An Abstract Machine for Parallel Graph Reduction*

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

george@cs.utah.edu

Abstract

An abstract machine for parallel graph reduction on a shared memory multiprocessor is described. This is intended primarily for normal order (*lazy*) evaluation of functional programs. It is absolutely essential in such a design to adapt an efficient sequential model since during execution under limited resources available, performance will be reduced in the limit to that of the sequential engine. Parallel evaluation of normal order functional languages performed naively can result in poor overall performance despite the availability of sufficient processing elements and parallelism in the application. Needless context switching, task migration and continuation building may occur when a sequential thread of control would have sufficed. Furthermore, the compiler using static information cannot be fully aware of the availability of resources and their optimal utilization at any moment in run time. Indeed this may vary between runs which further aggravates the job of the compiler writer in generating optimal and compact code for programs. The benefits derived from this model are: 1) it is based on the G-machine so that execution under limited resources will default to a performance close to that of the G-machine; 2) the additional instructions needed to control the complexities of parallel evaluation are extremely simple, almost trivializing the job of the compiler writer; 3) attempts are made where possible to avoid context switching and task migration by retaining a sequential thread of control (made more clear in the paper), and 4) the method has demonstrated good overall performance on a shared memory multiprocessor.

*This work was supported by an IBM Academic Fellowship and in part by grant CCR-8704778 from the National Science Foundation.

1 Introduction

We define an abstract machine suitable for parallel graph reduction on a shared memory multiprocessor. This provides a level of abstraction constituting an important step towards building a compiler. The machine is intented for an ML[11]-like language with *compound datatypes* with lazily evaluated components. Our interest in lazy functional languages for multiprocessors is motivated by several reasons:

1. Awkward annotations for runtime synchronization of parallel activity are not required, since parallelism in lazy functional programs is implicit.

2. Due to the side effect free nature of functional languages, expressions can be executed in parallel without fear of having violated data dependencies.

3. Functional languages are highly amenable to static analysis and program transformation which are often non-trivial for other kinds of programming languages.

Since little is known about programming *general purpose* multiprocessors the benefits afforded by functional languages makes them a good starting point compared to other alternatives.

1.1 BBN Butterfly Multiprocessor

The investigation reported in this paper was performed on a *shared memory* MIMD multiprocessor, the delta connected BBN Butterfly[22]. Each processing unit (PU) of the Butterfly is a MC68020 microprocessor with a MC68882 floating coprocessor and a 68851 paged memory management unit. The microprocessor and coprocessor operate at their max speed of 16MHz. Each PU has 4 Mbytes of local memory and can access the full address space of 4 Gbytes. Associated with each PU is a process node controller (PNC) that services non local references from the host microprocessor to modules across the switch and likewise references from other processors. The PNC also implements various atomic operations. The local to remote access time is 5:1 with a local reference taking 1 microsecond. The ratio of 5:1 is ideal and could be much larger based on switch traffic and congestion.

The machine at our site has 18 processors of which 2 are reserved for system use. This investigation was performed under the Chrysalis operating system.

1.2 Implementation

The abstract machine to be described is mapped onto each node of the multiprocessor with shared memory used to implement the heap. The abstract machine consists of a number of processing elements (APE) that share common resources like the heap and task pool. In this implementation there is one APE per processor or in other words, there is only one process per processor and this implements the APE. For this reason we may use processor, processor unit/element (PE) or just process to mean an instance of an APE. The usage should be clear from the context. The design considerations that went into this work would be appropriate for other kinds of tightly coupled shared memory machines like the bus connected Sequent multiprocessor, but may not be appropriate for loosely coupled machines like the Intel iPSC.

Parallelism on this abstract machine is obtained primarily from:

1. Evaluation of strict arguments to a function. Such information can be derived from strictness analysis, user annotations or even the static structure of patterns in a function definition.
2. Evaluation of anticipatory work from the top level print function (Section 7).
3. User annotations surrounding program expressions.

1.3 Organization of Paper

The rest of the paper is organized as follows. In Section 2 we describe the basic components of our abstract machine. This is based on the G-machine; most of the major components are recognizable and used identically, although we must make some additions to account for the needs of parallel execution. Section 3 is largely devoted to the conventions we will use to describe the state of the machine and state transitions. In Section 4 we describe a function whose compilation we will consider throughout the rest of the paper. Providing a description of a complete compiler would needlessly distract us from the main issues and would not add significantly to the contents of this paper. Section 5 represents the core of the paper. The main instructions used to control parallel activity are, **set_wtcnt**, **demand** and **block**. We progressively refine their meaning from a naive definition and employ them in the compilation of the function described in Section 4. We use some other instructions along the way and may not offer a detailed explanation; these are all identical to those presented in the original G-machine paper[13], like **eval**, **hd**, **update**, **pushnil**, etc. Their meaning should be obvious from the context. Recently, there have been significant improvements to the basic G-machine model[3] that have been proposed, but in this exposition we adhere to the original G-machine as it is widely understood and acts as a base reference. The subsequent optimizations applied to the G-machine are not particularly germane to this discussion. In Section 6 we discuss issues related to a complete compiler and in particular the role of strictness analysis. Sections 7 through 11 discuss other issues related to performance on a parallel machine. We discuss related work in Section 13 and conclude in Section 14.

2 Abstract machine

The abstract machine is derived from Johnsson's G-machine[13], modified for parallel execution. A processing element of the abstract machine is described by the tuple `<<S, C, F, D>, G, T>` where:

- `S` = the evaluation stack containing pointers to heap nodes.
- `C` = code sequence being executed.
- `F` = a flag pertaining to the sequential thread of control (Section 5.3).
- `D` = sequence of return addresses and saved stack segments.
- `G` = heap space shared by all processors.
- `T` = task pool of *bounded* size implemented as a dual queue[1] also shared by all processors.

The nesting structure of the tuple is used to indicate that the heap `G` and the task queue `T` are shared among all PEs while the rest of the components are PE specific. The abstract machine tuple differs from that presented by Johnsson[13] in the following ways :

- The presence of the global task pool, `T`.

[1]A dual queue is a queue that can be in one of 3 states, i.e., *empty*, containing *process* descriptors or containing *tasks* to be evaluated. A dual queue containing process descriptors indicates that thoses processes are idle and an excess of processes over tasks. A dual queue containing tasks indicates that all processes are busy and here there is an excess of tasks over processes. The usual kinds of operations on queues are available such as **Enqueue**, **Dequeue**, **TryEnqueue**, etc.

- The presence of the per PE flag, **F**.
- The absence of the arithmetic stack. With certain optimizations described in Section 10 we have found this to be unnecessary.
- The absence of the output stream and environment which have been omitted for compactness.

Each PE executes an infinite loop where it repeatedly dequeues tasks from the task pool, T, and attempts a reduction. When the printing of the user top level expression has completed (assuming termination), a *termination* task is put into the dual queue for each PE. When its termination task is received each PE dumps out certain statistics and terminates gracefully.

