting *P* within *Q*'s subtree could be used.

If *Q* cannot find any busy worker with excess load in the system, it will try to position itself at a suitable node in its branch as will be described in Section 5.3.

We divide the matching mechanism based in the idea described earlier into two parts. The first part is described in Section 5.2.1, and it concerns selecting *P* within *Q*'s subtree. The second part is described in Section 5.2.2, and it concerns selecting *P* outside *Q*'s subtree. The matching mechanisms described here use the following global information:

1. A counter, *load*, is associated with each worker containing the current load of the corresponding worker. The measure of load used in Muse is the number of private unexplored alternatives.

2. A register, *currentnode*, is associated with each worker containing a reference to its current position (shared node) on the tree.
3. A bitmap, *idlemap*, contains the current idle workers in the system.

As mentioned in Section 2, each shared node is associated with a shared frame which contains a bitmap, *workersbitmap*, representing workers within the subtree rooted by the node.

5.2.1. Matching Workers within the Current Subtree

The idle worker Q selects P within its subtree as follows. Q checks whether there are busy workers in its subtree. If there are, it will determine a set of those busy workers which are not closer to any other idle worker. Then it will select for sharing the worker P which has maximum load in this set.

The worker Q can determine which workers are busy (*busy-set*) and which are idle (*idle-set*), in its subtree, from *workersbitmap* of its current node and from the *idlemap*. (*busy-set* and *idle-set* are two local bitmaps.) Q can determine a subset of busy workers, in its subtree, which are not closer to any other idle worker, as follows. Q removes from its *busy-set* busy workers that are in the *idle-set* subtrees. Positions of *idle-set* workers are known from their *currentnode* registers. Finally, Q determines the one with maximum load in the remaining subset of those busy workers by investigating their *load* counters.

The mechanism allows Q to request sharing from P in situations shown in Figs. 2 (a) and (b), but not in (c). (Here, we always refer to the current idle worker by Q and the other idle workers by $Q1$, $Q2$, etc.) In Fig. 2a, P is not in $Q1$'s subtree. We allow Q to request sharing work from P, because $Q1$ could take long time to backtrack to a common node with P. $Q1$ could, for instance, reach a sequential node that should be

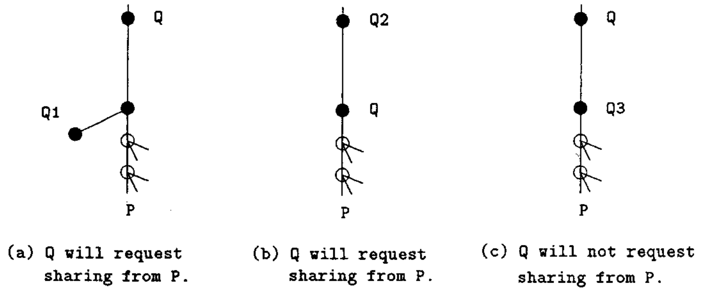
Q
Q1
P
(a) Q will request sharing from P.
Q2
Q
P
(b) Q will request sharing from P.
Q
Q3
P
(c) Q will not request sharing from P.

Fig. 2. Matching Workers within Q's Subtree.

processed. In Fig. 2b, there is also no other idle workers between P and Q. In Fig. 2c, there is an idle worker ($Q3$) between P and Q, i.e., $Q3$ is the nearest idle worker to P.

5.2.2. Matching Workers outside the Current Subtree

When Q cannot find any busy worker with excess load in its subtree, it will try to find one outisde its subtree as follows. It first determines a set of busy workers outside its subtree that are not closer to any other idle worker. Then, it determines the nearest node N that has any P with excess load of those busy workers by investigating nodes in its branch. After that, Q performs fast backtracking to the node N as long as none of the following situations occurs:

1. reaching a sequential node that should be processed, or
2. P's private work is exhausted, or
3. P is requested by another idle worker.

After backtracking to the node N, P will be in Q's subtree. Then Q uses the mechanism described in Section 5.2.1 to select a new P that currently has maximum load within its new subtree.

In order to allow other idle workers, in the same situation as Q, to select other busy workers simultaneously, we make P and Q invisible to those idle workers. The mechanism used for supporting invisible workers is as follows. There is an additional global bitmap, *invisibleworkersmap*, that contains workers similar to Q and P. When an idle worker Q wants to reserve a busy worker P while backtracking, Q sets the two bits corresponding to Q and P in the *invisibleworkersmap*. When Q either requests sharing from P or will do something else, Q resets these two bits. This *invisibleworkersmap* will be considered only by backtracking idle workers to determine the visible workers in the system.

Notice that P could be requested for sharing by an idle worker that has P in its subtree as described in Section 5.2.1. When Q stops backtracking for any of these reasons, it will make itself and P visible again.

The idle worker Q determines busy workers, that are outside its current subtree and are not closer to any other idle worker as follows. It first determines which workers are visible and busy (*visible-busy-set*), and which are visible and idle (*visible-idle-set*) outside its subtree. It determines those workers from *workersbitmap* of its current node, the *invisibleworkersmap*, and the *idlemap*. Then, Q removes from its *visible-busy-set* workers that are in *visible-idle-set* subtrees. Positions of those idle workers are known from their *currentnode* registers. Notice that if Q itself is in any idle worker subtree, Q should not backtrack because that idle worker is in a better position on the tree for requesting sharing. Therefore, before Q backtracks it has to check whether it is in a subtree of any idle worker of the *visible-idle-set*. One possible way to perform this check efficiently is that Q adds itself to the *visible-busy-set* before removing any worker, and Q checks if it is removed from this set when some workers are removed. If removing workers from the *visible-busy-set* terminates and Q is still in this set, Q could backtrack and not otherwise.

The mechanisms described in this section allow Q to stop backtracking in situations shown in Fig. 3 (a) and (b), but not in (c). In Fig. 3 (a), P is in $Q1$'s subtree. In Fig. 3b, Q is in $Q2$'s subtree, i.e. $Q2$ is closer to P. In Fig. 3c, P is not in $Q3$'s subtree, and Q finds P visible before $Q3$ does.

5.3. Idle Workers

The third part of the basic algorithm is the role of idle workers when there is neither any accessible live nodes nor busy workers with excess private load. In this case, idle workers will stay at nodes that allow newly generated work to be shared with low overhead. In Section 5.3.1, we discuss heuristics that attempt to achieve that.

