Fast Parallel Implementation of Lazy Languages – The EQUALS Experience*

O. Kaser   S. Pawagi   C.R. Ramakrishnan   I.V. Ramakrishnan

Department of Computer Science
SUNY at Stony Brook, NY 11794.
{owen, shaunak, cram, ram}@cs.sunysb.edu

R.C. Sekar

Bellcore, 445 South Street
Morristown, NJ 07962.
sekar@thumper.bellcore.com

Abstract

This paper describes EQUALS, a fast parallel implementation of a lazy functional language on a commercially available shared-memory parallel machine, the Sequent Symmetry. In contrast to previous implementations, we detect parallelism automatically by propagating exhaustive (normal form) demand. Another important difference between EQUALS and previous implementations is the use of reference counting for memory management instead of garbage collection. Our implementation shows that reference counting leads to very good scalability, low memory requirements and improved locality. We compare our results with sequential SML/NJ as well as parallel $\langle \nu, G \rangle$-machine and GAML implementations.

1   Introduction

It is well known that functional languages offer a conceptually simple vehicle for programming parallel computers. The main reason for this is that expressions may be evaluated in any order, due to absence of side-effects. Therefore detection as well as exploitation of parallelism is much simpler than in imperative languages. This has been exploited in many previous parallel implementations such as ALICE [Dar81], FLAGSHIP [WW87], GRIP [PJ87], Buckwheat [Gol88a], $\langle \nu, G \rangle$-machine [Aug89] and GAML [Mar91]. Whereas ALICE, FLAGSHIP and GRIP make use of specialized hardware, the other three implementations are based on commercially available shared-memory multiprocessors. In this paper, we focus on the latter approach and describe EQUALS, a fast parallel implementation of a lazy functional language[1] on Sequent Symmetry.

One of the earliest lazy parallel implementation on shared-memory multiprocessor was Buckwheat. It demonstrated the feasibility of parallel implementation, but was not tuned for performance. On the other hand, $\langle \nu, G \rangle$-machine and GAML showed performance improvement over sequential implementations such as LML [Aug84], starting from two processors. Both these implementations were able to reduce the parallel overheads, and consequently their performance continued to improve even when the number of processors was increased to ten or more.

However, these implementations do not satisfactorily exploit one of the primary advantages of functional languages for parallel evaluation, namely, automatic detection of parallelism. For instance, $\langle \nu, G \rangle$-machine and GAML use program annotations as the only means to identify parallelism   These annotations can be quite cumbersome for the programmer. The approach used in Buckwheat relies on strictness information to identify parallelism, but the strictness information used is based on head-normal form (HNF)[2] and is not sufficient (as we show later) to identify significant parallelism in most programs. To alleviate this problem they make assumptions such as *cons* and *append* being strict, which runs counter to the goals of lazy evaluation.

A second problem with $\langle \nu, G \rangle$-machine and GAML is that they use memory management techniques that do not scale well. $\langle \nu, G \rangle$-machine uses a sequential garbage collector; performance figures of $\langle \nu, G \rangle$-machine given in [Aug89] do not include garbage collection times. GAML uses a parallelized garbage collector, but it scales poorly, e.g., one of the garbage collector speedup curves flattens at a speedup of 2 when the number of

[1] A lazy implementation is one that performs only those computations that are necessary to obtain the normal form of an input expression [HL79].

[2] We use HNF and weak HNF interchangeably.

335

processors is increased to 7 or 8. The EQUALS implementation overcomes both these drawbacks as follows:

- Automatic detection of parallelism by propagating exhaustive demand as far as possible. We accomplish this using *ee*-strictness analysis developed in [SPR90][3]. As we show in section 2, exhaustive evaluation also increases task granularity by packing several HNF tasks into one task. It also improves sequential performance by avoiding repeated traversals of the graph.

- Using reference counting for memory management. Our implementation results show that it scales well and also has low memory requirements without compromising efficiency.

In this paper we present EQUALS, a fast parallel implementation of a lazy, first-order functional language on a six-processor Sequent Symmetry. The EQUALS system consists of an optimizing compiler and a runtime system. Functional programs are first translated into an intermediate code which is then optimized. Following this, the intermediate code is macro expanded into C code. (We chose to use C for portability reasons.) The C programs are then compiled and executed under the control of a runtime system which consists of routines for task and memory management.

The rest of this paper is organized as follows. The next section elaborates on the issues raised above, namely, parallelism detection and memory management. An overview of the EQUALS system is presented in section 3. The EQUALS compiler is described in section 4. Sections 5 and 6 describe our task and memory management schemes. A detailed discussion of performance of EQUALS is presented in section 7. Our results show that sequential performance is comparable to SML/NJ, one of the fastest functional language implementations. The parallel performance of EQUALS is about the same as $\langle \nu, G \rangle$-machine, even though EQUALS times include memory management whereas $\langle \nu, G \rangle$-machine times exclude garbage collection time. Results also show that reference counting mechanism scales well, uses less memory and has better memory locality. Concluding remarks about our experience with EQUALS appear in section 8.

