Buckwheat: Graph Reduction on a Shared Memory Multiprocessor

Benjamin Goldberg
Department of Computer Science
New York University
251 Mercer Street
New York, NY 10012

Abstract

Buckwheat is a working implementation of a functional language on the Encore Multimax multiprocessor. It is based on a heterogeneous abstract machine model consisting of both graph reduction and stack oriented execution. Buckwheat consists of two major components: a compiler and a run-time system. The task of the compiler is to detect the exploitable parallelism in programs written in ALFL, a conventional functional language. The run-time system supports processor scheduling, dynamic typing and storage management.

In this paper we describe the organization, execution model, and scheduling policies of the Buckwheat run-time system. A large number of experiments have been performed and we present the results.

1 Introduction

Functional languages have recently gained attention as vehicles for programming in a concise and elegant manner [2,12]. In addition, it has been suggested that functional programming provides a natural methodology for programming *multiprocessor* computers. Unfortunately, there has been little empirical evidence, until now, to support this hypothesis. This paper describes a working implementation of a functional language on a commercially available shared memory multiprocessor.

1.1 Objectives

This paper addresses the following question:

> Is it feasible to execute conventional functional programs on current shared memory multiprocessors such that a significant reduction in the execution time is achieved?

Some of the terms used in the above question need to be defined:

- *Conventional functional programs*: We seek to create an implementation for a functional language that does not contain any special constructs for specifying the parallel behavior of a program. Our implementation must be able to *automatically* decompose functional programs to run on a multiprocessor.
- *Reduction in execution time*: We are investigating whether a functional program can run significantly faster on a shared memory multiprocessor than on a sequential (uniprocessor) computer. We would ultimately like to show that functional programming is the *most* appropriate method for programming parallel computers. However, in this paper we restrict ourselves to the investigation of the advantages of using parallel machines instead of sequential machines to execute functional programs.

Buckwheat is a research project started at Yale University (and continued at Yale and NYU) that attempts to address these issues via an implementation of a functional programming language on the Encore Multimax, a shared memory multiprocessor. There are two major components to Buckwheat, a compiler and a run-time system. The task of the compiler is to detect the exploitable parallelism in programs written in ALFL, a

This research was supported in part by the US Government under DOE grant FG02-86ER25012

conventional functional language. The Buckwheat runtime system provides much of the mechanism for the distributed execution and performs such tasks as processor scheduling, dynamic typing, and storage management.

In addition to Buckwheat, an ALFL implementation has been built on an Intel iPSC hypercube multiprocessor and is called **Alfalfa**. A description of Alfalfa, along with experimental results, can be found in [4,5].

1.2 The Encore Multimax

The Encore Multimax [3] is a bus-based shared memory multiprocessor. Buckwheat was implemented on a system that contained twelve processors. Each processor is a 10 MHz National Semiconductor NS32032 microprocessor. Any location in memory can be accessed by any processor over a very fast bus called the Nanobus. An important feature of the shared memory in the Multimax is that any byte can be used as a lock (for enforcing mutual exclusion, etc.). Atomic test-and-set instructions are supported in order to set and reset these locks.

1.3 The Source Language

The functional programs that get executed by Buckwheat are written in ALFL [7], a non-strict functional language designed at Yale. It is similar in many ways to other existing functional languages such as Miranda [11] and LML [1]. It is weakly typed, however, and forces Buckwheat to perform run-time type checking.

Perhaps the most important aspect of ALFL, as it applies to this paper, is the fact that it contains no explicit constructs for expressing parallelism. The programmer is dependent on the Buckwheat compiler and run-time system to detect and exploit the inherent parallelism in his program.[2]

1.4 Graph Reduction

Graph reduction [13] is the evaluation method most often used to execute functional programs. It can be thought of as the graphical equivalent of reduction in the lambda calculus, and supports higher-order functions and lazy evaluation in a very natural manner. In graph reduction a program, along with its data, is represented as a graph. During execution, reductions (conversions) are applied to the graph until it has been reduced to a *normal form*, to which no more reductions can be applied. For example, the initial graph representing the ALFL program

```
{  f x y == h (x y) y;
   g a b == a + b;
   h c d == c * d;
   result f (g 2) 3;
   }
```

is shown in figure 1. The "@" symbol represents function application. Notice that the application of `f` to two

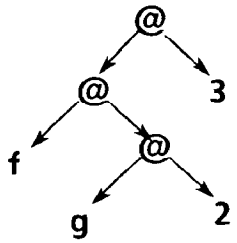

Figure 1: The initial graph

arguments is curried and is represented by two application nodes. Reduction proceeds via the construction of an instance of `f`'s body with its formal parameters replaced by pointers to the corresponding arguments. This is shown in figure 2. According to the definitions

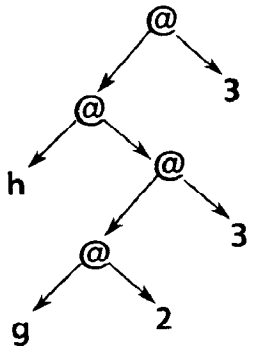

Figure 2: The graph after the first reduction step

of `g` and `h`, the reduction of the graph proceeds as shown in figure 3.

