Dynamic Partitioning of Parallel Lisp Programs

Eric Mohr

YALEU/DCS/RR-869
October 1991

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Dynamic Partitioning of Parallel Lisp Programs

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Eric Mohr
October 1991

Copyright © 1991 by Eric Mohr
ALL RIGHTS RESERVED

Abstract

Dynamic Partitioning of Parallel Lisp Programs

Eric Mohr
Yale University

1991

Many parallel algorithms are most easily expressed at a fine level of granularity, but most parallel systems cannot execute fine-grained programs efficiently. In search of efficiency, researchers often work to increase the granularity of parallel tasks by grouping a number of potentially parallel operations together into a single sequential thread. Most researchers have relied on either the programmer or a parallelizing compiler to make such *partitioning* decisions; this work explores the alternative approach of dynamic partitioning at runtime. Dynamic partitioning can simplify parallel programs by eliminating the need for explicit partitioning by the programmer and can produce a better partition than compile-time methods which must predict program execution paths and costs in advance.

Two dynamic partitioning strategies are presented and analyzed: *load-based partitioning* (LBP), where tasks are combined based on dynamic load level, and *lazy task creation* (LTC), where tasks are created only retroactively as processing resources become available. Methods like load-based partitioning have been proposed before; here I describe several deficiencies of the approach and show how lazy task creation overcomes them, both theoretically and in practice. In particular, lazy task creation will never introduce deadlock or worsen load balancing, as can happen in some programs with load-based partitioning. And lazy task creation consistently creates significantly fewer tasks than load-based partitioning.

The advantage of LTC over LBP arises because LTC delays all partitioning decisions and thus can recover flexibly in situations where LBP has made an unfortunate and irrevocable partitioning decision. Further, LTC performs only a minimal set of operations when delaying a partitioning decision, allowing surprisingly low runtime overhead. This set of operations is much smaller than in related approaches because of the key design decision to delay the parent task rather than the child task at potential fork points.

I present details of a successful implementation of lazy task creation in Mul-T, an efficient parallel version of Scheme running on the shared-memory Encore Multimax. Measurements of Encore Mul-T demonstrate that absolute performance and relative speedup are both quite good for a suite of parallel Lisp benchmark programs run with dynamic partitioning.

Acknowledgments

My advisor Paul Hudak supported me in many ways over the years, not least by having enough faith in my abilities to let me find my own path at my own speed. He also taught me about the academic world; about writing papers and giving talks, and encouraged me to be positive. Thank you, Paul, for truly considering your students' best interests.

In many ways Bert Halstead was as much my advisor as Paul. His breadth of knowledge, incisive observations, scientific and personal integrity, and direct and thorough communication are a constant source of inspiration. His ideas motivated much of this work, and his insightful feedback improved it greatly.

Marina Chen rounded out the committee, lending her experience with parallel computing issues.

Many thanks go to the members of the T project at Yale, both for building the high-quality T system and compiler which provided the foundation for Mul-T and for giving their knowledge, advice, and friendship over the years. Early conversations with Jim Philbin started me thinking about parallel Lisp. Norman Adams graciously ported his assembler for the NS 32000 processor. Richard Kelsey taught me about compilers, Scheme, and software engineering, and was a sounding board for my early bad ideas. He spotted a good one though; when we kicked around a version of lazy task creation in 1986 he said "Now *that* would make an interesting thesis." David Kranz was a valuable collaborator throughout the entire project and an invaluable source of information about the Orbit compiler and Mul-T runtime system. I am indebted to him for developing the first Mul-T system and for inviting me to Cambridge in 1988 to help finish it. Finally, T would not have existed without the creative vision of Jonathan Rees.

This work profited also from conversations with Marc Feeley, Dan Nussbaum, Randy Osborne, Charles Consel, and Tom Blenko as well as from periodic bouts with the dreaded Hudak Jello Wrestling Squad.

Many department cohorts helped a long seven years slide by more amiably, notably Ajit Agrawal, Rob Bjornson, Tom Blenko, Adrienne Bloss, Charles Consel, Chris Darken, Scott Fertig, Jim Firby, Eric Jones, Richard Kelsey, Amir Kishon, David Kranz, Zhijing Mou, Jim Philbin, Raman Sundaresh, and many others I must apologize for omitting.

For providing me with a full life outside the department I must thank a whole New England full of musicians and dancers, notably the members of Froggie on the Carport and New Haven Morris and Sword. Malcolm Sanders gets special appreciation for rolling along through all my worlds and helping me avoid terminal seriousness. Nicole Faulkner made the final year and a half a lot more fun.

Finally I would like to thank my family; my parents Larry and Jean and my sister Carol, who in their own ways supported me tremendously over the last seven years. My father's sustained interest provided a constant basis for perseverance. This thesis is dedicated to him.

This research was supported in part by the National Science Foundation (Grant DCR-8451415) and the Department of Energy (Grant FG02-86ER25012).

Contents

Abstract

Acknowledgments i

1 Introduction 1
- 1.1 The Granularity Problem 1
- 1.2 Domain: Parallel Shared-Memory Lisp 3
  - 1.2.1 `future`: Imperative or Declarative? 3
  - 1.2.2 Dynamic Partitioning with `future` 4
- 1.3 An Example 6
- 1.4 Encore Mul-T 9
- 1.5 Scope 11
- 1.6 Overview of Main Ideas, Results, and Contributions 12

2 Dynamic Partitioning Methods Described and Compared 14
- 2.1 Load-Based Partitioning 14
  - 2.1.1 Deadlock 15
  - 2.1.2 Load Balancing 18
  - 2.1.3 Number of Tasks Created 19
  - 2.1.4 Programmer Involvement 21
- 2.2 Lazy Task Creation 21
  - 2.2.1 Deadlock 23
  - 2.2.2 Load Balancing 26
  - 2.2.3 Number of Tasks Created 27
  - 2.2.4 Programmer Involvement 28

3 Lazy Task Creation: Data Structures, Algorithms, and Implementation 29
- 3.1 Why Implementation is a Challenge 30
- 3.2 The Lazy Task Queue 31
- 3.3 Encore Implementation 34
  - 3.3.1 Data Structures: The Stack and Lazy Task Queue 35

3.3.2 Synchronization Between Producers and Consumers 37
3.3.3 Example of Lazy Future Call and Return 43
3.3.4 Steal Operation 47
3.3.5 Blocking 48
3.3.6 Miscellaneous Considerations 50

4 Implementation: Improvements and Assessment 51
4.1 Optimizing ETC and LBP 51
4.1.1 Favoring the Child Task 52
4.1.2 FIFO or LIFO Queues? 53
4.1.3 Polite Stealing 54
4.2 Runtime Overhead of Partitioning Strategies 54
4.2.1 Eager Task Creation 55
4.2.2 Lazy Task Creation 57
4.3 Overhead of Copying in LTC Steal Operation 57

5 Goals and Methodology of Performance Measurements 60
5.1 Why You Should Believe These Numbers 60
5.1.1 Overhead of Lisp 61
5.1.2 Overhead of Sequential Mul-T 61
5.1.3 Overhead of Parallel Algorithms 63
5.1.4 Overhead of Multiprocessing 64
5.2 Methodology 64
5.3 Benchmark Programs Described 66

6 Benchmark Results 71
6.1 Overhead of Partitioning Strategies 72
6.2 How Granularity Affects Efficiency 73
6.3 How Tree Depth Affects Efficiency 75
6.4 Effect of ETC and LBP Scheduler Changes 77
6.5 Number of Tasks Created 78
6.6 Performance of Benchmark Programs 82
6.6.1 `fatwalk` 83
6.6.2 Four Programs with Excellent Speedup 83
6.6.3 Two Programs with Not Quite as Good Speedup 86
6.6.4 Numerical Data 87

7 Related Work 90
7.1 Methods Resembling Load-Based Partitioning 90
7.2 Methods Resembling Lazy Task Creation 91
7.3 Other Implementations of Lazy Task Creation 93
7.4 Other Related Methods 94

8 Future Work **96**
8.1 Handling Fine-grained Iterative Parallelism 96
8.2 Implementing LTC in Support of Other Parallel Languages 99
8.2.1 The Algol Family 99
8.2.2 Lazy Functional Languages 101
8.3 Dynamic Partitioning on Larger Parallel Machines 102
8.4 Demonstrating that Parallel Lisp is Useful 103

9 Conclusions **106**

Bibliography **108**

Trademarks **114**

List of Figures

1.1 To fork or not to fork. 5
1.2 Direct execution of `psum-tree`. 7
1.3 BUSD execution of `psum-tree` on four processors. 8

2.1 Program `find-primes` could deadlock with load-based partitioning. 17
2.2 Three lazy task creation scenarios. 22
2.3 With oldest-first scheduling a retroactive fork would occur at *a*. . 23

3.1 Stack contains necessary context for retroactive fork. 32
3.2 Lazy task queue data structures and operations. 33
3.3 Stack and lazy task queue in Encore Mul-T, before and after steal. 36
3.4 3 lazy task queue scenarios. 39
3.5 "Lockless" synchronization algorithm. 40
3.6 Program `fib`, compiled. 44
3.7 Program `pfib`, compiled with lazy task creation. 45
3.8 Blocking a task by tail-biting. 48
3.9 Stealing a task in `pfib`. 49

5.1 Standard and revised algorithms for cyclic reduction of tridiagonal systems. 70

6.1 Granularity *vs.* efficiency for `tree` benchmark, by partitioning strategy. 74
6.2 Effect of ETC and LBP scheduling improvements on `tree` benchmark. 75
6.3 Tree depth *vs.* efficiency for `tree` benchmark, by partitioning strategy. 76
6.4 Task counts for fixed tree height or fixed processor count. 81
6.5 Performance of `rantree`. 82
6.6 Performance of `fatwalk`. 84
6.7 Performance of `abisort`, `tridiag`, `queens`, and `fib`. 85
6.8 Performance of `allpairs` and `mst`. 86

List of Tables

5.1 Overhead introduced by Mul-T compiler in sequential code 62

6.1 Overhead per call of `future` and `touch` in sequential code, in microseconds, for each benchmark program and partitioning strategy. 72
6.2 How scheduler changes to favor the child task and adopt a FIFO stealing policy improve performance for `tree` and `rantree`. 78
6.3 How scheduler changes improve performance for benchmark programs. 79
6.4 Number of tasks created in Mul-T benchmarks, by partitioning strategy. 79
6.5 How task count varies with tree depth and number of processors. 80
6.6 Performance of Mul-T benchmark programs. 88

Chapter 1

Introduction

Many parallel algorithms are most easily expressed at a fine level of granularity, but most parallel systems cannot execute fine-grained programs efficiently. In search of efficiency, researchers often work to increase the granularity of parallel tasks by grouping a number of potentially parallel operations together into a single sequential thread. Most researchers have relied on either the programmer or a parallelizing compiler to make such *partitioning* decisions; this work explores the alternative approach of dynamic partitioning at runtime. Dynamic partitioning can simplify parallel programs by eliminating the need for explicit partitioning by the programmer and can produce a better partition than compile-time methods which must predict program execution paths and costs in advance.

This chapter sets the stage for a detailed investigation of dynamic partitioning in parallel Lisp systems, introducing two specific dynamic strategies. These strategies will be explored using Mul-T, an efficient parallel version of Scheme running on the shared-memory Encore Multimax.

1.1 The Granularity Problem

The problem of specifying how an algorithm's operations are divided into potentially parallel tasks is commonly known as *partitioning*; the *granularity* of a given partition describes the size of its tasks. Granularity is usually specified fairly loosely; tasks in a *coarse-grained* partition perform a relatively large number of operations while *fine-grained* tasks perform relatively few operations.

Although an algorithm can often be partitioned in more than one way, there is often a certain partition which is much easier to express than any others because it fits compatibly within the algorithm's control structure. Ideally, a programmer should be able to use the most convenient partition without worrying about granularity. But, executing operations in parallel always involves extra expense due to synchronization, communication, and/or task creation costs in the underlying parallel system. If a partition is too fine-grained these costs will be

overwhelmingly large and the program will perform poorly.

To execute fine-grained programs efficiently some researchers have looked to hardware specially designed to handle fine-grained tasks [Arvind & Culler 86, Gurd *et al* 85]. More common are software methods, where an alternate partition is sought which is coarse enough to allow efficient execution. The question is, how shall this coarser partition be chosen? The most common approaches involve a *static* partitioning of operations to tasks, by either a programmer or compiler. In this work we will explore the alternative of *dynamic* partitioning, motivated by some of the difficulties inherent in static partitioning.

In many systems the programmer is responsible for all granularity decisions, specifying partitions explicitly in the source code. While specifying an efficient partition can be quite easy for some programs, for other programs this approach can require substantial programmer effort and can decrease program clarity. Building tasks of an acceptable size may require additional control constructs, making the resulting program more difficult to understand and modify. For example, a program where each task handles a 10×10 section of a matrix is likely to be much more complex than a program where each task handles one matrix element. Further, finding the best value for a "chunking" parameter (such as 10×10) may require time-consuming experimentation.

At the other end of the spectrum, the programmer may leave partitioning decisions to a parallelizing compiler. To achieve good performance the compiler must create tasks of sufficient size by estimating the cost of executing various pieces of code [Goldberg 88, Hudak & Goldberg 85, Sarkar & Hennessy 86, Debray *et al* 90]. But when execution paths are highly data-dependent (as for example with recursive symbolic programs) the cost of a piece of code is often unknown at compile time. If only known costs are used the tasks produced may still be too fine-grained. And for languages that allow mutation of shared variables it can be quite complex to determine where parallel execution is safe, and opportunities for parallelism may be missed.

There are two additional difficulties with static partitioning. First, making a good static partition requires the programmer or compiler to have some knowledge of the amount of processing resources that will be available when the program is run. But if the code in question is run in parallel with other code or on a multi-user machine the built-in assumptions will be incorrect because other tasks will be competing for the same resources. Secondly, some programs do not lend themselves well to static partitions. Processing resources may need to be concentrated in different areas as execution progresses; in this case a good partition needs to change adaptively to maintain a balanced load.

Dynamic partitioning, where partitioning decisions are made on-the-fly at runtime, has the potential of alleviating all of these difficulties. Programs are simpler when free of explicit partitioning directives. No parameters need to be calibrated and no assumptions about available processing resources are built in.

And the partition can adapt to actual program behavior, even when execution paths are highly data-dependent. But, care must be taken to minimize runtime overhead when building a system using dynamic partitioning; a perfect partition is of no use if creating it adds overwhelmingly to a program's execution time.

This discussion of the granularity problem and the various approaches to solving it has been rather general. After introducing the domain of parallel Lisp and outlining the specific dynamic partitioning strategies to be explored we will look at a specific program which clearly shows the granularity problem, the limitations of static partitioning, and the potential of dynamic partitioning.

1.2 Domain: Parallel Shared-Memory Lisp

Many parallel systems provide some form of *lightweight processes* to execute the tasks in a chosen partition of an algorithm's operations. These *tasks* or *threads* are created at runtime and often have a short lifetime compared to the total execution time of the program. Lightweight processes are a common implementation technique for parallel dialects of languages from both the applicative family and the Algol family. In the applicative family (Lisp, functional languages, logic languages), directives to create lightweight processes may appear as language constructs [Halstead 85, Gabriel 84] or as annotations in source code [Hudak 86] or may be inserted by a compiler [Hudak & Goldberg 85, Larus 89, Debray *et al* 90]. In the Algol family, lightweight processes may be created by language constructs of the "fork/join" variety or may be provided as part of a multitasking library or "threads package".

The granularity problem arises in all of these languages, with potential solutions spanning the spectrum of approaches outlined in the previous section. Likewise, dynamic partitioning ideas may be applied in any of these languages.

Of the above languages, Multilisp [Halstead 85, Halstead 86, Halstead 89] has been one of the most influential. Multilisp's `future` construct offers a concise and flexible mechanism for introducing parallelism in a variety of styles. The dynamic methods explored in this work will use a descendent of Multilisp known as Mul-T [Kranz *et al* 89], a parallel version of Scheme [Rees *et al* 86] based on `future`. The applicability of the ideas developed here to other languages will be discussed in Chapter 8.

1.2.1 `future`: Imperative or Declarative?

The dynamic partitioning methods we will be discussing are based on changing the operational interpretation of `future`. Before discussing the variations however, let us review the original semantics of `future`.

In Multilisp, the expression (`future` X) creates a separate task (the *child*) to evaluate X, and returns a *placeholder*[1] for the eventual value of X. The placeholder is said to be *unresolved* until X's value has been computed, at which point the placeholder is *resolved*. An operation which uses a placeholder's value is said to *touch* the placeholder; any task which touches an unresolved placeholder will be suspended, or *blocked*, until the placeholder is resolved.

This original semantics will be called *eager task creation* (ETC) because executing `future` *always* causes a fork, that is, always causes a separate task to be created.

The dynamic partitioning strategies we will be exploring depend on a crucial shift in how `future` is viewed. We will shift from viewing (`future` X) *imperatively*—as a statement that X *must* be executed in a separate task—to viewing it *declaratively* as a statement that X *may* be executed in a separate task.

With this declarative view the programmer uses `future` to identify potential fork points in the source code. The runtime system may choose not to create a separate task for a given instance of `future`, thereby increasing task granularity and decreasing task creation overhead. The programmer takes on the burden of identifying *what* can be computed safely in parallel and leaves the decision of exactly *how* the division will take place to the runtime system; the programmer's task is to *expose* parallelism while the system's task is to *limit* parallelism.

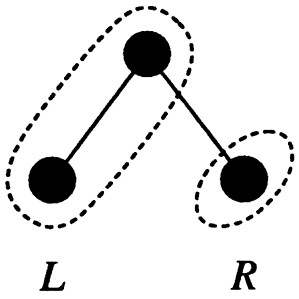
There is a conscious choice here that parallelism should be identified by the programmer rather than discovered by a compiler. It happens that the dynamic partitioning methods we will be discussing would also be useful with parallelizing compilers; but, experience with numerous programs written in the "mostly functional" style common with Scheme suggests that a program's parallelism is usually expressed quite easily using a small number of `future` forms. The challenge in writing a parallel program usually lies in recasting a sequential algorithm in a way that exposes its parallelism rather than in deciding where to place `future` forms. This recasting work must be done regardless of whether the programmer or a compiler is identifying parallelism, so the programmer must do roughly the same amount of work in either type of system. Specifying potential fork points "by hand" allows greater control, as well as insurance that an important fork point will not be overlooked by a parallelizing compiler in a program with complex control structure.

1.2.2 Dynamic Partitioning with `future`

When `future` is viewed as a potential fork point the runtime system has two choices. Figure 1.1 shows these two options for the example code fragment

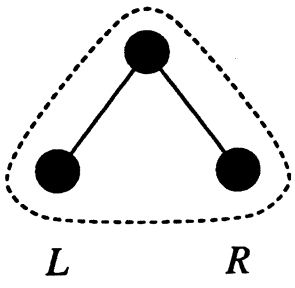
[1] Halstead uses the term *future* to describe the object returned by a `future` form; the term *placeholder* is used here to avoid overloading the word "future".

`(+ (future` *R*`)` *L*`)`



Fork

- Parent task handles *L*
- Child task created to handle *R*
- *R* and *L* evaluated in parallel



No Fork

- Parent task handles *R* and *L*
- *R* is "inlined" — no task creation overhead
- *R* and *L* evaluated sequentially
- Granularity of parent task increased

Figure 1.1: To fork or not to fork.

`(+ (future` *R*`)` *L*`)` [2]

where *R* and *L* denote subexpressions to be evaluated. This expression declares that the subexpressions *R* and *L* may be computed in parallel before they are added. The diagrams show two partitions of the call tree for this expression, with dotted lines encircling the nodes executed by a single task. If a fork occurs the parent task computing *L* and the child task computing *R* are executed in parallel; if no fork occurs the parent task computes *R* "in-line" so that *R* and *L* are evaluated sequentially. This *inlining* of *R* by the parent task eliminates the overhead of creating and scheduling a separate task and creating a placeholder to hold its value, and increases the granularity of the parent task.

Altering the semantics of `future` raises the question of whether combining tasks in this manner is safe, and the related question of what fairness guarantees are made by the runtime system's task scheduler. Issues of safety, fairness and the potential for deadlock will be discussed in Sections 2.1.1 and 2.2.1.

[2]For simplicity the examples in this chapter are written as if procedure arguments were evaluated left-to-right; in fact Mul-T's argument evaluation order is indeterminate and this example should be written as:

```
(let* ((r (future R))
       (l L))
  (+ r l))
```

Inlining can mean that a program's *runtime granularity* (the size of tasks actually executed at runtime) is significantly greater than its *source granularity* (the size of code within the `future` constructs of the source program). A program will execute efficiently if its average runtime granularity is large compared to the overhead of task creation, providing of course that enough parallelism has been preserved to achieve good load balancing.

The question is, *how* shall the runtime system decide whether to fork or not for a given instance of `future`? We shall consider two strategies, *load-based partitioning*[3] (LBP) and *lazy task creation* (LTC) [Mohr *et al* 90, Mohr *et al* 91].