3 Function Evaluation

A task[2] has several fields but for the purpose of PE state transitions as described in subsequent sections we shall only be concerned with the following:

- TAG: A tag value that may be either CLOS or BUSY.
- f: A code pointer.
- wc: A wait count for synchronization.
- nc: A notification chain which is a list of closures to be notified upon evaluation of this closure to normal form.
- env: A pointer to an environment or argument block.

A task having the tag CLOS indicates that it is unevaluated and the tag BUSY indicates that it is under evaluation. Associated with the task is a *lock* bit that is used to provide mutual exclusion when updating the notification chain or setting the tag field.

We use the following conventions when representing the state of the machine. The evaluation stack will normally be represented by $s_0..s_k..s_n$ where s_0, s_k and s_n are references to objects on the heap and s_0 is the bottom of the stack. A code sequence is represented as a list within [and], with the Prolog style of using | to represent the **cons** operation. A similar notation will be used for the task pool. We will use $\mapsto$ to dereference a pointer. Rather than displaying the entire heap, only the references of interest will be shown within { and }. This will usually be certain references from the evaluation stack. For compactness, if $S_0, S_1, ..S_n$ are states of a specific PE, and $\mathbf{cond_1}, ..\mathbf{cond_n}$, boolean expressions, then we will use the notation :

$$\begin{array}{lll} S_0 & \Rightarrow & S_1, \quad \mathbf{cond_1} \\ & & S_2, \quad \mathbf{cond_2} \\ & & \ldots \\ & & S_n, \quad \mathbf{cond_n} \end{array}$$

to represent a conditional state transition. **otherwise**, may be used as a catchall boolean condition.

As in the original G-machine, the S stack is used to *cache* the arguments of a closure and maintain an environment during execution of the closure. Prior to the execution of the code pointer associated with the closure, the argument block is *unwound* or copied onto the S stack. The state of the machine immediately after the unwind is shown in Figure 1. Note that the tag associated with the task is BUSY since it is currently under evaluation. A PE that picks a task off the task pool, will only attempt a reduction if the task is not being evaluated by any other processor, indicated by the tag being CLOS. In this case the tag is set to BUSY. The code sequence is $[f \mid C]$; f comes from the closure being evaluated and C represents the code forming the infinite top level processing loop.

We will frequently refer to the redex referenced by s_0 (Figure 1) as the *root* redex.

4 Language Compiler

The thrust of this paper involves the compilation of parallel activity, its control, synchronization and performance on a parallel machine all in the context of graph reduction and the G-machine. Much of the compiler would be similar to that described in the G-machine paper[13] which is to be expected since this model is based on it. Consequently, we do not provide a compilation strategy for a complete language or reasonable subset thereof; such a description would needlessly distract us from the main issues we want to present. For our purposes it will be sufficient to consider the compilation of just one function which we will use throughout the paper namely:

```
fun plus x y = x + y
```

where the + operator invokes the parallel evaluation of its arguments. Issues involved in a complete compiler are discussed in Section 6.

5 Demand/Block

The main instructions added to the G-machine related to parallel activity are **demand**, **block** and **set_wtcnt**. In order to motivate their final definition we progressively refine their meaning in the subsequent sections. The basic intuition is that **demand** should spawn the parallel evaluation of a task specified as an argument and

[2] In this paper we use **task** and **closure** interchangably. A closure is structure containing besides others a code pointer and a environment required by the former.

$$<<s_0 s_1..s_n, [f \mid C], F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ env \mapsto s1..s_n \end{array} \right\}, T>$$

Figure 1: Machine state after graph unwind.

block should implement the synchronization or wait for all the parallel activity to complete before proceeding.

In the next section we assume that there are an infinite number of processing elements and a task pool of unbounded size. This allows us to offer a spin waiting implementation for **block** for example. These assumptions are of course unrealistic, but we consider them so that the reader may get a feel for the nature of code generated since this does not really change as we make improvements.

In Section 5.2 we improve the definition of the **block** instruction so that the process executing this instruction *finds some other work* to do if all the parallel activity it is waiting for has not completed. A small change will be required in the code sequence we initially generated.

In Section 5.3 we improve both instructions to essentially retain a task in the sequential thread of control and take into account the reality of a limited number of processing elements and a task pool of bounded size.

5.1 Naive Definition

We forward the following naive definitions for **demand**, **block** and **set_wtcnt**, whose PE state transitions are shown in Figures 2 and 3.

demand(n) *Spawns* the parallel evaluation of the graph pointed by the reference on top of the stack assumed to be of height **n**. In the state transition of Figure 2, a reference to the *demanded* task and the parent is put onto the task pool so that the appropriate notification can take place. The notification is accomplished by the demanded task decrementing the wait count associated with the parent.

block(n,a,f) *Spin waits* for all parallel activity related to the root redex to complete before proceeding. This is equivalent to waiting for the wait count to fall to zero. The continuation is represented by the function **f** applied to **a** number of arguments whose references are on the top end of the evaluation stack assumed to be of height **n**. When the wait count falls to zero a tail recursion call is performed with the usual stack rearrangement and a jump to the continuation code (Figure 2).

set_wtcnt(w) Sets the wait count field of the root redex to **w**.

A possible compilation for the **plus** function (Section 4) using the above instructions and their definitions is shown below. As in the G-machine, the redex is at the bottom of the stack, and the arguments *unwound* onto the stack.

```
set_wtcnt(2);
push(1);
demand(3);
push(2);
demand(4);
block(4, 2, do_add);
```

do_add is a label to code that performs the update of the root redex under the *guarantee* that its arguments have been evaluated. **push(n)** is identical to that on the G-machine and copies the $\mathbf{n}^{\mathbf{th}}$ references in the evaluation stack to the top of the stack. The formal parameter **x** is at offset 2, and **y** at offset 1 after the unwind operation. This piece of code pushes these arguments on top of the stack using **push** and tries to invoke their parallel evaluation. The synchronization takes place at the **block** instruction that waits for them to complete. Given an *infinite* number of processing elements and a task pool that is *unbounded* in size, the definitions provided for **demand** and **block** are quite appropriate because, any task spawned on the dual queue would be picked up and a PE spin waiting is of no consequence although it could significantly increase the traffic on the communication medium due to a potential remote access to read the wait count field of the closure. These assumptions in practice are of course unrealistic and we relax them in a systematic manner in the subsequent sections.