In the above example, the function identifier in an application always resided at a leaf in the graph to support currying. Each interior node represented the

[2] There is a dialect of ALFL, called ParALFL [9], that provides explicit constructs for expressing parallelism.

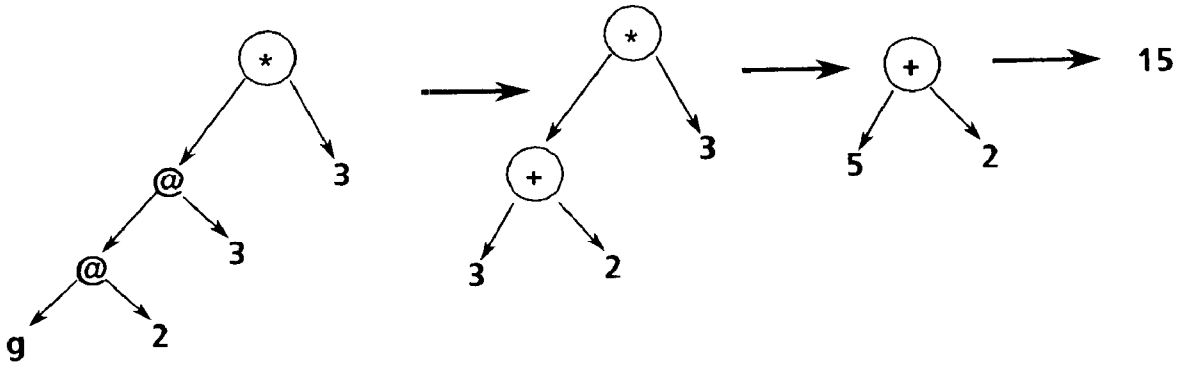

Figure 3: The reduction of the graph

application of its left child to its right child. If a function is supplied with all the arguments it needs (as in the above application of `f`), an *uncurried* application could be represented by a node containing the function and arguments. For example, the uncurried version of `f (g 2) 3` in the above program could be represented as shown in figure 4. In this case, the node serves as an

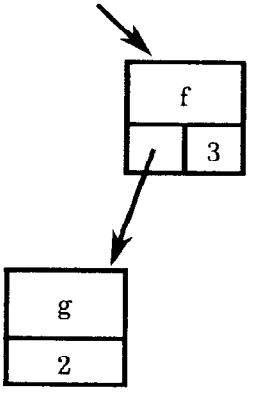

Figure 4: The uncurried version of `(f (g 2) 3`

activation record for the function call.[3]

Graph reduction can be used to support *parallel execution* of functional programs. Any sections of the graph that are eligible to be reduced (without violating ALFL's non-strict semantics) can be reduced in parallel. For example, the parallel reduction of `f (g 1 2) (h 3 4)`, represented in figure 5, could proceed by the evaluation of `(g 1 2)` and `(h 3 4)` in parallel, as long as `f` required both of their values. If this

[3]In applications that cannot be uncurried, an explicit apply node is required.

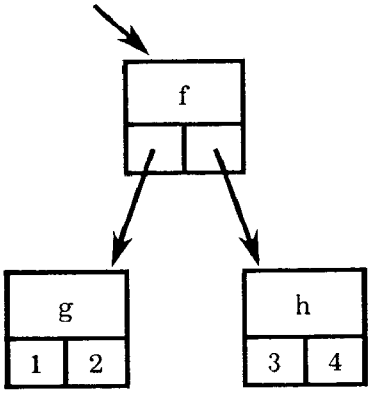

Figure 5: The applications of `g` and `h` can be evaluated in parallel

approach is taken, the functions in a program, such as `g` and `h` above, specify the behavior of tasks that run in parallel and determine the granularity of the computation.

1.4.1 Serial Combinators

We have previously [4,8] discussed transforming the source program into a new set of functions, called *serial combinators*, that exhibit an appropriate grain size for the target architecture. The body of each serial combinator is sequential and can be executed on a single processor without sacrificing useful parallelism. Serial combinators are *combinators*, functions that contain no free (non-local) variables. This means that the activation record for each serial combinator call contains all of the variables that may be referenced in the serial combinator body.

In section 3 we describe the mechanism for executing serial combinators. It is worth mentioning here, however, that the code generated for each serial combinator is native code for the multiprocessor. It is not a sequence of instructions for an abstract machine, such as a "graph reduction engine", that needs to be interpreted at run time. If the execution of a serial combinator requires synchronization—or some other mechanism not provided by the hardware—then the compiler simply generates a call to a system routine provided by the Buckwheat run-time system.

2 Heterogeneous Evaluation Model

We would like to minimize the overhead involved in execution by utilizing the mechanisms to support parallelism and lazy evaluation only when required. In doing so Buckwheat becomes a hybrid of two evaluation models: graph reduction and sequential stack-based execution[4].

In distributed execution of functional languages, graph reduction serves two basic purposes: It supports heap (i.e. graph space) allocation of closures, and supports a multi-threaded dynamic chain for parallel function calls. Heap allocation of closures is necessary for lazy evaluation and higher order functions; these closures represent "delayed" expressions during execution.