With load-based partitioning, the forking decision is based on the system's current load of tasks—(`future` X) means, "If the system is not loaded, make a separate task to evaluate X; otherwise evaluate X in the current task." A load threshold T indicates how many tasks must be queued before the system is considered to be loaded. Whenever a call to `future` is encountered, a simple check of task queue length determines whether or not a separate task will be created.

As we shall see, load-based partitioning can produce a satisfactory partition in many programs. But, difficulties arise because partitioning decisions are based only on a momentary snapshot of system load and cannot be altered if conditions change later on (see Section 2.1). Lazy task creation was devised to address these difficulties by relaxing the constraints on when a partitioning decision must be made: at *every* potential fork point LTC chooses *not* to fork, but saves enough information so that idle processors can make *retroactive* forks at a later time. Tasks are only created *lazily* as processing resources become available.

With lazy task creation (`future` X) means "Start evaluating X in the current task, but save enough information so that its continuation (representing all remaining work except X) can be moved to a separate task if another processor becomes idle." We say that idle processors *steal* tasks from busy processors; task stealing becomes the primary means of spreading work in the system.

The call tree of a fine-grained program has an overabundance of potential fork points. If each results in a separate task as with the static partition of eager task creation, bookkeeping costs will be overwhelmingly high and performance will suffer. With dynamic partitioning our goal is to fork at only a small subset of potential partitioning points, maximizing runtime task granularity while preserving parallelism and achieving good load balancing.

[3]This method was referred to as *load-based inlining* in [Mohr *et al* 90, Mohr *et al* 91]; "partitioning" seems more descriptive than "inlining" and also avoids confusion with the common compiler technique of *procedure* inlining.

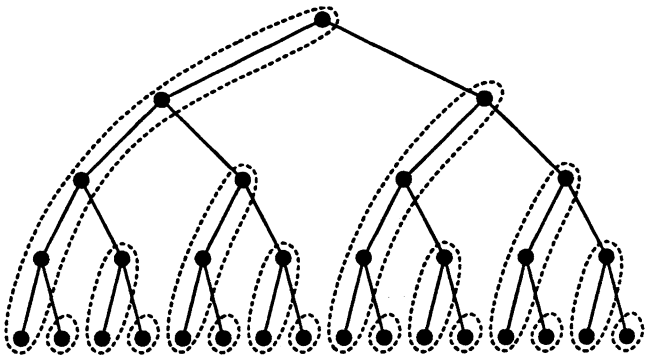


Figure 1.2: Direct execution of **psum-tree**.

1.3 An Example

Let us consider a simple example of the granularity problem and its potential solutions. The following Scheme program sums the leaves of a binary tree:

```
(define (sum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (sum-tree (right tree))
         (sum-tree (left tree)))))
```

(where **leaf?**, **leaf-value**, **left**, and **right** define the tree datatype). The natural way to express parallelism in this program is to indicate that the two recursive calls to **sum-tree** can proceed in parallel. This is accomplished by adding one **future**:

```
(define (psum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (future (psum-tree (right tree)))
         (psum-tree (left tree)))))
```

The natural expression of parallelism in this algorithm is rather fine-grained. With eager task creation this program would create 2^d tasks to sum a tree of depth d; the average number of tree nodes handled by a task would be 2. Figure 1.2 shows this partition pictorially; each circled subset of tree nodes is handled by a single task. Unless task creation is *very* cheap this partition will lead to poor performance.

What would be the *ideal* partition of this task tree? The ideal partition would maximize runtime task granularity while maintaining a balanced load. For a

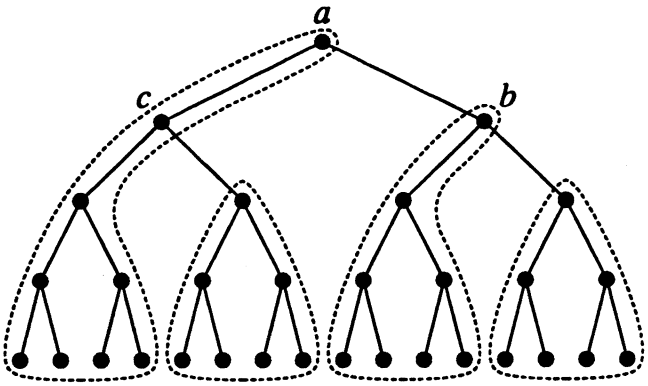


Figure 1.3: BUSD execution of `psum-tree` on four processors.

divide-and-conquer program like this one, that means expanding the task tree breadth-first by spawning tasks until all processors are busy and then expanding the tree depth-first within the task on each processor. This ideal partition will be called *BUSD*, for *B*readth-first *U*ntil *S*aturation, then *D*epth-first. Figure 1.3 shows a BUSD partition for a system with 4 processors.

How can this ideal partition be achieved? A parallelizing compiler might be able to increase granularity by unrolling the recursion and eliminating some `futures`, but in this example we *want* fine-grained tasks at the beginning so as to spread work as quickly as possible (breadth-first). The compiler might possibly produce code to do this as well if supplied with information about available processing resources, but making such a transformation general is a difficult task and would still have the parameterization drawbacks noted earlier.

What about specifying a partition directly in the source code? In Qlisp [Gabriel 84, Goldman 88], the programmer may control partitioning by supplying a predicate which, when evaluated at runtime, will determine whether or not a separate task is created.[4] `psum-tree` could be rewritten using Qlisp's `spawn` construct (equivalent to `future` with an additional predicate argument), yielding a program very similar in style to an example in [Gabriel 84]:

```
(define (psum-tree-2 tree cutoff-depth)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (spawn (> cutoff-depth 0)
           (psum-tree-2 (right tree)
                        (- cutoff-depth 1)))
         (psum-tree-2 (left tree)
                      (- cutoff-depth 1)))))
```

[4]One such predicate, `(qemptyp)` [Goldman *et al* 89], tests the length of the work queue, achieving the same effect as load-based partitioning.

In this example, `cutoff-depth` specifies a depth beyond which no tasks should be created. The predicate `(> cutoff-depth 0)` tells `spawn` whether or not to inline the recursive call. A `cutoff-depth` value of 2 would achieve BUSD execution similar to that shown in Figure 1.3; below level 2 all tasks are inlined.

This solution has two problems. First, the code has become more complex by the addition of `cutoff-depth`—it is no longer completely straightforward to understand what this program is doing. Second, supplying a value for `cutoff-depth` may require time-consuming experimentation. And the best value will vary with the amount of available processing resources, which as argued earlier may not be reliably known until runtime.

It appears that finding an acceptable static partition for this program introduces unattractive difficulties. With a dynamic partition on the other hand we may be able to achieve the BUSD partition attempted in `psum-tree-2` without sacrificing the clarity of `psum-tree`. How would execution of `psum-tree` proceed under each of our dynamic partitioning strategies?

First let us consider load-based partitioning. In an ideal run of `psum-tree` on a four-processor system with LBP the first three occurrences of `future` (at nodes *a*, *b*, and *c* of Figure 1.3) find that processors are free, and separate tasks are created (breadth-first). Depending on the value of the load threshold parameter T, a few more tasks might be created before the backlog becomes high enough to suppress task creation. But since there is a large surplus of work, most tasks are able to defray the cost of their creation by inlining a substantial subtree (depth-first).

Now consider an ideal run of `psum-tree` with lazy task creation. At the first potential fork point *a* no fork is performed so execution continues with the subtree rooted at *b*. But an idle processor immediately imposes a retroactive fork and takes over execution of the subtree rooted at *c*. Similarly, no forks occur initially at *b* and *c* but are made retroactively by the two remaining idle processors. Now all processors are busy so no further stealing takes place and each processor winds up executing one of the circled subtrees of Figure 1.3.

This execution pattern depends on an *oldest-first* stealing policy: when an idle processor steals a task, the oldest available fork point is chosen. In this example the oldest fork point represents the largest available subtree and hence a task of maximal runtime granularity.

It appears then that dynamic partitioning can solve the granularity problem for `psum-tree`—achieving a BUSD partition without the need for explicit partitioning—if actual implementations can live up to the ideal presentation above. Both lazy task creation and load-based partitioning will be examined much more closely in the coming chapters, after a few more introductory topics are covered.

1.4 Encore Mul-T

All three of the execution strategies under consideration—eager task creation, load-based partitioning, and lazy task creation—have been built into an implementation of Mul-T which runs on the Encore Multimax system, a shared-memory/shared-bus multiprocessor. Encore Mul-T is a fully realized system containing a well-integrated user interface which allows interactive error recovery in parallel programs.

A primary design goal (and achievement!) of Encore Mul-T was to show that a parallel Lisp system based on `future` could give "production-quality" performance [Kranz *et al* 89]. Previous implementations of Multilisp had been interpreter-based and had not performed well in absolute terms. In contrast, Mul-T uses the T system's Orbit compiler [Kranz *et al* 86, Kranz 88], a high-performance optimizing compiler. In addition, the Mul-T runtime system was carefully optimized to make (eager) task creation with `future` as cheap as possible.

The resulting system achieves its performance goals for medium to coarse-grained programs, showing that eager task creation can be quite acceptable. The overhead of creating, scheduling, executing, and retrieving the value from a task in Encore Mul-T is 143 instructions on the Multimax's NS 32000 Series processors. However, this overhead is definitely *not* acceptable for finer-grained programs (as will be seen in the performance figures of Chapter 6), motivating this investigation of alternative partitioning strategies.

The performance of ETC will figure strongly in evaluating the performance of the alternative strategies, so it is important at this point to cover a few details about the standard Encore Mul-T scheduler.

Each Mul-T processor (actually a UNIX process acting as a virtual processor) manages two task queues, its *new task queue* and its *suspended task queue.* When `future` is executed by a processor P a child task object is created and placed on P's new task queue. P also creates a placeholder object which is returned to the parent task as the result of the `future` call. If the parent task touches the placeholder before the child task has resolved it (or if the parent touches some other unresolved placeholder), the parent's state is saved and the parent task object is placed on a queue of blocked tasks associated with the placeholder. When the placeholder becomes resolved each blocked task is placed on the suspended task queue of the processor where it last executed.

Tasks run to completion unless they block. When a processor becomes idle because its current task either blocks or runs to completion the processor will search queues for a new task in the following order:

1. its local suspended task queue
2. its local new task queue
3. remote new task queues
4. remote suspended task queues

The purpose of the suspended task queue is to increase data locality and reduce cache turbulence by favoring suspended tasks over new tasks and resisting the migration of suspended tasks.

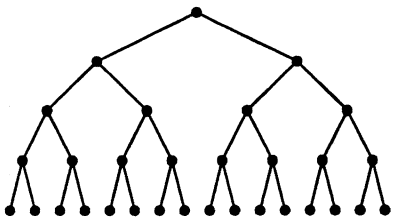
All task queue operations follow a last-in-first-out (LIFO) policy, so the most recently enqueued tasks are executed first. An optimization to this policy, as well as some other improvements to the standard Encore Mul-T scheduler, were discovered in the course of this work and are discussed in Section 4.1.

An understanding of the standard Encore Mul-T scheduler at the level presented here should suffice for the topics covered in this thesis.

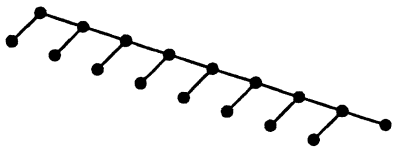
1.5 Scope

This work restricts its focus in two important ways. First, lazy task creation requires a shared-memory hardware model. However, the shared memory need not be a centralized resource; lazy task creation has also been implemented on distributed (and therefore scalable) shared memory machines as discussed in Section 7.3.

Second, dynamic partitioning is only effective when a program's task execution tree is fairly *bushy* as opposed to *spindly*:



A bushy task tree.



A spindly task tree.

Bushy task trees result for example from divide-and-conquer algorithms and recursive search algorithms; spindly task trees result for example from iteration over a linear data structure.

With both load-based inlining and lazy task creation the efficiency of a fine-grained program is only increased when most tasks are able to inline a substantial subtree of other tasks. With spindly task trees there are too few opportunities for inlining and runtime task granularity remains rather fine.

In evaluating the performance of LBP and LTC we will consider primarily fine-grained programs with bushy call trees. Ideas for increasing the efficiency of fine-grained programs with spindly task trees appear in Section 8.1.

Still, we will keep in mind throughout that dynamic partitioning must not *degrade* the performance of any program—even though efficiency may not be improved in programs with spindly task trees, we must ensure that efficiency is not worse than with eager task creation. If this requirement is met, a system may safely apply dynamic partitioning to all programs. Explicit partitioning may still be added to some programs (like fine-grained programs with spindly task trees) if performance is not satisfactory.

1.6 Overview of Main Ideas, Results, and Contributions

As an overview of the entire thesis this section summarizes specific ideas, results and contributions, with the appropriate section numbers shown in parentheses.

Fine-grained parallel programs are difficult to execute efficiently. Dynamic partitioning methods have the potential to overcome the limitations inherent in commonly used methods of static partitioning, if runtime overhead can be controlled. (1)

Load-based partitioning is an easily implemented dynamic partitioning method. But LBP has several drawbacks, not usually noted by other researchers. Because partitioning decisions are based only on momentary load level and are irrevocable, LBP can lead to bad load balancing and even deadlock. LBP also creates more tasks than necessary, and requires programmer involvement in setting the load threshold T. (2.1)

Lazy task creation addresses the drawbacks of LBP. With LTC no task is ever irrevocably inlined, solving the deadlock and load-balancing problems. LTC requires no parameterization and, with its oldest-first policy for task stealing, creates many fewer tasks than LBP. (2.2)

Although LTC has several advantages over LBP, LBP can be implemented simply with low runtime overhead, just 2 instructions per `future` call in Encore Mul-T. Cheap implementation of LTC is a challenge because of the need to save information at every potential fork point. In fact though, LTC's *lazy task queue* can be implemented simply as a block of pointers into a task's stack. Using a lockless algorithm to synchronize a task *producer* with stealing *consumers*, the overhead of a lazy future call and return in Encore Mul-T is reduced to only 8 instructions. (3)

In 3 areas, insight gained from studying LTC can be used to improve the performance of both eager task creation and LBP despite increasing the overhead of task creation. In particular, comparative performance measurements show that it is better to give scheduling preference to the child task than the parent task when forking even though the latter strategy requires fewer operations. (4.1)

Performance measurements of Encore Mul-T reveal the benefits of dynamic partitioning, as well as supporting numerous claims made in the text. Both LTC

and LBP perform substantially better than ETC. LTC creates substantially fewer tasks than LBP and performs better than LBP for all but one of the benchmark programs; the runtime overhead of LTC has been acceptably minimized. (6)

A study of all potential sources of overhead in the execution of a Mul-T program shows that the overhead of dynamic partitioning really *is* low, rather than just *looking* low because it is masked by overhead in the rest of the system. (5.1)

Other research efforts have adopted a strategy resembling LTC whereby the set of operations executed at potential fork points is reduced to only those needed if both branches are ultimately executed locally. But LTC is able to reduce the set substantially more than these other approaches because of the key design decision to delay the parent task rather than the child task at potential fork points. Thus the overhead of task creation is significantly lower with LTC than with these other approaches. (7.2)

The main contributions of this work are a detailed investigation of load-based partitioning and its drawbacks, a detailed development of the idea of lazy task creation, a robust and efficient implementation of LTC in Encore Mul-T, and thorough performance measurements demonstrating the effectiveness of dynamic partitioning.

Chapter 2

Dynamic Partitioning Methods Described and Compared

In this chapter load-based partitioning and lazy task creation are considered in more detail. Several drawbacks of load-based partitioning are identified and used as the basis for a qualitative comparison of the two strategies; quantitative comparisons will appear in later chapters.

2.1 Load-Based Partitioning

The idea behind load-based partitioning is to use the system's workload as a basis for forking decisions, by forking whenever the workload is *light* and inlining whenever the workload is *heavy*. When the workload is light processors are likely to be idle; forking supplies them with work and increases parallelism. On the other hand a backlog of work means plenty of parallelism is available and suggests that task creation overhead can be saved by inlining.

The idea of using system load information as a basis for dynamic partitioning is not new. Methods resembling load-based partitioning have been proposed for systems based on a variety languages from Lisp [Weening 89] to C [Gabber 90]. In most approaches as well as in the Mul-T implementation of load-based partitioning, task queue length is used as a measure of the system's current workload.

In Mul-T, a threshold parameter T indicates how many tasks must be queued before the system is considered to be loaded. `(future` X`)` means, "If the executing processor's local task queue contains fewer than T tasks, make a separate task to evaluate X; otherwise evaluate X in the current task." In fact, load-based partitioning can be implemented efficiently as a simple macro. Here is an example expansion:

```
(future (f x))  ⟹  (if (< (local-queue-length) T)
                       (eager-future (f x)) ;fork
                       (f x))               ;inline
```

(where **eager-future** guarantees that a separate task will be created). Local queue length is only an approximate measure of system load; however, using global information would require contention, introducing overhead which would increase in impact as more processors were added.

Load-based partitioning has an appealing simplicity, and as we shall see later it performs well for many programs. But there are several factors, not usually noted by other researchers, which decrease its effectiveness. A major factor is that partitioning decisions are irrevocable—once the decision to inline a task has been made there is no way to revoke the decision at a later time, even if it becomes clear at that time that doing so would be beneficial.

The following list summarizes the drawbacks of load-based partitioning; the following sections discuss each in turn.

1. LBP can cause deadlock in some programs.
2. LBP can significantly degrade the performance of some programs because of poor load balancing.
3. LBP creates many more tasks than are necessary for a good partition.
4. The programmer must decide when to apply LBP and may need to set the load threshold T.

2.1.1 Deadlock

Perhaps the most serious problem with load-based partitioning is that, for some programs, *irrevocable inlining is not a correct optimization.* Load-based partitioning can lead to deadlock because the decision to inline a task imposes a specific sequential evaluation order on tasks whose data dependencies might require a different evaluation order: the inlined child *must* complete before its parent can continue.

We shall consider two examples where load-based partitioning might cause deadlock in programs which are deadlock-free under eager task creation. The first example is straightforward: an inlined child waits for a semaphore which its "welded-on" parent will never be able to release. The second example, a prime-finding program, shows that deadlock is possible even in programs without explicit inter-task synchronization. If the wrong tasks are inlined a task testing

the primality of a number could deadlock trying to access divisor primes which haven't yet been computed by its welded-on parents.

The question of deadlock is related to the concept of fairness in scheduling; however, because the fairness issue will be more relevant when discussing a *solution* to the deadlock problem, a precise characterization of fairness assumptions is deferred until Section 2.2. A precise understanding of fairness assumptions is not required to understande the following examples.

The first example of deadlock, due to Halstead, appeared in [Kranz *et al* 89]:

```
(let ((s (make-semaphore))
      (x 0))
  (semaphore-p s)
  (let ((a (future (begin (semaphore-p s)
                          (+ x 1)))))
    (set! x (f 17))
    (semaphore-v s)
    (+ a 1)))
```

The parent creates a semaphore, acquires it, makes a `future` call, and ultimately releases the semaphore; meanwhile the child waits on the semaphore before proceeding. Deadlock would occur if the `future` call were inlined; the inlined child would block on the semaphore and the welded-on parent would never be able to reach the `semaphore-v` operation.

This example suggests that load-based partitioning and explicit inter-task synchronization do not mix well. A prudent programmer could respond by electing to forego load-based partitioning in programs with explicit synchronization; it is possible that dynamic partitioning would not be missed in such programs. But, as the next example shows, deadlock is possible even without explicit inter-task synchronization. Recognizing all programs where the potential for deadlock exists is not a simple task; further, there is no guarantee that some such programs might not require dynamic partitioning for good performance.

The second example `find-primes`, shown in Figure 2.1, is rather more complex than the first. It was not contrived to show deadlock however; this algorithm seemed the most natural way to write a prime-finding program in Mul-T. It uses a standard (and unsophisticated) prime-finding algorithm, checking each odd integer n for primality by looking for divisors among the primes found so far, up to $\sqrt{n}$. In addition to introducing parallelism `future` adds a flavor of lazy evaluation; primes are conveniently added to the end of the same list which holds the smaller primes used in divisor testing.

Initially, `all-primes` is bound to a lazily generated list of all the odd primes.[1] The function `find-primes>=n` generates a tail of `all-primes`—first, a `future`

[1] `delay`, a Scheme primitive which creates a placeholder object but does not spawn a task, is used instead of `future` to avoid a race condition in the `letrec` binding.

```
(define (find-primes limit)
  (letrec ((all-primes (cons 3 (delay (find-primes>=n 5))))
           (find-primes>=n
            (lambda (n)
              (if (> n limit)
                  '()
                  (let ((rest (future (find-primes>=n (+ n 2)))))
                    (if (prime? n all-primes)
                        (cons n rest)
                        rest))))))
    (cons 2 all-primes)))

(define (prime? n primes)
  (let ((prime (car primes)))
    (cond ((> (* prime prime) n)  #t)
          ((zero? (mod n prime))  #f)
          (else (prime? n (cdr primes))))))
```

Figure 2.1: Program `find-primes` could deadlock with load-based partitioning.

call is made to find all (odd) primes above n; second, n is checked for primality by walking down the list `all-primes` of primes already generated. If n is found to be prime it is added to the head of the local list of all primes.