The measure of excess load used in Muse is the number of private unexplored alternatives. According to this measure, there can be a situation in which there is work at some shared nodes which is not visible to idle workers and this work is only visible to busy workers that will not generate any new work. In this situation, idle workers will be idle forever, although there is excess work in the system. In Section 5.3.2, we will discuss a mechanism that detects this situation and allows the idle workers to share that work.

5.3.1. Distributing Idle Workers

The following heuristics attempt to distribute idle workers over the tree in such a way that when a busy worker generates work, an idle worker

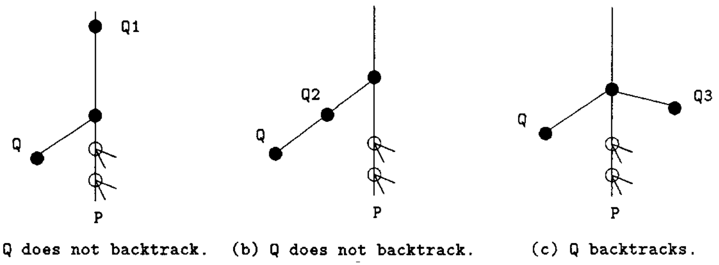

Fig. 3. Matching Workers outside Q's Subtree.

will share this work with low overhead. An idle worker, Q, leaves (backtracks from) its current node only if one of the following two situations occurs:

1. there are busy workers outside Q's subtree and Q is not in the subtree of any other idle worker, or
2. all workers in the current subtree are idle.

In the first situation, each idle worker will block all other idle workers in its subtree from backtracking. Only topmost idle workers in nonoverlapping subtrees will backtrack until one of them reaches a node common to all workers in the system. Moving some idle workers to nodes common to many busy workers allows sharing of work to be started earlier when a worker generates work. If the new position of an idle worker becomes closer to a worker that generates work, sharing overhead decreases. But if the old position was closer to the worker that generates work, sharing overhead increases. Blocking idle workers from backtracking when they are in any idle worker's subtree allows none of those idle workers to lose its position. Experimental results have indicated performance improvements for this heuristic in comparison to different other heuristics for distributing idle workers over the tree.

In the second situation, we allow idle workers to do fast backtracking to either the nearest sequential node or a node with a busy worker below. This is the right position for an idle worker to stay at for the following reasons. The last worker backtracking to a sequential node will take a branch from that node for processing and it might generate work. Also, for a node that has busy workers below, any of those workers might generate work.

5.3.2. Finding Invisible Work

The mechanism that finds invisible work for idle workers and allows sharing of that work is based on the following idea. When an idle worker, Q, cannot find work in the system and there are busy workers in its subtree, Q asks each of them to check if it has work in its branch. Q will try to ask workers, that have been busy for a long time and never asked before by any worker in the system. If any of them finds work, such work will be shared with Q.

To support this idea, there is a global bitmap, *mayaccessworkmap*, which contains workers that may access shared work. It is initially empty (all bits are reset). On every sharing the two bits corresponding to the two workers involved on sharing will be set. When a busy worker is asked for sharing and it turns out that the worker does not access shared work, the worker bit is reset.

There is also a local bitmap for every worker, *stablebusy*, which contains workers that were busy for a certain amount of time within the current subtree. When a worker becomes idle, it sets its own *stablebusy* bitmap to the busy workers within its subtree. While there are busy workers within its current subtree, it updates its *stablebusy* bitmap by resetting bits corresponding to workers that become idle. After *k* iterations of investigating the busy workers within its current subtree, it determines workers that were busy during the *k* iterations and never asked for sharing after their last sharing by investigating its *stablebusy* bitmap and the *mayaccessworkmap*. (The selected value of *k* for the Muse implementation on bus-based machines is 10.) Bits that are set in both bitmaps correspond to these workers. The idle worker could ask these workers for sharing, but it asks only those that are not closer to any other idle worker as described in Section 5.2. If any of these workers accesses shared work, that work will be shared by the idle worker. Otherwise, the busy worker will refuse the sharing request, reset its bit in the *mayaccessworkmap*, and mark shared nodes that are not marked as dead nodes by setting their *nearest-livenode* field to *dummy value* (see Section 5.1).

This mechanism allows a busy worker, that does not generate new work after sharing and does not access a live node, to be asked for sharing at most once. It also allows that worker to mark shared nodes as dead to avoid doing that later on. That is, this mechanism incurs almost no extra overhead for busy workers.

6. CUT AND SEQUENTIAL SIDE-EFFECTS

The current implementation of Muse supports full Prolog language with its standard semantics. It also supports asynchronous (parallel) side-effects and internal database predicates. In this implementation we have simple mechanisms for supporting cut and all standard Prolog side-effects predicates (e.g., read, write, assert, retract, etc.).

A simple way to guarantee the correct semantics of sequential side-effects is to allow execution of such side-effects only on the leftmost branch of the whole search tree. The current Muse implementation does not support suspension of branches. That is a worker that executes a sequential side-effect predicate on a branch, which is not leftmost of the whole tree, will wait until that branch is leftmost of the tree. Similarly, for supporting *findall* predicate a worker that generates a findall solution will wait until its branch is leftmost of a proper subtree.

The effect of *cut N* on a branch is to prune all branches to the right of the branch in a subtree rooted by the node *N*. In the current implementation of cut, when a worker executing cut is not leftmost in the relevant subtree, it will prune as much as it can and leave the pruning of the remaining branches to a worker to its left and then proceed with executing operations following the cut. When a worker detects that all left branches have failed, it will try to prune as much as it can until all branches within the scope of the cut are pruned. An idea of this cut algorithm has been proposed in Ref. 24.

In general, supporting cut, findall, and side effects requires efficient mechanisms for checking whether or not the current branch is leftmost in a tree. Also, efficient mechanisms for identifying workers working on branches to the right are required for supporting cut. In this section we discuss such mechanisms.

In the current Muse implementation, we have a very simple representation of a Prolog search tree. A worker can move only on its branch; i.e., there are no extra pointers in the nodes to the sibling nodes. The advantage of this simple representation is that adding and removing nodes are very efficient operations—neither locking overheads nor extra overheads for maintaining the topology of the tree are needed. The disadvantage is that a worker cannot traverse nodes on the other branches. This complicates the task of the Muse scheduler to perform a leftmost check, and to identify workers on branches to the right. To illustrate how the Muse scheduler carries out these operations, we first describe the representation of a Prolog search tree in the current Muse implementation.