5.2 Improvement 1

In this section we improve the **block** instruction by avoiding the spin wait. The PE that performs the block instruction could create the continuation closure and pick up another task if all the parallel activity it is waiting for has not been completed. Notification will now involve both decrementing the wait count and inserting the task into the task pool should its associated wait count fall to zero. Previously notification did not involve re-inserting the task into the task pool since the process executing the **block** instruction would be spin

$$<\!\!<s_0..s_n, [demand(n) \mid C], F, D\!\!>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_n \mapsto \alpha \end{array} \right\}, T > \Rightarrow$$
$$<\!\!<s_0..s_n, C, F, D\!\!>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_n \mapsto \alpha \end{array} \right\}, [[s_n, s_0] \mid T] >$$

$$<\!\!<s_0..s_n, [block(n,a,f) \mid C], F, D\!\!>, \{\ s_0 \mapsto BUSY\ g\ wt\ nc\ env\ \}, T > \Rightarrow$$
$$<\!\!<s_0..s_n, [block(n,a,f) \mid C], F, D\!\!>, \{\ s_0 \mapsto BUSY\ g\ wt\ nc\ env\ \}, T >,\ \texttt{wt} \neq 0$$
$$<\!\!<s_0 s_{n-a+1}..s_n, [f \mid C], F, D\!\!>, \{\ s_0 \mapsto BUSY\ g\ wt\ nc\ env\ \}, T >,\ \texttt{wt} = 0$$

Figure 2: Naive implementation of **demand** and **block**

$$<\!\!<s_0..s_n, [set\_wtcnt(w) \mid C], F, D\!\!>, \{\ s_0 \mapsto BUSY\ f\ wt\ nc\ env\ \}, T > \Rightarrow$$
$$<\!\!<s_0..s_n, C, F, D\!\!>, \{\ s_0 \mapsto BUSY\ f\ w\ nc\ env\ \}, T >$$

Figure 3: PE transition on **set_wtcnt** instruction

waiting and would eventually detect that the wait count had reached zero. With this change to **block**, in our running example above, there are really three processes that meet at the synchronization barrier; the two parallel tasks to evaluate **x** and **y** and the process generating the parallel activity which may need to modify the root redex with the continuation. Thus we need to modify the first line of our instruction sequence to be :

```
set_wtcnt(3);
```

In general for this reason, the wait count will have to be set to one more than the number of parallel tasks that are being created and spawned.

The state transition for this modification is shown in Figure 4. Notice the case when `wt = 1` all parallel activity that was generated has completed and a tail recursion call is implemented where a direct jump is made to the continuation with the appropriate arguments. The only process left at the synchronization barrier is that which is executing the **block** instruction. Note that in this case there is no need to update the root redex to reflect the fact that a specific tail call has been made.

The case when `(wt > 1)` corresponds to the situation where parallel activity that was generated has not completed. In an atomic operation implemented by locking, the root node is updated with the continuation and the wait count decremented. Any task that subsequently decrements this wait count to zero should enqueue it into the task pool.

This improvement now allows better utilization of the processor since there is no spin waiting while executing the **block** instruction.

5.3 Improvement 2

The naive definition of **demand** described in Section 5.1 and shown in Figure 2 has several drawbacks:

- If the argument is already evaluated to weak head normal form there would have been a needless insertion into the task pool and a synchronization step.
- In practice the task pool may have a finite size and the spawn or insertion may not always succeed. If

$$<<s_0..s_n, [block(n,a,f) \mid C], F, D>, \{\ s_0 \mapsto BUSY\ \ g\ \ wt\ \ nc\ \ env\ \}, T > \Rightarrow$$

$$<<s_0 s_{n-a+1}..s_n, [f \mid C], F, D>, \{\ s_0 \mapsto BUSY\ \ g\ \ 0\ \ nc\ \ env\ \}, T >, \texttt{ wt = 1}$$

$$<<empty, C, F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ \ f\ \ (wt-1)\ \ nc\ \ env_{new} \\ env_{new} \mapsto s_{n-a+1}..s_n \end{array} \right\}, T >, \texttt{ wt > 1}$$

Figure 4: Non spin waiting **block** instruction.

the spawn does fail we must guarantee that the task does eventually get evaluated.

- When there is only one task that can be generated for parallel evaluation, it is meaningless to spawn it off to another processor since one must incur the additional overhead of spawning and blocking. In such a situation it is better to retain the task in the sequential thread of control. Likewise when there are several parallel tasks that can be spawned it is meaningful to retain one of them in the sequential thread of control. By choosing the most expensive task to retain in the sequential thread one may decrease the likelihood of blocking in the synchronization step[9].

The occurrence of most of these conditions *cannot* be determined through static analysis at compile time, since they are runtime dependent. This means that the responsibility is on the design of the abstract machine and its instruction set to recognize these conditions and take the most appropriate action. The code generated for our running example remains exactly the same but we modify the definitions of **demand** and **block** to take the above into account. This actually means that the job of the compiler writer is *significantly simplified* since he may generate code under the assumption of infinite processing elements and a unbounded task pool. As we will show below, at saturation the execution of each processing element resorts to the execution model of the G-machine (or close to it) which is arguably the best known execution model for normal order languages.

Our approach will be to retain an unevaluated graph (one that is not in normal form) in the sequential thread of control. One possibility going back to our **plus** function would be to generate the following sequence of instructions (as a reminder, **x** is at offset 2 and **y** at offset 1):

```
set_wtcnt(2);
push(1);
demand(3);
push(2);
eval;
block(4, 2, do_add);
```

This spawns the parallel evaluation of **y** using **demand** and retains **x** in the sequential thread of control using **eval**. Hopefully, by the time the **block** instruction is executed, **y** would have been evaluated and the tail recursion call performed immediately. The problem with this is that **x** every time is *predestined* to be evaluated in the sequential thread of control which may not always be desirable. It may be the case that **x** is already in normal form in which case we would have needlessly spawned the parallel evaluation of **y** which instead should have been retained in the sequential thread of control.

Our solution is to introduce an additional state variable into the abstract machine called the **F** flag, which is a per-processor flag. This is reset at the beginning of the code sequence and records the reference to the first unevaluated graph encountered by the **demand** instruction. This will be made more clear after a detailed discussion of both **demand** and **block** below.

demand(k)

The state transition for the demand instruction is shown in Figure 5. Each conditional test in Figure 5 is explained in more detail:

i) If the object being demanded is already in normal form (determined by the **whnf** test) then the wait count associated with the root is decremented to indicate that the evaluation has already been performed. The Butterfly has an atomic decrement instruction which we use for this purpose.

ii) If the graph is busy as indicated by the presence of the BUSY tag, then a notification is set up to the root. This graph would be busy if it were shared and there was another processor that was evaluating it.

iii) If the graph is unevaluated and the flag `F` is reset (i.e., 0) then the reference is retained for the sequential thread of control. The information that the flag is set is utilized in the `block` instruction.

iv) If the graph is unevaluated and a task has been retained in the sequential thread of control then an attempt is made to insert the task onto the task pool. `TryEnqueue` is a boolean valued function that attempts to insert the task into the dual queue[3].

v If none of these conditions hold true then the default action is to perform the evaluation inline using the `eval` instruction. In the original G-machine paper[13] `eval` implied a machine state save on the dump and a restore upon completion. We have shown them explicitly, so our usage of `eval` here is not quite the same. The save involves pushing the stack 'top', the code sequence to be executed and the `F` flag onto the dump `D`.

block(n,a,f)

The state transition for the block instruction is shown in Figure 6. Each conditional test in Figure 6 is explained in more detail :

i) If the wait count has fallen to 1 then all parallel activity generated has completed and a tail recursion optimization can be performed. The function pointer is obtained from the instruction and the arguments must have been created on top of the stack. In this case nothing could have been retained for the sequential thread of control or the wait count would not be 1.

ii) If something was retained for the sequential evaluation indicated by the per-processor flag being set, then the equivalent of a G-machine `eval` is performed.

iii) Otherwise there is still pending parallel activity and we merely update the root redex with information pertaining to the continuation and decrement its wait count.