With eager task creation a separate task is created to test each value of n for primality:

These tasks could be scheduled in any order, but a task testing a large value of n might block when walking down the list of known primes if the tasks testing small values of n had not yet resolved their placeholders. Here is one possible scenario during eager execution:

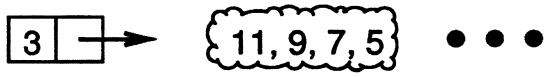
The tasks for $n = 3, 5, 9, 13, 15, 17$ have been executed, while the tasks for $n = 7, 11$ are still unresolved. 5, 13, and 17 have been found to be prime and added to the list; 9 and 15 have been found to be composite and have not been added.

No blocking has occurred; for example, the task for $n = 17$ was able to run to completion because only the first two elements of the list of primes (3 and 5) were needed to determine that 17 is prime.

Next, consider this possible execution snapshot of `find-primes` with load-based partitioning:

Inlining a `future` call in `find-primes` causes the parent task to test an additional value of n; if several successive `future` calls are inlined, a single task will test several values of n. The crucial point to glean from the above picture is that a task testing several values of n *tests the largest value first.* This happens because the parent computation to test n is not executed until *after* the inlined child computation (which tests $n + 2$) has returned.

If the wrong tasks are inlined, deadlock can result:



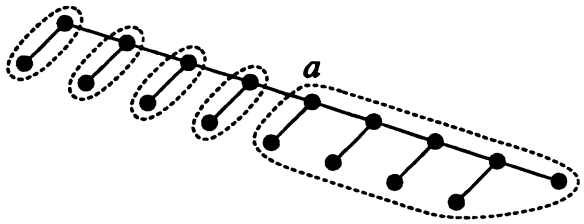
Here, inlining three `future` calls has created a task (call it T) to test $n = 11, 9, 7, 5$ in that order. This task would ultimately resolve a placeholder (the "cloud" in the above picture, call it P) with its return value, namely a list containing 5, 7, 11, and a second placeholder for the rest of the list. T's first act will be to call `(prime? 11)`, which will attempt to access the second element of `all-primes`. But this access will cause T to block on the unresolved placeholder P since the second element of the list has not yet been computed. Unfortunately, T itself is responsible for computing this second element (5). T has blocked on a placeholder which only T can resolve, so deadlock has occurred.

A dynamic partitioning method which could be used without the danger of deadlock would be preferable to load-based partitioning.

2.1.2 Load Balancing

Although designed to balance processor loads adaptively, load-based partitioning can actually degrade rather than improve the performance of some programs by creating decidedly unbalanced loads. Because partitioning decisions are based only on a momentary snapshot of system load, early decisions to inline tasks may cause processors to be idle at a later time when there are too few opportunities to create more tasks.

As an example of this effect, consider the following call graph of a coarse-grained iterative program executed on 4 processors with load-based partitioning:



Early partitioning decisions lead to task creation because system load is light; however, once all processors are busy the processor faced with a partitioning decision at node *a* chooses to inline, and proceeds to quickly do the same down the remainder of the spine. Once the other processors have finished their tasks they must remain idle as no more opportunities for task creation exist and the tasks already inlined are inaccessible. The result is bad load balancing and poor performance.

With eager task creation for the same call graph a task would be created at each fork point. The coarse task granularity means there will be plenty of parallelism and that task creation overhead will be minimal, leading to good performance. So in this case ETC would significantly outperform LBP.

In Chapter 6 we will measure the performance of a program (`fatwalk`) with this type of call graph. Tasks in that program are not partitioned exactly as depicted above, but the general idea is the same; the result is that performance is degraded with LBP compared to ETC.

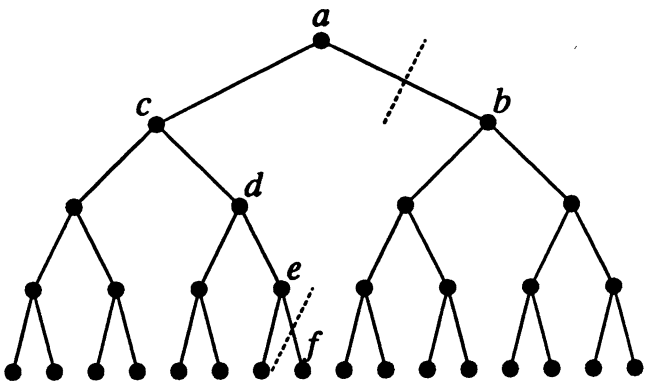
Load-based partitioning can also lead to bad load balancing in another way, although this second effect was not noticeable in any of the benchmarks considered here. If an inlined child becomes blocked waiting for a placeholder to resolve (or some other event), the parent is blocked as well and is not available for execution. As above, this can result in idle processors.

A dynamic partitioning method which did not compromise load balancing would be preferable to load-based partitioning.

2.1.3 Number of Tasks Created

Although load-based partitioning reduces task creation overhead by eliminating many tasks, the resulting partition falls short of the ideal BUSD partition described earlier. The overall pattern of the partition resembles BUSD somewhat: early on system load is light and tasks are created breadth-first; later on system loads are heavier and more tasks are inlined for depth-first execution. But, deviation from BUSD arises because forks can occur at essentially random spots in the call tree.

To see how this can happen, consider the following scenario of a call tree expanded using load-based partitioning with a load threshold of $T = 1$:



Say the executing processor P begins with an empty task queue so that the first partitioning decision, at node a, results in task creation (indicated by the dashed line separating nodes a and b). A task representing the subtree rooted at b is queued and P continues execution with node c. Because one task is now queued the partitioning decision at node c leads to inlining, and execution continues with node d. Inlining is chosen there as well, and we proceed to node e. Now suppose that at this point another processor becomes idle and removes the single task from P's queue. Since the queue is now empty, P's next partitioning decision at node e will result in task creation—but, the task created will evaluate only the single node f!

Because partitioning decisions are based only on momentary load level the call tree position of actual forks is largely a matter of chance, determined only by the timing of steal operations. Since the majority of potential fork points lie toward the leaves of the call tree, the tasks created by actual forks are more likely to represent small subtrees. Thus more tasks are created than would be created in an ideal BUSD division.

The behavior of load-based partitioning for programs with complete binary call trees has been analyzed theoretically by Weening [Weening 89, Pehoushek & Weening 89]. He assumes, as here, that each processor maintains its own local task queue and that partitioning decisions are based only on the local queue's length. He shows two additional ways in which the need to maintain at least one task on the local queue leads to non-BUSD execution. First, a lone processor P executing a subtree of height h (with a load threshold of $T = 1$) creates h tasks instead of just one; second, stealing a task from P's queue at an inopportune moment (as above) can lead to the creation of $O(h^2)$ tasks. He derives an upper bound of $O(p^2 h^{3+T})$ tasks using p processors.

Weening points out that this bound guarantees asymptotically minimal task creation overhead as the problem size grows exponentially in h. The unstated implication is that the overhead of task creation with load-based partitioning is not an issue. While the asymptotic argument is technically true, the performance measurements in Chapter 6 show that reducing the number of tasks created by

load-based partitioning would in fact measurably improve the performance of actual programs.

It is possible that using one central queue instead of several distributed queues would decrease the number of tasks created, but the contention introduced by this alternative would probably be unacceptable and would certainly not be scalable.

A dynamic partitioning method which more closely approximated a BUSD partition would be preferable to load-based partitioning.

2.1.4 Programmer Involvement

The final drawback to load-based partitioning is that even though LBP is an automatic mechanism, programmer control is still required in two ways. First, the dangers of deadlock and performance degradation mean that LBP is not safe in all programs; the programmer must therefore specify on a case-by-case basis whether or not load-based partitioning should be applied. Second, the programmer must supply a value for the threshold parameter T. Pehoushek and Weening found that the best value for T varied substantially among programs and was difficult to determine except by experimentation [Pehoushek & Weening 89]. Less variation was observed in the programs studied here; all but one ran best with $T = 2$, with $T = 1$ performing better in the one case (see Chapter 6).[2]

The extra work required to invoke LBP and set the threshold parameter is unappealing in itself; further, the issue of how best to exercise this control is also problematic. For example, should control be exercised over individual `future` calls, or over entire programs? Annotating individual `future` calls offers the most control but also complicates source code (*e.g.*, should `future` take an extra parameter?) and contradicts the philosophy that the programmer shouldn't have to specify *how* partitioning occurs. Making one specification for an entire program (as is done via a compiler switch in the Encore Mul-T implementation of LBP) is simpler, but may not offer enough control. For example, a single program could contain instances of `future` which required different settings.

A dynamic partitioning method valid for all programs and requiring no parameterization would be preferable to load-based partitioning.

2.2 Lazy Task Creation

We now consider lazy task creation in more detail to evaluate whether it can address the drawbacks of load-based partitioning. We need an alternate method of dynamic partitioning which:

[2]Pehoushek and Weening present results for three programs. Of these three, only the synthetic benchmark `tak` had an optimal threshold higher than $T = 2$; in `tak` $T = 5$ through $T = 11$ gave the best speedup. My benchmark suite does not contain `tak`, but in brief experiments using LBP it showed similar behavior.

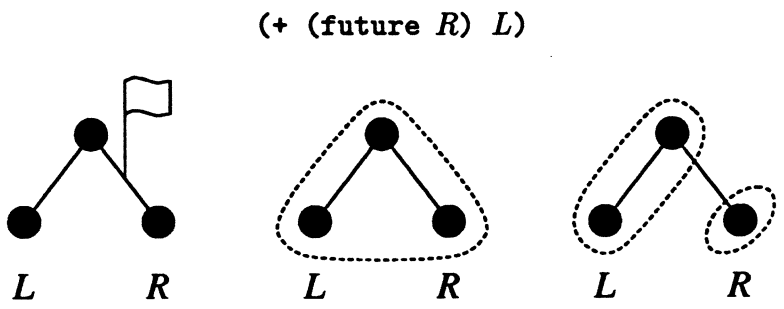


Figure 2.2: Three lazy task creation scenarios.

- Won't introduce deadlock
- Won't hurt load balancing
- Creates fewer tasks than LBP
- Requires no parameterization

With lazy task creation, as with load-based partitioning, `future` is viewed as a potential fork point. But with LTC partitioning decisions are made lazily rather than being made when `future` is executed. When executing `future`, LTC always chooses *not* to fork, but saves enough information so that forks can be made retroactively later.

Consider Figure 2.2, which shows a task tree for the expression

```
(+ (future R) L)
```

in three possible states. The leftmost picture shows an initial stage—the executing processor P has begun computing the right branch R but has flagged this branch as representing a potential fork point. The center picture shows what happens if P finishes computing R without any intervention by idle processors: P ends up computing the entire expression, thereby saving the overhead of task creation. The rightmost picture shows what happens if instead an idle processor I finds the flagged fork point while P is still executing R. I makes a retroactive fork, creating a task to execute the rest of the expression; this task will compute the left branch L and sum the results of R and L.

The key feature of lazy task creation *vis-à-vis* load-based partitioning is that *no task is ever irrevocably inlined.* Although every task is inlined initially, this decision can always be revoked by a retroactive fork. As will be discussed in detail below, this laziness addresses the problems of deadlock and load balancing found with LBP.

In addition, when lazy task creation is augmented with an *oldest-first* scheduling policy, many fewer tasks are created than with LBP. Consider Figure 2.3,

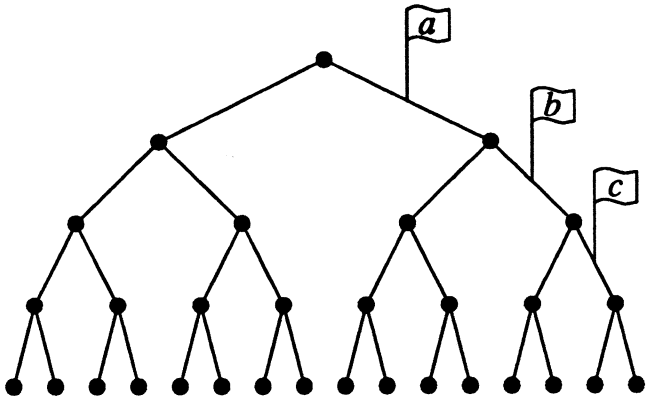


Figure 2.3: With oldest-first scheduling a retroactive fork would occur at *a*.

where executing three instances of **future** has led to three potential fork points flagged as *a*, *b*, and *c*. With the oldest-first policy, a stealing processor would choose *a* rather than *b* or *c* as a retroactive fork site, thereby transferring the largest possible subtree and creating a task with maximum potential granularity.

Let us now consider how well lazy task creation addresses each of the drawbacks of load-based partitioning.

2.2.1 Deadlock

The type of deadlock shown with load-based partitioning can never arise with lazy task creation. Deadlock arose in both examples when a parent and child task were irrevocably welded together by an unfortunate partitioning decision; with LTC a parent and child can always be decoupled. In fact, we shall see that any program that is deadlock-free with eager task creation is also deadlock-free with lazy task creation.

A defense of these claims requires a well-defined notion of what standard of fairness in scheduling is promised by the semantics of the underlying language. Halstead's ideas of fairness in Multilisp, as published in [Halstead 89] and communicated privately, form the basis for the discussion here.

Both the ETC and LTC schedulers distinguish two task states—tasks are either *runnable* or *blocked*. A task becomes blocked by touching an unresolved placeholder or by attempting to acquire an unavailable semaphore. Such a blocked task becomes runnable only when the semaphore is acquired or the placeholder becomes resolved. All tasks which are not blocked are in the runnable state, including those actually being executed by a processor.

For the purposes of this discussion, *deadlock* describes the state where no task is able to make progress. A blocked task is by definition unable to make progress; in addition, a runnable task which is busy-waiting is not making progress.

The ETC scheduler makes only a very weak guarantee of fairness which may be stated thusly:

- If at least one task is runnable, at least one runnable task will run.

In order to write programs that are deadlock-free under this weak guarantee, programmers must ensure that their programs fulfill two conditions:

- There is always at least one runnable task.
- Every runnable task will make progress if run.

The first condition reflects the fact that some algorithms can deadlock under even the strongest guarantees of fairness (*e.g.*, a one-task program performing two consecutive P operations on a binary semaphore); as in any system, programmers must use deadlock-free algorithms. The second condition is more restrictive, and has the effect of prohibiting busy-waiting. The ETC scheduler guarantees only that at least one runnable task will run, so it could choose to run a busy-waiting task forever and no other task. The programmer is responsible for ensuring that this cannot happen. If a program fulfills these two conditions it will be deadlock-free, since the addition of the scheduler's guarantee achieves the assurance that at least one task will always make progress.

The LTC scheduler makes the same weak guarantee of fairness stated above for the ETC scheduler, and maintains analogous sets of runnable and blocked tasks. In addition, the LTC scheduler maintains a set of *stealable continuations*, and adds a second guarantee:

- If no task is runnable and the set of stealable continuations is non-empty, a runnable task will be created to execute one of the stealable continuations.

In order to compare the ETC and LTC schedulers on the deadlock issue we must describe how Mul-T language constructs cause tasks to move among the scheduler states described above. Actually we need discuss only `future`, as the ways in which constructs such as `touch`, `resolve-placeholder`, `semaphore-p`, and `semaphore-v` cause task transitions between running and blocked states are the same for the ETC and LTC schedulers. In particular, note that when a task becomes blocked with LTC, the set of stealable continuations is not affected.

Let E be the expression (`future` X), let K be its continuation, and let T be the (runnable) task executing it. When E is executed with ETC, a second runnable task is created to execute X; the original task T remains runnable and will execute the continuation K. When E is executed with LTC, the original task T remains runnable and will execute X; K is added to the set of stealable continuations. If K is still in the set of stealable continuations when X completes, K is removed from the set and executed in task T.

Armed with all these definitions, we may now prove the following:

Theorem 1 *Any program P that is deadlock-free with eager task creation is also deadlock-free with lazy task creation.*

Proof (by contradiction): We shall assume that executing P with LTC causes deadlock and show that this assumption contradicts the given axiom that P is deadlock-free with ETC. Note that this axiom means that no possible task execution order can produce deadlock with ETC.

Assume that executing P with LTC has caused deadlock, and consider the entire set of LTC scheduler decisions that led up to the deadlock. We can construct a set of ETC scheduler decisions which exactly parallels the LTC decisions, so that the order of computation under both schedulers is the same.

As noted earlier, task transitions between running and blocked states are the same for the ETC and LTC schedulers. Thus to construct a matching execution order we must merely specify which tasks the ETC scheduler should choose to execute in each of three scenarios involving the execution of `future`:

1. Where with LTC the expression (`future` X) is executed by flagging its continuation K as stealable and beginning execution of X, the ETC scheduler's matching decision will be to queue the task for K and then create and execute the task for X.

2. If with LTC X returns before K is stolen, the corresponding ETC task computing X will complete and the ETC scheduler will choose to dequeue and execute the task for K.

3. If with LTC a processor completes execution of its current task and steals K, the corresponding processor with ETC will complete the corresponding task and the ETC scheduler will choose to dequeue and execute the task for K.

These guidelines are sufficient to ensure that the chosen order of computation with ETC exactly matches the deadlock-producing order of computation with LTC. In particular, there are always as many runnable tasks with the chosen ETC execution as there are runnable tasks plus stealable continuations in the LTC execution.

Now consider the state of the LTC scheduler when deadlock occurs. By the definition of deadlock no task can make progress; but, since P is deadlock-free with ETC we know that no task can be busy-waiting, so it must be that there are no runnable tasks. Since in such a state the LTC scheduler converts stealable continuations to runnable tasks, there must also be no stealable continuations. At the corresponding point in the chosen ETC ordering then, the ETC scheduler must also have no runnable tasks because of the task count equivalence argued in the previous paragraph.

Thus we see that the chosen ETC execution order must also lead to deadlock. But this contradicts the original axiom that P is deadlock-free with ETC. Therefore, our original assumption must be false and P must be deadlock-free with LTC as well. ∎

The following quote from [Halstead 89] is an appropriate postscript to the fairness discussion:

> The weakness of Multilisp's scheduling guarantees may seem surprising when compared with the specifications of other schedulers, such as those for operating systems or real-time control. It is acceptable for Multilisp's scheduling semantics to be weaker because the only design goal for the basic Multilisp language is to execute mandatory parallel computations quickly. This definition of Multilisp's mission does not include goals that other schedulers must meet, such as apportioning a computer system's resources fairly among multiple users or juggling tasks of differing priorities.

The decision to adopt a weak definition of fairness for Mul-T is not radical; many systems supporting lightweight tasks have similar goals to those described for Multilisp and a similar (if often implicit) definition of fairness. Because preemptive scheduling is expensive, many systems adopt the policy that a task will run to completion uninterrupted unless it becomes blocked on a system-provided primitive. Such a policy implicitly carries a weak guarantee of fairness in scheduling.

Feeley's Multilisp system [Feeley 91] makes a strong guarantee of fairness in scheduling, at some cost in runtime scheduling overhead.

2.2.2 Load Balancing

We saw how performance can suffer with LBP when unfortunate partitioning decisions leave one processor with many pending inlined continuations while other processors are idle. In contrast, performance of such a program would not suffer with lazy task creation because idle processors can always steal continuations by making retroactive forks.

In a program with a relatively small number of coarse-grained tasks ETC is likely to perform well because the overhead of task creation is negligible. But the small number of tasks means that performance will be quite sensitive to load balancing; combining tasks irrevocably as with LBP can hurt performance. With LTC the capability of making retroactive forks allows load balancing to be as good as it is with ETC.

Because idle processors can always make retroactive forks, LTC can be used with any program without the danger that load balancing will suffer. This con-

clusion is supported by performance figures for the program **fatwalk** in Section 6.6.1.

2.2.3 Number of Tasks Created

We saw how load-based partitioning creates more tasks than necessary because partitioning decisions are based only on momentary load level. A much better alternative is the oldest-first scheduling policy used with lazy task creation; as will be seen in Section 6.5, LTC results in many fewer tasks than LBP. Tasks created by oldest-first scheduling tend to encompass larger subtrees, giving a much better approximation to BUSD execution.

That LTC creates fewer tasks than LBP may also be seen by a theoretical analysis.

Theorem 2 *With lazy task creation and p processors, executing a program with a perfect binary task tree of height h results in the creation of no more than p^2h tasks.*

When compared with Weening's result of $O(p^2h^{3+T})$ tasks for load-based partitioning [Weening 89], this complexity figure shows that LTC creates theoretically fewer tasks than LBP.

A proof of this theorem depends on an important fairness claim relating to the scheduler's policy for stealing tasks. If a processor P_i steals a task from another processor P_j, P_i must poll all other processors for stealable tasks before stealing from P_j again.