A serial combinator may require the values of serial combinator calls being evaluated by other processors. Thus, the system has to provide a mechanism for suspending the evaluation of a serial combinator if a required value is not yet available. This necessitates the creation of activation records in a heap, and not in a stack, in order to preserve the state of the serial combinator call during suspension. The processor has to be free to use the stack for other purposes while a serial combinator call is suspending.

In many programs, however, there are function calls in which the function is sequential and is not passed any unevaluated arguments. In this case, no heap allocation is necessary and the invocation of the function can be allocated on the stack, just as if it were part of conventional (i.e. sequential and call-by-value) language.

In many programs, however, there are sequences of function calls that contain no parallelism and require no mechanism for suspending. In these cases, heap allocation of activation records is unnecessary. The activation records for these function calls should be (and are in Buckwheat) allocated on a stack. The stack provides a sequential return mechanism at very low cost.

Briefly, we say a function is sequential (and can be evaluated using a stack) if it satisfies the following conditions:

1. It makes no parallel function calls, i.e. it only calls one function at a time.
2. It never "forks" a function call, i.e. it never proceeds without waiting for a value of a function call to return.
3. It only calls functions that are themselves sequential.

Determining if a function, f, is sequential in a first order language involves solving a simple recursive set equation. In the higher order case, f may make calls to unknown functions (bound to local variables) and a more sophisticated analysis is needed to determine if the unknown function is sequential. If a sequential function f has arity n then an application of f to n arguments can be executed on a stack if all arguments to the invocation have already been evaluated. The Buckwheat compiler currently solves the set equation for sequential functions in the the first order case. For those functions f_i that it has found to be sequential, it generates two definitions. One is for sequential stack based execution when the compiler can determine that the actual parameters in a call to f_i have been evaluated. A graph reduction based definition is also generated for the case where a call to f_i may involve unevaluated arguments.

3 Shared Memory Graph Reduction

In graph reduction, the program graph logically resides in a single graph space. Thus, a shared memory multiprocessor is the most natural architecture on which to implement graph reduction. On the Multimax any processor can access any component of the program graph. Naturally, access to any node in the graph that is being mutated must be restricted to the processor performing the mutation.

Buckwheat's processors are *self-scheduled*. That is, when a processor becomes free it removes a task from a shared task queue and performs the action dictated by the task. No processor needs to be aware of the state of any other processor in the system.

3.1 System Organization

The organization of buckwheat is shown in figure 6. Each processor has a private copy of the graph reducer

[4] The G-machine [10] uses a similar hybrid evaluation model for sequential evaluation of functional programs

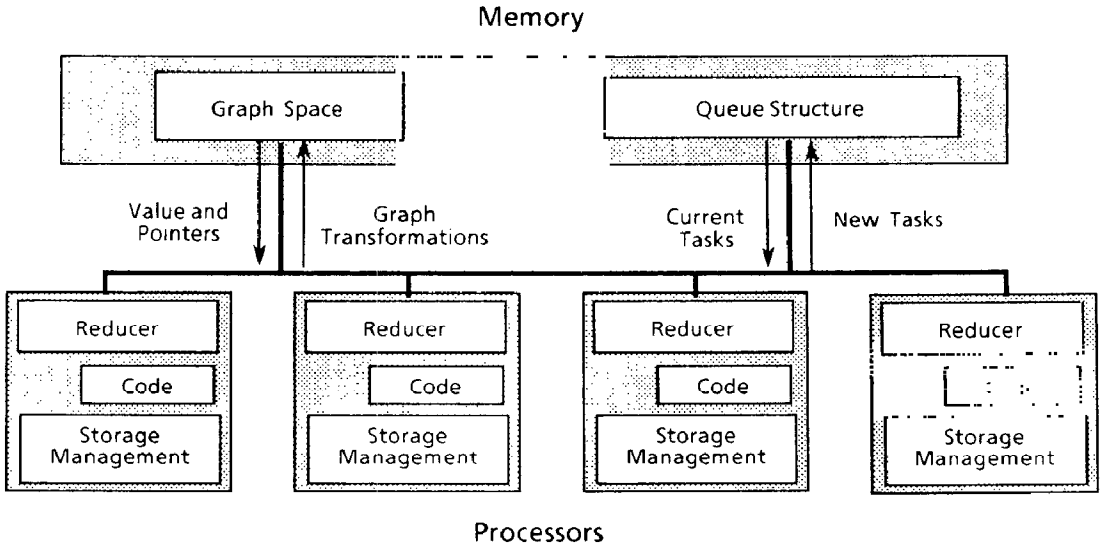

Figure 6: The Buckwheat system

module, serial combinator code, and storage manager. Even though the Multimax has a single physical memory, multiple copies of these modules allows the processors to execute the routines without memory contention. Of course, there may still be contention for the bus. However, the Nanobus is fast enough that the effect of bus contention is minimal.

The graph space and task queue structure reside in a shared area of memory. In its simplest form, the queue structure consists of a single queue from which all processors access tasks to be executed. A more sophisticated task queue structure is described in section 4.

3.2 Synchronization by serial combinators

Unlike ALFL functions, serial combinators contain constructs that specify the synchronization necessary for parallel execution. The algorithms used by the compiler to detect the inherent parallelism in an ALFL program—and to generate the necessary synchronization constructs—can be found in [4,8].