6.1. Tree Representation

A Prolog search tree is represented in the current implementation of Muse as follows. Each shared node is associated with a shared frame containing a *workersbitmap* identifying workers accessing the node (and other information see Section 2). Each worker sharing a node has a choicepoint frame pointing to the shared frame associated with the node. Each alternative of a node is associated with its number. Each worker exploring an alternative of a node keeps the alternative number, *alt-number*, in its choicepoint frame associated with the node (or in a separate stack). Figure 4 shows the representation of a tree of two nodes *a* and *b*, and three workers *X*, *Y*, and *Z*. Each worker keeps two choicepoint frames in its choicepoint stack corresponding to the nodes *a* and *b*. The *alt-number* field in each choicepoint frame contains the corresponding alternative number of the node. For worker *X*, *alt-number* is 1 in each of its choicepoint frames.

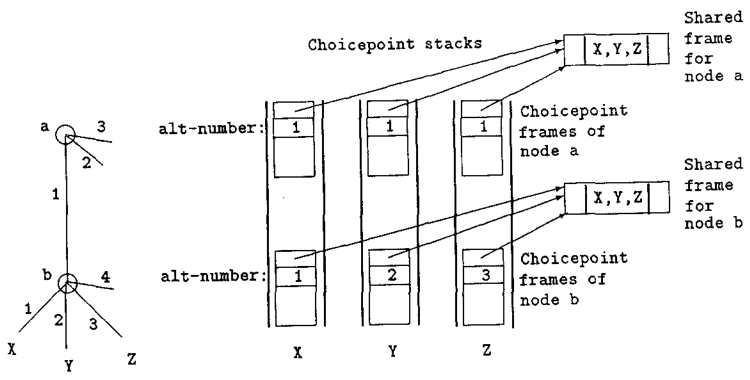

Fig. 4. Representation of a Tree.

6.2. Leftmost Check

Let us suppose that the worker Y in Fig. 4 wants to check if its branch is leftmost. It starts from the bottom-most shared node, b, determines the other workers accessing b (from the shared frame associated with b), and then investigates *alt-number* in choicepoint frames of these workers to determine if any of them has *alt-number* less than 2. It finds X with *alt-number* 1. So, Y is not leftmost. But if X wants to perform the same check, it will not find any other that has *alt-number* less than its *alt-number* in choicepoint frames associated with the nodes b and a. Thus, X is leftmost.

A leftmost algorithm based on this idea is as follows. When a worker P wants to check wether it is leftmost on a tree rooted by a shared node N, it starts from the bottom-most shared node on its branch, reads *workersbitmap* associated with the node to determine other workers sharing the node, investigates *alt-number* corresponding to that node for the other workers in the *workersbitmap*. The worker P finds *alt-number* field of each other worker by calculating the remote address from the offset of its *alt-number* field in its choicepoint stack and the base address of each choicepoint stack of the other workers. If P finds any of these *alt-number* less than its *alt-number*, P is not leftmost. But if P does not find any *alt-number* less than its *alt-number* and the current node is not N, P will repeat the same operation at the next upper node and so on until reaching N.

According to this algorithm, the worker X in Fig. 4 investigates four remote choicepoint frames (two at the node b, and then two at the node a). For a branch with n shared parallel nodes and a system of W workers, the number of remote choicepoint frames to be investigated will be $O(n * W)$. Also, all the n nodes will be investigated. This means that using this algorithm without substantial improvements is impractical for efficiency reasons.

We present improvements to this algorithm that make it practical and reasonably efficient. The first improvement reduces the number of nodes in a branch to be investigated. The second improvement reduces the number of remote choicepoint frames to be investigated from $O(n * W)$ to only $W-1$. The third improvement reduces the number of remote choicepoint frames to be investigated to zero.

6.2.1. Reducing Investigated Nodes

The first improvement reduces the number of nodes to be investigated by a worker checking if its branch is leftmost. It is based on associating with each shared node a pointer, *nearest-leftnode*, pointing to the nearest upper node with a branch to its left. That node will be investigated next when a worker is leftmost at the current node. A worker updates *nearest-leftnode* field of nodes on its branch while performing its leftmost check and sharing its private nodes. When a worker performs a leftmost check and detects a node N in its branch with a branch to its left, it updates all shared nodes below N to point to N. When a worker completes a leftmost check, all nodes in its branch will point to the root of the tree. A next leftmost check in the same branch reaches the root from any node on the branch indicating determination of a leftmost check on the whole tree.

Each Muse worker shares several private nodes at a time. The nearest upper node that could have a branch to its left is the bottom-most shared node. So, when a worker makes its private nodes shareable, it sets *nearest-leftnode* field in each private node to the bottom-most shared node. An optimization is to allow a leftmost worker at a bottom-most shared node to set *nearest-leftnode* field of its private nodes to the nearest upper node with a branch to its left. This improvement is also useful to all the Aurora schedulers, specially the Bristol scheduler.

6.2.2. Reducing Remote Access

The idea of the second improvement is that when a worker in an alternative L at node N performs a leftmost check, it will investigate only workers in the other alternatives at N. It determines these workers from the difference between *workersbitmaps* at N and its child node in the alternative L. (Each choicepoint frame associated with a shared node contains a *child* field that refers to choicepoint frame associated with its child node.) If N is the bottom-most shared node in a branch, all other workers in N will be investigated. For instance in Fig. 4, the worker X will investigate choicepoint frames of *Y* and *Z* at the node *b*, but none at the node *a*; there are no new workers at *a*.

By this improvement only one choicepoint frame of each of the other workers will be investigated in a complete leftmost check on the whole tree.

6.2.3. Avoiding Remote Access

The third improvement reduces the number of remote choicepoint frames to be investigated to zero on performing a leftmost check. It is based on associating with each shared node a bitmap, *active-alternativesmap*, for indicating active alternatives at the node. When an alternative is taken from a shared node the corresponding bit is set, and when an alternative fails (i.e., when all workers on the alternative are backtracked) the corresponding bit is reset. (The size of the *active-alternativesmap* is equal to the maximum number of alternatives in a parallel node; i.e., number of clauses in a parallel predicate.) A worker on an alternative *L* of a shared node is leftmost at that node only when all bits less than *L* are reset.

This improvement reduces the number of remote choicepoint frames to be investigated on each leftmost check to zero. Efficient implementation of this improvement requires limiting the maximum number of clauses in each parallel predicate to the size of one machine word. (For instance, 32 clauses can be represented by a 32-bits word). This can be done by program transformation. This improvement has not been implemented yet.