This policy is known as *polite stealing* and is discussed more fully in Section 4.1.3. Such a fairness policy is also crucial to Weening's analysis but is not stated explicitly in [Weening 89].

The theorem may now be proved, by a method roughly following Weening's.

Proof: At any point in the computation, each processor P_i is either idle or executing a subtree of height H_i, called its *local height.* The maximum of the H_i's is H, the *global height.* We will count the number of steals necessary to reduce the global height in h steps from h to 0.

With lazy task creation using oldest-first scheduling, stealing a task reduces a processor P_i's local height H_i by 1. If $H_i = H$ and $H_i > H_j \ \forall\, j \neq i$ (that is, if only P_i had the global height), then the global height is also reduced by 1.

What is the maximum number of steals s that can happen without reducing the global height H? If the global height is to be the same after s steals then some processor P_i must have height H before and after the steals. Each of the other $p-1$ processors could steal from all remaining processors except P_i, making a total of $s = (p-1)(p-2)$ steals. After these s steals, $H_j < H \ \forall\, j \neq i$; that is, only P_i can have a subtree of the global height. Since all processors except P_i have

stolen from all other processors except P_i, the polite stealing policy guarantees that the next steal has to be from P_i, reducing the global height by 1. (Or, P_i might reduce the global height by 1 itself by finishing half its tree and starting in on the other half.)

Since p^2 steals suffice to reduce the global height by 1 and since the global height must be reduced by 1 h times, a maximum of p^2h tasks will be created. ■

2.2.4 Programmer Involvement

Lazy task creation requires no parameterization and may be safely used with any program, so programmers need exercise no additional control.[3] In particular, the debate of how to specify when dynamic partitioning should be used (should control be exercised at the level of individual fork points or entire files?) is moot because it can be used safely on all programs.

The last four sections have shown that lazy task creation addresses each of the drawbacks of load-based partitioning. These arguments will be further supported by performance figures in later chapters; the next chapter paves the way for these performance experiments by considering several important topics in the implementation of lazy task creation.

[3]One situation requiring programmer awareness arises because of the design choices made in the Encore Mul-T implementation of LTC and will be discussed in Section 4.3.

Chapter 3

Lazy Task Creation: Data Structures, Algorithms, and Implementation

We have seen that lazy task creation has several strong advantages over load-based partitioning. But, load-based partitioning can be implemented simply with low runtime overhead, and in fact gives good performance for many programs. The question becomes: can lazy task creation be implemented efficiently enough to perform *at least as well* as load-based partitioning?

Both dynamic methods increase efficiency by converting only selected instances of `future` to actual forks. But with lazy task creation enough information must be saved at every `future` call to allow a retroactive fork at that point; minimizing the cost of maintaining this information is critical to the efficiency of LTC. By comparison the cost of actually stealing a task is somewhat less critical since in a fine-grained program few potential fork points will be converted to actual forks. Still, we would prefer that stealing a task with LTC have comparable cost to creating a task with ETC.

This chapter presents details of the algorithms, data structures, and implementation of lazy task creation, both abstractly and for Encore Mul-T. First, a brief summary.

The primary data structure for LTC is the *lazy task queue*, a double-ended queue supporting push, pop, and steal operations. In Encore Mul-T the lazy task queue is implemented simply as a block of pointers into a task's stack; by using a lockless synchronization algorithm the crucial operations of lazy future call and return can be performed very efficiently. Salient details are given for all LTC operations, including lazy future call/return, stealing a task, and blocking on an unresolved placeholder.

In the following sections an execution of `future` with lazy task creation is known as a *lazy future call*; a processor making lazy future calls is called a

producer of tasks and processors making retroactive forks by stealing these tasks are known as *consumers.*

3.1 Why Implementation is a Challenge

To see why it might be difficult for LTC to match the efficiency of LBP we will borrow some figures from the discussion of runtime overheads to be presented in Section 4.2. To achieve comparable efficiency we must ensure that the runtime overhead introduced by LTC is no greater than the runtime overhead introduced by LBP.

With both methods `future` is treated as a potential fork point so that in a fine-grained program only a small fraction of all `future` calls result in task creation. Thus the runtime overhead of a method has two components: the cost incurred at potential fork points and the cost of actually creating tasks.[1] This overhead O may be expressed as:

$$O = N_{fp}C_{fp} + N_{tc}C_{tc}$$

That is, as the number of potential fork points N_{fp} times the cost per fork point C_{fp} plus the number of tasks created N_{tc} times the cost of task creation C_{tc}. Using this equation together with the known cost of load-based partitioning in Encore Mul-T and some assumptions about lazy task creation, we can get a rough idea of the implementation constraints for lazy task creation.

With load-based partitioning the cost per fork point C_{fp} is low, just 2 instructions to test the local queue length, while C_{tc} is the same as with eager task creation, 133 instructions. And in a typical program roughly 10% of fork decisions lead to task creation (see Section 6.5). With lazy task creation, let us assume that only 1% of fork decisions lead to task creation and that C_{tc} is roughly the same as with eager task creation.

From all of this we may conclude that if lazy task creation is to perform at least as well as load-based partitioning, the cost per fork point of lazy task creation must not exceed 14 instructions.

What must be accomplished in these 14 instructions? A task encountering `future` must save enough information to allow lazy task creation, that is, enough that an idle processor may make a retroactive fork. One branch of the potential fork must be packaged up so that execution can begin when a processor becomes idle. This operation is somewhat similar to lazy *evaluation*, where one branch of a computation is packaged up so that execution can begin when its value is demanded.

[1] Other runtime overheads are not considered here because their costs are comparable for the two methods; see Section 4.2.

Performing the LTC packaging operation in 14 instructions begins to seem challenging indeed when one considers that the comparable operation with lazy evaluation costs about 60 instructions![2] But as we shall see there is an important difference—with lazy task creation the *parent* branch must be packaged up while with lazy evaluation it is the *child* branch that is packaged up. It is this key difference which will allow efficient implementation of LTC.

This difference also sets LTC apart from methods developed by other researchers where the child is packaged, discussed in Section 7.2. When the child is packaged certain bookkeeping operations just cannot be eliminated, but the parent can be packaged very economically.

3.2 The Lazy Task Queue

Packaging up the parent turns out to be very easy because in most cases it requires no work at all! The important observation is this: *After a lazy future call, the stack contains all information necessary for a retroactive fork.*[3] For example, consider again the program `psum-tree` shown in Figure 3.1, which also shows a partial call tree and a stack (growing downward). Imagine that we are running `psum-tree` with lazy task creation—at each `future` we provisionally choose not to fork, instead making the recursive procedure call `(psum-tree (right tree))` in the current task.[4] Before making the first such call ("call 1" in Figure 3.1), we must save on the stack the context necessary to continue execution after returning from the call. Or, in the parlance of continuations, we must push the continuation to call 1 on the stack. The frame labelled A contains all information necessary to execute the continuation to call 1; that is, to sum the left subtree and add the two partial sums. Likewise, after call 3 the frames labelled A, B, and C contain all information necessary to execute the continuation to call 3.

This stacked information is exactly what is required for a retroactive fork. For example, an idle processor could use the information stored in frame A to begin executing the left part of the call tree while the original processor was still executing the right part. The only missing link is identifying which points in the stack represent possible fork points; that is, continuations to lazy future calls.[5]

For this purpose we define the *lazy task queue*, whose structure and operations are shown abstractly in Figure 3.2. It is a double-ended queue where each entry

[2]With some optimizations to the current implementation, evaluating an expression lazily using `delay` in Mul-T would add about 60 instructions of overhead.

[3]An exception is dynamic binding information, discussed in Section 3.3.6.

[4]To simplify the presentation I assume once again that + evaluates its arguments left-to-right.

[5]I refer to continuations here and elsewhere only for clarity; packaging up the parent conveniently does not depend on the implementation technique of continuation-passing style used in [Kranz *et al* 86].

```
(define (psum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (future (psum-tree (right tree)))
         (psum-tree (left tree)))))
```

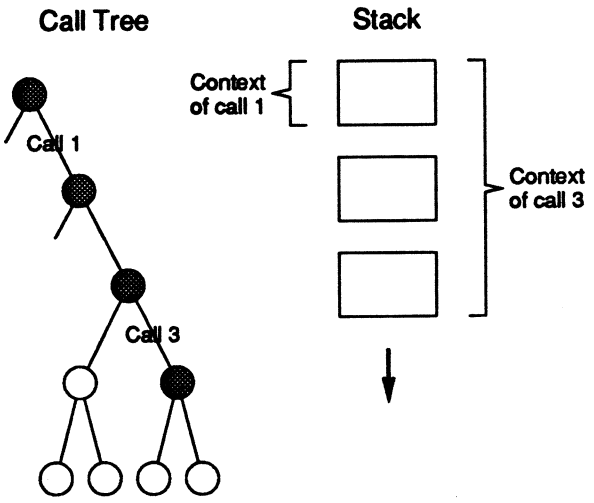


Figure 3.1: Stack contains necessary context for retroactive fork.

points to a stack frame representing a potential fork point. A task making lazy future calls (called the *producer*) pushes and pops items from the tail of the queue, while idle processors (called *consumers*) steal tasks from the head of the queue. Note that the oldest task is at the head (top) of the queue while the newest task is at the tail (bottom) of the queue.

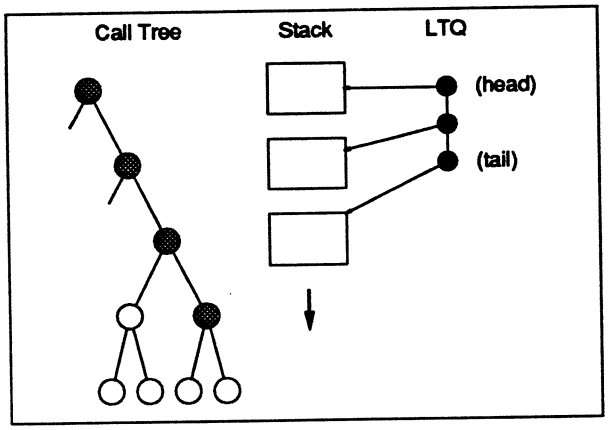
In Figure 3.2a, the producer has made 3 lazy future calls and is executing the 4th shaded node of the call tree. Figure 3.2b shows the situation after the producer makes a 4th lazy future call, executing the code

```
(future (psum-tree (right tree)).
```

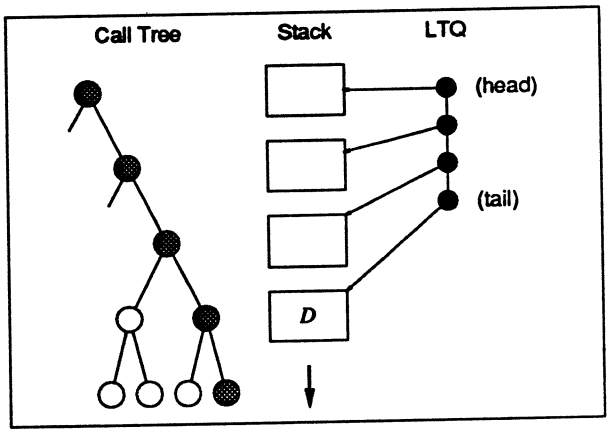
The producer pushes a new frame D on the stack, representing the call's continuation

```
(lambda (rsum)
  (+ rsum (psum-tree (left tree)).
```

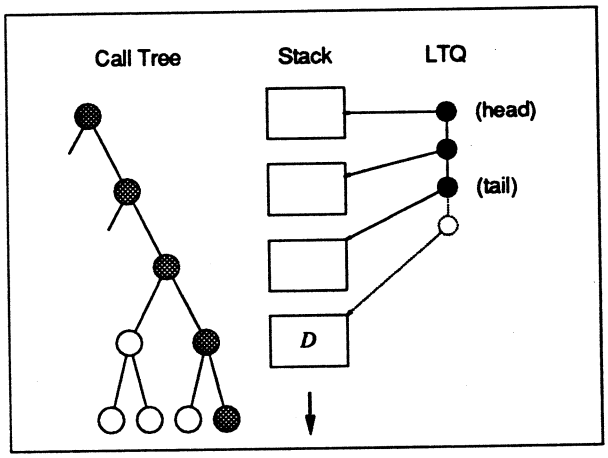
The producer also pushes a pointer to D onto the tail of the lazy task queue, and begins executing the bottom rightmost tree node.



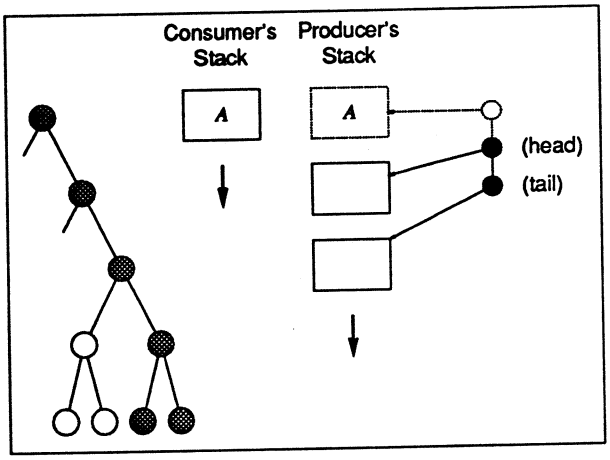
(a) The lazy task queue.



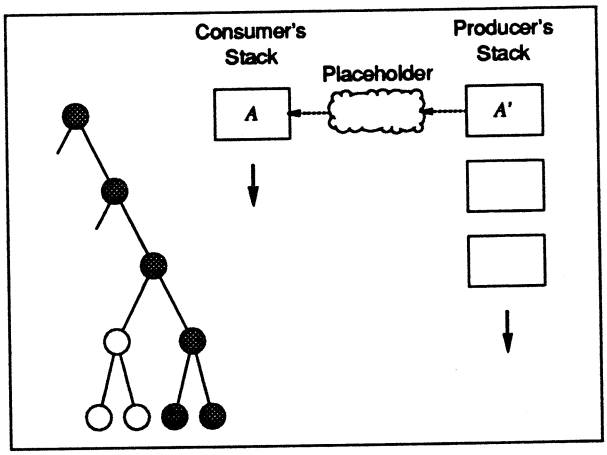
(b) Lazy future call.



(c) Return from lazy future call.



(d) Stealing a task.



(e) Placeholder communicates value.

Figure 3.2: Lazy task queue data structures and operations.

In the next scene (Figure 3.2c), the producer returns from the lazy future call before any stealing has occurred. The producer pops the tail of the lazy task queue, returns its result (calls the continuation represented by frame D), and begins to sum the left subtree. Lazy task creation has saved the expense of creating a separate task!

Next an idle consumer steals a task from our producer's lazy task queue, by removing the oldest part of the producer's stack (Figure 3.2d). The head item of the lazy task queue points to frame A, representing the continuation to the very first lazy future call. The consumer pops the head of the lazy task queue, moves frame A from the producer's stack to its own stack, and calls the stolen continuation. The consumer has made a retroactive fork!

But we've skipped an important detail. What will happen when the producer finishes summing the entire right subtree and tries to return a value `rsum` to frame A? The consumer has stolen A, so `rsum` must somehow be passed to the consumer. This communication happens through a placeholder P, shown in Figure 3.2e. When stealing, the consumer must replace frame A on the producer's stack with a new frame A', representing an alternate continuation which will resolve the placeholder:

```
(lambda (rsum)
  (resolve-placeholder P rsum)).
```

When the consumer begins executing the stolen continuation, the unresolved placeholder is used in lieu of an actual return value. When the producer eventually returns, `rsum` will resolve the placeholder.

Figure 3.2e shows the completed steal operation. The configuration of stacks after creating a task lazily looks exactly like the configuration of stacks after creating a task eagerly—one processor evaluates the child (the producer, summing the right subtree) and passes its result via a placeholder to another processor which is evaluating the parent (the consumer, summing the left subtree and adding the results).

This discussion has omitted the details of synchronization between producer and consumer; in fact, two kinds of race condition are possible during the stealing operation. First, two consumers may race to steal the same continuation; second, a producer trying to return to a continuation may race with a consumer trying to steal it. These details will be discussed in the next section. But an important feature of the stealing operation is evident even now: *the consumer never interrupts the producer.*

3.3 Encore Implementation

The lazy task queue and associated operations described abstractly in the previous section may be implemented in several ways; the remainder of this chapter

describes a successful implementation for the Encore Multimax. Implementations have also been built for the Butterfly multiprocessor [Feeley 91] and the experimental ALEWIFE machine (by David Kranz, reported in [Mohr *et al* 91]); more will be said about these implementations in Chapter 7.

The Multimax is a shared-bus shared-memory multiprocessor. Yale's Multimax system has 18 processors; the National Semiconductor 32332 processors used have relatively few general-purpose registers (8) but fairly powerful memory addressing modes. An atomic hardware test-and-set operation provides the fundamental means of inter-processor synchronization; however, the machine's multi-cache coherency also allows flag-based synchronization.

As suggested earlier, minimizing the overhead of lazy future call and return is critical to an acceptably efficient implementation of lazy task creation; the implementation strategies presented in the ensuing sections strongly reflect this concern. In Encore Mul-T a task's stack and lazy task queue share a contiguous section of the heap, allowing a lazy task queue entry to be represented very simply as a pointer into the stack. This representation of the lazy task queue still allows proper synchronization between producers and consumers—a novel *lockless* algorithm is presented (with a correctness proof) which obviates the need to store a lock with each lazy task queue entry. Using the lockless synchronization algorithm the critical lazy future call and return sequence can be implemented with only 8 instructions of overhead, comfortably below the goal of 14 instructions identified in Section 3.1. The algorithms for stealing a task and blocking on an unresolved placeholder are straightforward and are presented last.

3.3.1 Data Structures: The Stack and Lazy Task Queue

In "standard" Encore Mul-T each task has an associated stack, stored in a contiguous section of the heap. If a stack overflows its allocated size, its contents are copied to a new stack of twice the original size. How must this basic data structure be altered to support lazy task creation? As seen in Section 3.2, stealing a task requires the producer's stack to be split into two pieces. If we stick with a conventional contiguous-memory stack representation, stealing a task will require a consumer to copy frames from the producer's stack to its own stack. This seems undesirable at first glance because the cost of a steal operation is unbounded. However, as detailed in Section 4.3, it turns out that this copying does not in fact add significant overhead. Sticking with a conventional contiguous-memory stack representation is in fact quite a viable design choice, and is the one chosen for the Encore implementation. An alternative linked-frame stack representation for stacks in support of lazy task creation has been built as part of the ALEWIFE project, and is discussed in [Mohr *et al* 91].

For lazy task creation each stack must have an associated lazy task queue. Rather than creating a separate data structure of potentially unbounded size, the

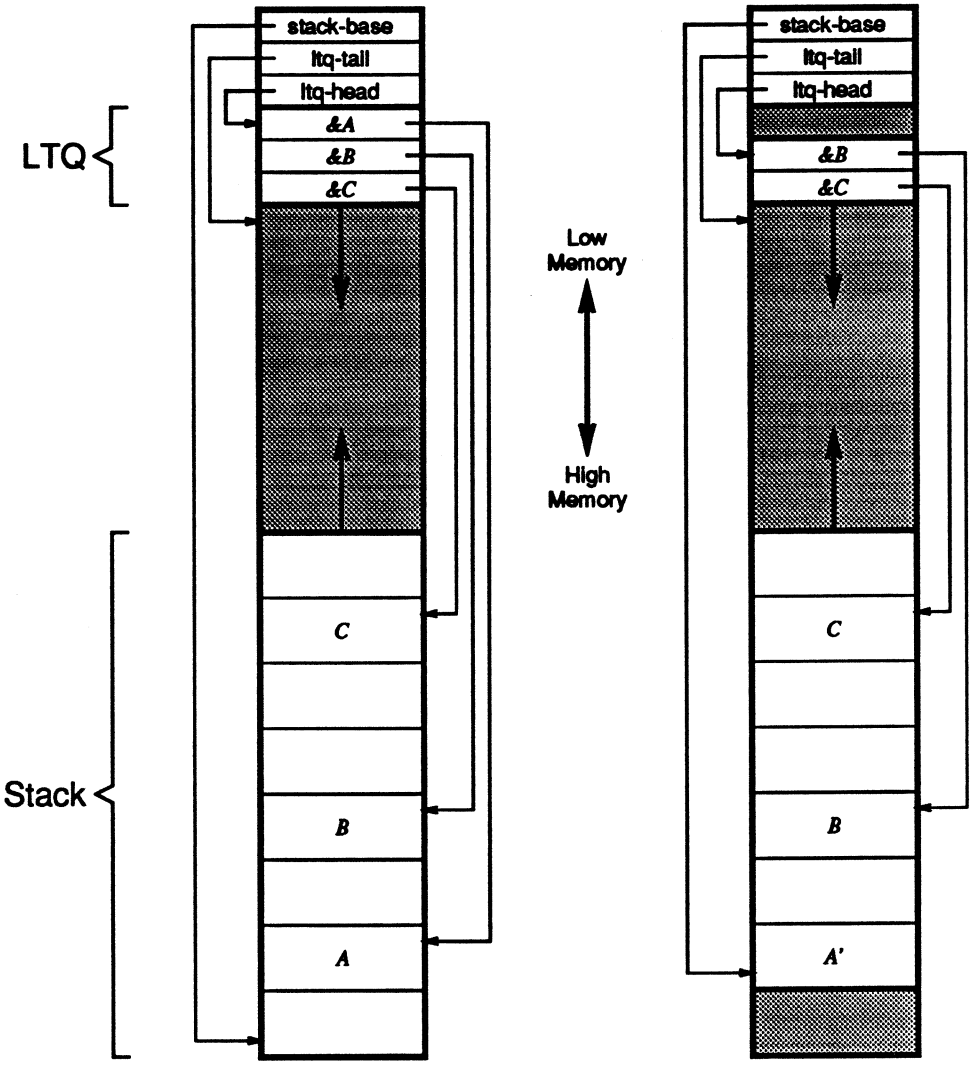


Figure 3.3: Stack and lazy task queue in Encore Mul-T, before and after steal.

lazy task queue is stored in the same heap object as the stack (known henceforth as the *stack/ltq object*). This combined stack and lazy task queue object is shown in Figure 3.3; the stack grows upward (from high memory to low) while the lazy task queue grows downward. Consider the left hand stack/ltq object, whose stack contains 8 frames. 3 of these, labelled A, B, and C, are continuation frames of lazy future calls and thus represent potential fork points. Each fork point is identified by a one-word entry in the lazy task queue which points to the stealable stack frame; for example, lazy task queue entry $\&C$ contains the address of the top word of frame C. Storing the lazy task queue in contiguous memory allows efficient push and pop operations, which is important to minimizing the overhead of lazy future call and return.