2   Parallelism and Memory Management

There have been two approaches to identify parallelism in lazy languages. One approach, used in $\langle \nu, G \rangle$-machine and GAML requires programs to be annotated for parallelism. This can be quite cumbersome since

$$
\begin{aligned}
qs(x : xs) &\rightarrow qs1(split(xs, x, nil, nil)) \\
qs(nil) &\rightarrow nil \\
split(x : xs, y, u, v) &\rightarrow if(x > y, split(xs, y, x : u, v), \\
&\qquad\qquad split(xs, y, u, x : v)) \\
split(nil, y, u, v) &\rightarrow < u, y, v > \\
qs1(< x, y, z >) &\rightarrow append(qs(x), y : qs(z)) \\
append(x : xs, y) &\rightarrow x : append(xs, y) \\
append(nil, y) &\rightarrow y
\end{aligned}
$$

Figure 1: QuickSort (':' denotes *cons* operator)

annotations are always required. Moreover, in order that laziness not be compromised, the task scheduler must have a mechanism to ensure that the normal order branch of computation makes progress. An alternative approach that does not suffer from these drawbacks is to use strictness information to identify parallel components, as in [Geo89] and [Gol88b]. However, their model of computation is based on head evaluation and does not at all deal with exhaustive evaluation. Hence the strictness information they use also deals with head-normal forms alone[4]. This strictness information is not sufficient (as shown below) to detect significant parallelism in many programs. Therefore they assume that even non-strict functions such as *cons* and *append* are strict.

To illustrate why strictness based upon HNF alone is not sufficient to identify parallelism, consider the *QuickSort* example shown in fig. 1. Note that $qs(l)$ first splits the list into l_1 and l_2 and subsequently calls $append(qs(l_1), qs(l_2))$. Unless we know that *append* needs to evaluate its second argument, $qs(l_1)$ and $qs(l_2)$ cannot be evaluated in parallel. However, *append* is not strict in its second argument, and so strictness information does not detect any parallelism here. Even if we annotate *append* as being strict, no parallelism is detected because $qs(l_2)$ will be evaluated concurrently with that l_1 only till its HNF is obtained. After this, its evaluation will be suspended till *append* consumes all of its first argument, i.e., until $qs(l_1)$ is completely evaluated. This forces us to the extreme measure of annotating *cons* (in addition to *append*) as being strict in both arguments so as to identify any parallelism at all.

2.1   Propagating Exhaustive Demand and its Merits

In order to overcome the problems mentioned above, we need to take the context of evaluation into account when performing strictness analysis. Observe that if the output of *append* (or *cons*) is exhaustively demanded (i.e., needed in normal form) then both its arguments are needed in normal form. In other words, *append* and

---

[3]Most of the other strictness methods do not deal with (exhaustive) normal form demands. For the parallelism detection discussed here, any strictness method that deals with normal form demands can be used.

[4]i.e., it provides information about which arguments of a function are to be head-normalized in order to head-normalize the function application.

cons are *ee*-strict (see [SPR90] for details) in their arguments. By propagating exhaustive demand in this manner and utilizing *ee*-strictness information we can identify all the parallelism in the examples discussed in this paper. Observe that since parallelism detection is still based on strictness, laziness is never compromised. Therefore, the task scheduler needs no explicit mechanism to ensure progress of normal order computation.

In previous implementations, tasks compute weak head-normal forms of terms. (Henceforth, we use "terms", "graphs" and "expressions" interchangeably.) However, HNF tasks are typically fine grained and therefore can easily lead to significant overheads. Although this problem can be alleviated to a large extent by a careful design of task management (as is done in [Aug89] and [Mar91]), nevertheless it is quite advantageous to use larger grain tasks. Use of exhaustive demand (or *e*-demand) aids us in achieving this, since it packs several HNF tasks into a single task.

Propagating exhaustive demand also increases the efficiency of sequential evaluation since it avoids repeated traversals of the graph. For instance, observe that in the QuickSort example, $qs(l)$ eventually reduces to

$$append(append(\cdots append(t_1, t_2)\cdots)^5$$

If we do not propagate exhaustive demand, then the request to head normalize the outermost *append* results in another call to head-normalize the inner *append*. This proceeds all the way to the innermost *append*, which then outputs a single element. This element is consumed by the next outer *append* and so on until the top-level *append* outputs one element. Then the whole process is repeated, thereby resulting in n traversals of the graph in order to compute n elements. In contrast, if we propagate *e*-demand then the top-level *append* will force complete evaluation of inner *append*, which in turn will force full evaluation of its inner *append* and so on. This results in just a single traversal of the graph to compute all the elements in the output list. Moreover, Due to exhaustive demand propagation, EQUALS code is similar to that generated by a strict language and hence its sequential performance is comparable to strict implementations. In summary, propagating exhaustive demand leads to:

- easier detection of parallelism
- larger task granularity
- avoidance of repeated traversals of the graph.

2.2 Memory Management

Most previous implementations use garbage collection to reclaim storage. This approach suffers from several drawbacks in the context of parallel evaluation. First, the garbage collector scales poorly when the number of processors is increased. This is because there are certain inherently sequential components and hot spots in the copying phase of the collector such as the need to lock *every* structure before moving[6]. This problem is compounded by the fact that the garbage collectors traverse a considerable portion of the heap space and consequently produce a considerable amount of paging activity, which is again inherently sequential.

Another problem with the garbage collection approach is that it can lead to poor locality of reference, which is quite important in a virtual memory/cache environment. In evaluation of functional programs, very often we build structures that are used just once. In the garbage collection approach, this space is not reused until after the next collection. This means that a page may be brought in from the disk, accessed very few times and then written back.

