A Scheduling Strategy for Shared Memory Multiprocessors

Lal George[1]

UUCS-90-002

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

September 26, 1990

Abstract

An efficient scheduling strategy for shared memory multiprocessors is described. The rapid dissemination of tasks to available procesors and ready queues is crucial to the performance of any parallel system. Such overheads determine the attainable speedup and performance of the system. Poor techniques used to address this can lead to severe degradation in performance particulary with high processor counts. This work has been conducted in the context of a parallel functional language-CoF, where the parallelism is usually fine grained and the efficient assignment of tasks to processors even more important. In such systems, observing strict queue semantics (i.e., FIFO) is not essesntial. This allows for very efficient algorithms such as that described here. On the BBN GP1000, our technique was superior in performance to the centralized queue and has the potential of performing well on a fully configured GP1000.

[1]This work has been supported in part by the National Science Foundation under Grant CCR-8704778 and an IBM Fellowship Award.

A Scheduling Strategy for Shared Memory Multiprocessors

Lal George*
Parallel Programming Research Group
Dept. of Computer Science
University of Utah
Salt Lake City, UT 84112

george@cs.utah.edu

Abstract

An efficient scheduling strategy for shared memory multiprocessors is described. The rapid dissemination of tasks to available processors and ready queues is crucial to the performance of any parallel system. Such overheads determine the attainable speedup and performance of the system. Poor techniques used to address this can lead to severe degradation in performance particularly with high processor counts. This work has been conducted in the context of a parallel functional language - CoF, where the parallelism is usually fine grained and the efficient assignment of tasks to processors even more important. In such systems, observing strict queue semantics (i.e., FIFO) is not essential. This allows for very efficient algorithms such as that described here. On the BBN GP1000, our technique was superior in performance to the centralized queue and has the potential of performing well on a fully configured GP1000.

1 Introduction

Efficiently allocating tasks to available processors is crucial to the performance of any multiprocessor system. The simpliest approach on a shared memory machine is to use a centralized queue structure. The queue can contain either tasks, indicating an excess of tasks over available processors or process ids - *pids*, indicating an excess of processors over tasks. The drawbacks with using a centralized queue are that it becomes a severe bottleneck for even a small number of processors and there is no concurrency in the basic operations. Figure 1 shows statistics gathered from execution of the prime number sieve program (Section 3.1) varying the number of processors. When a processor is unable to acquire the lock associated with the centralized queue it *backs off* for an *exponentially* increasing amount of time before it tries again. To optimize the execution, a failure to acquire the lock when attempting to migrate a task results in the task being executed locally. While there is no observable

*This work has been supported in part by the National Science Foundation under Grant CCR-8704778 and an IBM Fellowship Award.

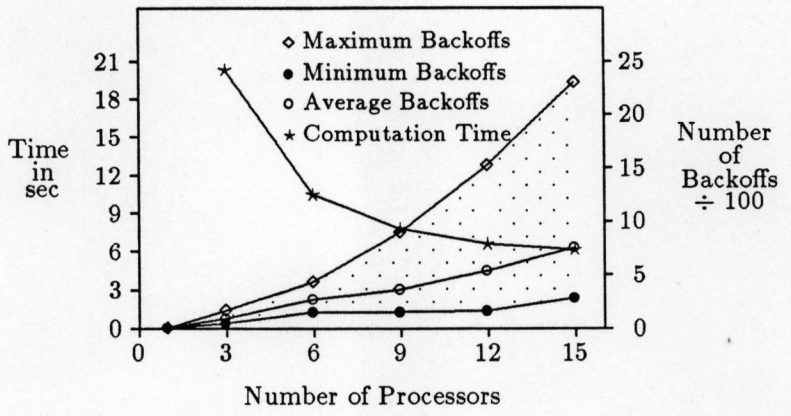

Figure 1: Backoff Statistics for the Prime Number Sieve

degradation in overall performance, the maximum number of backoffs with 15 processors was as high as 2300 and rapidly increasing!

The work reported in this paper has been done in context of a parallel functional language CoF[5], being developed at the University of Utah. With high processor counts the degradation in performance, lack of concurrency and the limitations of the centralized queue were found to be unacceptable. We describe a technique which:

1. Is superior to the centralized queue for large processor counts,
2. Admits concurrency within the basic queue operations,
3. Drastically reduces the hot spot contention for a centralized lock and
4. Requires little bookeeping and is therefore quite efficient.