Three important pointers are stored in the header of the stack/ltq object. `ltq-head` points to the head of the lazy task queue, which represents the oldest potential fork point (A in this case). `ltq-tail` points just beyond the tail of the lazy task queue, which represents the newest potential fork point (C in this case). Finally, `stack-base` points to the bottom word of the stack.

An important feature of this implementation of the lazy task queue is that only the producer modifies `ltq-tail` (during lazy future call and return) and only consumers modify `ltq-head` (when stealing tasks). This key observation leads to the lockless synchronization algorithm presented in Section 3.3.2.

The effect on these data structures of stealing a task may be seen in the right-hand stack/ltq object of Figure 3.3. A consumer has located the oldest stealable task A (via the `ltq-head` pointer) and has copied the bottom two stack frames to its own stack, replacing frame A by a new frame A' as described in Section 3.2. Finally, the consumer has updated two pointers in the header—`stack-base` now points to the new stack base while `ltq-head` points to the next item on the lazy task queue. We see that stealing effectively shrinks the active area of the stack/ltq object by removing information from both ends; that is, from both the head of the lazy task queue and the oldest frames of the stack.

Splitting a stack by copying is safe in Encore Mul-T because the compiler guarantees that there are no pointers into the stack. Nested lexical environments are heap-allocated instead of stack-allocated; this introduces little overhead in the programs studied (see Section 5.1.2).

Stack overflow testing is straightforward with the combined stack/ltq data structure; we must merely detect when the gap between stack and lazy task queue falls below a safe threshold. When an overflow is detected, the stack and lazy task queue may be copied to a larger stack. Or, if the active area of the stack/ltq object has been substantially reduced by many steal operations, the stack and lazy task queue may just be re-packed into the same space. Stack overflows are relatively rare in programs with bushy call trees.

3.3.2 Synchronization Between Producers and Consumers

The lazy task queue is a shared data structure in a multiprocessing environment. To implement the lazy task queue operations correctly requires proper synchronization between producers and consumers—we must ensure that only one processor will claim any given task on the lazy task queue. Two kinds of race condition are possible. First, two consumers may race to steal the same task; second, a producer trying to return to a continuation may race with a consumer trying to steal it.

The potential race between two consumers affects only the steal operation and thus is not critical to the efficiency of lazy task creation. This race is easily arbitrated by a single lock on the lazy task queue (see Section 3.3.4). In con-

trast, the race between producer and consumer affects the critical lazy future call and return sequence; arbitrating this race efficiently is crucial to minimizing the overhead of lazy task creation. The arbitration of this race must require minimal work from the producer in the common case where no race in fact exists.

Consider that the producer removes tasks from the tail of the lazy task queue (by returning to the associated continuation) while consumers remove tasks from the head (by stealing the associated continuation). Usually the queue contains several entries and there is no conflict between producer and consumer; however, when the queue contains just one entry we must ensure that either the producer or consumer claims it but not both. A simple solution, reported in [Mohr *et al* 91], employs a lock on each lazy task queue entry. Before a task is claimed its associated lock must be acquired, preventing producer and consumer from both claiming the same task.

This lock-based synchronization method has two undesirable features. First, the test-and-set instruction needed to implement locks is rather expensive on the Multimax, taking roughly the same amount of time as 20 register-to-register moves on Yale's machine. Second, storing a lock for each lazy task queue entry increases the storage needed by the lazy task queue.

These considerations motivated the development of an algorithm which eliminates the locks on lazy task queue entries. I will call this the "lockless" algorithm even though one lock remains—consumers still lock the entire lazy task queue to ensure that only one consumer at a time may steal tasks from that queue.

The Lockless Synchronization Algorithm

In the lockless algorithm, rather than locking individual queue entries before claiming them, producer and consumer both claim tasks optimistically and then check to see if a conflict has occurred, retracting their claim if necessary.[6] Both optimistic claiming and conflict detection can be done using only the head and tail pointers of the lazy task queue.

Recall that a producer's lazy task queue is stored in a block of adjacent memory locations and accessed via two pointers (both in global memory) called `head` and `tail`. `tail` points just beyond the newest task on the queue and is modified only by the producer; `head` points to the oldest task on the queue and is modified only by the consumer.

For example, consider the leftmost scenario in Figure 3.4. In this picture, the producer returning to C would decrement `tail`, while a consumer stealing A would increment `head`. Since each agent has exclusive control over its pointer, no additional locks are needed to ensure atomic updates. The middle scenario shows that the queue is empty when when `tail` = `head`; the rightmost shows

[6] A similar algorithm, discovered independently, is discussed in [Feeley 91].

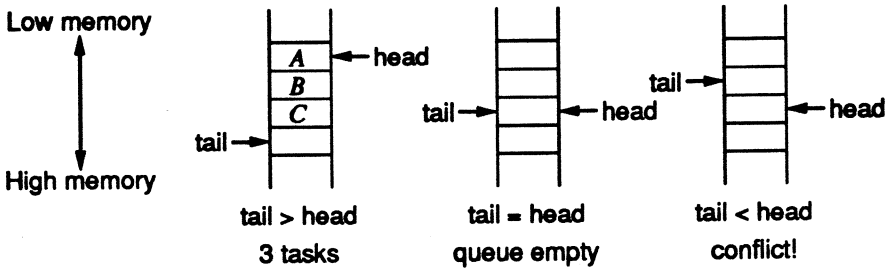


Figure 3.4: 3 lazy task queue scenarios.

that there is a conflict when `tail` < `head`.

In the common case where the queue contains several tasks no conflict will occur and no optimistic update will need to be retracted. If on the other hand a conflict does occur it may be detected by just the producer, by just the consumer, or by both. If the consumer detects the conflict it yields to the producer. If the producer detects the conflict it waits to see whether the consumer also detected it and then acts accordingly.

The lockless algorithm appears in Figure 3.5. The first section of code is executed by the producer returning from a lazy future call. When the lazy future call was made, the producer pushed a continuation on the stack and pushed a task (a pointer to the continuation) on the lazy task queue. Now the producer must determine whether the task has been stolen by a consumer. The producer optimistically claims the task by decrementing `tail`,[7] and checks for conflict by a comparison with `head`. If no conflict is detected, the producer concludes that the task was not stolen and so returns to the original continuation.

If on the other hand a conflict is indicated, the task in question may or may not have been stolen yet. A consumer may have stolen the task long ago, or may have only just claimed the task. After waiting for the stack to be unlocked the producer can correctly distinguish between these cases by a second comparison of `tail` and `head`. (The producer will wait only a bounded length of time—the stealing consumer must eventually unlock the stack if it hasn't already; further, no consumer will re-lock the stack because of the test at line C_1.) If the comparison shows that the consumer has retracted its claim, the producer returns normally to the continuation in question. Otherwise the steal was successful and the producer returns to the modified continuation "swapped in" by the consumer (*e.g.* frame A' in Figure 3.3). In this case, the producer needn't retract its claim to the stolen task because no more operations are possible on this lazy task queue.

Note that in the common case where no conflict exists, this synchronization code adds minimal overhead to the return from lazy future call sequence—the

[7] I have borrowed the shorthand "`tail--`" from the C programming language to indicate that `tail` is decremented.

```
  { Executed by producer returning from a lazy future call }
P1   tail--    { optimistically claim tail task }
P2   if tail >= head then    { no conflict }
P3      return normally
     else    { this task may have already been stolen! }
P4      wait until stack unlocked
P5      if tail >= head then    { the steal was aborted }
P6         return normally
        else    { the steal was successful }
P7         return to modified continuation

  { Executed by consumer stealing a task }
C1   if tail > head then    { stealable tasks appear to exist }
C2      acquire stack lock    { so other consumers don't interfere }
C3      if tail > head then    { stealable tasks really do exist }
C4         head++    { optimistically claim head task }
C5         if tail >= head then    { no conflict }
C6            steal the head task
           else    { producer has already claimed this task }
C7            head--    { retract claim }
C8      release stack lock
```

Figure 3.5: "Lockless" synchronization algorithm.

producer merely decrements `tail` and compares it with `head`.

The second section of code in Figure 3.5 is executed by the consumer stealing a task. The consumer only locks the producer's stack if tasks appear to exist (otherwise it searches for another stack with stealable tasks; see Section 3.3.4). Locking the stack prevents interference by other consumers and also allows the consumer to tell for certain whether stealable tasks exist. If tasks exist the consumer claims one by optimistically incrementing `head` and then checks for conflicts by a comparison with `tail`. If no conflict is detected, the consumer steals the head task; however, if a conflict indicates that the producer has also claimed the task the consumer defers to the producer by decrementing `head`, retracting its claim. In all cases the consumer finishes by unlocking the stack.

Correctness Proof

The correctness of this lockless algorithm may be stated as a theorem:

Theorem 3 *The lockless synchronization algorithm ensures that each task on the lazy task queue is executed by one and only one agent.*

Proof: Clearly the theorem holds if only the producer is active or only consumers are active; it is also clear that the stack lock ensures mutual exclusion among consumers. Thus we concentrate on the case where a producer and consumer attempt to claim the same task. We assume that the queue contains a single entry and show that the theorem holds for all possible interleavings of statements P_i with statements C_i. In fact, only interleavings of the crucial statements P_1, P_2, C_4, and C_5 must be considered; here these statements are called P_{claim}, P_{test}, C_{claim}, and C_{test}.

We shall formally consider statement execution order using the "precedes" relation. A statement A *precedes* a statement B if execution of A has completed before execution of B begins. If A does not precede B it does not necessarily follow that B precedes A; the execution of A and B could overlap.

Three cases must be considered. An agent's test of `tail` *vs.* `head` will reveal a conflict only if the other agent's claim precedes the first agent's test.

Case 1: C_{claim} precedes P_{test} but P_{claim} does not precede C_{test}

	Producer		Consumer	
			C_{claim}:	`head++`
			C_{test}:	`if tail >= head`
time ↓	P_{claim}:	`tail--`		
	P_{test}:	`if tail >= head`		

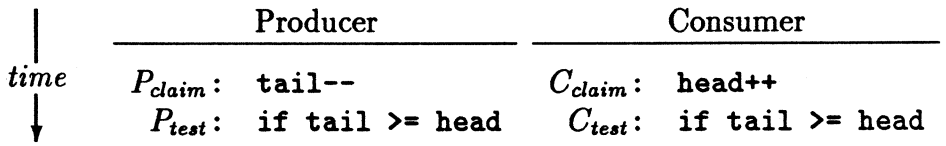
In this case, only the producer will discover the conflict. Since C_{claim} precedes P_{test} the producer will discover the conflict, but since P_{claim} does not precede C_{test} the consumer will not discover the conflict. As specified by the algorithm (and described above), the producer will defer and only the consumer will execute the task.

Case 2: P_{claim} precedes C_{test} but C_{claim} does not precede P_{test}

	Producer		Consumer	
	P_{claim}:	`tail--`		
	P_{test}:	`if tail >= head`		
time ↓			C_{claim}:	`head++`
			C_{test}:	`if tail >= head`

In this case, only the consumer will discover the conflict. Since P_{claim} precedes C_{test} the consumer will discover the conflict, but since C_{claim} does not precede P_{test} the producer will not discover the conflict. As specified by the algorithm (and described above), the consumer will defer and only the producer will execute the task. It is (remotely) possible in this case that the producer could enqueue another task before the consumer's test. That would lead to correct execution as well; the test would succeed and the consumer would steal the new task.

Case 3: P_{claim} precedes C_{test} and C_{claim} precedes P_{test}



In this case, both the producer and the consumer will discover the conflict. As specified by the algorithm (and described above), the consumer will defer and only the producer will execute the task.

All possible interleavings of the four expressions are covered by these three cases, so we may conclude that the lockless synchronization algorithm ensures that each task on the lazy task queue is executed by one and only one agent. ■

Practical Considerations

The correctness proof of the lockless synchronization algorithm makes an important implicit assumption—that if a variable update by one agent precedes a variable reference by another agent, the second agent will receive the updated value. When considering an implementation for a given multiprocessor, this will be true only if the following assumption holds:

Memory Coherence Assumption: *If a processor writes a memory location M, any other processor reading M after the write has completed will receive the updated value.*

One might expect this assumption to hold for the Encore Multimax, whose cache hardware ensures that a write by one processor to location M causes other processors to invalidate locally cached copies of M. But it turns out that the memory coherence assumption does not in fact hold for the Multimax [Wilson 91]. After writing M, a processor does not wait until all caches have been invalidated; instead, it continues with its next instruction.

As a result, lockless synchronization requires the use of extra instructions to ensure memory coherence. Applying the method sanctioned by Encore [Wilson 91] to our lockless synchronization algorithm results in an extra test-and-set instruction for both producer and consumer:

Producer		Consumer	
P_1:	`tail--`	C_4:	`head++`
	`test-and-set(dummy)`		`test-and-set(dummy)`
P_2:	`if tail >= head`	C_5:	`if tail >= head`

(As a side-effect, the test-and-set causes a processor's cache invalidation queue to be flushed; this guarantees for example that the producer will read the most current value for `head` in statement P2.) If this extra instruction is not used, it is possible that *neither* producer nor consumer will discover a conflict. Unfortunately, this test-and-set is the same expensive instruction we wanted to eliminate from the original "locking" algorithm!

The result for Encore Mul-T is that the lockless and locking algorithms have roughly comparable overhead since they both contain the expensive test-and-set instruction. The lockless algorithm is still the preferred choice however because it is in fact slightly faster and because it allows lazy task queue entries to consume less storage space.

3.3.3 Example of Lazy Future Call and Return

With the lockless synchronization algorithm in hand we may proceed to an example showing the full lazy future call and return sequence. The trusty doubly recursive Fibonacci program will serve as our vehicle.

First consider the sequential version `fib`, shown in Figure 3.6 together with a listing of assembler code generated by the T3.1 compiler. Both have been slightly simplified for presentation.[8] `fib` has been written using `let*` to allow an easier comparison with `pfib`; this does not affect the assembler code. In reading the assembler code, it may help to know the conventions that R1 holds the first procedure argument or return value, and that R0 always points to the environment of the currently executing procedure. Also, integer constants appear 4 times too big because the two low-order bits are used as a type tag (*e.g.*, the constant 2 appears as `$8`).

Consider the block labelled `first_call`. Before making the recursive jump to `fib` the live registers `r1` and `r0` and return address `second_call` are saved on the stack. Or, equivalently, we may say that a continuation closure (environment plus code pointer) is created on the stack. The live registers constitute the environment while the return address constitutes the code pointer. When the

[8]The sometimes obscure NS32000 Series assembler mnemonics have been replaced by more meaningful instruction names. The 3-instruction stack overflow check and procedure return sequences have been abbreviated to one line each. The code blocks have been re-ordered to match the source program. Finally, the actual source code directed the compiler to use integer arithmetic and not look up the value of `fib` before each recursive call.

```
(define (fib n)
  (if (< n 2)
      n
      (let* ((f1 (fib (- n 1)))
             (f2 (fib (- n 2))))
        (+ f1 f2))))
```

```
fib:                        # entry point
   stack overflow check
   compare    r1,$8         # is n < 2?
   jlt        return_n      # if so, return n
first_call:                 # make first call
   push       r1            # save n
   push       r0            # save env pointer
   push_addr  second_call   # push return address
   subtract   $4,r1         # compute n - 1
   jump       fib           # recursive call to fib
second_call:                # make second call
   push       r1            # save f1
   push_addr  add_results   # push return address
   move       12(sp),r0     # restore env pointer
   move       16(sp),r1     # restore n
   subtract   $8,r1         # compute n - 2
   jump       fib           # recursive call to fib
add_results:                # add results
   move       4(sp),r0      # restore f1
   add        r0,r1         # compute f1 + f2
   adjust_sp  $-20          # pop stack frames
   return                   # return f1 + f2 in r1
return_n:                   # return n
   return                   # return n in r1
```

Figure 3.6: Program `fib`, compiled.

first recursive call is complete this continuation will be called (*i.e.*, a result will be returned in `r1` and execution will continue at the return address `second_call`.)

Figure 3.7 shows an analogous compiler listing for the parallel version `pfib`, compiled using lazy task creation. `fib` has been changed only by inserting `future` around the first recursive call and `touch` around the reference to `f1`.[9] The com-

[9]The Mul-T compiler can insert touch operations automatically; an explicit `touch` was used in this and some other examples to enhance clarity.

```
(define (pfib n)
  (if (< n 2)
      n
      (let* ((f1 (future (pfib (- n 1))))
             (f2 (pfib (- n 2))))
        (+ (touch f1) f2)))))
```

```
pfib:                                  # entry point
    stack overflow check
    compare     r1,$8                     # is n < 2?
    jlt         return_n                  # if so, return n
first_call:                            # lazy future call
    push        r1                        # save n
    push        r0                        # save env pointer
    push_addr   return_from_lf_call       # push return address
    subtract    $4,r1                     # compute n - 1
    move        ltq-tail,r6               # get ltq-tail pointer
    move        sp,0(r6)                  # queue continuation!
    add         $4,ltq-tail               # increment ltq-tail (4 bytes)
    jump        pfib                      # recursive call to pfib
return_from_lf_call:                   # return from lazy future call
    subtract    $4,ltq-tail               # decrement ltq-tail
    test&set    dummy                     # wait for cache coherence
    compare     ltq-tail,ltq-head         # conflict?
    jlt         resolve_conflict          # if so, resolve it
second_call:                           # make second call
    push        r1                        # save f1
    push_addr   add_results               # push return address
    move        12(sp),r0                 # restore env pointer
    move        16(sp),r1                 # restore n
    subtract    $8,r1                     # compute n - 2
    jump        pfib                      # recursive call to pfib
add_results:                           # see if f1 is a placeholder
    move        4(sp),r0                  # restore f1
    test_bit    $0,r0                     # is r0 a placeholder?
    jclear      got_value                 # if not, proceed
    jsr         get_placeholder_value     # if so, get value
got_value:                             # add results
    add         r0,r1                     # compute f1 + f2
    adjust_sp   $-20                      # pop stack frames
    return                                # return f1 + f2 in r1
return_n:                              # return n
    return                                # return n in r1
```

Figure 3.7: Program `pfib`, compiled with lazy task creation.

piled code has changed only by the addition of the 3 boxed instructions for lazy future call, the 4 boxed instructions for return from lazy future call, and the boxed instructions for `touch` (of which only 2 are executed in the common case where no placeholder is found). With these changes, `fib` is transformed into an efficient parallel program! (Performance results for `pfib` appear in Section 6.6.2.)

Consider again the block labelled `first_call`. Exactly as before, a continuation closure is created on the stack. But now a pointer to this frame (obtained from the stack pointer `sp`) is placed on the lazy task queue.[10] Execution continues as in the sequential version with a recursive jump to `pfib`; however, with the sequence shown here all information necessary for an idle processor to steal the queued continuation has been saved.

The astute reader will have noticed that the return address for the first recursive call has been changed; instead of `second_call` as in `fib`, the return address is now `return_from_lf_call`. So when the first recursive call is complete, execution resumes with `return_from_lf_call`. This block implements the lockless synchronization algorithm described in Section 3.3.2, by which the returning producer determines whether or not the queued continuation has been stolen. We optimistically decrement `ltq-tail`, synchronize for cache coherence, and check for conflicts by a comparison with `ltq-head`. In the common case where no steal has occurred, execution proceeds with `second_call`. If on the other hand a conflict indicates that a steal is in progress, we jump to `resolve_conflict`, not shown here.