We use reference counting in EQUALS for memory management. We will show later how reference counting avoids memory contention and improves locality due to immediate reclamation and reuse of free space. It is also very economical on memory use and is quite efficient. For instance, our sequential run times are comparable to that of SML/NJ (with dereferencing typically taking less than 20% of the time) and heap space usage is typically 15 to 25% that used by SML/NJ.

Although the reference counting approach cannot handle cyclic structures, this is not a problem in a purely functional language implemented using combinators. For instance, Goldberg [Gol88b] also uses reference counting[7].

3   Overview

The EQUALS system consists of a compiler, a task manager and memory manager. In this section we present an overview of each of these modules, deferring the details to later sections.

The EQUALS compiler uses *ee*-strictness analysis [SPR90] to detect parallelism. This information is used in translating the source program into a combinator-based intermediate language. The intermediate language includes constructs for creating parallel tasks and synchronizing among them. Several optimizations are performed on the intermediate code before it is translated to C-code.

---

[5]This term may not be constructed explicitly in its entirety; parts of it (e.g., its spine) may be on the stack.

[6]It should be noted here that the Appel-Ellis [AEL88] collector is sequential; the concurrency arises in a single process performing collection while other process(es) are executing the program.

[7]Although reference counting was used in [Gol88b], its efficiency compared to garbage collection and effectiveness in overcoming the above problems was not established.

337

The goal of compiled code is to normalize a given input expression. If we identify multiple subterms that are to be evaluated to accomplish this, EQUALS evaluates these subterms in parallel. Evaluation of each subterm is taken up by a task, which executes the compiled code on its private stack. There may be many more tasks created than processors, hence tasks are created only when they are deemed useful. Some of these decisions are made at compile time, while others are deferred until run-time. For instance, the compiler will never emit code to create a new task, unless it can generate code for work to be done concurrently by the existing task. At run-time, on the other hand, opportunities for additional parallelism will be passed up, if many parallel tasks already exist.

Although these tasks can be executed as UNIX processes, it is very expensive to do so. Therefore tasks are executed under the control of evaluator processes (one per processor). Load sharing is based on a global queue from which idle evaluators can take up tasks.

If a task needs the value of a subterm that is currently being evaluated by another task, then it must suspend itself until the evaluation is complete. In the meantime, its evaluator runs other tasks. Once the evaluation of a term is completed, all tasks awaiting its evaluation are put back in the global queue.

4   Compiler

The EQUALS source program consists of a set of functions defined by pattern match. The abstract syntax of EQUALS source language is given below. Here f denotes a function symbol, c a constructor, d a functor or a constructor and x a variable.

program	::=	fundef; $\cdots$; fundef
fundef	::=	f(pat, . . . , pat) = expr
expr	::=	if expr then expr else expr
		$\mid$ d(expr, . . . , expr)
		$\mid$ x
pat	::=	c(pat, . . . , pat)
		$\mid$ x

Each function definition in the program is translated into a corresponding function in the intermediate language. Its body gets translated into a sequence of intermediate code statements. The constructs in the intermediate language are given below. This language bears some similarities to G-machine, but differs in many ways such as explicitly named variables and functions, and constructs for demand propagation. The target language (C) influenced some constructs; for instance, compound expressions were permitted since they are allowed in C and hence can be more efficiently compiled than a sequence of simple expressions achieving

the same objective. In the description of the intermediate language, *extent* specifies the extent of evaluation (NF or HNF); *expr* stands for constants, variables or compound expressions using predefined functions such as +; *block* stands for any sequence of statements in intermediate code. *Location* specifies whether a task is to be evaluated locally or at a remote site (i.e., on another processor). Some of the constructs such as **Barrier** and **Deref** that are introduced during optimization are not shown here.

- **Function** $f(context, x_1, \ldots x_n)$:
  Header for function f. Every function in the intermediate code takes a parameter named *context*. This parameter specifies (at run-time) the extent to which the output of (the current invocation of) a function needs to be evaluated. Using this parameter we propagate demand at run-time.
- **Assign** $var, expr$.
- **If** $expr$ then $block_1$ else $block_2$.
- **Switch** x case $c_1 : block_1, \ldots$, case $c_n : block_n$.
- **Evaluate** x to $extent$ at $location$.
- **FunctionEval** $f(v_1, \ldots, v_n)$ to $extent$ at $location$ result x.
- **BuildTerm** $d(v_1, \ldots, v_n)$ result x.
- **GetChild** i of x result y.
- **Return** x.

4.1   Compilation Algorithm

First, the pattern matching constructs of EQUALS are transformed to case expressions, using Huet-Levy [HL79] algorithm for lazy pattern matching[8]. After pattern matching, the only change to the structure of the source is introduction of case expressions. The code generator (see fig. 2) takes these transformed function definitions and produces intermediate code. It consists of the several functions listed below. Most of them take two parameters: the fragment of the source program to be translated and *extent*, which specifies the demand on this fragment. The value of *extent* can be UNK, which means that the demand is not known statically, or one of NF or HNF. This parameter *extent* is *not* to be confused with *context*: the former is a *compiler* parameter whereas the latter is a parameter to functions in intermediate code and is used to propagate demand at *run-time*. In the figure, best(*extent, context*) stands for 'NF' if *extent* = NF and '*context*' otherwise.

$\mathcal{F}$: the top-level code generator for a function.

$\mathcal{E}$: translates an expression. It takes three parameters: the expression to be translated, the name of the

---

[8]Our current system does not handle prioritized patterns. It can be done using the techniques of Laville [Lav88], Puel and Suarez [PS90] and those in Sekar et al. [SRR92].