6.3. Cut

The effect of *cut N* on a branch is to prune all branches to the right of the branch in a subtree rooted by the node *N*. It is implemented in Muse as follows. *N* can be either a private node or a shared node. In the former case, cut is processed by a worker which executes it exactly as in the sequential Prolog implementation. In the latter case, we have two different situations: (1) cutting branch is leftmost, or (2) cutting branch is not leftmost. In situation (1), the worker executing cut signals all other workers in the node *N* to backtrack to parent of *N*, it removes itself from *N* and all shared nodes below *N* in its branch, deallocates its choicepoint frames associated with those nodes, and then proceeds with processing the operations following the cut. (Deallocation of a shared frame is done by the last worker backtracking from the corresponding node.)

In situation (2), the worker executing cut finds a shared node *M* below *N* that has a branch to its left. It removes unexplored alternatives at *M*, saves at *M* information describing alternative number in the current branch at M, say L, and scope of cut (i.e., N) only if there exist upper branches to be pruned, and signals all workers on branches to its right to backtrack to parent of M. A simple way of identifying these workers is to investigate choicepoint frames of all the other workers at M and signal each worker that has its *alt-number* equal to or greater than L. Another efficient way for identifying these workers (based on the idea presented in Section 6.2.2) is to first find all the other workers on the child node of M in the alternative L, and then investigate only choicepoint frames of the new workers at M. The new workers at M are obtained from the difference between *workersbitmaps* of M and its child node in the alternative L.

In order to avoid extra overhead by every worker backtracking to M for performing a leftmost check, we save also at M the name of one of the workers that are on branches to the left of L. Only that worker will perform a leftmost check for the alternative L at M. If while performinig a leftmost check there is another worker Q, that has its *alt-number* less than L and there exist upper branches to its right to be pruned, Q will be saved on the node M and the backtracking worker tries to find work on the tree. Otherwise, the backtracking worker will restart pruning upper branches by examining every node in its branch starting from M's parent until reaching the node N. The worker will perform the following at each node: (1) it determines the alternative number, $L1$, of the node in the current branch from its choicepoint frame associated with that node, (2) finds the difference between *workersbitmaps* associated with the node and its child node in the current branch, (3) signals workers on alternatives greater than $L1$, (4) removes the remaining unexplored branches, and (5) checks if there is any worker on alternatives less than $L1$. If there exists such worker, the name of that worker and $L1$ will be saved at the node. But if there is no such a worker and N is not reached, the same procedure will be performed on the next upper node in the current branch.

7. PERFORMANCE RESULTS

In this section we discuss the extra overhead for maintaining information that support cut, findall, and sequential side-effect constructs. We also show the performance of the Muse scheduler for different class of benchmarks. The timing results of Muse will be compared with the corresponding results for Aurora[7] with the Manchester scheduler.[24] This gives some ideas about how the Mure scheduler performs in comparison with another good scheduler for a similar system. Both Aurora and Muse are based on the same sequential Prolog, SICStus version 0.6. Neither Muse nor Aurora with the Manchester scheduler handles speculative work properly. [A new Aurora scheduler which handles speculative work properly is under development at the University of Bristol.] The main difference between Muse and Aurora on implementation of cut, findall, and sequential side-effect constructs is that Aurora supports suspension of branches whereas Muse does not. For instance, an Aurora worker executing cut in a nonleftmost branch of the cut node will suspend the branch and try to find work outside the cut subtree. In Muse, the pruning operation suspends while the worker proceeds with the next operation following the cut as described in Section 6.3. Other differences between Aurora and Muse are that: Aurora uses a more general representation of a Prolog search tree than the one used in Muse, and Aurora is based on another model for Or-parallel execution of Prolog.(29) Finally, both run on the same Sequent Symmetry S81, with 16 processors and 32 Mbytes of memory, which is available to us at SICS. The total scheduling overhead in Muse will also be discussed.

7.1. Benchmarks

The group of benchmarks used in this paper can be divided into two sets: the first set (*8-queens1*, *8-queens2*, *tina*, *salt-mustard*, *parse2*, *parse4*, *parse5*, *db4*, *db5*, *house*, *parse1*, *parse3*, *farmer*) has relatively well understood granularity, and has been used by several researchers in previous studies.(1,2,24,27) *8-queens1* and *8-queens2* are two different *N* queens programs from ECRC. *tina* is a holiday planning program from ECRC. *salt-mustard* is the "salt and mustard" puzzle from Argonne. *parse1 — parse5* are queries to the natural language parsing parts of Chat-80 by F. C. N. Pereira and D. H. D. Warren. *db4* and *db5* are the data base searching parts of the fourth and fifth Chat-80 queries. *house* is the "who owns the zebra" puzzle from ECRC. *farmer* is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC. This set contains benchmarks with coarse grain parallelism (*8-queens1*, *8-queens2*, *tina*, *salt-mustard*), with medium grain parallelism (*parse2*, *parse4*, *parse5*, *db4*, *db5*, *house*), and with fine grain parallelism (*parse1*, *parse3*, *farmer*). It is divided into tree groups known in the following sections by *High*, *Med*ium, and *Low* respectively. This set of benchmarks does not contain major cuts. All the benchmarks of the first set look for all solutions of the problem.

The second set of benchmarks (*mm1*, *mm2*, *mm3*, *mm4*, *num1*, *num2*, *num3*, *num4*) contains major cuts and has been used for studying different cut schemes.(22) *mm* is a *m*aster*m*ind program with four different secret codes. *num*bers program generates the two largest numbers consisting of given digits and fulfilling specified requirements. The *num* was run four different queries. This set is divided into two groups known in the following sections by *mm* and *num*. All the benchmarks of the second set look for the first solution of the problem.