With these definitions of **demand** and **block**, in our running example it is immaterial which of `x` or `y` is in normal form. The appropriate task will be retained in the sequential thread of control. If neither `x` or `y` are in normal form then one will be spawned off and the other retained in the sequential thread of control. If the machine is saturated then the spawning will default to an inline evaluation by creating a new environment (*exactly what the G-machine would do in a sequential execution*). When the number of parallel tasks in the system greatly exceeds the number of processors, this inline evaluation is going to occur very frequently. *It is for this reason that the model must be based on the best sequential model of execution.* The trick towards efficiency in our model is to make the tests involved as cheap as possible. It is exactly this that adds to the cost of parallel evaluation.

[3]Since the task pool is a critical resource it needs to be locked before any operations can be performed on it. In an implementation that we developed if the lock could not be obtained on the first attempt then the function `TryEnqueue` returned `false` immediately making this a very inexpensive function.

6 Role of Strictness Analysis.

It is possible to develop a system based on what has been presented here, where the primary source of parallelism is from strict binary operators. The Butterfly implementation of SASL+LV[2] exploits exactly this using the Turner's S, K and I combinator technique. However, it is possible to get better performance and unravel more opportunities for parallelism by using strictness information[24]. For example, consider the `filter` function :

```
fun filter p [] = []
  | filter p (x::y) =
        if (x mod p) = 0 then filter p y
        else x :: (filter p y)
```

The strictness analyzer developed by Yeh[24] was able to determine that the `filter` function was strict on its second argument since there is an empty list test in the pattern match of the first *clause*, `fun filter p [] = ....` Should this test fail then it was able to determine that the first argument and the *head* of the second argument should both be *atomic* or integers since they are involved in a `mod` computation in the conditional statement. We adopted a style of compilation where the arguments were first raised to the level determined by the strictness analysis so that context switching in the body of the function to evaluate the first usage of an argument would be minimized[7,16]. This gave us good overall performance using a sequential implementation as compared to commercial implementations of LISP. We propose to adopt a similar style here. So for the above function we could generate the following code :

```
r_filter:  set_wtcnt(2);
           push(1);
           demand(3);
           block(2, 2, g_filter)
```

Since `filter` is strict on its second argument this piece of code *pre-evaluates* the second argument (recall that the second argument is at offset 1). Due to the nature of `demand` and `block` an attempt will be made to retain its execution in the sequential thread of control. *Note that we have used the instructions related to*

$$<\!<s_0..s_k, [demand(k) \mid C], F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \alpha \end{array} \right\}, T > \Rightarrow$$

$$\text{i)}\quad <\!<s_0..s_k, C, F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ (wc-1)\ nc\ env \\ s_k \mapsto \alpha \end{array} \right\}, T >\ ,\ \mathtt{whnf}(\alpha)$$

$$\text{ii)}\quad <\!<s_0..s_k, C, F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto BUSY\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, T >,\ \mathtt{s_k = BUSY\ f_k\ wc_k\ nc_k\ env_k}$$

$$\text{iii)}\quad <\!<s_0..s_k, C, k, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto CLOS\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, T >,\ \begin{array}{l} \alpha = \mathtt{CLOS\ f_k\ wc_k\ nc_k\ env_k} \\ \mathtt{F} = 0 \end{array}$$

$$\text{iv)}\quad <\!<s_0..s_k, C, F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto CLOS\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, T >,\ \begin{array}{l} \alpha = \mathtt{CLOS\ f_k\ wc_k\ nc_k\ env_k} \\ \mathtt{F} \neq 0 \\ \mathtt{TryEnqueue(s_k, T)} \end{array}$$

$$\text{v)}\quad <\!<s_0..s_k, \left[\begin{array}{l} save\_machine(); \\ eval(); \\ restore\_machine() \mid C \end{array} \right], F, D>, \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto CLOS\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, T >,\ \mathtt{otherwise}$$

Figure 5: Final definition of the **demand** instruction.

parallel execution since the execution could indeed block, particularly if the second argument is shared and is currently being evaluated by some other processor. In this situation the PE executing this piece of code will set up the continuation and pick up another task.

The code generated for the label **g_filter** would perform the pattern match with the *guarantee* that the arguments have been sufficiently evaluated. Should this pattern match fail then a similar set of operations could be performed to raise the arguments sufficiently for the second pattern matching step, i.e., `filter p (x::y) = if...`. The code may resemble :

```
g_filter:
    push(2);
    null;                         Conditional test
    jfalse L1;
    pushnil;
    update(3);                    Update with NIL
    ret(2);
L1:
    set_wtcnt(3);
    push(2);
    demand(3);          Pre-evaluate first argument,p
    push(1);
    hd(4);
    demand(4);  Pre-evaluate head of second argument
    block(2, 2, g_filter2);
```

Note the parallel activity generated after the label **L1**, which pre-evaluates the first argument using **demand(3)**, and the head of the second argument using **demand(4)**. This sequence of instructions handles all possibilities related to these components already in normal form and the state of the machine. The function **g_filter2**[4] can be similarly defined with the guarantee that the arguments have been sufficiently evaluated to the extent determined by the strictness analysis.

7 Top Level Print

The purpose of the top level print function is to perform output of the top level expression being evaluated. Further, parallelism can be obtained by spawning *if possible* tasks to evaluate the remaining components of the expression, *only if* they are in unevaluated form. We call this work *anticipatory* work since its need is anticipated. *Mandatory* work on the other hand is related to the current print object. In the terminology used by Burn[4], this is equivalent to evaluating the top level expression

[4]The prefix **r_** and **g_** to function names are no accident. They were initially used to convey *red* and *green*. Red saying, "STOP don't go ahead but pre-evaluate certain arguments" and green saying, "All pre-evaluation done, shoot into the body of the function".

$$<<s_0..s_n, [block(n,a,f) \mid C], F, D>, \{\ s_0 \mapsto BUSY\ \ g\ \ wc\ \ nc\ \ env\ \}, T > \ \Rightarrow$$

$$\text{i)}\quad <<s_0 s_{n-a+1}..s_n, f \mid C, F, D>, \{\ s_0 \mapsto BUSY\ \ g\ \ wc\ \ nc\ \ env\ \}, T >, \ \texttt{wt = 1}$$

$$\text{ii)}\quad <<s_0..s_n, \left[\begin{array}{l} push(k); \\ save\_machine(); \\ eval(); \\ restore_machine(); \\ block(n,a,f) \mid C \end{array}\right], 0, D>, \{\ s_0 \mapsto BUSY\ \ g\ \ wc\ \ nc\ \ env\ \}, T >, \ \mathbf{F = k \neq 0}$$

$$\text{iii)}\quad <<empty, C, F, D>, \left\{\begin{array}{l} s_0 \mapsto BUSY\ \ f\ \ (wc-1)\ \ nc\ \ env_{new} \\ env_{new} \mapsto s_{n-a+1}..s_n \end{array}\right\}, T >, \ \texttt{otherwise}$$

Figure 6: Final definition of the **block** instruction

using a ξ_3 *evaluator*, which evaluates the structure of a list and every element of the list to head normal form. A major problem occurs when this evaluator gets applied to a graph more than once. Each evaluator will be running down the list structure looking for unevaluated graphs to spawn, which is wasteful. If the evaluator is represented as the application of a built in function then this traversal could involve a substantial amount of graph building as it recursively descends the **cons** structure.