338

$$\mathcal{F} \; [\![f(x_1, \ldots, x_n) = e]\!] \quad = \quad \text{Function } f(context, x_1, \ldots, x_n);$$
$$\mathcal{E} \; [\![e]\!] \; y \; \text{UNK};$$
$$\text{Return } y$$

$$\mathcal{E} \; [\![\text{case } x \text{ in } (pm_1, \ldots, pm_n)]\!] \quad = \quad \text{Eval } x \text{ to HNF at Remote};$$
$$y \; extent \qquad\qquad \text{Switch } x \quad \mathcal{P} \; [\![pm_1]\!] \; x \; y \; extent;$$
$$\vdots$$
$$\mathcal{P} \; [\![pm_n]\!] \; x \; y \; extent$$

$$\mathcal{E} \; [\![x]\!] \; y \; extent \quad = \quad \text{Eval } x \text{ to } best(extent, context) \text{ at Remote};$$
$$\text{Assign } y, x;$$

$$\mathcal{E} \; [\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!] \quad = \quad \mathcal{E} \; [\![e_1]\!] \; z \; \text{NF};$$
$$y \; extent \qquad\qquad \text{If } z \text{ then } \mathcal{E} \; [\![e_2]\!] \; y \; extent \text{ else } \mathcal{E} \; [\![e_3]\!] \; y \; extent$$

$$\mathcal{E} \; [\![d(e_1, \ldots, e_n)]\!] \; y \; extent \quad = \quad \mathcal{A} \; [\![e_1]\!] \; 1 \; d \; z_1 \; extent;$$
$$\vdots$$
$$\mathcal{A} \; [\![e_n]\!] \; n \; d \; z_n \; extent;$$

BuildTerm $d(z_1, \ldots, z_n)$ result y if d is a *constructor*

Assign $y, d(z_1, \ldots, z_n)$ if d is a *predefined function*

FunctionEval $d(z_1, \ldots, z_n)$ to   if d is a *user-defined function*
     $best(extent, context)$ at Remote result y

$$\mathcal{P} \; [\![c(x_1, \ldots, x_n) \rightarrow e]\!] \; x \; y \; extent \quad = \quad \text{case } c: \quad \text{GetChild 1 of } x \text{ result } x_1;$$
$$\vdots$$
$$\text{GetChild n of } x \text{ result } x_n;$$
$$\mathcal{E} \; [\![e]\!] \; y \; extent$$

$$\mathcal{A} \; [\![e]\!] \; i \; d \; y \; extent \quad = \quad \begin{cases} \mathcal{E} \; [\![e]\!] \; y \; \text{NF} & \text{if } i^{th} \text{ arg. of } d \text{ is } ee\text{-strict and } extent = \text{NF} \\ \text{If } context = \text{NF} & \text{if } i^{th} \text{ arg. of } d \text{ is } ee\text{-strict and } extent \text{ is not NF} \\ \qquad \text{then } \mathcal{E} \; [\![e]\!] \; y \; \text{NF} & \\ \qquad \text{else } \mathcal{B} \; [\![e]\!] \; y & \\ \mathcal{B} \; [\![e]\!] \; y & \text{otherwise} \end{cases}$$

$$\mathcal{B} \; [\![x]\!] \; y \quad = \quad \text{Assign } y, x$$

$$\mathcal{B} \; [\![d(e_1, \ldots, e_n)]\!] \; y \quad = \quad \mathcal{B} \; [\![e_1]\!] \; z_1;$$
$$\vdots$$
$$\mathcal{B} \; [\![e_n]\!] \; z_n;$$
$$\text{BuildTerm } d(z_1, \ldots, z_n) \text{ result } y;$$

Figure 2: **Compilation Scheme**

variable into which the value of the expression is to be stored (y) and *extent*.

$\mathcal{P}$: is like $\mathcal{E}$, but handles pattern-matching.

$\mathcal{A}$: generates code for evaluating argument expressions to a function. $\mathcal{A}$ takes five arguments: the expression e to be evaluated, the argument position of e, the name d of the function whose argument e is, the name of the variable y into which the result is stored and *extent*. $\mathcal{A}$ differs from $\mathcal{E}$ in that it takes the strictness of d into account to determine the demand on e.

$\mathcal{B}$: generates code to build the graph representing its argument e. The result is stored into the argument y.

Note that all function evaluations have been labeled for possible remote evaluation, and no synchronization barriers have been placed. Moreover, code for managing the free space (e.g., dereferences) have not been generated. These are generated after a flow analysis performed during optimization.

The code generated by the above scheme is not efficient. It uses many temporary variables and manipulates all values in boxed form. Also, the code is generated without regard to common sub-expressions. Several optimizations are performed to improve the time and space efficiency of the final code. These optimizations are not new: in EQUALS known techniques have been combined to achieve good efficiency.

4.2 Optimizations

1. Unboxing: Any boxing operation on a value followed by an unboxing operation is removed, eliminating all unnecessary boxing operations When a function needs a parameter in a fully evaluated state (which is determined from strictness), it is passed as an unboxed value. Also, functions return unboxed values when-

ever possible. Such inter-procedural unboxing improves speed by more than a factor of 2.

2. Tail Recursion Elimination: Direct tail recursion and certain linear recursions (with associative functions as outermost symbols) are converted into loops.