7.2. Results of Benchmarks with no Major Cuts

To evaluate the extra overhead for maintaining information that support cut, findall, and sequential side-effect constructs, we compare runtimes of the first set of the benchmarks on a version of Muse supporting full Prolog with another version that supports a parallel version of Prolog, named Commit Prolog.(1) Commit Prolog is a Prolog language with cavalier commit(3) instead of cut, asynchronous (parallel) side-effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. The standard Prolog semantics of cut and sequential side-effects was obtained on Commit Prolog by following a few rules that restrict the degree of Or-parallelism.(1)

Table I presents the runtimes (in seconds) from the execution of the

Table I. Runtimes (in Seconds) on Muse Version of Commit Prolog for the First Set of the Benchmarks

	Workers					
Benchmarks	1	4	8	12	15	SICStus
8-queens1	6.83(0.92)	1.72(3.65)	0.87(7.22)	0.59(10.6)	0.48(13.1)	6.28
8-queens2	17.38(0.95)	4.36(3.77)	2.21(7.43)	1.49(11.0)	1.20(13.7)	16.43
tina	14.44(0.95)	3.67(3.74)	1.90(7.22)	1.31(10.5)	1.08(12.7)	13.71
salt-mustard	2.10(0.96)	0.54(3.74)	0.28(7.21)	0.19(10.6)	0.16(12.6)	2.02
• High Σ	40.75(0.94)	10.29(3.74)	5.26(7.31)	3.59(10.7)	2.93(13.1)	38.44
parse2*20	5.99(0.95)	1.78(3.20)	1.28(4.45)	1.10(5.17)	1.11(5.13)	5.69
parse4*5	5.53(0.95)	1.50(3.51)	0.93(5.66)	0.76(6.92)	0.73(7.21)	5.26
parse5	3.92(0.95)	1.02(3.66)	0.57(6.54)	0.46(8.11)	0.45(8.29)	3.73
db4*10	2.39(0.95)	0.65(3.49)	0.39(5.82)	0.30(7.57)	0.28(8.11)	2.27
db5*10	2.91(0.95)	0.80(3.45)	0.47(5.87)	0.36(7.67)	0.33(8.36)	2.76
house*20	4.41(0.96)	1.33(3.18)	0.83(5.10)	0.66(6.41)	0.62(6.82)	4.23
★ Med Σ	25.15(0.95)	7.14(3.35)	4.53(5.28)	3.69(6.49)	3.58(6.69)	23.94
parse1*20	1.59(0.94)	0.60(2.50)	0.56(2.68)	0.59(2.54)	0.63(2.38)	1.50
parse3*20	1.36(0.96)	0.56(2.32)	0.50(2.60)	0.52(2.50)	0.54(2.41)	1.30
farmer*100	3.19(0.96)	1.38(2.22)	1.39(2.21)	1.38(2.22)	1.40(2.19)	3.07
* Low Σ	6.14(0.96)	2.54(2.31)	2.46(2.39)	2.49(2.36)	2.57(2.28)	5.87
Σ	72.04(0.95)	19.98(3.42)	12.25(5.57)	9.79(6.97)	9.11(7.49)	68.25

first set of the benchmarks on the Muse version of Commit Prolog. The runtimes given are the shortest obtained from eight runs. Times are shown for 1, 4, 8, 12, 15 workers with speedups given in parentheses. These speedups are relative to running times of SICStus0.6 on one Sequent processor shown in the last column. For benchmarks with small runtimes the timings shown refer to repeated runs, the repetition factor being shown in the first column. $\sum$ in the last row corresponds to the goal: (*8-queens1, 8-queens2, tina, salt-mustard, parse2*20, parse4*5, parse5, db4*10, db5*10, house*20, parse1*20, parse3*20, farmer*100*). That is, the timings shown in the last row correspond to running the whole first set of the benchmarks as one benchmark. In the following tables, the last row for each group of a set of benchmarks represents the whole group as one benchmark.

For all programs in Table I, except *parse1 – parse3* and *farmer*, increasing the number of workers results in shorter runtimes. For *parse1 – parse3* and *farmer*, increasing the number of workers beyond a certain limit results in slightly longer runtimes. This degradation is due to the extra runtime scheduling overhead for programs with fine granularity. The scheduling overhead will be discussed in Section 7.4.

Table II presents the runtimes (in seconds) from the execution of the first set of the benchmarks on Aurora and the Muse version of full Prolog. The runtimes given are for each group of the first set of the benchmarks and for the whole first set of the benchmarks. If we compare the runtimes for Muse in Tables I and II, we find that Muse of full Prolog is slower than Muse of Commit Prolog by around 0% for the 1 worker case, 2% for the

Table II. Runtimes (in Seconds) on Aurora and Muse Version of Full Prolog, and the Ratio Between Them for the First Set of the Benchmarks

	Workers					
Benchmarks	1	4	8	12	15	SICStus
○ High Aurora	51.97(0.74)	13.28(2.89)	6.76(5.69)	4.58(8.39)	3.74(10.3)	38.44
◇ Med Aurora	30.20(0.79)	9.09(2.63)	6.25(3.83)	5.37(4.46)	5.09(4.70)	23.94
o Low Aurora	7.31(0.80)	3.76(1.56)	3.70(1.59)	3.94(1.49)	4.14(1.42)	5.87
Σ	89.48(0.76)	26.16(2.61)	16.75(4.07)	13.94(4.90)	13.00(5.25)	68.25
● High Muse	41.00(0.94)	10.61(3.62)	5.32(7.23)	3.64(10.6)	2.98(12.9)	38.44
★ Med Muse	25.24(0.95)	7.23(3.31)	4.64(5.16)	3.85(6.22)	3.70(6.47)	23.94
* Low Muse	6.16(0.95)	2.59(2.27)	2.52(2.33)	2.58(2.28)	2.63(2.23)	5.87
Σ	72.40(0.94)	20.46(3.34)	12.50(5.46)	10.09(6.76)	9.31(7.33)	68.25
Aurora/Muse	1.24	1.28	1.34	1.38	1.40	—

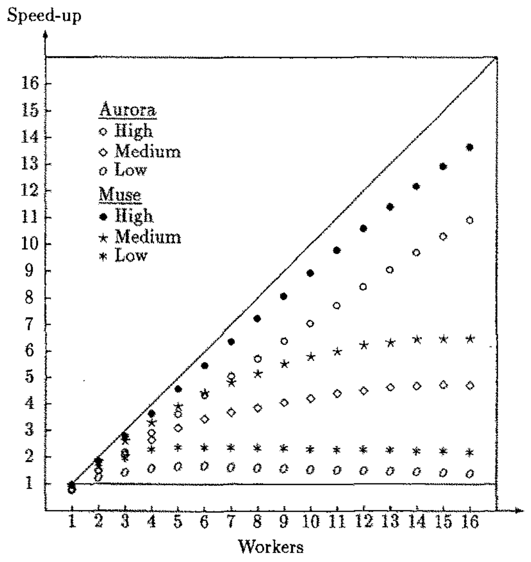

Fig. 5. Speedups of Muse and Aurora for the First Set of the Benchmarks.