The basic synchronization constructs are *demand*, *wait*, and *spawn*. For ease of explanation, we will represent serial combinators using S-expression syntax much like that of LISP.

In a serial combinator, the *demand* construct

(**demand** (v_1 ... v_n)
body)

indicates that the values of variables $v_1 \ldots v_n$ may be safely demanded (in parallel). Because serial combinators preserve laziness, we must be certain, using strictness analysis, that the values of $v_1 \ldots v_n$ will be needed at some point in the computation. The values of $v_1 \ldots v_n$ do *not* have to return before *body* is evaluated.

The *wait* construct

(**wait** (v_1 ... v_n)
body)

indicates that the values of $v_1 \ldots v_n$ must be available before the evaluation of *body* can even begin. If any of $v_1 \ldots v_n$ are still being evaluated then evaluation of the current serial combinator is suspended. Even though evaluation of a serial combinator is blocked, the processor is free to evaluate any other serial combinator application whose value has been requested.

The *spawn* construct

(**spawn** ((v_1 exp_1) ... (v_n exp_n))
body)

indicates that $exp_1 \ldots exp_n$, as well as *body*, should be evaluated. If there are a sufficient number of available processors, the spawn construct will cause these expressions to be evaluated in parallel. When each exp_i has been evaluated its value is bound to the variable v_i. Since evaluation of *body* proceeds without blocking on the values of $v_1 \ldots v_n$, each v_i must occur within a wait before being referenced in *body*.

Figure 7 shows a divide and conquer factorial as it appears in ALFL and serial combinator form. Notice that serial combinators also contain a LISP-like *let* construct for evaluating simple expressions sequentially.

3.3 The Graph Reducer

The synchronization constructs that serial combinators contain are simply calls to routines in Buckwheat's

```
{ pfac l h == l=h->l, { mid == (l+h)/2;
                        result pfac l mid + pfac (mid+1) h;
                        }
  result pfac 1 10;
  }


pfac l h == (demand (l h)
               (wait (l h)
                  (if (= l h) l
                     (let ((mid (/ (+ l h) 2)) (v1 (+ mid 1)) )
                        (spawn ((v2 (pfac l mid)) (v3 (pfac v1 h)))
                           (wait (v2 v3)
                              (+ v2 v3)))))))
```

Figure 7: Divide and conquer factorial in ALFL and serial combinator form

graph reducer module. These routines perform the necessary transformations on the graph. Before discussing how these transformations are performed, we describe the data structures involved.

3.3.1 Data Structures

A *node* in the graph is a contiguous block of bytes that contains the following fields:

- *State*: Either "unevaluated", "pending" (which means that the node is in the process of being evaluated), or "evaluated".
- *Value*: If the node has been evaluated, the value field contains the result. Otherwise it contains a pointer to code that specifies the computation to be performed when the value of the node is requested.
- *Args*: This is a vector containing the values of the arguments in the function call represented by the node. Each element contains either a value or a pointer to another node in the graph.
- *Requests*: A list of other nodes that have requested the value of this node.
- *Evalfield*: A bitfield indicating the status of each element in the args vector. If the ith bit of the bitfield is 1 then the ith argument has already been evaluated and contains a value. Otherwise the ith argument is a pointer to another node.
- *Waitmask*: A bitfield indicating which arguments must be evaluated before evaluation of the node can proceed. Evaluation proceeds when, for every 1 in the waitmask, there is a corresponding 1 in the evalfield.
- *RefCount*: The reference count of the node for storage reclamation purposes.
- *Lock*: A lock for enforcing mutual exclusion.

A *task* is an instruction that specifies a step for the run-time system to take in order to reduce the graph. Program execution proceeds by repeatedly removing tasks from the task queue of each processor and performing the action specified by the task. There are two kinds of tasks:

- An *evaltask* contains pointers to a target node and a source node. It indicates that the value of the target node is being requested by the source node.
- A *returntask* contains a pointer to a target node and a value. It indicates that some other node has been evaluated and is returning its value to the target node.

3.3.2 Execution

Execution begins via the creation of a collection of nodes representing the initial graph. An evaltask requesting the value of the root node of the graph is placed on the shared task queue. Execution proceeds with each processor removing tasks from the shared task queue.

If an evaltask is encountered and the target node n is unevaluated, evaluation of n proceeds by a jump to the code pointed to by n's value field. This code is the code generated for a serial combinator by the Buckwheat compiler. When the serial combinator code is finished

executing, a returntask is created to return the resulting value v to any requesting node. n's state is modified to "evaluated" and its value field is overwritten with v. Otherwise, if n has already been evaluated when the evaltask is encountered, the reducer immediately creates a returntask with n's value.

When a processor encounters a returntask returning a value to a node n, the appropriate elements of n's args vector and evalfield are updated. If the evalfield now has a 1 in every bit position that the waitmask does, n is ready to be awakened. This is accomplished by simply jumping to the code pointed to by n's value field. Otherwise, no action is taken.

Figure 8 shows the state of a node n before it is evaluated. The code for the serial combinator application represented by n will contain the synchronization constructs (spawn, demand, and wait) and will specify transformations to be applied to n. The list of variables in the demand, wait, and spawn constructs will have been translated into a list of indices i into n's args vector.