3. Lifetime Analysis: A variable is determined to be in evaluated (to different extents), unevaluated or unknown state at each point in the code, by lifetime analysis (performed beyond basic blocks). Using this analysis the following optimizations are performed:

• *Reducing the Number of Temporary Variables* using a graph coloring heuristic.

• *Placement of Synchronization Barriers* just before the point where the value is needed. If no significant work is done between a remote evaluation and its corresponding Barrier (e.g., only straight line code) it is changed to a local evaluation.

• *Common Subexpression Sharing* is done to share expressions with different evaluation extents and across basic blocks.

• *Immediate Reclamation of Free Space*. At the end of the lifetime of a variable, the term it points to is dereferenced. This significantly reduces heap space usage, e.g. heap space requirement is brought down by a factor of 4 in QuickSort.

4. Generating two versions: Finally, we can eliminate the *context* parameter (together with all the tests on its value) by generating two versions of the code for each function. The two versions are invoked when the result is required in NF and HNF respectively.

Following is an example EQUALS program and its optimized (NF-demand version of) intermediate code:

```
sumfrom(i,j) = if (i==j) then i
   elseumfrom(i, (i+j)/2))
      + sumfrom( (i+j)/2 + 1, j)
```

Function *sumfrom_NF*$(x1, x2)$
If $x2 = x1$ then
   Assign $y3, x1$
else
   Assign $y1, (x1 + x2)/2$
   FunctionEval *sumfrom_NF*$(x1, y1)$ at Remote result $y2$
   Assign $y3, (y1 + 1)$
   FunctionEval *sumfrom_NF*$(y3, x2)$ at Local result $y1$
   Barrier $y2$, **waitfor** NF
   Assign $y3, (y1 + y2)$
Return $y3$

---

5   Task Management

The task management system consists of subsystems to create, synchronize and load-balance tasks. Each of these subsystems is described in detail below.

Task Creation: Recall that the purpose of a *task* is to evaluate a term. Task creation consists of building the term to be evaluated (if it does not already exist) and allocating a stack for its evaluation. In contrast with our scheme for task creation, $\langle \nu, G \rangle$-machine avoids explicit graph construction by building it directly on a stack. This is not suitable for our approach since stack frames are of variable size and will complicate memory management due to fragmentation. Moreover, contiguous stacks are much simpler to manage since they do not require any dereferencing or garbage collection.

Task Synchronization: While evaluating a term, a task T may find that it needs to evaluate one of its subterms[9]. If this subterm s is already being evaluated by another task then T must suspend itself until s is evaluated. In such a case, T executes a **Barrier** instruction which places it in a wait-queue corresponding to s. Its evaluator then proceeds to execute the next task from the global ready queue. In general, there may be several such tasks in the wait-queue of s. Some of these tasks may need s in NF whereas the others may need it in HNF. When s is evaluated, if it is in NF then all tasks in the wait-queue of s are moved to the global ready queue. Otherwise only those tasks that need s in HNF are released. If there are tasks waiting for NF of s, these tasks are not released and s is taken up for normalization.

An important point to be noted here is that wait queues are associated with terms, rather than tasks. This is because a task T created to evaluate a term s may also take up many subterms $s_1, ..., s_k$ of s for (local) evaluation. Suppose that another task T' needs to evaluate s_1. Observe that in this case T' needs to wait only until s_1 is evaluated. However, T will complete only after evaluating all of $s_2, ..., s_k, s$. Thus the parallelism that can result in simultaneous execution of T and T' is lost if T' waits on T instead of s_1.

Load Balancing: Note that new or resumed tasks are placed in the global ready queue from which free evaluators take up tasks[10]. Thus, the global ready queue is the mechanism for load balancing. To reduce contention at the global queue, we create tasks only when the system is lightly loaded. When the number of tasks in the ready queue exceeds some threshold, the evaluators avoid creating tasks and instead perform the intended computation locally. Using this mechanism we have been able to significantly reduce contention at the ready queue. Experiments show that ready queue is *not* a bottle neck in our implementation.

---

[9] This subterm may not be a part of the original term, but may be created during its normalization.
[10] A task may be taken up by different evaluators during its lifetime. Thus, all memory accessed by a task, including its stack, must be kept in shared memory.

6 Memory Management

Memory is divided into two sections, namely, stack space and heap space. The stack space is divided into stacks of two different sizes: 2K and 16K. Initially each task is allocated a 2K stack. There can be many suspended tasks at any time, many of them waiting without having performed much computation. Allocating a small stack initially ensures that very little space is tied up by such tasks. When the stack of a task overflows, it is extended by linking a 16K stack[11].

The heap space is divided into fixed-size nodes that are used to build graphs. To avoid contention when allocating memory, the heap is split into several free lists, one list per evaluator. Each evaluator allocates from (and returns free memory to) its own list. When an evaluator runs out of nodes, it may borrow nodes.

6.1 Node Design and Locking

In EQUALS, graph nodes have value and status fields. Value fields include the constructor or function symbol at the node, its type and pointers to its arguments. When a functor node is normalized, these fields are updated. Status fields show whether the term beneath that node is entirely normalized, or whether the node is being normalized. They also include the wait queue for the node and its reference count.