1.1 Previous Work

There has been considerable work related to scheduling tasks on a shared memory multiprocessor. The BBN Butterfly I and Butterfly Plus supported in hardware the idea of a *dual queue*, which found extensive use in programming systems[8,9,11]. The queue could contain either process-ids or data and hence the name. The dual queue forms the basis of our implementation, however, we use a spin-waiting implementation and enqueue the *address* of a location that a processor is spin-waiting on rather than its process-id. Notification involves writing a non-zero value on this location. With large processor counts a single dual queue becomes a severe bottleneck.

Anderson[4,3] discusses the effects of varying data structure and thread management techniques on the performance of shared memory multiprocessors. While there are some broad similarities, we feel our technique is more suitable to the Butterfly than the schemes they considered.

Rao and Kumar[7], and Jones[6] discuss techniques to implement priority queues where they adhere to strict queue semantics. Such a requirement is not essential in our context and would impose too high an overhead; such as that of *reheapification* on every dequeue[7].

1.2 BBN GP1000

The features of the BBN GP1000[2,10] relevant to our discussion are:

1. A maximum configuration of 256 nodes each employing a Motorola MC68020 microprocessor.
2. Each node contains 4 megabytes of memory providing a total of 4 gigabytes of shared memory on a fully configured machine.
3. A remote to local access time ratio of 5:1.
4. Node interconnection via a delta connected switch.
5. Process and virtual memory management via the MACH operating system.

Concurrent execution under MACH is obtained by *forking* child processes onto other processors. Forking a UNIX process is equivalent to creating a *task* with a single *thread*[1]. The GP1000 implementation of MACH does not permit more than one thread per task as this would permit a thread to execute in a task created on another processor, and thus incur severe performance penalties, e.g. in code access over the switch. Consequently, none of the thread management primitives such as *thread_join*, *thread_suspend*, etc, are supported. The usual mode of usage on the GP1000 is to have one process per processor communicating via shared memory. Hence, we may frequently use the words process and processor synonymously.

A *spin-waiting* implementation has been used in all experiments described in this paper where a processor waiting for a task periodically reads the value of a *local variable*. A non-zero value indicates the presence of a task.

2 Distributed Dual Queues (DDQs)

The main data structure used in our technique is a binary tree with dual queues at the leaves of the tree. The initial state of the DDQ is shown in Figure 2. Each internal node contains two fields `N` and `D`. The `D`-field[1] is a *static* constant that represents the *capacity* of the queues below that node in the tree *assuming no processors presents in the system.* Thus node `d` has its `D`-field set to 3 as the queue directly below it has a capacity of 3, while the internal node `b` has its `D`-field set to the sum of the `D`-fields of its immediate children. The `N`-field is dynamic (modifiable) and is equal to the `D`-field at startup. Informally, if the value of the `N`-field at a node x is n, then at *least* n data items may be buffered at the queues rooted at node x. During startup each processor is assigned to a specific leaf of the tree which we shall refer to as the `HOME_LEAF` of the processor. The labels p1-p8 represent processor ids and are positioned below their assigned `HOME_LEAF`.

2.1 DDQ Operations

The basic queue operations are (i) spawning of parallel work, corresponding to an *insert* operation, and (ii) obtaining work from other processors corresponding to a *remove* operation. These involve a traversal of the tree structure.