Note that if a steal has already been completed the producer will never execute the `return_from_lf_call` code because the continuation containing this "return address" will have been changed by the consumer to `resolve-placeholder`. The producer will return to this new continuation instead.

We have now covered the most important points in the lazy future call and return sequence, although a few interesting details remain.

The very astute reader may be wondering what the consumer will do upon stealing the continuation we have been discussing—if the consumer began execution with `return_from_lazy_future_call` (as indicated by the continuation's code pointer) some rather unpleasant behavior would ensue since that code must be executed only by the producer. Instead, the consumer must offset the code pointer by a known constant (the length of the "return from lazy future call" sequence) to find the true starting address, `second_call` in this case.

When calling the stolen continuation, the consumer passes an unresolved placeholder as an argument. In our example execution would begin at `second_call`,

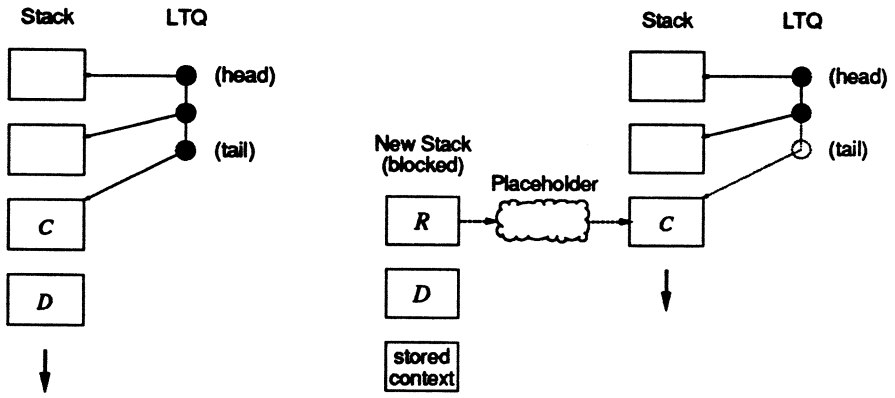
[10] Referencing `ltq-tail` and `ltq-head` directly in the assembler code is a simplification. In actuality, these pointers are extracted from the current stack/ltq object, which is accessed through a block of data kept locally by each processor. This is accomplished economically using the double indirection capability of the NS 32000 Series processor; as a result however, return from lazy future call actually requires 5 instructions instead of 4.

with the placeholder (representing `f1`, the result of the first recursive call) stored in `r1`. Note that this value is saved on the stack during the second recursive call, and touched before being used in the block labelled `add_results`. If a steal has in fact occurred, the placeholder will be dereferenced by the out-of-line routine `get_placeholder_value`, possibly requiring the caller to be blocked until the value has been computed and the placeholder resolved.

3.3.4 Steal Operation

The algorithm for stealing a task from another processor's lazy task queue is shown below; referring to the "before" and "after" pictures in Figure 3.3 may be helpful. As the algorithm is straightforward and the "stealing story" has already been told in some detail, further discussion is omitted.

- Allocate and initialize data structures: a placeholder P and a new stack/ltq object S_{new}.
- Look for a task to steal.
  - Poll other processors to find one whose current stack/ltq object S_{old} has a non-empty lazy task queue (*i.e.*, `ltq-tail` > `ltq-head`).
  - Try to lock S_{old}; if it's already locked, skip to next processor.
  - Make sure tasks exist by a second check that `ltq-tail` > `ltq-head`.
  - Optimistically increment `ltq-head` to claim head item; if a conflict is indicated, skip to next processor.
- Steal the continuation. The head item of the lazy task queue is a pointer $\&A$ into the stack S_{old}. $\&A$ points to a stack frame A representing a stealable continuation. The oldest part of the stack (the portion between A and S_{old}'s `stack-base` pointer) must be copied to the new stack S_{new}.
  - Copy oldest portion of S_{old} into S_{new}.
  - Update `stack-base` and `ltq-head` pointers in S_{old}.
  - Replace A in S_{old} with a new continuation A', directing (`resolve-placeholder` P).
  - Unlock stack S_{old}.
  - "Return" to continuation A in new stack S_{new}, passing placeholder P as the argument.



(a) Before tail-biting. (b) After tail-biting.

Figure 3.8: Blocking a task by tail-biting.

3.3.5 Blocking

There is one remaining loose end in this discussion: what happens to the lazy task queue when a task T blocks by touching an unresolved placeholder? It is not sufficient to save the lazy task queue as part of T's state because the queued tasks would become inaccessible. We would then have the same potential deadlock problem that arises with load-based partitioning.

The solution adopted here is for T to "bite its tail." As shown in Figure 3.8, T's stack is split at the newest potential fork point (found through the tail of the lazy task queue), and only the newest piece is blocked. As with a steal operation, a placeholder is created to communicate a value between the two pieces of the split stack; in this case a frame R is added to the new stack directing that the result returned by continuation D should be used to resolve the placeholder. The producer can continue using the older piece of the stack, which contains all frames referenced in the lazy task queue. Deadlock cannot be introduced because all tasks remain accessible to potential consumers. After dequeuing the tail task, the producer returns to the continuation it represents, passing the newly-created placeholder as an argument.

In essence, T has stolen a task from the *tail* of its own lazy task queue. One problem with this solution is that it goes against our preference for oldest-first scheduling, since we have effectively created a task at the newest potential fork point. Performance can suffer because this task is more likely to have small granularity; also, further blocking may result, possibly leading to the dismantling of the entire lazy task queue.

```
(define (pfib n)
  (if (< n 2)
      n
      (let* ((f1 (future (pfib (- n 1))))
             (f2 (pfib (- n 2))))
        (+ (touch f1) f2)))))
```

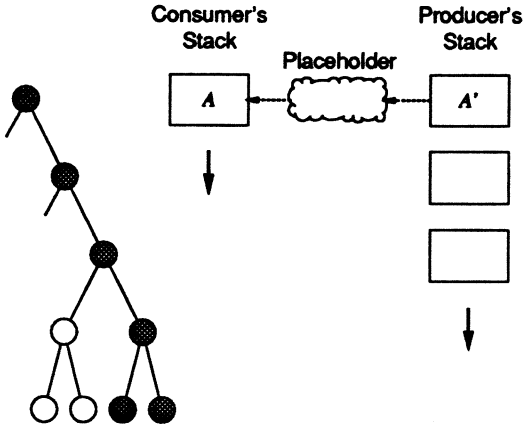


Figure 3.9: Stealing a task in `pfib`.

However, these problems only arise in certain kinds of programs and in fact caused no difficulty for any of the programs described in Chapter 5. Consider for example Figure 3.9, which shows the program `pfib` together with the post-steal stack configuration from Figure 3.2e. For this program we shall see that tail-biting can never occur because the lazy task queue of any blocked task is always empty.

Recall that blocking can only occur as the result of touching a placeholder, and that placeholders are only created when a task is stolen. When stealing a task during the execution of `pfib` the consumer first creates a placeholder to contain the eventual value of `f1`. The consumer then begins executing the stolen continuation A, which will make the second recursive call, touch `f1`, and add `f1` and `f2`. When the consumer begins executing this stolen continuation, its stack contains only frame A as shown; the consumer's lazy task queue (not shown) is empty. The crucial observation is that these data structures will have the same configuration *after* the second recursive call. In particular, the lazy task queue will be empty when the consumer executes `(touch f1)`; thus if blocking is necessary it can be accomplished without tail-biting.

In general, tail-biting will never occur if the value produced by a `future` form

is used only by the parent task. Tail-biting can only occur if the value is passed to other tasks or stored in a data structure.

Still, it is interesting to consider algorithms which address the shortcomings of the tail-biting approach; one such algorithm was implemented as part of the ALEWIFE project and is described in [Mohr *et al* 91].

3.3.6 Miscellaneous Considerations

To wrap up this chapter let us briefly consider the relationship between lazy task creation and the dynamic binding constructs common in Lisp. When restoring the state of a task from stolen stack frames, the consumer must also be able to restore any dynamic bindings which were in effect when the task was queued by a lazy future call. Certainly this could be accomplished if the dynamic state were stored in the lazy task queue entry as part of the lazy future calling sequence, but this would add overhead to this crucial sequence for *all* tasks, not just those using dynamic binding or even just those stolen. In the alternative solution developed by Feeley [Feeley 91], the data structure storing dynamic bindings is augmented by pointers into the lazy task queue. Thus the job of restoring the correct dynamic state is relegated to the consumer and costs are incurred only in dynamic binding operations and the stealing operation.

A related issue is maintainence of debugging information. When an exceptional condition halts execution of a multi-task program, debugging will be much easier if the source code corresponding to every extant task can be retrieved. With eager task creation such information can be stored in the task object and easily retrieved by a debugger. But with lazy task creation this is more complicated, since the source code corresponding to a given task T changes when the oldest continuation is stolen from its lazy task queue. To establish the correct source code for T after the steal would require saving a pointer to the code X for each lazy future call (`future` X). Again, this would add overhead to the lazy future calling sequence for *all* tasks, not just for those stolen. The best solution may be to store this information only when a debugging flag is specified, as the extra overhead is probably tolerable during program development.

Chapter 4

Implementation: Improvements and Assessment

Three topics relating to improvement and assessment of our dynamic partitioning strategies bear discussion before we proceed to benchmark performance results in the next two chapters. First, it turns out that insight gained from studying lazy task creation can be used to improve the performance of both eager task creation and load-based partitioning, even though these changes actually increase the cost of task creation. After discussing these improvements we begin an assessment of the runtime overheads of the three partitioning strategies, comparing overhead here by instruction counts. Finally we consider the Encore LTC implementation of task stealing, arguing why the cost of copying stack frames from producer to consumer is not a significant source of overhead.

4.1 Optimizing ETC and LBP

Having spent substantial effort on the optimization of lazy task creation we must be sure that the "adversary" strategies (eager task creation and load-based partitioning) are similarly well-implemented.

A primary goal of the original Encore implementation of Mul-T [Kranz *et al* 89] was to show that `future` could be implemented with low overhead, and design choices tended to favor the shortest possible instruction sequence for task creation. Subsequent experience has shown however that several design changes which actually *lengthen* the task creation sequence will in fact lead to better performance results for benchmark programs. Three such improvements are discussed in this section. To summarize:

Favor child: When forking, queue the parent task and execute the child task. This policy greatly reduces the number of blocked tasks.

Double-ended task queues: Remove tasks from local queues in a LIFO manner, but remove tasks from remote queues in a FIFO manner. Larger subtrees are executed locally, giving a better approximation to BUSD execution.

Polite stealing: Remember which processor you stole from previously; poll all other processors before stealing from that one again. Larger subtrees are executed locally.

4.1.1 Favoring the Child Task

The first and most important improvement was motivated by the observation that blocked tasks are rare with lazy task creation but common in standard Mul-T. For every benchmark program run with either ETC or LBP, the number of blocked tasks was nearly the same as the number of tasks created; by contrast only a small percentage of tasks created with LTC would block. Because blocking is fairly expensive, reducing the number of blocked tasks is important to good performance.

With any of the partitioning strategies two tasks exist after an actual fork occurs. Recall that the child task is the one which executes X, while the parent task executes the continuation to `(future` X`)`.

These definitions may seem slightly counter-intuitive for lazy task task creation; one might expect the producer to be the parent since it executes `(future` X`)`. But, after a consumer steals the continuation to `(future` X`)` the original definitions are accurate. The producer runs the child task, evaluating X, while the consumer runs the parent task, evaluating the continuation to `(future` X`)`.

The important observation in analyzing the frequency of blocked tasks is that typically the parent task can block waiting for the value of the child task, but not the other way around. If the child finishes executing before its value is needed by the parent, the parent will not block. Thus, giving scheduling preference to the child by starting it immediately is likely to lead to fewer blocked tasks.

This desirable scheduling behavior of favoring the child rather than the parent is an automatic feature of lazy task creation, since the producer begins executing the child code immediately after making a lazy future call. The parent code only begins to run later, after a steal operation or after the child code has completed. This "head-start" for the child explains the observed behavior that few tasks block with lazy task creation.

Halstead has discussed the desirability of favoring the child task [Halstead 89, Halstead 86]; in the Concert Multilisp implementation of eager task creation the processor executing `future` always suspends the parent task and begins executing the child immediately.

However, because suspending the parent task at this time is more expensive

than suspending the nascent child task, the opposite policy of queuing the child and continuing the parent was chosen for the original Mul-T implementation. As detailed in Section 4.2.1, favoring the parent in this way lowers task creation overhead by 13 instructions.

Experience has shown, though, that favoring the parent carries a high price tag in task blocking overhead. Because blocking a task costs 84 instructions and most tasks block, it makes more sense to pay the overhead of 13 instructions to queue the parent. The performance measurements presented in Section 6.4 support the conclusion that favoring the child leads to greater efficiency; this experiment provides the first direct evidence I am aware of for this conclusion.

A further advantage of this scheduling change is that it allows processors to manage just one queue rather than two. Only the "suspended" task queue is needed since new child tasks are always executed immediately.

4.1.2 FIFO or LIFO Queues?

The second improvement alters the way tasks are dequeued in order to more closely approximate BUSD execution. Recall that with ideal BUSD execution single processors execute task subtrees which are as large as possible. Besides allowing fewer tasks to be created, BUSD execution reduces "cold start" cache problems by executing tasks locally rather than remotely. Thus BUSD execution is beneficial even with eager task creation.

With lazy task creation we have seen how oldest-first stealing helps maximize BUSD behavior: picking the oldest task to steal expands the call tree breadth-first, increasing the likelihood that a large subtree will be moved to the stealing processor.

The same breadth-first effect can be achieved in standard Mul-T by adopting a FIFO queuing discipline when stealing tasks from remote processor queues; FIFO is equivalent to oldest-first. Conversely, a LIFO queuing discipline is desirable when removing tasks from local processor queues; LIFO corresponds to a depth-first expansion of the call tree on a single processor.

Because operations can be cheaper when tasks are always added and removed from the same end of the queue, a LIFO discipline was chosen for both types of dequeuing operation in the original Mul-T implementation. Experiments using double-ended task queues have shown, however, that the increased cost of queuing operations is more than compensated by the efficiency of executing larger subtrees locally. Fewer tasks are created (with load-based partitioning) and fewer tasks are stolen (with both LBP and eager task creation). See Section 6.4.

4.1.3 Polite Stealing

Any thief knows that robbing the same bank twice in a row is a bad idea. In the section's final improvement, this wisdom is applied to the polling strategy adopted by idle processors. In the original implementation of Mul-T an idle processor numbered i always begins its search for stealable tasks with processor number $i + 1$ (modulo p, the number of processors), polling the other processors in numerical sequence if the first has no tasks.

The result of this policy is that some processors can be polled more often than others. For example, if processors 3 and 4 both have a large queue of tasks, idle processors will always poll 3 before 4 and nothing will be stolen from 4's queue until 3's queue is empty. This unfairness in stealing decreases the likelihood that the globally oldest task will be stolen and tends to decrease the size of subtrees executed on a single processor.

To address this problem we adopt a simple "polite" stealing policy, where a stealing processor stores the number v of the victim processor. When the stealer is next idle it will begin polling with processor $v + 1 \pmod{p}$, ensuring that the original victim will be polled last.

This fairness in stealing improves the likelihood that the globally oldest task will be stolen, tending to increase the size of subtrees executed on a single processor. As the experiments of Section 6.5 show, polite stealing improves performance.

4.2 Runtime Overhead of Partitioning Strategies

The prime motivation for investigating dynamic partitioning strategies was to find an alternative to the high runtime overhead of eager task creation. We may now begin to evaluate the success of this endeavor by a direct analysis of the runtime overheads of load-based partitioning and lazy task creation. Here we will measure overheads by counting machine instructions; benchmark timings will appear in later chapters.

How much does `future` cost? The answer depends, naturally, on the program in question, but we can get a good idea by considering `pfib`.[1] For each partitioning strategy then, how much overhead is introduced by executing one `future` in `pfib`?

The following table shows the number of instructions of overhead for each partitioning strategy (the assumptions on which these counts were based are discussed below). Two implementations of eager task creation are shown; "Old ETC" is the original Mul-T implementation, while "New ETC" (as well as "LBP") includes the changes described in Section 4.1.

[1] The code for `pfib` appeared in Figure 3.7.

	old ETC	new ETC	LBP	LTC
Cost if no task created	—	—	2	8
Cost if task created	133	153	155	146
Additional cost if task blocks	84	94	94	94

The first line shows the crucial case where no task is created; load-based partitioning requires 2 instructions of overhead while lazy task creation requires 8.

The second line shows that the cost of actually creating a task is about the same for all strategies. This is no surprise with load-based partitioning since it uses the same implementation as eager task creation, but it is perhaps surprising that stealing a task with lazy task creation is actually slightly cheaper than creating a task eagerly. Note also that the cumulative overhead of the optimizations for "new" ETC is 20 instructions.

The third line shows the additional cost if a task blocks, also about the same for all strategies.

The cost of the `touch` operation in `pfib` has not been counted in the table because the overhead is the same for all strategies. If no task is created, `touch` requires 3 extra instructions (one to move the value of `f1` from the stack to a register). If a task is created, `touch` must extract the value from the resulting placeholder, raising the cost to 10 instructions.

The instruction counts for task creation are further broken down in the following two sections.

4.2.1 Eager Task Creation

Implementing lightweight tasks is not easy. The purpose of the following breakdown of instruction counts for eager task creation is not to defend in detail why x instructions were required for operation y; rather, the goal is to convince the reader that ETC is not a straw man. Creating, scheduling, running, and terminating a task that can run on any processor and be blocked or interrupted at any time requires a significant amount of work in the best of systems.

With another round of fine-toothed optimizations and some nontrivial design changes, I estimate that the total could be shaved by perhaps 24 instructions. While this would improve the performance of eager task creation somewhat, efficiency would still lag far behind the average of 14 instructions per task achievable with dynamic partitioning.

In several cases of the following breakdown a common inexpensive path was counted rather than a less common, more expensive path:

- Task queue access is arbitrated by spin locks. In the common case a lock is acquired on the first try (requiring 4 instructions); no busy-waiting time is included.

- Stacks are placed on a freelist when no longer needed. In the common case a stack will be available from the freelist so the cost of allocating a new stack is not included.
- Any tasks waiting for a placeholder's value must be re-queued when the placeholder is resolved. In the common case there are no waiting tasks.
- In the common case a processor finds a task on its local queue; polling time is not included.

Here then is the breakdown for the original Mul-T implementation of eager task creation, showing the overhead introduced by executing one `future` and one `touch` in `pfib`:

Creating processor:
- 17 Build closure (2 slots) for code in `future`
- 9 Call task creation subroutine and return
- 25 Allocate and initialize placeholder + task object
- 9 Enqueue task on local new task queue
- 4 Disable and enable interrupts

Executing processor:
- 14 Dequeue a new task (after finding no suspended tasks)
- 8 Check for exceptions and initialize processor globals
- 12 Get stack from freelist and initialize
- 7 Call the closure and move free variables to registers
- 12 Resolve the placeholder and check for waiters
- 12 Prepare the stack for re-use
- 4 Disable and enable interrupts

Touching processor:
- 3 `placeholder?` check on `x` (includes move from stack to register)
- 7 Extract value from placeholder object (it might contain another placeholder)

Total:
- 143 instructions of overhead

As reported in Section 4.1, improving the runtime scheduling behavior of eager task creation increased its overhead somewhat. When the child task is given scheduling preference instead of the parent, the parent must be suspended and restarted. This increases the instruction count by 13. And, using double-ended task queues raises the cost of queuing operations, increasing the instruction count by 7.

4.2.2 Lazy Task Creation

The operations performed when creating a task lazily are mostly similar. Assuming that a task is actually created, here is a breakdown for LTC showing as above the overhead introduced by executing one `future` and one `touch` in `pfib`:

Consumer:
- 23 Allocate placeholder + task object, fill with zeros for gc safety
- 19 Allocate and initialize a stack/ltq object
- 30 Find a stealable continuation
- 12 Initialize new task and stack, modify old task object
- 21 Copy continuation (here, 4 slots) from old stack to new
- 5 Modify old stack
- 4 Disable and enable interrupts
- 5 Call stolen continuation

Producer:
- 3 Lazy future call (no return because `resolve-placeholder` swapped in)
- 12 Resolve the placeholder and check for waiters
- 12 Prepare the stack for re-use

Touching processor:
- 3 `placeholder?` check on `x` (includes move from stack to register)
- 7 Extract value from placeholder object (it might contain another placeholder)

Total:
- 156 instructions of overhead

Lazy task creation seems more complex than eager task creation, so how can the total here be less than with "new" ETC? Well, queuing a task is certainly much cheaper with LTC than with ETC, although dequeuing a task (*i.e.* stealing it) is more expensive. LTC is also cheaper because there is no need to build a heap closure to store the state of the child task. With LTC the state of the parent task is saved in a stack closure, but in `pfib` (and most other programs studied) that closure must be built anyway because of the recursive call, so no extra cost is incurred.