Another status field is used to implement a lock for the node. We have built shadow locks, in which we spin on the copy of the lock bit in cache until it is reset and then try to obtain the lock [Seq87]. These locks are built around the atomic test-and-set (btsw) instruction that Sequent supports and generate less bus traffic than naive locks. In general, a node is accessed only after the lock is acquired. However, if a node has been normalized, the extent of its normalization guarantees that either the entire term, or perhaps just the root, will not require locking. The implementation ensures that the status bits indicating the extent of normalization can be safely read without locking. Since EQUALS is a lazy language, tests for normalization are done very often and the above optimization is quite important. Another case when locking is not required is when a node cannot be referenced by another processor, e.g., when the node has just been allocated. Finally, the reference count is manipulated exclusively with atomic increment and decrement instructions and the normal locking convention is not used.

Two other locks are used in our implementation. One serializes access to the global queue, and the other serializes access to the pool of free stacks. Our experiments

---

[11] Checking for stack overflow or underflow normally involves only a test-and-branch. This check is performed at the beginning and end of every function.

indicate that these locks are not significant since too many small tasks are not created in practice.

6.2 Reference Counting

As mentioned before, reference counting is used to reclaim free space. Since there is no separate phase in which all processes collect free space, opportunities for contention at memory reclamation are minimized. Moreover, using reference counts permits the following trick to avoid locking when a node is freed. Observe that a node about to be freed (i.e., a node being dereferenced with reference count = 1) will be referred to only by the current evaluator. Thus there is no need to lock it before freeing. Since shared nodes are uncommon, this trick succeeds most of the time and hence is important in practice. Note that this trick cannot be used to avoid locking at copying time in a garbage collector, since the reference information is not available.

Using reference counting we can immediately reclaim freed space. This results in greatly reducing the heap space usage. Furthermore, by maintaining the free list as a LIFO, we immediately reuse memory that is freed. Since nodes are created and destroyed very quickly in typical programs, this strategy greatly increases the chances of using the same set of memory locations again and again, resulting in improved locality.

One caveat is that if the compiler generates code that manipulates the counts improperly, the system may fail. We have found the LIFO strategy useful in debugging such problems, as the immediate reallocation of a prematurely deallocated node tends to generate detectable problems soon after the problem occurs.

7 Implementation Results and Discussion

In this section we present the results of our implementation based on example programs adapted from [Geo89, Gol88a, Aug89, SPR90]. First we study the sequential performance of EQUALS and show that it is comparable to the latest release of Standard ML of New Jersey (SML/NJ). Following this we compare our speeds and scalability with that of $\langle \nu, G \rangle$-machine and GAML. We then discuss the impact of reference counting on scalability and performance. In particular, we provide experimental evidence to show that memory requirements are significantly less and that locality is improved.

7.1 Sequential Performance of EQUALS

Table 1 compares the performance of EQUALS to

341

	EQUALS	SML/NJ
Euler	88.0	104
9queens	10.4	8
MatMult	19.7	14
Sieve	59.0	33
QuickSort	8.6	4

Table 1: Comparison of EQUALS and SML/NJ. (All timings in secs. on a Sun 3/260)

	EQUALS	$\langle \nu, G \rangle$	GAML
MatMult	22.6	NA	NA
QuickSort	9.5	NA	NA
Euler	116.9	128.4	430
10queens	64.0	73.9	467
Nfib30	32.1	62.1	213

Table 2: Comparison of EQUALS with $\langle \nu, G \rangle$-machine and GAML. (NA: Not Available)

SML/NJ (release 0.75). SML/NJ is a sequential implementation of SML, a strict language, and is among the fastest functional language implementations. In the table, *Euler* computes the Euler totient function from 1 through 1000. In addition to performing substantial amounts of computation, this program also spends a lot of time creating and destroying lists. *MatMult* computes the product of two 100×100 matrices. *Sieve* computes list of primes between 2 and $10,000$. *QuickSort* sorts a list of 5000 integers.
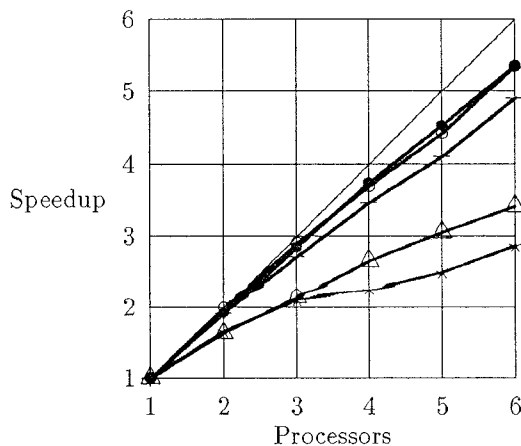
Observe that speeds of SML/NJ and EQUALS are

Figure 3: Speedup curves for EQUALS.
Ideal: —   Euler: •   Nfib: o
MatMult: +   10queens: △   QuickSort: ⋆

comparable in *Euler*, *MatMult* and *9queens*. By propagating exhaustive demand and generating two versions, our code is similar to that generated for a strict language, and hence the speeds are comparable. In *QuickSort* and *Sieve*, where there are very few computation steps and most of the time is spent in creating and destroying list structures, SML/NJ is significantly faster because it uses unboxed lists, whereas EQUALS uses boxed lists[12]. This is not a problem in the first three examples, since the number of steps that access lists or perform any other computations are much larger than those that create or destroy lists. (e.g., in *MatMult* there are 10^6 operations of the first kind versus 10^4 list creation/deletion steps.). Boxing can increase the work involved in copying by as much as 100%. Moreover, the performance of EQUALS can be substantially improved by generating assembly code, as is done in SML/NJ. We are quite encouraged to get performance comparable to SML/NJ in spite of these factors.