4 workers, 2% for 8 workers, 3% for 12 workers, and 2% for 15 workers. This means that the extra overhead for maintaining information that support cut, findall, and sequential side-effect constructs in Muse is very low. Table II shows in the last row the ratio of the running times on Aurora to the running times on Muse for the first set of the benchmarks. Aurora timings are longer than Muse timings by 24% to 40% between 1 to 15 workers.

Figure 5 shows the speedup curves of Muse and Aurora for the three groups of the first set the benchmarks: High, Medium, and Low. Notice that all spedups in this paper are relative to SICStus0.6. The results shown in Table II and Fig. 5 illustrate how the Muse scheduler performs well on each group of the first set of the benchmarks.

7.3. Results of Benchmarks with Major Cuts

Here we show timing results for programs with major cuts. Table III presents the runtimes (in seconds) from the execution of the second set of the benchmarks on the Muse version of full Prolog. The runtimes given are the mean values obtained from eight runs. For programs with (major) cuts, mean values are more reliable than best values because scheduling of speculative work changes from one run to another causing larger variations of timing results.

Table III. Runtimes (in Seconds) on Muse for the Second Set of the Benchmarks

	Workers					
Benchmarks	1	4	8	12	15	SICStus
mm1	4.10(0.96)	2.05(1.93)	1.16(3.41)	0.99(3.99)	0.93(4.25)	3.95
mm2	3.23(0.98)	1.09(2.90)	0.81(3.90)	0.59(5.36)	0.51(6.20)	3.16
mm3	9.26(0.97)	3.17(2.83)	2.12(4.24)	1.55(5.79)	1.39(6.46)	8.98
mm4	15.80(0.96)	5.06(3.00)	2.75(5.52)	1.79(8.47)	1.56(9.72)	15.17
● mm Σ	32.39(0.97)	11.37(2.75)	6.84(4.57)	4.93(6.34)	4.39(7.12)	31.26
num1	1.63(0.99)	0.94(1.72)	0.52(3.12)	0.31(5.23)	0.28(5.79)	1.62
num2	2.67(0.99)	0.91(2.91)	0.47(5.64)	0.34(7.79)	0.27(9.81)	2.65
num3	2.86(0.99)	0.83(3.41)	0.44(6.43)	0.31(9.13)	0.27(10.5)	2.83
num4	3.69(0.99)	0.95(3.84)	0.50(7.30)	0.33(11.1)	0.28(13.0)	3.65
★ num Σ	10.85(0.99)	3.63(2.96)	1.93(5.57)	1.29(8.33)	1.10(9.77)	10.75
Σ	43.24(0.97)	15.00(2.80)	8.77(4.79)	6.22(6.75)	5.49(7.65)	42.01

Table IV presents the runtimes (in seconds) from the execution of the second set of the benchmarks on Aurora. The runtimes given are for each group of the second set of the benchmarks and for the whole second set of the benchmarks. It also shows in the last row the ratio of the running times on Aurora to the running times on Muse for the second set of the benchmarks as one benchmark. Aurora timings are longer than Muse timings by 19% to 101% between 1 to 15 workers. There are two possible explanations for this difference of performance results between Muse and Aurora for this set of benchmarks. The first one is that dispatching on the bottom-most used in Muse allows less speculative work than dispatching

Table IV. Runtimes (in Seconds) on Aurora and the Ratio Between Aurora and Muse Timing for the Second Set of the Benchmarks

	Workers					
Benchmarks	1	4	8	12	15	SICStus
○ mm Aurora	39.39(0.79)	22.73(1.38)	13.28(2.35)	10.75(2.91)	9.38(3.33)	31.26
◇ num Aurora	12.09(0.89)	4.54(2.37)	2.56(4.20)	1.95(5.51)	1.65(6.52)	10.75
Σ	51.48(0.82)	27.27(1.54)	15.84(2.65)	12.70(3.31)	11.03(3.81)	42.01
Aurora/Muse	1.19	1.82	1.81	2.04	2.01	—

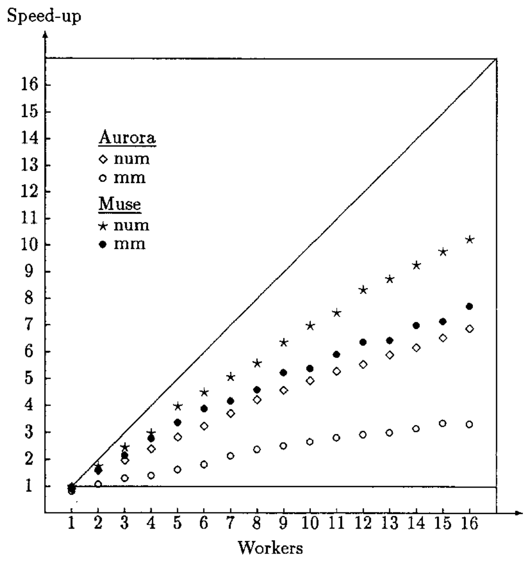

Fig. 6. Speedups of Muse and Aurora for the Second Set of the Benchmarks.

on the topmost used in Aurora (by the Manchester scheduler). The second reason is that for those programs suspension of branches as used in Autora is not the best. In general, better handling of speculative work for cut programs is the best.

Figure 6 shows the speedup curves of Muse and Aurora for the two groups of the second set of the benchmarks: mm and num. These curves correspond to speedups obtained from Tables III and IV.

7.4. Scheduling Overheads

In this section, we present and discuss briefly the time spent in the scheduling activity in the Muse version of full Prolog. A Muse worker time is distributed over the following three basic activities:

1. *Prolog*: time spent in executing Prolog, checking arrival of interrupt signals, and maintaining value of the local load.
2. *Idle*: time spents in looking for a worker with excess local work and distributing idle workers on the tree when there is no available work in the system.

3. *Scheduling*: time spent in scheduling activity. It includes time spent in sharing nodes, grabbing work from shared nodes, selecting busy workers for sharing nodes, copying data, synchronization, moving up within the shared region, spin lock, signalling, etc.

Table V shows the total time spent in each activity, with the percent of that time relative to the total time, for the first set of the benchmarks. Times shown in Table V have been obtained from an instrumented system of Muse on Sequent Symmetry. Those times include the time spent in the measurements. The times obtained from an instrumented system are longer than those obtained from an uninstrumented system by around 10%. We believe that the percentage of time spent in each activity obtained from the instrumented system reflects what is happening in the uninstrumented system.