[1] We only make use of this field in Section 2.2

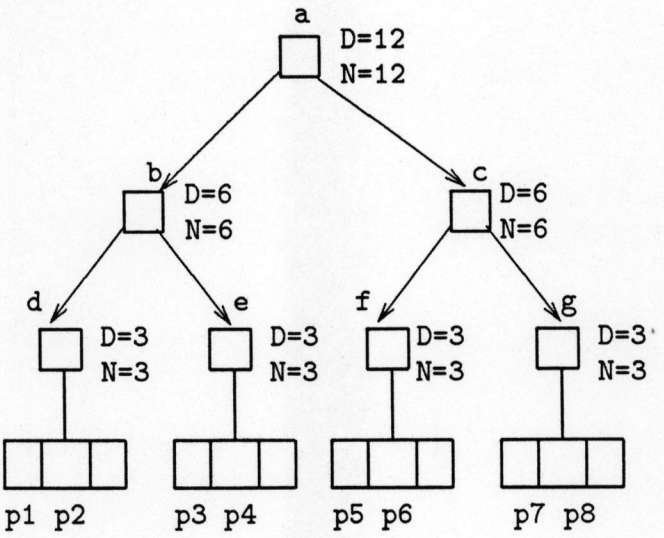

Figure 2: Distributed Dual Queue Structure

Insert

An *insert* operation involves descending the tree structure from the root until an empty slot at one of the queues is found. A processor performing an *insert*, reads the value of the N-field at the root. From our initial condition, if this is greater than 0, then there must be an available slot below it in the tree structure. Thus, if it can atomically decrement the root to a value greater than or equal to zero, it is has effectively *reserved* a slot space. Practically all machines, both single and multiprocessor, provide some sort of *test and set* operation where the old value of the operand is returned, hence this is not a special hardware requirement. The process of visiting a node is split into two phases: 1) reading the value at the node, and 2) conditionally decrementing the value by 1. We could alternatively omit the reading phase and just perform the atomic decrement immediately. However, atomic operations are expensive and an incorrect atomic decrement resulting from decrementing a value less than or equal to zero must be compensated for by an atomic increment. If the value at the root is zero (or transiently less than zero) then there is no free space below it in the tree and the *insert* fails.

If the root decrement operation *succeeds* in the manner described above, then the *insert* operation continues by finding a child node with a value greater than zero and repeating the decrement until a queue with an empty slot attached to a leaf is found, at which point the insertion can take place. From the initial condition, once the processor has crossed the root there is guaranteed to be an empty slot[2] at one of the queues (Section 2.1).

C language pseudo code for *insert* is shown below. The function *notify_reserved_processor*

[2]Since the queues are really dual queues, the inserting process may find a process-id instead of an empty slot. We defer an explanation of the action taken until after we have described how process-ids are entered into the dual queue.

is not shown but is discussed in Section 2.2.

```
boolean insert (task, tree) Task * task; DDQ * tree;
{
    if (tree→N >= 0) {
        if (reserve_processor(tree))
            notify_reserved_processor(tree);
        else return FALSE;
    } else return FALSE;
}

boolean reserve_processor (tree) DDQ * tree;
{
    if (atomadd(tree→N, -1) > 0) return TRUE;
    else {   atomadd(tree→N, 1);
             return FALSE;}
}
```

Remove

If there is data or tasks in the queue at the **HOME_LEAF** of a processor, the item is dequeued and the **N**-field of each internal node on the path to the root unconditionally incremented. This indicates that there is one more empty slot in the queue corresponding to the one from which data was just dequeued. If there is no data in the queue, then the process id or its spin waiting location is inserted into the queue and as before, the **N**-field of each internal node on the path to the root unconditionally incremented. The *insert* operation described above must be modified so that if the queue contains process ids, the spin-waiting processor is notified. It is permissible for a processor to be notified while it is ascending the tree as it will discover the task as soon as it enters its spin waiting loop. The C pseudo code for *insert* is:

```
remove()
{
    lock (HomeQueue);
    if (HomeQueue→status == DATA) dequeue_data();
    else insert_pid();
    unlock (HomeQueue);
    for (n= HOME_LEAF; n >0; n = n >> 1)
        atomadd(n→N, 1);
    spin_wait();
}
```

Correctness

In order to guarantee correct operation we must maintain the invariant that the **N**-field at a node be less than or equal to the number of data items that can be inserted into the queues below it. From the initial condition (when **N = D**), this invariant is trivially satisfied at startup. We show that the *insert* and *remove* operations do not violate this invariant. If a *remove* operation dequeues a data item then there is one more slot available for data and the unconditional increment to the root maintains the invariant. If the *remove* enqueues it

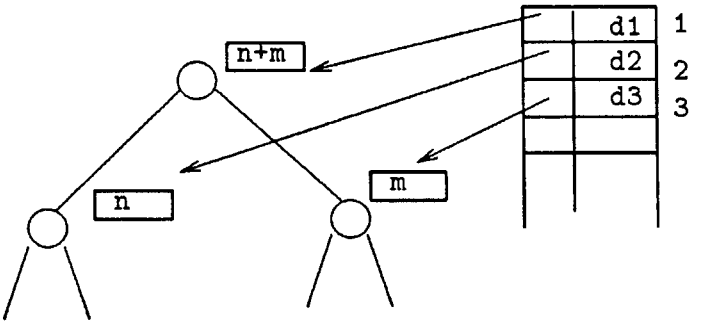

Figure 3: Implementation of Access Tree

process-id, then that same slot can hold two data items; one that can be absorbed by the processor and the other that can be enqueued into the slot itself. A processor descending at a node after successfully decrementing its N-field indicates that there is one less slot available below that node and hence maintaining the invariant.