A solution to this problem would be to have a flag in the list structure indicating that it has been previously traversed by an ξ_3 evaluator thus avoiding further propagation. We adopted a slightly different strategy by adding an extra bit called the **exhaustiveness** bit to the closure representation. If the closure is updated as a **cons** structure with this bit set, then this bit is reset and the bit propagated to unevaluated components of the **cons** structure. If *possible* these unevaluated components are inserted into the task pool. The main drawback with this approach is ensuring that the **exhaustiveness** bit gets propagated to the unevaluated components since this appears to be quite expensive and not very easy to implement.

The parallelism from the top level print is not always sufficient or useful. For example the sieve program (Section 16.1) with all **cons** operators being lazy exhibits sequential performance with this form of parallelism. The reason for this is that when the top level expression gets reduced to normal form the *head* component is already in normal form and is printed immediately. The spawning of the *tail* part has little consequence since it will be demanded by the top level print almost immediately. The parallelism from the top level print is useful when the *head* computation involves a substantial amount of work. In this situation, while the head is being evaluated and printed, a substantial portion of the tail may be evaluated in parallel. On our 18 node Butterfly we ran the program of Section 16.2 with the parallelism from the top level print and without it. The performance was almost the same indicating that the machine was probably being swamped by the parallelism from evaluating the *head* of the stream itself.

Our intuition at this point is to omit the parallelism from the top level print in future implementations due to its complexity and overhead in book keeping and concentrate more on the parallelism from strictness analysis (Section 6) and insights provided by user annotations.

8 Memory Management

Our heap is a shared memory space where each abstract machine has a segment of the total heap, that is typically local to the memory of its associated physical processor. Accesses to the local segment of the heap is *usually* faster than accesses to nonlocal segments. Each abstract machine makes allocations out of its local heap segment and when exhausted will try to allocate from a remote heap segment. This means that the heap allocation routine must be a critical section. Since functional programs tend to be memory intensive this is a bottleneck as locking would be required for every allocation. The heap allocation can be optimized by locally managing a sufficiently large buffer space allocated out of the heap. Test programs showed an improvement of between 7-17% with a local buffer space of 1K bytes.

It was also observed that the local memory usage among the processors was very even for the benchmarks tried. This strongly favors a complete stop and collect style of garbage collection since, when one processor has exhausted its memory most of the other processors will be in a similar situation. Our implementation does not presently contain a garbage collector.

In our state transition description we have abstracted away the locking that is involved at various stages. However, the locking issue does impact the design of the node structures used. Consider two processors competing for the lock on a node, one to update the node to normal form and the other to attach a notification marker. Assuming the processor that is performing the update acquires the lock first then upon release of the lock the node is no longer *busy* or in unevaluated form. The second processor would then lock a node in normal form. For this reasons the lock bit must be in the same position for both the closure/task and its normal form.

9 Scheduling

Since there is just a single task pool in the abstract machine, scheduling is trivial. Tasks are always removed from the front of the queue but may be inserted at both ends. Our policy is to put all parallelism generated as a consequence of advisory information or the top level print at the back of the queue and mandatory work from **demand** in the front. This has worked out to be marginally better than a FIFO or LIFO scheduling policy.

10 Mixed Evaluation Stack

The Butterfly is a byte addressable, 32 bit word machine. If all nodes are word aligned then all pointers to nodes will have zeros in the lower most two bits. These lower two bits can be used as a tag to distinguish between between nodes and pointers. There are two kinds of node structures: 1) *unboxed* objects that can fit inside a word used to represent constants and nullary constructors and 2) *boxed* objects that are multiple words used to represent closures and constructors of arity greater than zero[17]. In boxed objects the first word is used as a descriptor indicating the length, type, etc. The C language declarations are shown below :

```
struct unboxed {
    unsigned tag : 27;
    unsigned int_flag : 1;
    unsigned exhaustive : 1;
    unsigned lock : 1;
    unsigned id : 2;
};

struct Desc {
    unsigned tag : 16;
    unsigned length : 12;
    unsigned exhaustive : 1;
    unsigned lock : 1;
    unsigned id : 2;
};

struct boxed {
    struct Desc desc;
    WORD comp[0];
};
```

Note that the **lock** and **exhaustiveness** bits are in the same position for both boxed and unboxed objects, a requirement we discussed in Section 8. The field **id** (which happen to be the lowest two bits of the word) is used to distinguish between pointers, boxed and unboxed objects. The **int_flag** in the **unboxed** declaration is used to distinguish between integers and nullary constructors, where 27 bits are used to store the integer value. With this design we can use the evaluation stack to hold unboxed objects *as well as* pointers to heap nodes. In the G-machine the evaluation stack (**S** stack) contained exclusively pointers to heap nodes. The advantages of having data in the evaluation stack are:

- Computations can be performed using the evaluation stack to store temporary results. This was achieved by using a special stack (the arithmetic stack) in the G-machine.
- The most important benefit is that when closures are created the graph need not have pointers to unboxed objects in the heap but can store the unboxed objects themselves *directly* in the graph. This further means that when an **unwind** takes place the unboxed objects are unwound directly onto the evaluation stack and an initial reference need not go to the heap which may involve a non local access.
- A consequence of the fact that unboxed objects are stored directly in graphs rather than storing pointers to them, results in fewer requirements on the heap space.

A drawback is a small amount of additional complexity in the implementation but this was not sufficient to affect the performance which was improved over the "pointer evaluation stack" case. In Figures 8 and 7 we show the effect of a mixed evaluation stack on the 8 queens and sieve programs respectively.

11 Two level scheduling

To avoid contention on the centralized task pool, a small local task pool can be maintained. Thus excess work spills over from the local task pool to the centralized task pool. Search for work begins at the local task pool and ends at the centralized task pool. We have found in the examples tried that a large local task pool (i.e., greater than 2) degrades the performance.