As mentioned in Section 7.1, this set of benchmarks contains four benchmarks with coarse grain parallelism, six with medium grain parallelism, and three with fine grain parallelism. This set also represents benchmarks with lack of parallelism. Lack of parallelism explains the reasons of decreasing the percentage of the *Prolog* time and increasing the percentage of the *Idle* time when increasing the number of workers in Table V. The summation of these two percentages is almost constant (only 4.1% difference) from 8 workers to 15 workers. The *Scheduling* overhead increases from the 4 workers case to the 8 workers case by 5.7%, from the 8 workers to the 12 workers by 2.3, and from the 12 workers to the 15 workers by 1.8%.

A possible explanation for the increase of overhead when increasing the number of workers is shown in Table VI, which shows the effect of increasing the number of workers on the number of tasks and task size (expressed as a number of Prolog calls per task) for the first set of the benchmarks. In Table VI granularity of parallelism is decreased from the 4

Table V. Total Times (in Milliseconds) Spent in Basic Activities of a Muse Worker for the First Set of the Benchmarks

	Muse Workers			
Activity	4	8	12	15
Prolog	80885(90.0)	83158(85.7)	84829(63.8)	86424(56.2)
Idle	2893(3.2)	13010(11.8)	28488(21.4)	41765(27.2)
Scheduling	6069(6.8)	13684(12.5)	19731(14.8)	25576(16.6)
Total	89847(100.0)	109853(100.0)	133048(100.0)	153765(100.0)

Table VI. Average Number of Tasks and Task Sizes for the First Set of the Benchmarks

	Muse Workers			
	4	8	12	15
Total Number of Tasks	13807	28145	34530	39537
Prolog Calls per Task	64	31	26	22

Table VII. Scheduling Overhead per Task for the First Set of the Benchmarks

	Muse Workers			
	4	8	12	15
Scheduling Overhead per Task in Prolog calls	4.80	5.10	6.05	6.51

workers case to the 8 workers case by a factor around 2 whereas from the 8 workers to the 12 workers by a factor 1.2, and from the 12 workers to the 15 workers by a factor 1.2.

Tables V and VI illustrate that the scheduling overhead increases when reducing the granularity of parallelism.

The number of Prolog calls that are equivalent to scheduling overhead per task are shown in Table VII. Figures in Table VII are calculated from Tables V and VI. Table VII illustrates that the scheduling overhead is equivalent to around 5–7 Prolog calls per task, where the time of a Prolog procedure call is between 83 to 100 microseconds. Similar figures for scheduling overhead in terms of Prolog calls per task are reported for Aurora with the Manchester scheduler.(27) The time of a Prolog procedure call for Aurora is longer than the corresponding time for Muse by a factor around 1.20–1.25, the relative speed of the Muse engine to the Aurora engine. That is, the time of scheduling overhead of Muse is less than the time of scheduling overhead of Aurora with the Manchester scheduler by a factor around 1.20–1.25.

Both Muse and Aurora with the Manchester scheduler attempt to minimize continuous increase of runtime scheduling overhead as the number of workers is increased. They achieve that by supporting mechanisms that avoid a continuous decrease in task sizes as the number of workers grows. The idea used by the Muse system is that when a busy worker reaches a situation at which it has only one private parallel node, it will make its private load visible to the other workers only when that node is still alive after a certain number, *n*, of Prolog procedure calls. The value of n is a constant value selected in the order of the number of Prolog procedure calls equivalent to the scheduling overhead per task. It is 5 on Sequent Symmetry.

The idea used by the Manchester scheduler to avoid a continuous decrease in task sizes with increasing workers is that each busy worker checks for arrival of signals from other workers on every N Prolog procedure calls. The value of N is also 5 on Sequent Symmetry.

The Muse solution costs extra runtime overhead only when a worker has only one private parallel node. It also limits the parallelism at this situation only, which in turn avoid a continuous decrease in task size. The Manchester scheduler solution costs extra runtime overhead on every Prolog procedure call for updating a counter. The private work will be released on every N Prolog procedure calls.

8. FUTURE WORK

Our future work on the current Muse system is to support *free findall* construct as it is defined in Ref. 19, and to use more advanced implementation schemes for cut, commit, findall, and sequential side-effects. Note, in the current Muse implementation, commit is translated into cut.

In the current implementation of cut, we do not maintain information describing alternatives with cut. Cut schemes based on maintaining such information reduces speculative work. An advanced cut scheme based on one of the scheme presented in Ref. 22 will be implemented.

In the current implementation of *assert* when a worker is going to assert a rule, it waits until its branch is leftmost on the whole tree. An idea is to allow the worker to perform most of the work needed for assert and delays insertion of the rule until the branch is leftmost on the whole tree, and the worker proceeds with operations following assert. Information describing the uncompleted assert will be saved in the nearest upper node, M, in the branch with a branch to its left. The last worker, which backtracts to M from left branches that have caused delay of completing assert, will take care of that uncompleted assert, if it is not already removed by a cut. That worker either moves the information describing the uncompleted assert to another upper node with a branch to its left, or completes it if current branch is leftmost on the whole tree.

Similarly, adding a generated solution of findall requires the current branch to be leftmost on a proper subtree. This idea could be used also here to perform most of the work needed for generating a solution and delay insertion of the solution until its branch is leftmost on the subtree, and the worker proceeds with the next operation.

Regarding calling dynamic predicates and the other side effects, like *retract* and *input/out*, a mechanism for suspending branches is needed. An idea of suspension is to save in the shared-memory space the difference between the current state of a worker, which is going to suspend its branch, and the state corresponding to the nearest upper node, *M*, with a branch to its left. Information describing the suspended branch and the location of saved part of state will be stored in *M*. As described earlier, the last worker, which backtracks to *M* from left branches that have caused suspension of a branch, will take care of the suspended branch, if it is not already removed by a cut. That worker either moves information describing the suspended branch(es) to another upper node, *N*, with a branch to its left along with the difference between the computation state at *M* and *N*, or restarts the suspended branch if it is leftmost on the whole tree. When a worker is going to restart a suspended branch, it first gets the state corresponding to that branch from the shared-memory space by using information stored in the node *M*.

Now how to determine the difference of states between two nodes. It is known in the Muse model by incremental copying (see Ref. 1 for details).