For each index i in a demand construct, if the ith bit of n's evalfield is 1 (i.e. the ith argument has already been evaluated) then no action is taken. Otherwise an evaltask is created to request the value of the node to which the ith argument points. As discussed above, the execution of n's code continues without blocking.

For each index i in a wait construct, the ith bit in n's waitmask is set to 1. n's value field is then modified so that it now points to a *continuation*—code that will be executed when the needed arguments return. When the required arguments return, execution proceeds via a jump to the continuation. Figure 9 illustrates a suspended node about to resume.

When a spawn is executed, a new subgraph is created to represent the activation record of each spawned expression. An evaltask is placed in the shared task queue for the root of each new subgraph. Ultimately, the result of the program is the value returned by the root node in the graph.

4 Queue-based Scheduling

Processor scheduling is accomplished by maintaining a central queue structure which every processor accesses. The simplest approach would be for every processor to remove tasks from the single shared queue. However, a shared queue causes contention between processors attempting to access the queue. This problem is exacerbated as the number of processors in the system grows. Unless the hardware supports efficient access to a central queue (as in the NYU Ultracomputer [6]), it is often necessary to modify the queue structure to prevent contention.

The solution we have implemented for Buckwheat is a *two-level* queue structure illustrated in figure 10. A processor can directly access a task queue, called a *primary queue*, that it shares with a small number of other processors. There may be many primary queues in the system. Each primary queue has a rather small fixed size. We define the set of processors accessing a single primary queue to be a *primary cluster*.

If a processor is ready to execute a task and its primary task queue is empty, it can access another queue, called the *secondary queue*, which is shared among all the processors in the system. Similarly, if a processor attempts to put a task onto is primary queue and its primary queue is full, then the task is put onto the secondary queue.

There are several advantages to the two-level queue structure:

1. Since a primary queue is shared by a relatively small number of processors, contention for the queue is reduced.
2. The secondary queue provides a way to send tasks from a busy primary cluster to other primary clusters. The cost of the extra indirection needed to access the secondary queue is only incurred by idle processors in idle primary clusters or when a primary cluster becomes very busy. If the size of the primary queue is chosen appropriately, the vast majority of queue accesses will be to primary queues.

5 Execution Results

Four applications programs were executed on Buckwheat to test the effectiveness of our approach. The four programs were `pfac`, a divide and conquer factorial, `queens`, a program to find all solutions to the 8-queens problem, `quad`, an adaptive quadrature algorithm, and `matmult`, a matrix multiplication program. Almost 600 runs were performed to measure the following:

1. The performance of Buckwheat using a single shared task queue.
2. The effect of using a two-level queue structure. The number of processors in a primary cluster as well as the sizes of the primary queues were varied in order to find the best task queue configuration.

Figures 11 through 14 plot the execution times (in microseconds) for the four programs as a function of the number of processors used.