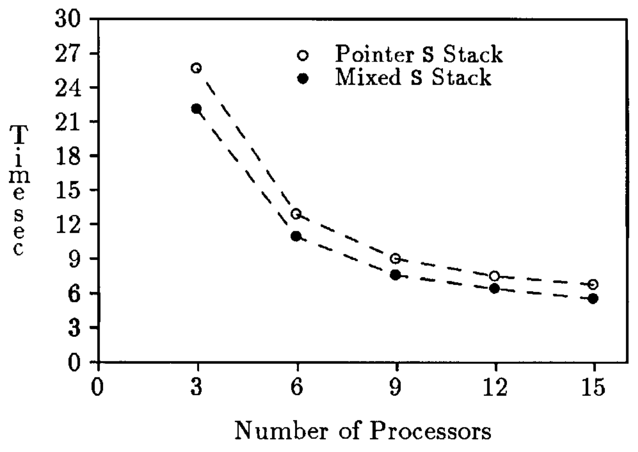

Figure 7: Sieve Program with Data Caching

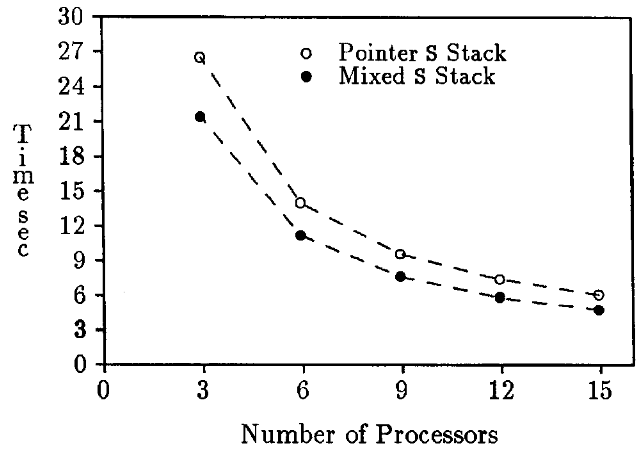

Figure 8: 8 Queens Program with Data Caching

Figure 9 and 11 shows the results obtained. In all these experiments there was one processor per *local* task pool. Goldberg reported better results with a small number of processors sharing a local task pool[9]. These experiments show that going overboard in trying to retain locality is not allows a good idea (particularly in the *fibstrm* program).

12 Performance

Figures 9 through 12 shows the best performance of our machine. Also shown is the performance of the Standard ML of New Jersey (NJML) compiler running on the VAX 8600, a machine *at least* 6 times faster than the BBN Butterfly. The NJML compiler (version 0.33) is the latest from AT&T and is highly optimized. The figure of 6 times, was obtained by taking the Dhrystone benchmark, and allocating all global variables on another processor. This was so that the switch and message overhead typical in programs would come into the picture. The Dhrystone benchmark is reported to have 9% of its accesses to global structures. We hesitate to show speedup curves based on our sequential performance. Since our sequential performance is not blazingly fast, we would obtain good (though misleading) speedup curves. Ideally, we should obtain our speed up curve based on the best sequential system running on one node of the Butterfly. This would have to be either Johnsson's LML system or even the New Jersey ML system. Unfortunately the latter is not as portable as we would have liked and at the time of writing we do not have access to the former.

13 Related Work

13.1 Architectures

There has been considerable effort in exploiting the benefits of compiling to a fixed set of combinators such as the S,K,I set[23]. These includes projects such as COBWEB[10], SKIM I and SKIM II[5,21], and NORMA[20] although the NORMA architecture could be microcoded to execute a different set of combinators. Our approach is to use commercially available shared memory multiprocessors and to follows the lead of the G-machine by having every user function represent a combinator. Our motivation for this was the need to adapt an efficient sequential model for reasons stressed throughout this paper. Other projects such as ALICE[6] and GRIP[19,18] use compilation techniques motivated by the G-machine but make use of specialized hardware to support graph reduction and garbage collection.

13.2 Buckwheat

Goldberg[9,8] describes a system for graph reduction on a shared memory machine called Buckwheat. The source language ALFL developed at Yale is transformed into a new set of combinators called *serial combinators*[12]. These serial combinators contain specific constructs that specify the synchronization for parallel execution. The basic synchronization constructs are *demand*, *wait* and *spawn* which are expressed at a fairly high level. For example, in a serial combinator, the *demand* construct:

$$\texttt{(demand (}v_1 \;..\; v_n\texttt{)}\ \textit{body}\texttt{)}$$

indicates that the variables v_1 .. v_n may be safely evaluated in parallel and need not return before ***body*** is evaluated. The synchronization constructs are simple calls to routines in the Buckwheat graph reducer module.

The implementation details of these routines are not readily available in the literature, but it seems very likely that the ideas introduced in this paper (with maybe certain minor enhancements) could be used to implement each of these constructs. An interesting prospect would be to compile the serial combinator code into the instruction set of the abstract machine defined in this paper.

13.3 $<\nu, G>$ Machine

Augustsson and Johnsson[14] describe an abstract machine called the $<\nu, G>$ machine which is a shared memory multiprocessor abstract machine for normal order evaluation of functional programs. Its characteristic feature besides being a simple formalization is that the evaluation stack is *contained* in the graph structure and consequently resides in the heap space. This raises several problems such as allocating a sufficient stack space to be associated with the graph.

A more difficult problem would be obtaining good performance on a machine like the Butterfly which does not have a data cache associated with each processor and for which a fairly high penality is paid for in accessing references across the switch. In this respect our adaptation of G-machine is better suited to the Butterfly because the per-process *private* evaluation stack, acts as a local cache for arguments once they have been unwound and for temporaries created during the reduction.

13.4 Evaluation Transformers

Burn[4] describes a model where it is possible through abstract interpretation to determine the amount of evaluation a particular argument could be subject to, give the *context* in which it is being evaluated. Four *evaluators* are discussed $\xi_{0..3}$; for example, ξ_1 evaluates an expression to head normal form. In the model proposed by Burn, each function is compiled so that it first executes a **switch** statement. The switch statement applies the appropriate evaluators to the existing arguments based on the context in which the function is being evaluated. The function need not wait for arguments to complete.

Our **demand** instruction actually implements the ξ_2 evaluator and we attempted to implement the ξ_3 evaluator when evaluating the top level expression in the program. The work by Burn is indeed more general, however, there are several points worth mentioning:

- We have found that in practise the ξ_3 evaluator is not trivial to implement and *could* incur a significant overhead. The approach we investigated involved using an *exhaustiveness* bit. In many of our programs we found that this evaluator applied to the top level print function was not very useful (Section 7), since either the head computation was so small that the ξ_3 was not able to progress far enough, or the head computation was so large that it swamped the machine thus inhibiting the ξ_3 evaluator.

- We compile functions so that the evaluation of parallel arguments is completed before the body is executed. This will certainly prevent blocking or a context switch to evaluate strict arguments.

- In our discussion of strictness analysis (Section 6) it seems conceivable that the idea of the **switch** statement to account of the context under which the evaluation is taking place, could be incorporated.

14 Conclusions

In this paper we have discussed possible extensions to the G-machine that account for the needs of parallel evaluation. While we have discussed this in the context of the G-machine[13] there is no reason why a newer and improved model of the G-machine cannot be incorporated, like the Spineless G-machine[3]. Our suggested method of compilation consists of two phases :

1. Spawning the parallel evaluation of strict arguments.

2. Waiting for the parallel activity above to complete before entering the body of the function being evaluated.

This paper largely deals with 1 above. The code for 2 could be substituted by ones favorite G-machine variant.