From the above, it follows that if the processor descends the root node it is guaranteed to find a queue in which to enqueue data.

2.2 Implementation

The implementation distributes the tree and queues across the machine as much as possible. Thus a queue is allocated on one of the processors that will be ascending from it. Furthermore, each internal node is allocated on a processor that can potentially spin-wait in a queue below that node. Thus node b and the left most queue in Figure 2 may be allocated on one of processors p1-p4. There is now a greater chance that an atomic operation is performed on a processor that is idle.

The binary tree is implemented as an array which is *copied* on every node of the machine and is *read-only*. This is possible as the tree is balanced and does not change shape dynamically at runtime. Each element of the array contains the static D-field and a pointer to the location of the dynamic N-field which is distributed across the machine. Figure 3, depicts a fragment of the table for the root node and its children. Movement up and down the tree are implemented using shift operations on the node number.

To avoid processors always picking the same branch to descend at a node the bit patterns on the data being enqueued are decoded as preferred directions for each level. Thus if the data has a bit pattern of 101001..., then this is decoded as LRLRRL... and consequently the left direction is preferred at level 0, the right direction at level 1, ... etc. This yields a rudimentary form of *pseudo randomness*. Furthermore, it is desirable that the *insert* operation *homes in* on queues with idle processors. Since the D-fields contains the number of empty slots assuming no processors (Section 2) the following given without proof is true. If at a node:

N=0 Then there is no space below the node to buffer any more data and all queues are full.

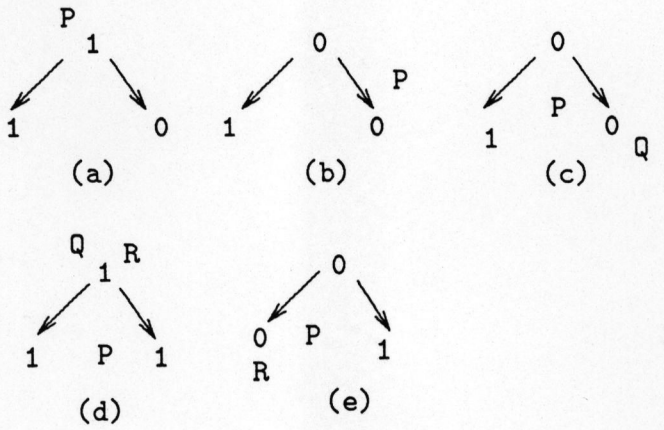

Figure 4: Starvation while Descending a Node

N $\leq$ D Indicates that there are N slots below this node to buffer the data item but no available processors to immediately take up the work.

N>D Indicates that there are `D` slots to buffer data and in addition there are $N - D$ idle processors below this node as well.

This criteria is first used to select the preferred path.

In what has been described so far, there is the possibility of *starvation* while trying to descend a node. Figure 4 shows the possible `N` values at the root of a `DDQ`.

In (a) the process `P` finds that the value of a node is greater than 0, successfully decrements the count to zero and moves down the right branch first. In (b) seeing that the count on the current node is `0` the process moves over to the left branch. Meanwhile, another process `Q` doing a *remove* operation ascends the right branch incrementing the count, leaving the tree in the state shown in (d). While `P` is sitting at the left branch another process `R` doing an *insert* operations sees that the value of the top node is greater than 0 and decrements the tree going down the left branch while `P` is still blocked for some reason. The solution to this is that `P` should go back and try the right branch. The chances of this actually happening in practice are likely to be very low.

2.3 Lazy Ascends

Having to ascend the tree each time a data item is pulled out of the queue is wasteful and inefficient. When there is more than one data item in the queue we need not ascend the tree but rather note the number of *lazy deferred ascends* in a field associated with the queue. So the processor merely dequeues the data item and increments the lazy ascends count. At some point a processor must ascend the tree with this accumulated sum of lazy ascends. There are two possibilities we considered for when the ascend may occur:

1. When the queue changes from *empty* to containing processors, the first processor to insert itself into the queue may ascend the tree.