4.3 Overhead of Copying in LTC Steal Operation

Although the choice of a traditional contiguous-memory representation for stacks in Encore Mul-T allows efficient stack operations, this design decision also means that with LTC a retroactive fork must copy stack frames to split an existing stack. It would appear that the amount of copying required is potentially unlimited so that the cost of the LTC steal operation is also unlimited, scotching our original goal to keep the cost of stealing close to the cost of creating an eager task.

While this is technically true it is somewhat misleading; the overhead of copying when stealing a continuation should be viewed against the cost of creating the continuation in the first place. A program with fine source granularity does little work between lazy future calls; thus it is not able to push enough items onto the stack to require significant copying. A program which creates large continuations (requiring consumers to copy many words) must do a fair amount of work to push all that information on the stack; the cost of copying is unlikely to be significant in comparison.

As evidence of this claim a "worst-case" program was written which creates large continuations with as little work as possible; timings for this program using both lazy task creation and eager task creation are shown below.[2] The programs used for ETC and LTC are slightly different, but timings for both measure the cost of pushing n numbers on the stack, creating a task, and adding up the numbers (all `future` calls led to actual forks with LTC). For example, the first row shows the elapsed time for each method to execute 10 tasks of size 1000 while the last row shows the time to execute 1000 tasks of size 10.

Number of Tasks	Continuation Size (words)	Time (seconds) ETC	LTC
10	2000	.08	.12
20	1000	.08	.12
50	400	.09	.13
100	200	.10	.14
200	100	.12	.15
500	40	.18	.21
1000	20	.28	.30

These figures show that even with a "worst-case" program the costs of task creation with LTC and ETC are of the same order. With the largest amount of copying (2000 words), task creation with LTC is only 50% more expensive than with ETC. And if a real program required this much copying it would probably do considerably more work in the course of creating such a large continuation. We can conclude that copying is very unlikely to introduce noticeable overhead in real programs.

However, one exception to this conclusion can occur when parallel tasks are created in a particular style from deep within a program. For example consider `parmap-cars`, a straightforward parallel version of Scheme's `map` procedure which applies a function `f` to every element of a list `l`:

[2]Information on timing methodology appears in Chapter 5.

```
(define (parmap-cars f l)
  (if (null? l)
      '()
      (let* ((elt (future (f (car l))))
             (rest (parmap-cars f (cdr l))))
        (cons elt rest))))
```

In each iteration `future` specifies that a child task may be created to call `f` on the current list element while the parent maps `f` down the rest of the list.

Now assume that `parmap-cars` is called from deep within a program after numerous procedure calls have created a large number of stack frames. With lazy task creation, stealing the continuation to the first `future` call will require copying all of the stack frames. As argued, the cost of this copying is unlikely to be significant compared with the cost of building up the stack in the first place. But in this example the stolen continuation (representing the `rest` calculation) immediately makes another `future` call, and the next steal must copy the same information all over again. In fact, spreading work to n processors in this example via lazy task creation would require the built-up stack information to be copied n times.

There are two easy solutions to this problem. First, `future` could be inserted around the original invocation of `parmap-cars`. Stealing the continuation to this original `future` call would require copying the built-up stack, but any further steals would require very little copying.

Second, an alternative parallel version of `map` could be used:

```
(define (parmap-cdrs f l)
  (if (null? l)
      '()
      (let* ((rest (future (parmap-cdrs f (cdr l))))
             (elt (f (car l))))
        (cons elt rest))))
```

In this version, `future` appears around the "rest" computation. Stealing the continuation to the `future` call in the first iteration would require copying the built-up stack, but again any further steals would require very little copying.

Although situations such as this should be relatively rare and can be fixed by either of the suggested methods, identifying such situations still requires a certain amount of awareness from the programmer. This is a case (the only one I have encountered) where lazy task creation cannot be applied blindly. The alternative linked-frame representation for stacks used in the ALEWIFE implementation of LTC [Mohr *et al* 90, Mohr *et al* 91] can alleviate this problem (steals require no copying); however, the costs of using this alternative stack representation have not yet been determined precisely.

Chapter 5

Goals and Methodology of Performance Measurements

Many of the claims made in previous sections remain to be demonstrated by actual experiments. Specifically, Chapter 6 will present data to support the following claims:

- Both LTC and LBP perform substantially better than ETC.
- LBP degrades the performance of some programs (compared with ETC).
- LTC creates substantially fewer tasks than LBP.
- LTC performs at least as well as LBP for most programs—the overhead of LTC has been acceptably minimized.
- The design changes for the standard Mul-T scheduler (see Section 4.1) improve the performance of ETC and LBP.

There are a few topics to consider before getting to the experimental data; this chapter describes the benchmark programs to be used and discusses how the experiments were run.

First though, I would like to argue the important point of

5.1 Why You Should Believe These Numbers

Most of the above claims relate to the runtime overhead of the various partitioning strategies—we will be drawing conclusions about these partitioning overheads based on various measurements of Encore Mul-T. But since these conclusions will be based on relative timings we must be careful; it is important to be sure that the overhead of partitioning really *is* low, rather than just *looking* low because it is masked by overhead in the rest of the system.

What other sources of overhead could make a parallel Mul-T program slow in absolute terms? In comparison with, say, a sequential C program, four categories of overhead come to mind:

1. The cost of using Lisp instead of a language like C, *e.g.* automatic storage reclamation, manipulation of type tags at runtime, and dynamic linking.
2. The cost of using sequential Mul-T instead of T, *e.g.* runtime checks for stack overflow.
3. The cost of using a parallel algorithm instead of a sequential algorithm, *e.g.* using recursive divide and conquer instead of an iterative loop.
4. The runtime costs of multiprocessing, *e.g.* contention for shared resources and cache turbulence.

To ensure that measurements of the partitioning strategies are meaningful we must distinguish overhead due to partitioning from overhead due to these other sources; each source is considered in turn. We shall see that although overhead from some of these categories is noticeable for the benchmarks to be presented, the impact is not enough to hide the overhead of dynamic partitioning.

5.1.1 Overhead of Lisp

Despite Lisp's reputation for inefficiency, the first category of overhead is not a major factor in the benchmarks to be presented. This is true largely because code produced by T's Orbit compiler is comparable in quality to code produced by other compilers for the same hardware [Kranz 88]. In addition, runtime overhead has been minimized in these programs by using type-specific arithmetic and avoiding run-time storage allocation in the benchmark code. Finally, the cost of garbage collection time has been excluded from performance statistics.

As a direct comparison, the "best" version of `tridiag` (see Section 5.3) was coded in C (3.33 sec) as well as in sequential T3.1 (3.92 sec). T's modest slowdown here arises because Orbit generates code to save and restore live values more often in this benchmark than the the C compiler does. `tridiag` has a lot of live data and Orbit has fewer registers available because several are dedicated to specific uses. This effect would probably be less pronounced on other processors (which tend to have more general-purpose registers) or other benchmarks (which often have less live data).

5.1.2 Overhead of Sequential Mul-T

Moving to the second category of overhead, we must consider factors that make a sequential Mul-T program run less efficiently than the same program in sequential

Program	Elapsed Time (seconds)				Overhead	
	T3.1	"Standard" Mul-T	Mul-T w/ LTC	Compiler Touches	of Mul-T	of Compiler Touches
`abisort`	6.91	8.23	8.27	11.78	19%	51%
`allpairs`	10.82	11.88	11.93	31.72	10%	183%
`fib`	1.40	2.37	2.48	2.70	69%	24%
`mergesort`	2.00	2.19	2.21	3.50	9%	66%
`mst`	20.96	22.68	22.75	35.44	8%	61%
`queens`	2.46	3.37	3.39	3.98	37%	25%
`rantree`	1.00	1.21	1.22	1.33	21%	12%
`tridiag`	10.45	10.69	10.13	15.52	2%	46%
`tribest`	4.04	3.91	3.94	5.99	-2%	51%

Table 5.1: Overhead introduced by Mul-T compiler in sequential code

T. While the need to support a parallel runtime system introduces potential overhead in several areas, only two areas have significant impact on the code generated for the benchmarks to be presented:

- Compiler support for multiple stacks—the compiler inserts instructions which test for stack overflow.
- Compiler support for `future`—the compiler may be directed to generate `touch` operations automatically (so the programmer needn't consider whether a variable might be bound to a placeholder and thus require explicit dereferencing with `touch`).

The effects of these sources of overhead for 9 programs are summarized in Table 5.1.[1] Each program was compiled 4 different ways: in T3.1, in "standard" Mul-T, in Mul-T with lazy task creation, and in standard Mul-T with compiler touches enabled. Since these versions were all run using just one processor, any instances of `future` and `touch` were ignored during their compilation.

The elapsed time for each version is shown, along with two overhead figures derived from the timings. The first overhead column shows the additional cost of running the program in standard Mul-T as a percentage of its time in sequential T; this essentially measures the cost of stack overflow tests.[2] The second overhead column shows the additional cost of running the program with compiler touches

[1]The programs are described in Section 5.3; benchmarking methodology is described in Section 5.2.

[2]The cost of allocating nested lexical environments in the heap instead of the stack is also a factor here, but is minor compared with the cost of stack overflow checking.

enabled, as a percentage of its time in standard Mul-T. The "Mul-T with LTC" column does not figure in either of the derived columns; it is included to show that support for lazy task creation introduces only a slight amount of overhead in sequential code. This overhead arises because the combined stack/ltq object used in the implementation of lazy task creation (see Section 3.3.1) makes stack overflow tests slightly more expensive.

The data show that compiler support for both multiple stacks and `future` carries a modest cost which varies from program to program.[3] Even though Encore Mul-T was engineered to minimize the cost of compiler touches [Kranz *et al* 89], Table 5.1 shows that the overhead can be non-trivial. In order to factor out as many sources of overhead as possible, all measurements discussed hereafter were made with compiler touches disabled (see Section 5.2 for a fuller discussion). The remaining "overhead of Mul-T" averages around 20% for the programs shown; this is not large enough to obscure the effects of the partitioning overheads we care about measuring.

Two anomalies in Table 5.1 are probably caused by the mechanisms to be described in Section 5.2: `tridiag` appears to be noticeably faster with lazy task creation than without, and `tribest` appears to be faster in Mul-T than in T. The presence of anomalies suggests that these measurements of overhead not be taken too precisely but rather as a rough guide to the costs involved.

5.1.3 Overhead of Parallel Algorithms

In some cases an efficient sequential algorithm does not parallelize well but a highly parallel algorithm for the same problem is much less efficient. There are two ways in which this kind of algorithmic overhead could emasculate claims made about a benchmark program.

First, if a parallel benchmark using the maximum number of processors didn't run significantly faster than the corresponding sequential version there would be no sense in using the parallel algorithm. Timings of two such parallel/sequential pairs appeared in Table 5.1—`abisort/mergesort` and `tridiag/tribest` (see Section 5.3 for a description). The table shows that the sequential versions of these algorithms run between three and four times faster than the corresponding parallel algorithm on one processor. As will be seen in Section 6.6, speedup of both `abisort` and `tridiag` is close to linear with dynamic partitioning so they outperform the sequential competition when more than 4 processors are used. These two programs are the worst cases in this regard for the benchmarks to be studied; the parallel versions of the other benchmarks have less algorithmic overhead.

[3]This set of programs may show a larger-than-average cost for stack overflow tests because they tend to have rather fine-grained inner loops.

Secondly, algorithmic overhead could increase a program's granularity enough to mask the effects of partitioning overhead. Although there is no question that some of the benchmarks have finer-grained sequential versions, we will see that the granularity of all but one of the benchmark programs is fine enough that eager task creation performs poorly. In addition we will measure the performance of the dynamic partitioning strategies using a synthetic program whose granularity may be made arbitrarily fine.

5.1.4 Overhead of Multiprocessing

The statistics to be presented do not allow precise conclusions about the extent of multiprocessing overhead from sources such as cache turbulence and contention for shared resources. However, as Section 6.6 will show, speedup for most of the benchmarks is close to linear with dynamic partitioning so we can conclude that the effect of these other sources of overhead is fairly small.

5.2 Methodology

It is perhaps an understatement to say that measuring the performance of complex computer systems is not straightforward. To summarize the behavior of a complex set of interacting systems into a few numbers for a table or graph, the effects of many of the interactions must be minimized to achieve a repeatable result. This section describes some of the steps taken to ensure that the results reported here are as accurate as possible.

All experiments were run on Yale's Encore Multimax system, configured with 18 NS-32332 processors and 64 megabytes of memory and running the Umax 4.3 operating system (R4_0.0). In the Encore implementation of Mul-T each of several UMAX processes acts as a virtual Mul-T processor; all experiments were run when the Multimax was otherwise empty of user processes, in a mode where each virtual Mul-T processor was given exclusive access to a physical Multimax processor. All pages of the processes' virtual address space were accessed on start-up to reduce paging effects.

Garbage collection time is not included in any of the performance statistics. While it might be interesting to include the amortized cost of storage reclamation when comparing Lisp with C, the cost is less important when making relative comparisons of partitioning strategies which use the same garbage collector. Actually, including garbage collection time would make eager task creation look worse than it already does because of the storage consumed by task objects. Likewise, lazy task creation would improve slightly relative to load-based partitioning; garbage collection is less frequent with LTC because fewer tasks are created.

Unless noted otherwise, all timings of eager task creation and load-based partitioning incorporate the scheduling improvements described in Section 4.1.

With load-based partitioning, the load threshold T was always chosen to give the fastest time on 16 processors after comparing trials using $T = 1$, $T = 2$, and $T = 3$. In most cases $T = 2$ performed best though $T = 1$ was sometimes better; using $T \geq 3$ always degraded performance. If in this report a value of T is not specified, assume $T = 2$.

Two attributes of the Encore Multimax architecture make repeatable timings especially difficult. First, because the caches are physically addressed and direct-mapped, two trials of the same sequential program with the same data can exhibit markedly different cache behavior. For example, page swapping may cause the second trial to run with a different mapping of virtual addresses to physical memory locations, with the result that two sections of live data which previously mapped to different cache locations now map to the same cache locations. The second trial would suffer more cache misses and run more slowly.

To reduce the effects of this variability, all timings reported are the average of numerous trials. Programs were always re-loaded before a trial so that the location of program instructions varied more widely between trials.

The second "frustration factor" relates to the decoding of machine instructions. NS Series 32000 processors have multi-byte instructions, so the time required to fetch a given instruction may be larger if the bytes comprising the instruction straddle two words or two cache lines. If a program is compiled in two different ways (say, once with load-based partitioning and once with lazy task creation) the inner loop instructions may fall on different word boundaries in the two object files, complicating a comparison of the two versions.

This dynamic has greatest effect in a very tight loop and did not seem to affect most of the benchmarks. However, the synthetic program `tree` (see Section 5.3) uses a tight delay loop to simulate computations of different granularities. Versions of `tree` compiled with different scheduling strategies had to be adjusted to ensure that the alignment of the delay loop did not adversely affect program timings. I suspect, but have not verified, that this dynamic is behind the anomalies observed in Table 5.1.

As mentioned in Section 5.1.2, all measurements were made with compiler touches disabled. For Encore Mul-T both implicit and explicit touches have points in their favor, but the decision between them is orthogonal to our primary topic of partitioning strategies. The important consideration for this study is to factor out as many sources of runtime overhead as possible, so compiler touches are disabled in favor of the less expensive explicit touches. This decision leads to finer-grained inner loops which are a more demanding test of dynamic partitioning.

As a side note, inserting `touch` operations explicitly was a trivial task for the benchmarks used here. Still, even this small amount of work would not

be necessary on the ALEWIFE machine [Agarwal *et al* 91, Agarwal *et al* 90, Chaiken *et al* 91] (once it is built), as implicit touches are supported in hardware and add no overhead to program execution in the common case where the touched datum is not in fact a placeholder.

5.3 Benchmark Programs Described

Most of the experiments to be presented use a common suite of benchmark programs. The choice of programs for this suite was primarily influenced by two factors. First, the program had to be fine-grained enough that it performed poorly with eager task creation; several programs which appeared at first to be fine-grained were rejected on closer inspection because a coarse-grained parallelization was easily attainable. One such program, `allpairs`, is included in the benchmark suite for comparison. Note also that no program was ever rejected because it was too fine-grained.

Second, the algorithm had to have the potential of close to linear speedup on 16 processors. That way, any deviation from linear speedup is attributable to multiprocessing overheads such as task creation or idle processors rather than to the algorithm itself. `mergesort` is one program rejected on these grounds. Programs containing speculative parallelism such as the Boyer-Moore theorem prover or travelling salesperson problem were also rejected on these grounds; ideal speedup is a slippery concept for such problems because the amount of work varies considerably depending on how the call tree is expanded.

Knowing the source granularity of a benchmark is very important in interpreting performance results. Recall from Section 1.2.2 that "source granularity" measures the work performed per `future` call in the source code while runtime granularity measures the work performed per task actually created at runtime. The quantity g will be used to measure source granularity; g is computed by dividing the sequential Mul-T execution time of a benchmark (measured in microseconds) by its total number of calls to `future`:

$$g = \frac{t_{seq}}{maxtasks}$$

Thus g measures the average amount of work done by a Mul-T program per `future` call, excluding partitioning overhead. Finer-grained programs will have smaller values of g. For the benchmarks measured here Yale's Multimax delivers about 1 Mips per processor[4], so g is roughly comparable to the average number of NS32332 instructions per task as well.

[4]Encore claims 2 Mips per processor, but the benchmark programs used here tend to have numerous memory referencing instructions, decreasing the execution rate.

The programs comprising the benchmark suite are summarized in the following table and then described individually. All benchmark code is free of unnatural parameterized grouping; these are straightforward parallel algorithms with 1 or 2 instances of `future` and `touch`. Each program was carefully coded for maximum efficiency and none uses floating-point arithmetic.

Program	g	Description
`abisort`	77	Adaptive bitonic sort
`allpairs`	875	All-pairs shortest paths (Floyd's algorithm)
`fib`	20	Fibonacci numbers
`rantree`	63	Sum nodes of a randomly shaped tree
`mst`	91	Minimum spanning tree (Prim's algorithm)
`queens`	97	n queens problem
`tridiag`	217	Tridiagonal matrix solver

`abisort` ($g = 77$) performs an "adaptive" bitonic sort [Bilardi & Nicolau 89] of $n = 16,384$ numbers. The adaptive algorithm achieves optimal complexity ($O(n \log n)$ rather than the $O(n \log^2 n)$ of the standard bitonic sort algorithm) by storing bitonic sequences in a special tree data structure. Still, adaptive bitonic sort performs about twice as many comparisons as a merge sort, and has somewhat greater bookkeeping costs. However, its parallel divide-and-conquer merge operation allows virtually linear speedup when $n \gg p$. Such speedup is not possible with straightforward implementations (on machines like the Multimax) of other divide-and-conquer sorts such as merge sort and quicksort which contain significant sequential phases.

The optimized sequential program `mergesort` is also measured for comparison with `abisort`.

`allpairs` ($g = 875$) solves the all-pairs shortest paths problem [Aho *et al* 83] on a directed linear graph of $n = 117$ nodes[5] using a parallel version of Floyd's algorithm. Starting with an $n \times n$ connectivity matrix C, where $C_{i,j}$ gives the length of the edge connecting nodes i and j (or 0 if i and j are not adjacent), execution continues until $C_{i,j}$ contains the length of the shortest path from i to j for all i and j. The algorithm iterates sequentially through all vertices k. During step k, all pairs of vertices are checked to see if going through vertex k produces a shorter path; that is, $C_{i,j}$ is updated if $C_{i,k} + C_{k,j} < C_{i,j}$. These operations may all proceed in parallel. To see why, note that in step k no element of row k or column k will change; this is so because $C_{k,k} = 0$. Since all computations in step k will only reference

[5]Why 117 nodes? Originally the test graph was to represent 117 cities of the Eastern United States, but since the algorithm for this problem is essentially oblivious to the connectivity of the graph, the simpler linear graph was used with the same number of nodes.

values from row k and column k, all vertex pairs can be safely handled in parallel during a given step.

Handling all vertex pair tests in parallel would produce a rather fine-grained program, but this is not necessary for the Multimax because a coarse-grained parallelization is easily obtained by having each (potential) task handle the n vertex pair tests in a single matrix row. This is the strategy adopted for `allpairs`, explaining its relatively high g value. Thus the parallel version of `allpairs` has n sequential steps separated by barrier synchronization; in each step there are n potentially parallel tasks. The tasks are created by a divide-and-conquer traversal of the index set of the matrix; the additional overhead of such a traversal compared to an iterative traversal is negligible because of the coarse grain of each task.