We feel that the added instructions to the G-machine significantly simplifies the job of the compiler writer in generating compact code for a parallel machine and that our suggested compilation technique is easy enough, to develop a simple and robust compiler. The instructions introduced avoid context switching and needless task migration by retaining a sequential thread of control and default to an inline sequential execution of the base model under saturation. The model on our 18 node Butterfly demonstrated good performance.

14.1 Future Directions

The work described here is part of an ongoing research project at Utah to develop a tightly coupled multi-paradigm language. This is essentially a functional language augmented with *logic variables*. While it embraces concepts such as constraint satisfaction, monotonic data types and bidirectional communication using logic variables (found also in *Id Nouveau*[1]), it also embraces the notions of committed choice non-determinism and guarded evaluation which we have found to be very useful in certain examples[2]. Using static analysis of programs with logic variables[15] we hope to be able to generate large grain combinator code (the results from this work) that can be executed efficiently on a multiprocessor machine. We are presently doing a port of our implementation to MACH on the Butterfly??.

15 Acknowledgments

I would like to gratefully acknowledge the comments and suggestions provided by Simon Peyton Jones and Gary Lindstrom which I found very useful while preparing the final version. If I were to fully acknowledge there comments and suggestions throughout the paper, I would need to add a footnote to almost every page. I would also like to thank Peter Kogge for taking an outside interest in the development and future directions of this work. None of this work would have been possible without the support of the excellent systems staff at the University of Utah, in particular Mark Swanson who patiently answered my questions on the Butterfly.

References

[1] Arvind, Nikhil, R. S., and Pingali, K. K. I-structures: data structures for parallel computing. In *Graph Reduction: Proceedings of a Workshop*, J. H. Fasel and R. M. Keller, Eds., Springer-Verlag, 1987, pp. 336–369. Lecture Notes in Computer Science No. 279.

[2] Båge, G., and Lindstrom, G. Committed choice functional programming. In *Proc. International Conference on Fifth Generation Computer Systems* (Tokyo, November 1988), Institute for New Generation Computer Technology (ICOT), pp. 666–674.

[3] Burn, G., Peyton Jones, S. L., and Robson, J. D. The spineless g-machine. In *Proc. of the 1988 ACM Conf. on Lisp and Functional Programming* (July 1988), ACM, pp. 244–258. Conf. at Snowbird, Utah.

[4] Burn, G. L. *Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages. LNCS 274*, Springer-Verlag, 1987, pp. 446–470. Portland Oregon Proc., Ed. Gilles Kahn.

[5] Clarke, T., Gladstone, P., and MacLean, N. A. SKIM - the s, k, i reduction machine. In *The 1980 LISP Conference* (August 1980), pp. 128–135. Stanford University.

[6] Darlington, J., and Reve, M. ALICE: a multiprocessor reduction machine for the parallel evaluation of applicative languages. In *Functional Programming Languages and Computer Architecture* (October 1981), ACM, pp. 65–76.

[7] George, L. *Efficient Normal Order Evaluation Through Strictness Information.* Master's thesis, University of Utah, March 1987.

[8] Goldberg, B. E. Buckwheat: graph reduction on a shared memory multiprocessor. In *Proc. of the 1988 ACM Conference on Lisp and Functional Programming* (July 1988), ACM, pp. 40–51. Snowbird, Utah.

[9] Goldberg, B. E. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, Dept. of Computer Science, April 1988. YALEU/DCS/RR-618.

[10] Hankin, C. L., Osmon, P. E., and Shute, M. J. *COBWEB: A Combinator Reduction Architecture. LNCS 201*, Springer-Verlag, 1985, pp. 99–112.

[11] Harper, R. Introduction to Standard ML. LFCS Report Series ECS-LFCS-86-14, Computer Science Department, Edinburgh University, November 1986.

[12] Hudak, P., and Goldberg, B. E. *Serial Combinators "Optimal" Grains of Parallelism. LNCS 201*, Springer-Verlag, 1985, pp. 382–388.

[13] Johnsson, T. Efficient compilation of lazy evaluation. In *Proc. Symp. on Compiler Construction* (Montreal, 1984), ACM SIGPLAN.

[14] Johnsson, T. The $< \nu, g >$-machine: an abstract machine for parallel graph reduction. In *Proc. WG 10.1 Workshop on Concepts and Characteristics*

of Declarative Systems (Budapest, October 1988), G. David, Ed., IFIP. Also appears in this proceedings.

[15] Lindstrom, G. Static analysis of functional programs with logical variables. In *Proc. International Workshop on Programming Language Implementation and Logic Programming (PLILP '88)*, Springer Lecture Notes in Computer Science Number 348, Orleans, France, 1989, pp. 1–19.

[16] Lindstrom, G., George, L., and Yeh, D. Generating efficient code from strictness annotations. In *Proc.Second Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)* (March 1987).

[17] Peyton Jones, S. L. *The Implementation of Functional Programming Languages. International Series In Computer Science*, Prentice-Hall, 1987.

[18] Peyton Jones, S. L. Parallel implementations of functional programming languages. *The Computer Journal 32*, 2 (1989), 175–186.

[19] Peyton Jones, S. L., Clack, C., Salkild, J., and Hardie, M. *GRIP - a high performance architecture for parallel graph reduction. LNCS 274*, Springer-Verlag, 1987, pp. 98–112.

[20] Scheevel, M. NORMA: a graph reduction processor. In *Proc. of the 1986 ACM Conference on Lisp and Functional Programming* (August 1986), ACM, pp. 212–219.

[21] Stoye, W. R., Clarke, J. W., and Norman, A. C. Some practical methods for rapid combinator reduction. In *Proc. of the ACM Conference on LISP and Functional Programming* (August 1984), ACM, pp. 159–166.

[22] Thomas, B., Gurwitz, B., Goodhue, J., and Allen, D. Butterfly parallel processor: overview. Tech. Rep. 6148, BBN Laboratories Incorporated, March 1986.

[23] Turner, D. A. A new implementation technique for applicative languages. *Software Practice and Experience 9* (1979), 31–49.

[24] Yeh, D. *Static Evaluation of a Functional Language Through Strictness Analysis.* Master's thesis, University of Utah, December 1987.

16 Appendix: Test Programs

We surround expressions with ^ to indicate that should resources exist then they should be spawned onto the task pool. Unless otherwise stated all the **cons** operators (`::`) are strict which means that the result is constructed with possibly unevaluated components but should resources exists the unevaluated components are spawned onto the task pool. Thus (e1 :: e2) is really (^e1^ :: ^e2^).