7.2 Parallel Performance

Table 2 shows wall-clock times for EQUALS, $\langle \nu, G \rangle$-machine and GAML on a single processor. Timings for both EQUALS and $\langle \nu, G \rangle$-machine were obtained on Sequent Symmetry (16 Mhz clock). However, $\langle \nu, G \rangle$-machine timings do not include garbage collection time, which can account for up to 30% of total time. GAML timings were obtained on Sequent Balance which is considerably slower. This impedes a reasonable comparison between our times and those of GAML. However, it is mentioned in [Mar91] that the sequential execution times for GAML are roughly of the same order as that of $\langle \nu, G \rangle$-machine.

Fig. 3 shows speedup curves on all of the examples run using EQUALS. MatMult and Euler create large grain tasks and hence speedup is almost linear. Although task granularity is very small in Nfib30, we still scale well, showing that we have managed to keep down task overheads and contention at the global queue. The *split* phase of QuickSort is inherently sequential and hence even the ideal speedup is considerably worse than linear. (For instance, the maximum speedup achievable for an input list of length n is only $\log n$ and this too requires n processors.) In 10queens there is a lot of vertical parallelism (i.e. parallelism that arises in simultaneous evaluation of a function and its arguments) which we do not currently exploit.

Figs. 4, 5 and 6 compare the scalability of EQUALS with that of $\langle \nu, G \rangle$-machine and GAML on Euler, Nfib30 and 10queens respectively. Observe that EQUALS scales as well as the $\langle \nu, G \rangle$-machine and GAML on Nfib30. On Euler, it scales as well as $\langle \nu, G \rangle$-machine

---
[12]Currently EQUALS unboxes only primitive data types.

342
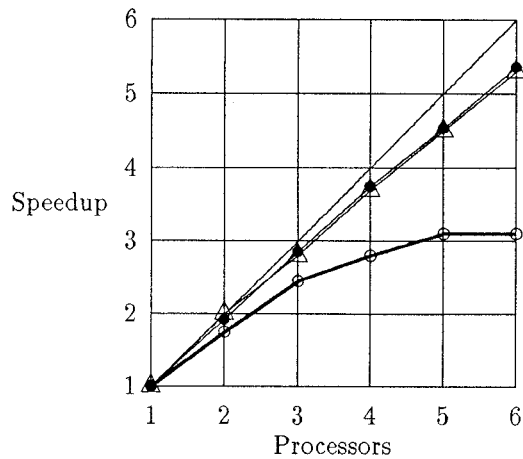
Figure 4: Speedups on Euler.

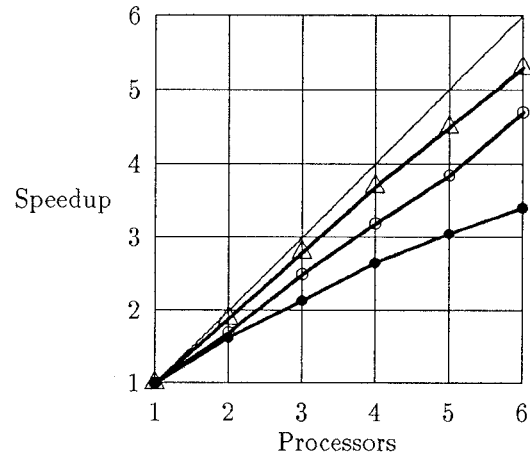Ideal: —   EQUALS:• $\langle \nu, G \rangle$: △   GAML: ○

Figure 6: Speedups for 10queens.

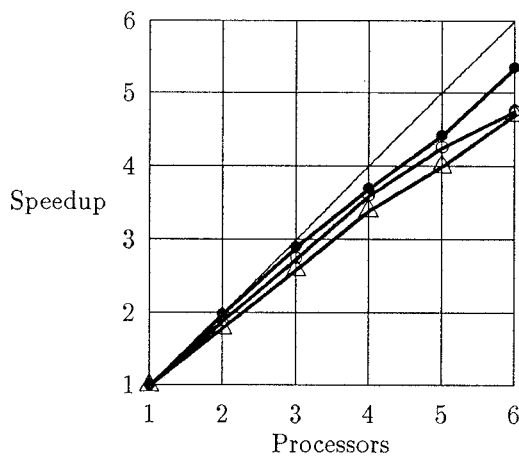Ideal: —   EQUALS: •   GAML: ○ $\langle \nu, G \rangle$: △

		EQUALS	
	SML/NJ	heap	stack
Euler	2.2	0.10	0.07
9queens	0.8	0.34	0.09
MatMult	0.8	0.64	0.01
Sieve	2.2	0.32	0.32
QuickSort	1.4	0.16	0.20

Table 3: Memory usage of EQUALS and SML/NJ (in MB's).

Figure 5: Speedups for Nfib30.

Ideal: —   EQUALS: •   GAML: ○ $\langle \nu, G \rangle$: △

low memory requirement and improved locality. In this section we give empirical evidence for these claims.

The Euler program spends over 40% of the total time in memory allocation and deallocation, creating and destroying as many as 3 million nodes. The fact that this program scales almost linearly clearly demonstrates the scalability of our memory manager. In contrast, the speedup of GAML appears to saturate even for 5 processors, largely due to poor scaling of the memory management techniques used. Table 3 shows the memory utilization of some programs in EQUALS and SML/NJ.

and better than GAML. But both $\langle \nu, G \rangle$-machine and GAML scale better in 10queens since they exploit vertical parallelism (unlike us) with the aid of programmer annotations. Furthermore $\langle \nu, G \rangle$-machine timings do not include garbage collection times. As can be seen from the results in GAML, garbage collection times scale poorly, e.g., in Euler, the garbage collection time decreases by only a factor of 2 when number of processors increases to 8.