2. When the queue changes from having data to being empty, the processor taking out the last data item may in the interest of the other working processors perform the ascend.

We have arbitrarily chosen the second alternative, believing that the best choice would be application dependent.

Likewise in the interest of locality, an *insert* operation may first check to see if there is space on the local queue before descending from the root. If successful in this endeavor it decrements the *lazy ascends* count.

2.4 Variations

The advantage with this structure is that it admits of a lot of concurrency. Many processors may be simultaneously performing a mixture of *insert* and *remove* operations. Furthermore, no locking is required traversing the tree other than that transiently in effect during atomic increment or decrement. The contention on the queue is reduced to those processors that have access to the queue and the entire structure is well distributed across the machine.

However, the concurrency comes at a price. For a machine with N processors and a queue size of q the height of the tree H is $\lceil log_2(N/q) \rceil$. An *insert* or *remove* will involve H atomic operations at best and probably much more for an *insert* due to starvation and picking the wrong branch. This can be offset by increasing q at the cost of introducing contention among processors sharing a queue. Also the root of the tree becomes a hot spot. This can be offset by having more than one tree. An *insert* operation now reads (or polls) the value of the roots in a round robin fashion until it can find one where it can descend.

3 Performance

The scheduling structure was tested on both artificial loads and programs compiled by our CoF compiler for typical application programs. In the first case, half the processors behaved as producers performing *insert* operations and the other half behaved as consumers performing *remove* operations until a termination message was received. The last producer to terminate sends out the termination messages. The relevant code is shown below:

```
#define ITERATIONS 20000
producer ()
{   int i, j;
    for (i = 0; i < ITERATIONS; i++) {
        j = 0;
        while (insert (i) == FALSE) backoff(j++);
    }
}
consumer ()
{   int i, data;
    while (TRUE) if (remove() == -1) return;
}
```

In Figure 5 we show the effect of multiple clusters with a queue size of 1. Varying other parameters yielded similar results.

3.1 Typical Programs

Artificial loads are sometimes a poor indication of typical programs so we tried our technique on code generated from our compiler for the prime number sieve program shown below.

```
fun from n m = if n < m then nil else n :: from (n+1) m
fun filter p x =
    if null x then nil
    else if x mod p = 0 then filter p (tl x)
        else (hd x) :: filter p (tl x)
fun sieve l =
    if null l then nil
    else (hd l) :: sieve (filter (hd l) (tl l))
```

Figure 6 summaries our results for: (i) the centralized queue; (ii) tree of height 1 (or 2 queues)and (iii) tree with queue size of 4. We draw the following observations:

- The parallelism in the problem peaks out between 16-24 processors.
- With high processor counts the DDQ does not degrad as badly as the centralized queue. This is due to the reduced contention for the root node and added concurrency in the basic operations.
- As with Figure 5, the contention for the root node may be reduced by having more than one tree.

4 Related Issues

4.1 Barrier Synchronization

The idea of using a distributed tree very similar to that described here has been employed in performing barrier synchronization by Yew, *et.al.*[12]. In their work, the data stored at each internal node is the number of processors that have yet to enter the barrier. Beyond this broad similarity the overall problem they have to tackle is quite different from the one we have addressed here.

Other Architectures

The DDQ is particularly suitable for clustered machines like the Evans and Sutherland ES-1 machine or the CMU Cm* machine. A DDQ may be used to effectively utilize the processors on a single cluster and the communication mechanism built into the machine may be used to communicate between trees.

Bus Architectures

Even though bus connected multiprocessors are radically different we conjecture that the same idea may be useful in minimizing the contention for a specific memory module in the system. The presence of a cache can have subtle effects on performance of a centralized queue.

Distributed Machines

These ideas may not be entirely appropriate for local area networks or machines like the Intel Hypercube since the amount of communication involved in traversing the tree is likely to be an overriding factor.

5 Conclusions

A scheduling technique for shared memory multiprocessors has been described. The technique requires little bookeeping, admits of concurrency within the basic operations and is superior to the centralized queue. The scheduling technique is currently employed in the runtime system for a parallel functional language.

6 Acknowledgements

I would like to thank Dr. Gary Lindstrom for valuable suggestions, Dr. K. Pingali for graciously granting me permission to use the Butterfly at Cornell and IBM for their generous support of this work.