16.1 Sieve

```
fun from n m =
        if (n>m) then nil else n :: from (n+1) m;
fun filter p [] = []
  | filter p (x::y) =
        if (x mod p = 0) then filter p y
        else x :: filter p y;
fun sieve [] = []
  | sieve (x::y) = x :: sieve (filter x y);

sieve (from 2 2000);
```

16.2 Fibonacci Stream

This is obviously not the best way to compute fibonnacci numbers or a stream of them. The intent was just to generate a lot of parallelism.

```
fun fib n = if n < 2 then 1
            else fib(n-1) + fib(n-2);
fun from n m = if (n > m) then nil
               else n :: from (n+1) m;
fun fibstrm [] = []
  | fibstrm (x::y) = fib x :: fibstrm y;

fibstrm (from 1 20);
```

16.3 Tak Function

```
fun tak x y z = if (y >= x) then z
                else tak (tak (x-1) y z)
                         (tak (y-1) z x)
                         ^(tak (z-1) x y)^;

tak 18 12 6;
```

16.4 8 Queens

```
fun append [] x = x
  | append (x::xs) ys = x :: append xs ys;

fun queens n = queensoln n n

and
    queensoln 0 _ = nil :: nil
  | queensoln n size =
      add_columns n size
    (queensoln (n - 1) size)
```

```
and
    add_columns _ _ nil = nil
  | add_columns x n (board :: boards) =
        append (add_column x n board)
      ^(add_columns x n boards)^

and
    add_column x 0 board = nil
  | add_column x y board =
        if (attacks x y (x-1) board) then
           add_column x (y-1) board
        else (y :: board) ::
             (add_column x (y - 1) board)

and
    attacks x y x1 nil = false
  | attacks x y 0 board = false
  | attacks x y x1 (y1 :: board) =
        attacks2 x  y x1 y1 board

and
    attacks2 x1 y1 x2 y2 rest =
        y1 = y2 orelse
        abs (y2 - y1) = abs (x2 - x1) orelse
        (attacks x1 y1 (x2-1) rest);
```

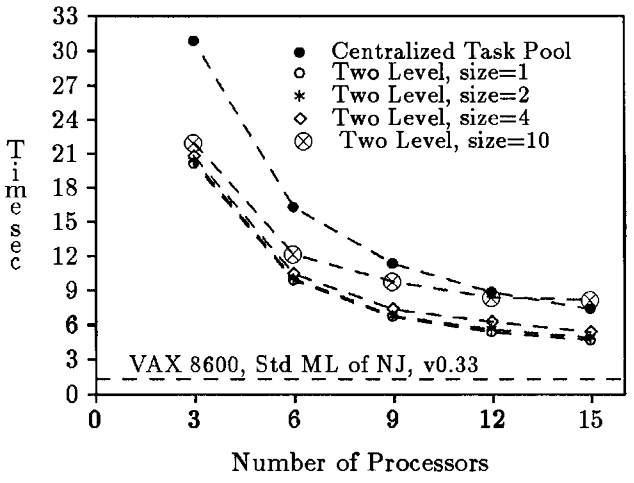

Figure 9: Two Level Scheduling and the Sieve Program

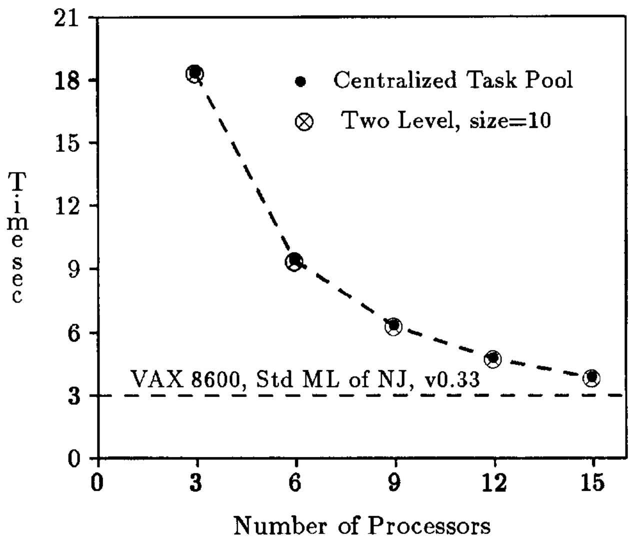

Figure 11: Two Level Scheduling and the 8 Queens Program

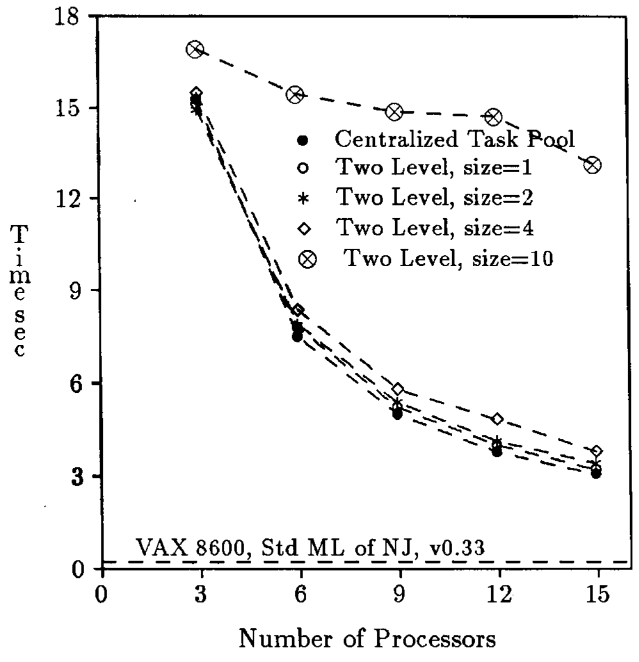

Figure 10: Two Level Scheduling and the Fibstrm Program

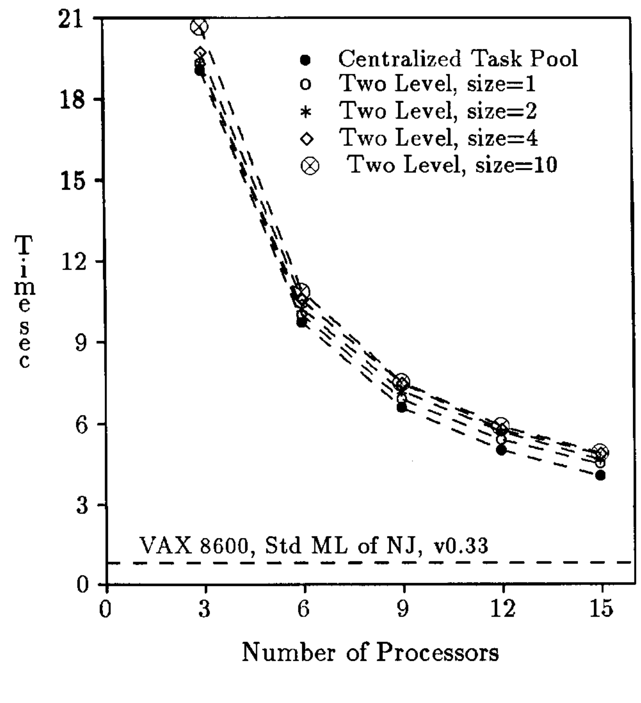

Figure 12: Two Level Scheduling and the Tak Program