9. CONCLUSIONS

The principles and implementation of scheduling work and supporting full Prolog in Muse have been presented. Many of the presented algorithms are also applicable to other OR-parallel Prolog approaches. The performance results are very encouraging and the extra overhead for maintaining information that support cut, findall, and sequential side-effect constructs in Muse on Sequent Symmetry is very low (around 0% for one worker, and 2%–3% for 15 workers). The total scheduling overhead per task on Sequent Symmetry for the set of benchmarks used in Ref. 27, is equivalent to around 5–7 Prolog calls per task. For programs with cuts, dispatching on the bottom-most gave much better performance results than dispatching on the topmost. The suggestions for improvements and new constructs mentioned in the paper will be implemented. The implementation of efficient and simple form of suspension of branches and speculative work will be examined. Using bitmaps in the presented algorithms limit the use of these algorithms for systems with too many workers. Mechanisms for large systems (like larger configuration of the Butterfly machine) will also be examined.

ACKNOWLEDGMENTS

We would like to thank Mats Carlsson and Andrzej Ciepielewski for helpful comments on an earlier draft of this paper.

REFERENCES

1. Khayri A. M. Ali and Roland Karlsson, The Muse Approach to OR-Parallel Prolog, *Int'l. J. of Parallel Programming*, **19**(2):129–162 (April 1990).
2. Uri Baron, Jacques Chassin de Kergommeaux, Max Hailperin, Michael Ratcliffe, Philippe Ropert, Jean-Claude Syre, and Harald Westphal, The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results, *Proc. of the Int'l. Conf. on Fifth Generation Computer Systems, ICOT*, pp. 841–850 (November 1988).
3. Ralph Butler, Ewing Lusk, Robert Olson, and Ross Overbeek, ANLWAM—A Parallel Implementation of the Warren Abstract Machine, Internal Report, Argonne National Laboratory (1986).
4. William Clocksin, Principles of the DelPhi Parallel Inference Machine, *Computer Journal* **30**(5):386–392 (1987).
5. Laxmikant V. Kalé, The Reduce-OR Process Model for Parallel Evaluation of Logic Programs, *Proc. of the Fourth Int'l. Conf. on Logic Programming*, MIT Press, pp. 616–632 (May 1987).
6. K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma, KABU-WAKE: A New Parallel Inference Method and Its Evaluation, IEEE, *COMPCON Spring 86* (1986).
7. Ewing Lusk, David H. D. Warren, Seif Haridi, *et al.*, The Aurora OR-Parallel Prolog System, *New Generation Computing* **7**(2, 3):243–271 (1990).
8. H. Yasuhara and K. Nitadori, ORBIT: A Parallel Computing Model of Prolog, *New Generation Computing* **2**:277–288 (1984).
9. Doug DeGroot, Restricted And-parallelism, *Proc. of the Int'l. Conf. on Fifth Generation Computer Systems*, Tokyo, pp. 471–478 (November 1984).
10. Manuel Hermenegildo, An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, *Third Int'l. Conf. on Logic Programming* (Ed. Ehud Shapiro), Springer-Verlag, London, pp. 25–39 (1986).
11. Yow-Jian Lin and Vipin Kumar, AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results, *Proc. of the Fifth Int'l. Conf. and Symp. on Logic Programming*, pp. 1123–1141 (1988).
12. Prasenjit Biswas, Shyh-Chang Su, and David Y. Y. Yun, A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND parallelism (RAP) in Logic Programs, *Proc. of the Fifth Int'l. Conf. and Symp. on Logic Programming*, MIT Press, pp. 1160–1179 (August 1988).
13. John S. Conery, Binding Environments for Parallel Logic Programs in Noshared Memory Multiprocessors, *Int'l. J. of Parallel Programming* **17**(2):125–152 (April 1988).
14. Laxmikant V. Kalé, B. Ramkumar, and W. Shu, A Memory Organization Independent Binding Environment for, AND- and OR-Parallel Execution of Logic Programs, *Proc. of the Fifth Int'l. Conf. and Symp. on Logic Programming*, pp. 1223–1240 (1988).
15. Harald Westphal, Philippe Ropert, Jacques Chassin de Kergommeaux, and Jean-Claude Syre, The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism, *Proc. of the Symp. on Logic Programming*, pp. 436–448 (1987).
16. Mats Carlsson and Johan Widén, SICStus Prolog User's Manual, SICS Research Report R88007B (October 1988).
17. Shyam Mudambi, Personal communication (September 1990).
18. Khayri A. M. Ali and Roland Karlsson, The Muse OR-Parallel Prolog Model and its Performance, *Proc. of the North American Conf. on Logic Programming*, pp. 757–776, MIT Press (October 1990).
19. Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens, Scheduling OR-Parallelism: an Argonne perspective, *Proc. of the Fifth Int'l. Conf. and Symp. on Logic Programming*, MIT Press, pp. 1590–1605 (August 1988).

20. Khayri A. M. Ali and Roland Karlsson, Scheduling OR-Parallelism in Muse, *Proc. of the Int'l. Conf. on Logic Programming*, Paris, pp. 807–821 (June 1991).
21. David H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, *SRI International* (1983).
22. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel Prolog, PhD thesis, Swedish Institute of Computer Science, SICS Dissertation Series 01 (SICS/D-90-9901) (March 1990).
23. Gopal Gupta and Bharat Jayaraman, On Criteria for OR-Parallel Execution Models of Logic Programs, *Proc. of the North American Conf. on Logic Programming*, MIT Press, pp. 737–756 (October 1990).
24. Alan Calderwood and Péter Szeredi, Scheduling OR-Parallelism in Aurora—the Manchester scheduler, *Proc. of the Sixth Int'l. Conf. on Logic Programming*, MIT Press, pp. 419–435 (June 1989).
25. Per Brand, Wavefront Scheduling, Internal Report, Gigalips Project (1988).
26. Anthony Beaumont, Muthuraman, and Péter Szeredi, Scheduling OR-parallelism in Aurora with the Bristol Scheduler, Report TR-90-04, University of Bristol (March 1990).
27. Péter Szeredi, Performance analysis of the Aurora OR-Parallel Prolog System, *Proc. of the North American Conf. on Logic Programming*, MIT Press, pp. 713–732 (March 1989).
28. Khayri A. M. Ali, OR-Parallel Execution of Prolog on BC-machine, *Proc. of the Fifth Int'l Conf. and Symp. on Logic Programming*, pp. 1531–1545 (1988).
29. David H. D. Warren, The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation Issues, *Proc. of the Symp. on Logic Programming*, pp. 92–102 (1987).