References

[1] Avadis Tevaninan, Jr., Rashid, Richard F., Golub, David B., Black, David L., and Young, Michael W. *Mach Threads and the Unix Kernel: The Battle for Control.* Technical Report CMU-CS-87-149, Carnegie-Mellon University (August 1987).

[2] Downey III, W. J. *Inside the GP1000.* BBN Advanced Computers Inc, revision 1.0 edition (1988).

[3] Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (January 1990) 6–16.

[4] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The performance implications of thread management alternatives. *IEEE Transactions on Computers*, 38, 12 (December 1989) 1631–1644.

[5] George, L., Mantha, S., and Lindstrom, G. *Monotonic Side Effects: Towards Amalgamating Functional and Logic Programming.* Technical Report UUCS-001-90, University of Utah (Jan 1990).

[6] Jones, D.W. Concurrent operations on priority queues. *Communications of the ACM*, 32, 1 (Jan. 1989) 132–137.

[7] Rao, V.N. and Kumar, V. Concurrent insertions and deletions in a priority queue. In *International Conference on Parallel Processing* (1988) 207–211.

[8] Swanson, Mark. *A Portable Standard Lisp System for the BBN Butterfly.* MS thesis, University of Utah (May 1988).

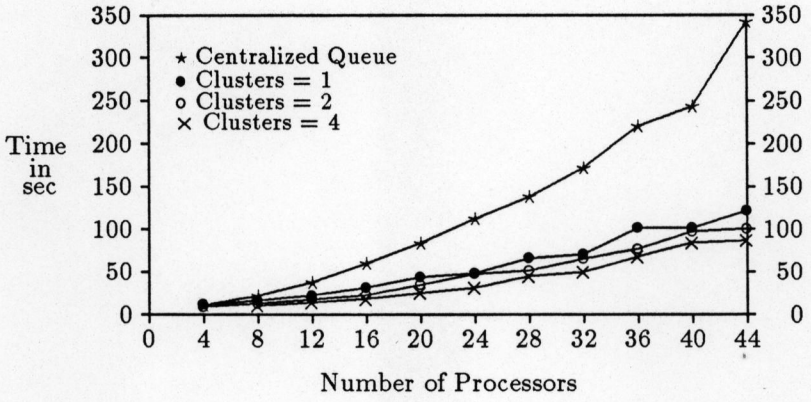

Figure 5: Effect of Multiple Clusters with Queue Size = 1

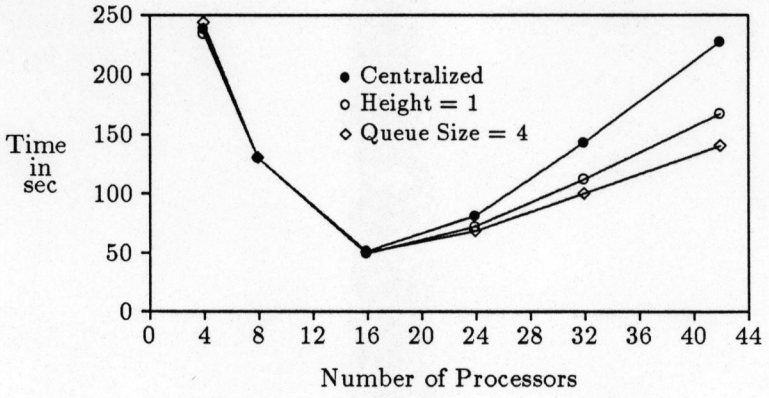

Figure 6: Prime Number Sieve Program

[9] Swanson, Mark R., Kessler, Robert, and Lindstrom, Gary. An implementation of Portable Standard Lisp on the bbn butterfly. In *Proc. Lisp and Functional Programming Conference*, ACM, Snowbird, Utah (July 1988) 132–141.

[10] Thomas, B., Gurwitz, B., Goodhue, J., and Allen, D. *Butterfly Parallel Processor: Overview.* Technical Report 6148, BBN Laboratories Incorporated (March 1986).

[11] Tinker, Peter A. Performance of an OR-parallel logic programming implementation. *International Journal of Parallel Programming*, 17, 1 (February 1988) 59–92.

[12] Yew, P. C., Tzeng, N. F., and Lawrie, D. H. Distributed hot-spot addressing in large scale multiprocessors. *IEEE, Trans. on Computers*, C-36, 4 (April 1987).