7.3   Impact of EQUALS Memory Management

We had mentioned in the introduction that memory management was a crucial component and that by using reference counting we can achieve very good scalability,

	EQUALS	SML/NJ
Euler	1.66	2.16
9queens	2.16	3.12
MatMult	1.77	2.14
Sieve	1.76	2.42
QuickSort	1.97	4.75

Table 4: Ratio of Diskless Sun timings over Server timings.

343

The table shows that EQUALS typically uses substantially less memory than SML/NJ[13]. For EQUALS, we show stack and heap use separately. (Stack usage is less critical, since it is easier to manage and its locality leads to less paging than comparable heap use.) Table 4 shows the relative speeds of SML/NJ and EQUALS programs on a Sun 3/260 (a server), in comparison to a diskless Sun 3/75. The slowdown of EQUALS is quite close to 1.75, which is the factor of difference in raw cpu power between the two machines used. In contrast, the performance of SML/NJ degrades considerably more due to excessive paging activity. This demonstrates that memory utilization and locality of reference are much better in EQUALS than in SML/NJ. The difference in degradation is large enough to make EQUALS perform better than SML/NJ on most of these examples on a Sun 3/75.

8   Experience with EQUALS

The EQUALS implementation results show it is possible to automatically detect and effectively exploit parallelism in functional programs by propagating exhaustive demand; there is no need to make assumptions such as *cons* and *append* being strict in all contexts. Furthermore, it also establishes reference counting as a valid mechanism for memory management. In addition to using much less memory and possessing improved locality, reference counting scales well and therefore appears appropriate for parallel implementation.

The implementation experience has also shown us the importance of minimizing task creation and management overheads. We assumed that we can minimize the impact of these overheads by minimizing task creation. We did succeed in reducing task creation: the number of tasks created in EQUALS is less than 10% of the total number that would be created without a throttle on task creation. Still, there is observable parallel overhead, and the task creation time needs to be further reduced.

Moreover, simple and efficient techniques typically perform better than more general and elaborate schemes. For instance, wait counter based synchronization (a generalization of the scheme in [Geo89], where a task waits for multiple subtasks at a single barrier) was initially implemented. Experience showed that most of the waits were performed on a single task and use of the wait counter (with associated overheads of initialization, increment and decrement) was wasteful.

The load balancing scheme used in EQUALS is quite simple, but may not always succeed on more complex programs. We are currently looking at static analysis

of programs for sophisticated load balancing There are also several other sources of improvement in EQUALS such as direct generation of assembly code instead of C-code. This will enable us to use registers effectively and reduce the overhead of function calls. The code can also be much tighter and we believe that considerable improvement in speed can be achieved.

References

[AEL88]   A. Appel, J. Ellis and K. Li, *Real-time concurrent collection on stock multiprocessors*, PLDI, 1988.

[Aug84]   L. Augustsson, *A compiler for lazy ML*, LFP,1984.

[Aug89]   L. Augustsson and T. Johnsson, *Parallel graph reduction with the $\langle v, G \rangle$ machine*, FPCA, 1989.

[Dar81]   J. Darlington, *Alice: A multi-processor reduction engine for the parallel evaluation of applicative languages*, FPCA, 1981.

[Geo89]   L. George, *An abstract machine for parallel graph reduction*, FPCA, 1989.

[Gol88a]   B. Goldberg, *Buckwheat: Graph reduction on shared-memory multiprocessor*, LFP, 1988.

[Gol88b]   B. Goldberg, *Multiprocessor execution of functional programs*, PhD Thesis, Yale Univ. Dept of Computer Science, YALEU/DCS/RR-618, 1988.

[Joh84]   T. Johnsson, *Efficient compilation of lazy evaluation*, Compiler Construction, 1984.

[HL79]   G. Huet and J.J. Levy, *Computations in nonambiguous linear term rewriting systems*, Tech Rep. No. 359, 1979, IRIA, Le Chesney, France, 1979.

[Lav88]   A. Laville, *Implementation of lazy pattern matching algorithms*, ESOP, LNCS 300, 1988.

[Mar91]   L. Maranget, *GAML: A parallel implementation of lazy ML*, FPCA, 1991.

[MKH90]   E. Mohr, D. Kranz and R. Halstead, *Lazy task creation: A technique for increasing the granularity of parallel programs*, LFP, 1990.

[PS90]   L. Puel and A. Suarez, *Compiling pattern matching by term decomposition*, LFP, 1990.

[PJ87]   S. L. Peyton Jones, *GRIP: A parallel graph reduction machine*, FPCA, 1987.

[SPR90]   R.C. Sekar, S. Pawagi, I.V. Ramakrishnan, *Small domains spell fast strictness analysis*, POPL, 1990.

[SRR92]   R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, *Adaptive pattern matching*, to appear in ICALP, 1992.

[Seq87]   Sequent Computer Systems, *Sequent guide to parallel programming*, 1987.

[WW87]   P. Watson and I. Watson, *Evaluating functional programs on the FLAGSHIP machine*, FPCA, 1987.

---

[13]The memory usage of SML/NJ was estimated from garbage collection messages.