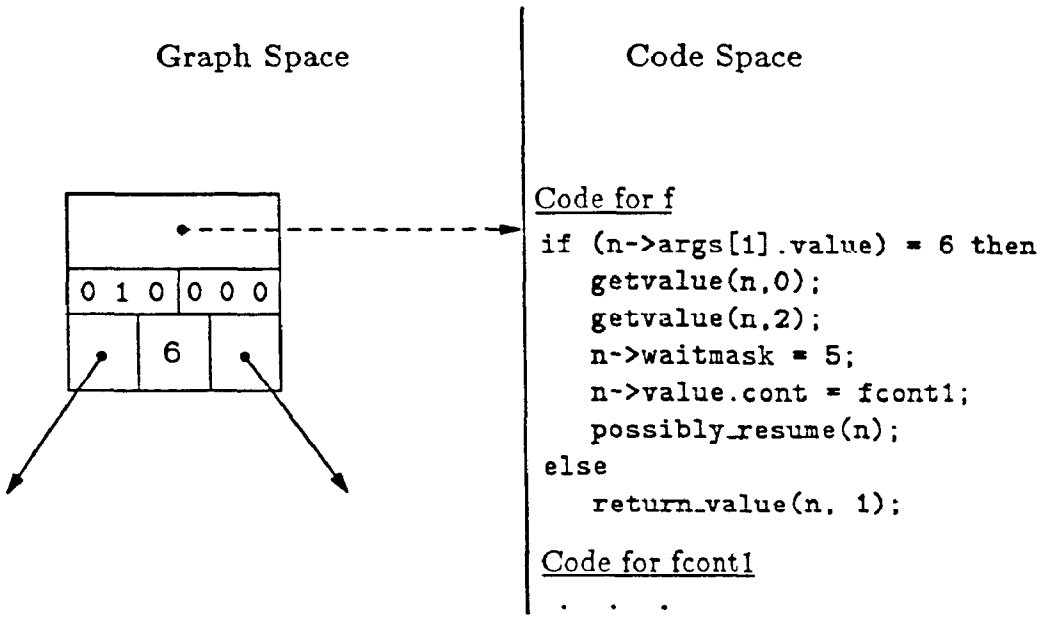

Figure 8: A node in its initial state

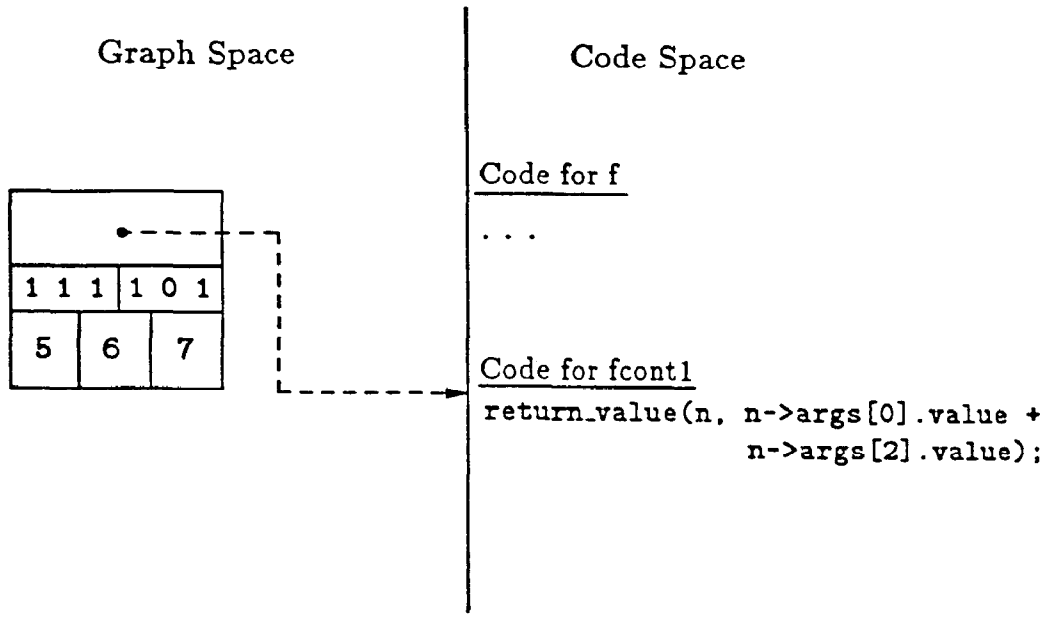

Figure 9: A suspended node about to resume

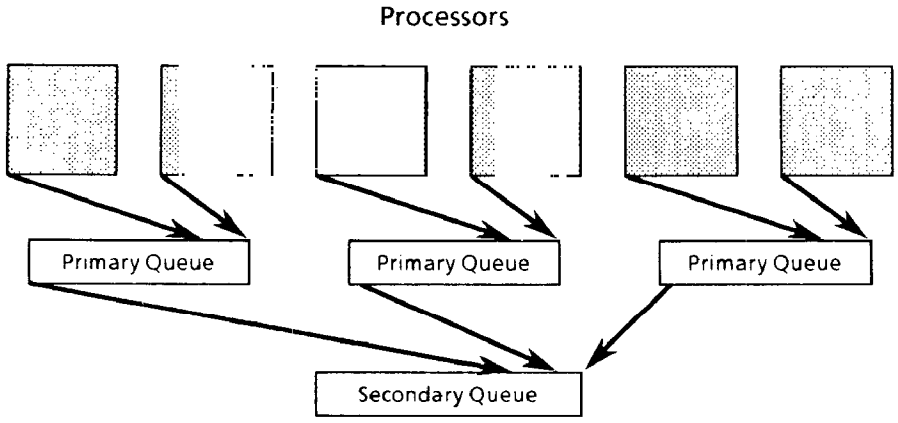

Figure 10: Buckwheat's two-level queue structure

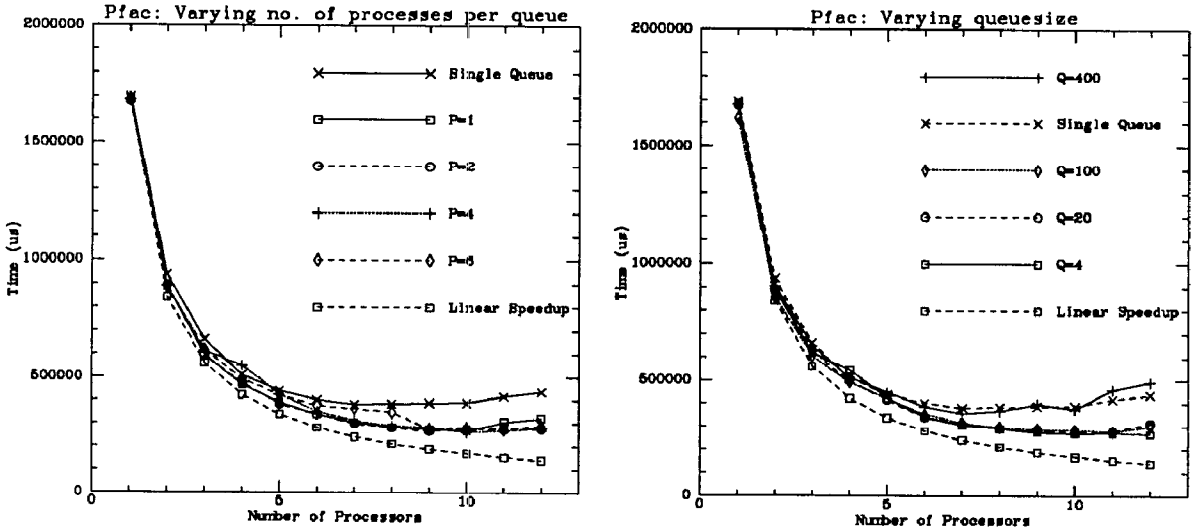

Figure 11: The execution times for pfac on Buckwheat

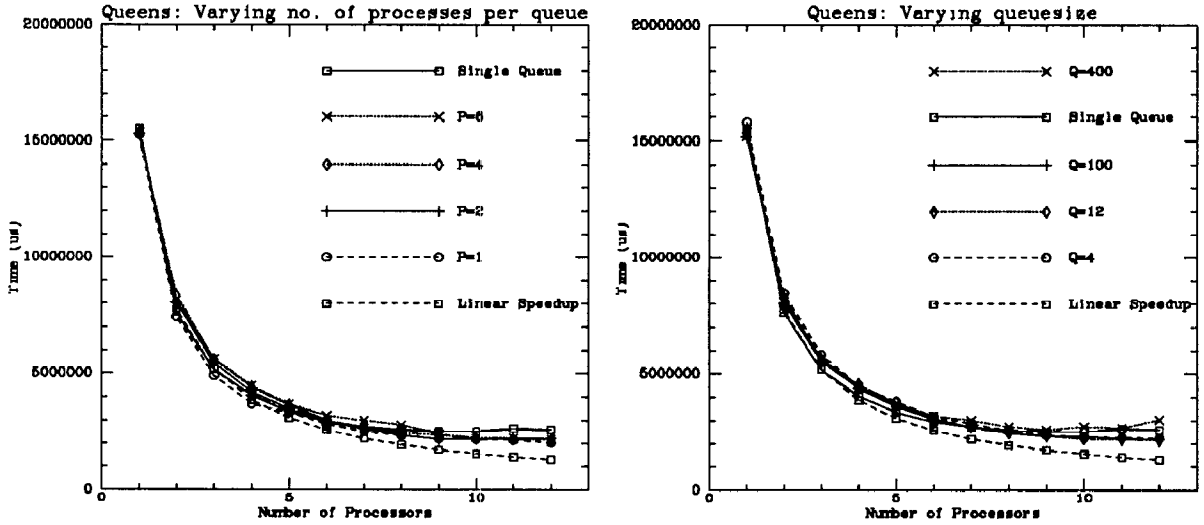

Figure 12: The execution times for queens on Buckwheat

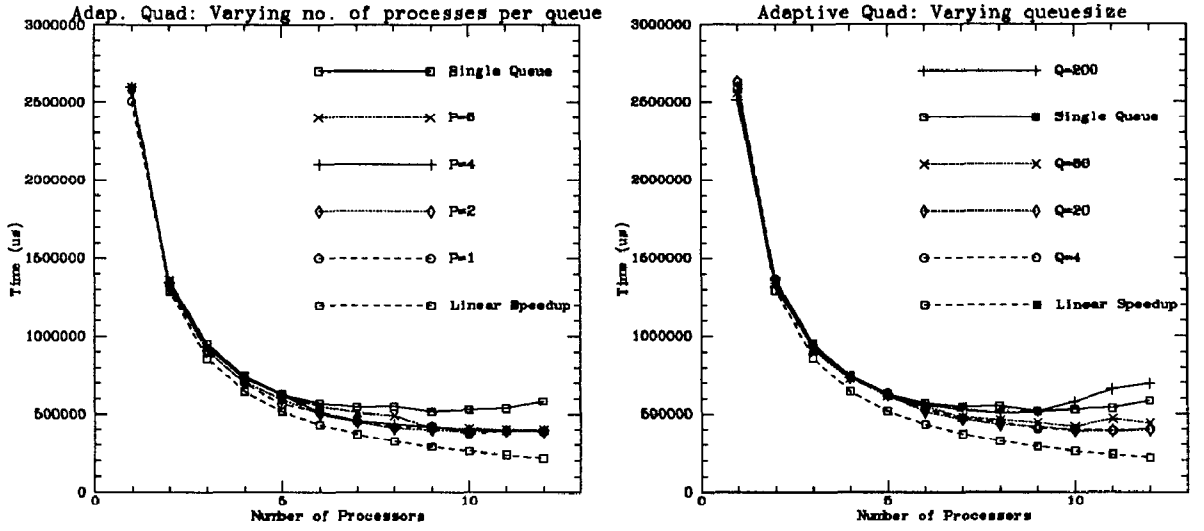

Figure 13: The execution times for quad on Buckwheat

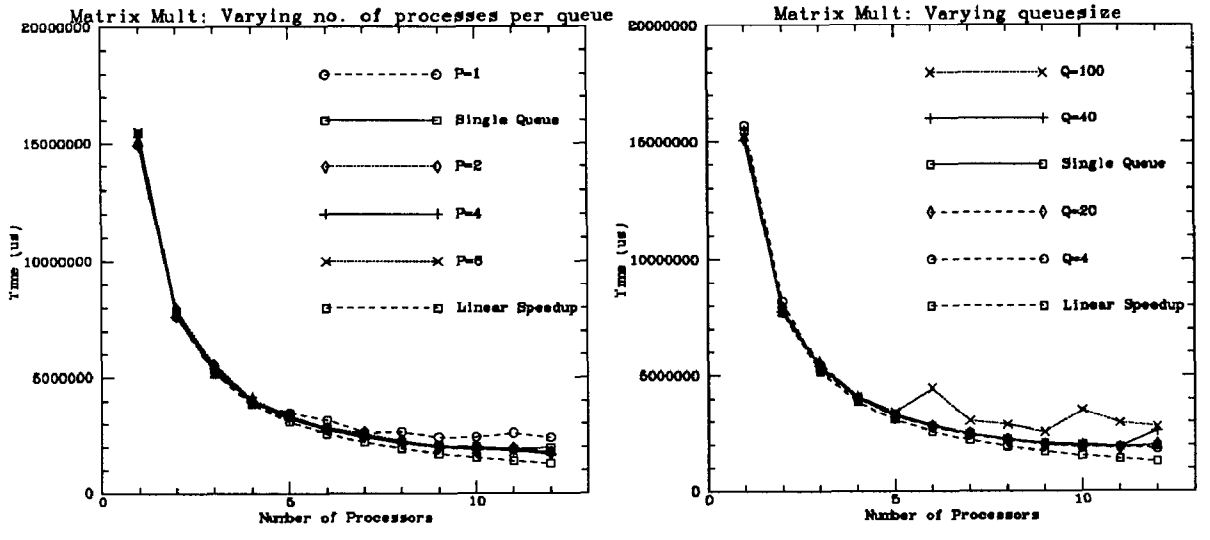

Figure 14: The execution times for `matmult` on Buckwheat

5.1 Finding the Appropriate Cluster Size

In each figure, the graph on the left plots the execution times using a single shared task queue. It also plots the execution times using a two-level queue structure for various values of P, the number of processors in a primary cluster. The size Q of each primary queue was fixed at 10 (tasks).

In every program the two-level queue structure performed better than a single queue. For small numbers of processors the difference was small, but as the number of processors grew the single queue caused the execution time to *increase*. This effect is due to contention for the access to the single queue. The two level queue structure significantly reduced the effect of contention for task queues.

With $Q = 10$ the two-level queue structure performed very well over the range of values for P. In `pfac` and `matmult`, the performance with $P = 1$ was poorer than for other values for P. With one processor per primary queue, parallelism can only be exploited by having tasks spill over onto the secondary queue. In three of the programs, the performance with $P = 1$ was still superior to the single queue case.

A P value of 4 provided the best performance over all the programs. Surprisingly, this proved to be program independent (although $P = 2$ performed just about as well). Having found an appropriate number of processors per cluster, it remained to find the best primary queue size for Buckwheat.

5.2 Finding the Appropriate Primary Queue Size

In each of figures 11 through 14, the graph on the right plots the execution times for various values of Q, the size of the primary queues. The number of processors P per cluster was fixed at four. In every case, a large value of Q performed poorly. When Q is large, fewer tasks will spill over to the secondary queue and the task distribution will be poor. The smaller Q values performed much better. The execution times for values of Q under twenty were very similar, although a Q value of 4 (the smallest value of Q we tested) performed the best. Again, the best Q value seemed to be program independent.

6 Conclusions

All the programs (with sufficient potential parallelism) that we tried performed extremely well. `Matmult` had the greatest reduction in execution time over the sequential case. Since the experiments were performed on a machine with only twelve processors, and since we have only examined speedup rather than absolute performance, we hesitate to draw conclusions about performance on massively parallel shared memory architectures. However, the results have reinforced our belief that functional language implementations on shared memory machines have the potential for providing a useful programming environment for parallel processing.

7 Acknowledgments

I would like to thank Paul Hudak for his contributions to this research. I would also like to thank those in the wrestling group at Yale, as well as the following people at Los Alamos National Laboratory: Joe Fasel, Randy Michelsen, and Bonnie Yantis.

References

[1] L. Augustsson. A compiler for Lazy ML. In *Proc. 1984 ACM Conf. on LISP and Functional Prog.*, pages 218–227, August 1984.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[3] *Multimax Technical Summary.* Encore Computer Corporation, Marlborough, MA, 1986.

[4] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, Department of Computer Science, May 1988.

[5] B. Goldberg and P. Hudak. Implementing functional programs on a hypercube multiprocessor. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, ACM, January 1988.

[6] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Trans. on Comp.*, C-32(2):175–189, February 1983.

[7] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[8] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In *Functional*

Programming Languages and Computer Architecture, pages 382–388, Springer-Verlag LNCS 201, September 1985.

[9] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proc. 12th Sym. on Prin. of Prog. Lang.*, pages 243–254, ACM, January 1986.

[10] T. Johnsson. *The G-machine: an abstract machine for graph reduction.* Technical Report, PMG, Dept. of Computer Science, Chalmers Univ. of Tech., February 1985.

[11] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16, Springer-Verlag LNCS 201, September 1985.

[12] D.A. Turner. The semantic elegance of applicative languages. In *Functional Programming Languages and Computer Architecture*, pages 85–92, ACM, 1981.

[13] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.