`fib` ($g = 20$) is the standard doubly recursive program to calculate the nth Fibonacci number ($n = 25$ here), where the two recursive calls may be performed in parallel. `fib` is not a program anyone would use; it is included mostly as a standard of comparison with other parallel implementations of applicative languages. Also, `fib` has finer granularity than any of the "real" benchmarks, providing a more demanding test of the partitioning strategies.

`rantree` ($g = 63$) is a synthetic benchmark motivated by the study of parallel Monte-Carlo particle transport algorithms [Miura 88]. A tree with n nodes is generated in a random shape using a branching probability of 50%. When branching is chosen the remaining nodes are divided randomly between the left and right subtrees. To ensure acceptable pseudo-random behavior, a pair of matched functions is used to update the "random seed" for the right and left subtrees. `rantree` has fairly fine source granularity and a highly irregular call tree.

`mst` ($g = 91$) finds the minimum spanning tree of an n node graph using (a parallel version of) Prim's algorithm [Aho *et al* 83]; here the input data is a fully-connected graph of 1000 points chosen randomly from a unit square, with edge lengths determined by Euclidean distance. Prim's algorithm builds the minimum spanning tree one node at a time; in each of $n - 1$ sequential steps the node closest to the existing partial tree is added. With each node i not yet in the tree is stored the minimum distance d_i to a node in the tree. In each step, each d_i is updated if node i is closer than d_i to the node most recently added to the tree, and the node with the smallest d_i is added to the tree. In each sequential step a parallel divide-and-conquer traversal of the free nodes is used to update the d_i's and find the closest node. Since the number of free nodes decreases by 1 in each step, the size

of the search tree decreases from $n-1$ nodes in the first step to 1 node in the last step.

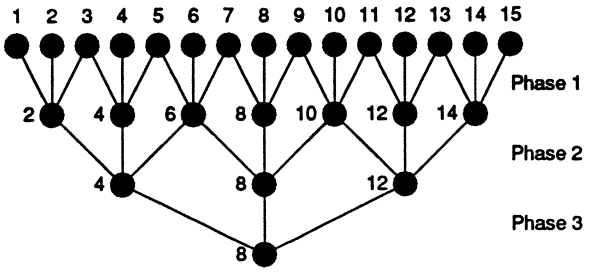
queens ($g = 97$) finds all solutions to the n queens problem, where n queens are placed on an $n \times n$ chessboard in such a way that no queen may capture another. $n = 10$ was chosen for these experiments. A queen is placed on one row of the board at a time; each time a queen is legally placed, **future** appears around a recursive call to find all solutions stemming from the current configuration. This version of n queens uses bit vectors for a very compact and efficient representation of the placed queens, resulting in fine source granularity.[6] Like **rantree** the call tree of **queens** is not well-balanced.

tridiag ($g = 217$) solves a tridiagonal system of $n = 2^k - 1$ equations using cyclic reduction [Hockney & Jesshope 88] and backsubstitution. In each of k phases of standard cyclic reduction half of the remaining equations are eliminated. Figure 5.1a shows the 3 phases of standard cyclic reduction for $n = 15$; a circle represents each equation and the reduction phases proceed from top to bottom. For example, in phase 2 equations 2, 6, 10, and 14 are eliminated, yielding new versions of equations 4, 8, and 12. After the final phase equation 8 may be solved; the other equations may then be solved by backsubstitution (not shown).

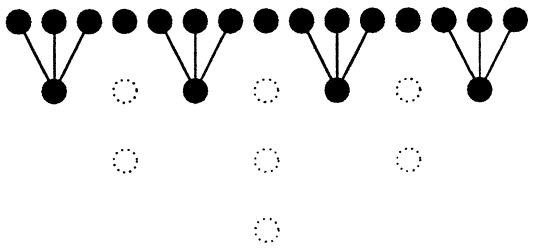
Note that a value computed in one phase may be used by two distinct computations in the subsequent phase; for example, the updated version of equation 10 computed in phase 1 is used in phase 2 to update both equations 8 and 12. This pattern of data dependencies complicates the synchronization needed for parallel implementations of cyclic reduction.

A novel solution, shown in Figures 5.1b–5.1d was adopted in **tridiag** to take advantage of the problem's natural divide-and-conquer structure. In a given phase only the "odd" equations (those which will be eliminated in the subsequent phase) are updated. This means that updating an equation may require computations at several "levels", as with the 3 updates shown for the central equation in Figure 5.1d. But, this approach also allows a simple control structure using only call/return synchronization. For example, to solve the central equation we recursively reduce the right and left subtrees (in parallel) and then reduce the central spine sequentially as in Figure 5.1d. This algorithm does have the effect of introducing a sequential component of $O(h)$ to the computation of a node with height h. In practice however, parallel efficiency can still be very close to 1.0 when $n \gg p$ because processors are only idle during the last few (cheap) phases.

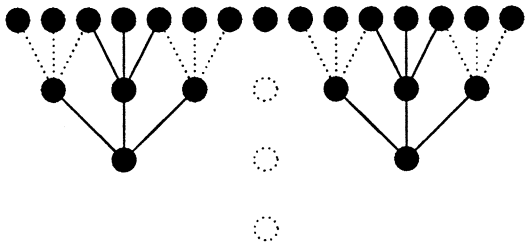
[6] The version of **queens** measured in [Mohr *et al* 90] had substantially coarser granularity.



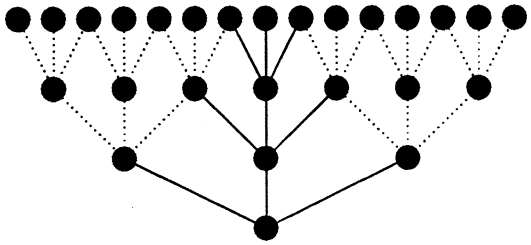
(a) Standard parallel algorithm.



(b) Phase 1 of revised algorithm.



(c) Phase 2 of revised algorithm.



(d) Phase 3 of revised algorithm.

Figure 5.1: Standard and revised algorithms for cyclic reduction of tridiagonal systems.

A rather large value of $n = 65,535$ was used here, reflecting a preference for non-trivial problems; performance was comparable for smaller values of n.

The sequential program `tribest` is also measured for comparison with `tridiag`; `tribest` solves the same tridiagonal system using standard Gaussian elimination. This algorithm performs fewer operations per equation than cyclic reduction (8 as opposed to 17) but contains essentially no opportunities for parallelism.

Two other synthetic programs will be used in experiments in addition to those in the standard suite described above.

`tree` adds up a perfect binary tree of 1's using a parallel divide and conquer structure very similar to `pfib` or `psum-tree`; however, at each tree node or leaf it executes a delay loop of a specified length, allowing granularity control. Varying the granularity, tree depth, number of processors, and partitioning strategy will produce some revealing comparisons.

Finally, `fatwalk` ($g = 140,000$) is a very coarse-grained iterative program designed as a worst case to show how load-based partitioning can degrade the

performance of a program. **fatwalk** is discussed only in Section 6.6.1 so further description is deferred until then.

Chapter 6

Benchmark Results

We are now ready to examine the experimental results. As an overview, here is a list of major conclusions, with section numbers in parentheses:

- In sequential code the overhead of ETC is an order of magnitude greater than the overhead of LTC, which in turn is an order of magnitude greater than the overhead of LBP. (6.1)
- ETC has low efficiency at fine granularities; both LTC and LBP have higher efficiency across the granularity spectrum. (6.2)
- Without the scheduling changes for "standard" Mul-T, LTC outperforms LBP at all granularities; with the scheduling changes, LTC and LBP have comparable performance. (6.2)
- With the scheduling changes, benchmark performance improves 4–30% with ETC and 3–43% with LBP. (6.4)
- All strategies are less efficient when task trees are shallow; LTC suffers the least. (6.3)
- LTC creates substantially fewer tasks than LBP. (6.5)
- Polite stealing significantly reduces the number of tasks created with both LTC and LBP. (6.5)
- LBP degrades the performance of `fatwalk`. (6.6)
- Both LTC and LBP perform substantially better than ETC for the fine-grained benchmarks. (6.6)
- LTC has close to linear speedup for most benchmarks; speedup of LBP is almost as good. Speedup of both methods suffers on programs with shallow task trees. (6.6)

	Overhead of operation (μsec)				
	`future`			`touch`	
Program	ETC	LBP	LTC	LBP	LTC
`abisort`	152.3	5.1	10.5	1.4	1.6
`allpairs`	177.6	20.6	25.1	5.2	3.7
`fib`	145.1	1.2	9.7	1.6	1.6
`mst`	149.9	0.2	14.2	2.7	1.0
`queens`	149.1	1.1	12.4	1.4	1.4
`rantree`	152.2	1.0	10.9	1.6	1.6
`tridiag`	156.7	6.1	14.6	12.2	0.2

Table 6.1: Overhead per call of `future` and `touch` in sequential code, in microseconds, for each benchmark program and partitioning strategy.

6.1 Overhead of Partitioning Strategies

How much overhead do the partitioning strategies add to the execution of real programs? Table 6.1 shows one angle on this question, reporting the overhead per call to `future` and `touch` in *sequential* code for each program in the benchmark suite. This may be seen as measuring the minimum impact of a partitioning strategy, as multiprocessing overheads like idle processors or contention for shared resources are not present.

For each partitioning strategy, three versions of each program were compiled: one with `future` and `touch` ignored, one with just `future` ignored, and one with neither ignored. Each version was executed with all three partitioning strategies using just one processor and averaged over 40 trials. The overhead figures were derived from these timings; for example, the cost of `future` was determined by computing the difference in execution time between the latter two versions and dividing by the total number of calls to `future`.

The ETC column shows the overhead introduced when a task is created on every call to `future`. The LTC column shows only the overhead of lazy future call and return; there was no task creation overhead because only one processor was active. Similarly, the LBP column shows only the overhead of testing the local task queue length—a threshold of $T = 0$ was used so that no tasks would be created.

These figures show a surprising amount of variability. I conclude from this variability not that the overheads vary highly from program to program, but rather that these measurements were only partially successful in factoring out other sources of variability. Support for this conclusion may be seen in the `touch` columns for LBP and LTC—the two implementations are executing exactly the

same instructions for `touch` but yet the computed overhead of `touch` can differ greatly between LBP and LTC for the same program. This may be caused by the variability in instruction alignment discussed in Section 5.2, or by some other dynamic not yet identified.

Despite the variability in Table 6.1, the overall picture which emerges is consistent with what one would expect after considering the instruction counts given in Section 4.2. LBP has very low overhead, LTC about an order of magnitude greater, and ETC yet another order of magnitude greater than that. It might seem surprising that LTC costs 10 times as much as LBP when only 8 instructions of overhead are required compared with 2 for LBP. The difference is that LTC requires an expensive synchronization instruction (as explained in Section 3.3.2) guaranteed to cause a cache miss, while the LBP instructions rarely cause a cache miss. Still, the gap could likely be narrowed on a machine allowing a less clumsy method of synchronization than that described in Section 3.3.2.

A somewhat more controlled comparison of the partitioning strategies using multiple processors is given in the next section.

6.2 How Granularity Affects Efficiency

We observed initially that eager task creation performs poorly with fine-grained programs. Now we may examine just how poor the performance of ETC actually is, and how much improvement is seen with dynamic partitioning. Using program `tree` (described in Section 5.3), we can measure the effectiveness of the various task-creation strategies over a range of task granularities. Recall that `tree` executes a delay loop of a specified length, allowing granularity control. By timing trials using a range of granularities we can get an "efficiency profile" for each task-creation strategy. Note that the first profiles we shall consider were measured without the scheduling improvements for ETC and LBP.

Figure 6.1 shows these profiles, with source granularity along the the x axis (in a log scale) and efficiency along the y axis. Each curve shows how efficiency varies for a particular partitioning strategy as granularity is increased from 3 to 3003 instructions to execute the delay loop at each node in `tree`. (The instructions which execute the basic divide-and-conquer loop are not counted here.) The efficiency E for a given trial is calculated using the formula

$$E = \frac{t_{seq}}{nt_{par}}$$

where in this case the sequential time t_{seq} is for a Mul-T program compiled with `future` and `touch` ignored and the parallel time t_{par} is measured using $n = 16$ processors. Efficiency of 1.0 means perfect speedup. The tree depth of 16 used in these trials ensures many tasks so that processor idle time at start-up and tail-off has minimal effect and close-to-perfect speedup should be achievable.

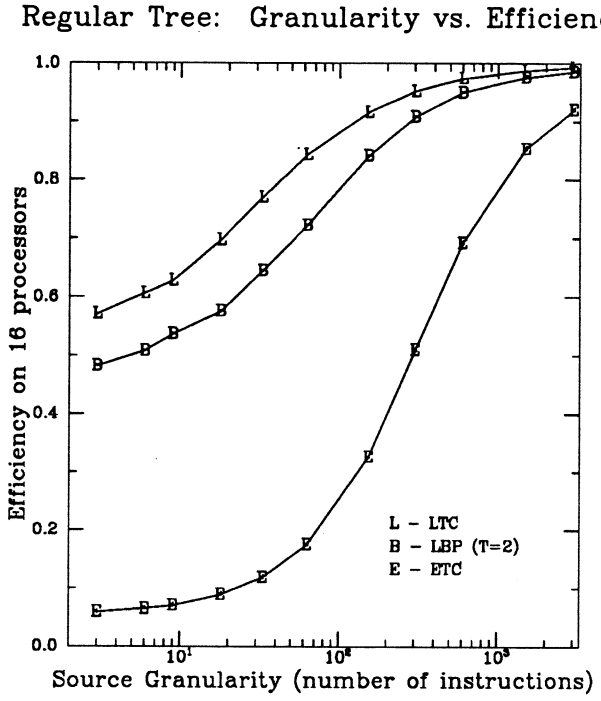


Figure 6.1: Granularity *vs.* efficiency for `tree` benchmark, by partitioning strategy.

As expected, the high cost of eager task creation leads to very poor efficiency at fine granularities. With load-based partitioning 90–95% of the 2^{16} tasks are eliminated, improving efficiency substantially. Lazy task creation makes an additional improvement by eliminating more than 99% of the tasks. Still, the overhead of lazy future calls is significant, hurting efficiency at the finest granularities.

Figure 6.2 shows the effect of adding the scheduling improvements for ETC and LBP. In the left-hand plot, the curves from Figure 6.1 for these strategies are shown with lower case labels while the corresponding curves in the "new" system have upper case labels. Performance is noticeably improved for both strategies. The right-hand plot retains these "new" curves and adds the LTC curve from Figure 6.1, showing that LTC and the improved LBP are essentially equivalent; both are still significantly better than ETC.

With the exception of Section 6.4, where the effects of the scheduling improvements are examined in more detail, all subsequent measurements of eager task creation and load-based partitioning were made in the "new" system, which incorporates the scheduling improvements.

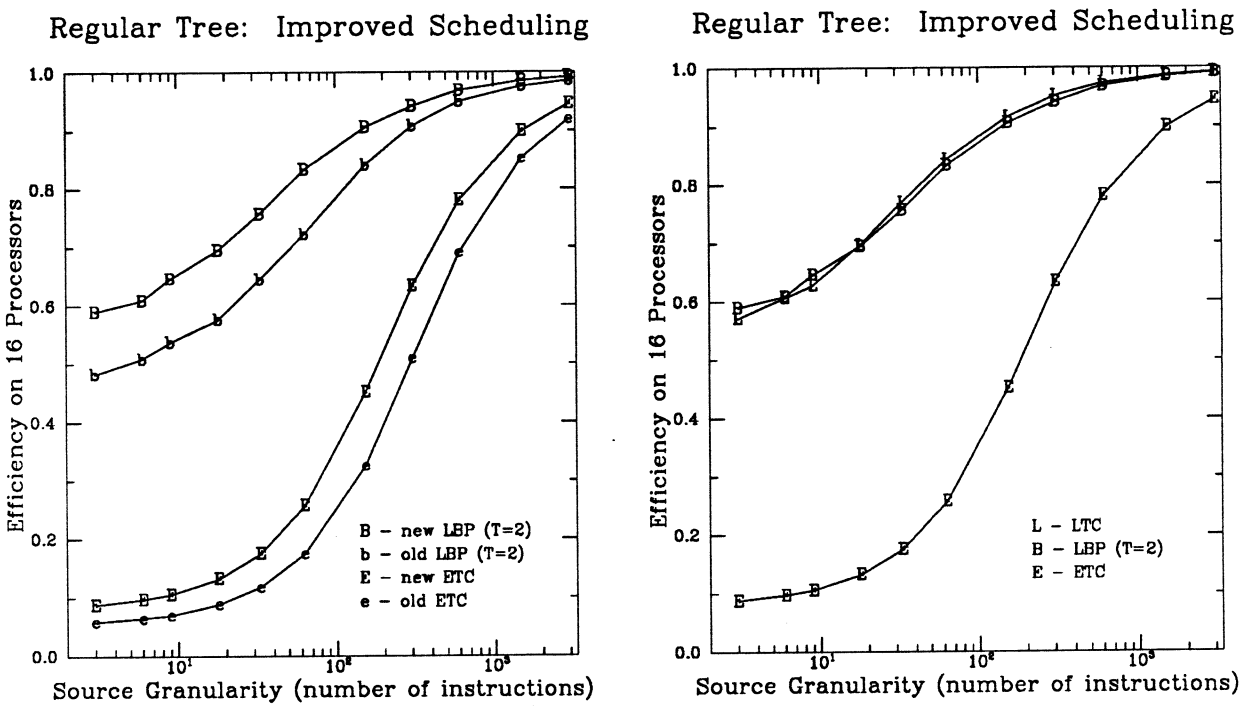


Figure 6.2: Effect of ETC and LBP scheduling improvements on `tree` benchmark.

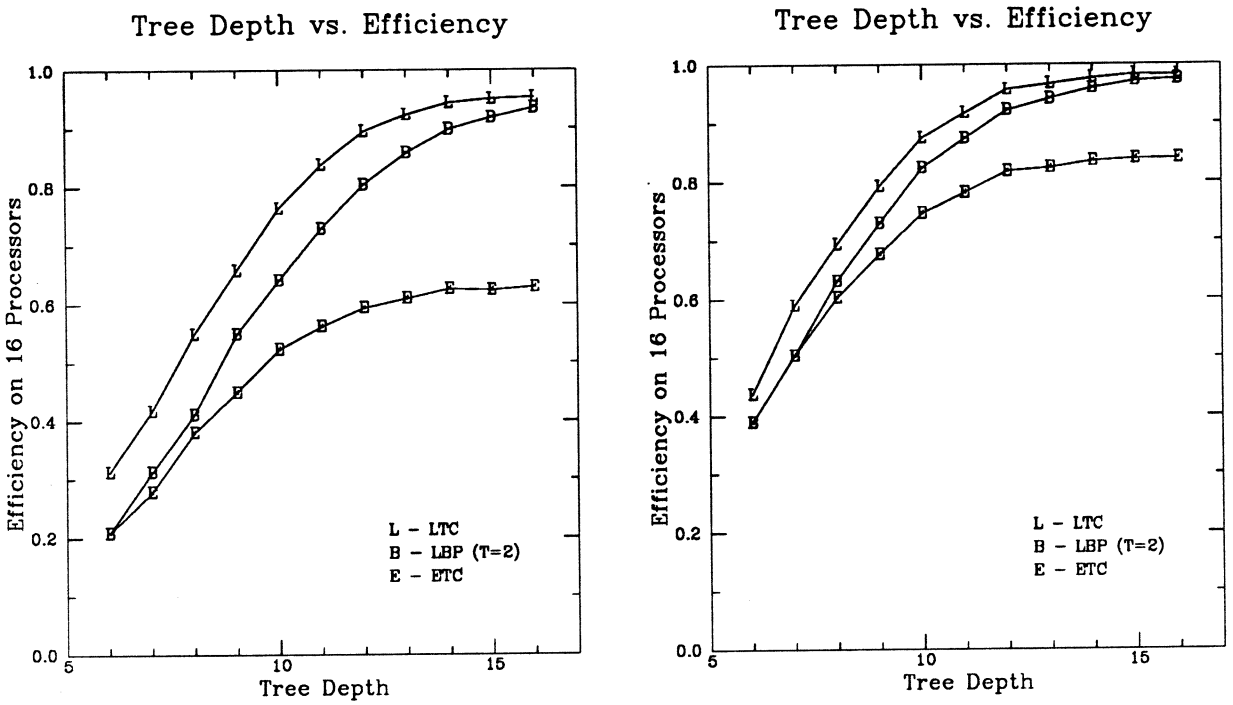
6.3 How Tree Depth Affects Efficiency

`tree` may also be used to see how efficiency varies with call tree depth. Figure 6.3 shows graphs similar to those in the previous section, except that tree depth varies along the x axis while granularity is constant at $g = 307$ for the left-hand plot and $g = 867$ for the right.[1] As before, 16 processors were used for all timings.

Two factors limit efficiency in this benchmark; first there is the familiar overhead of task creation. With small trees no strategy is particularly efficient because most potential fork points lead to task creation. But as the tree depth and thus the number of potential fork points rises, the efficiency of LTC and LBP also rises—as tasks execute larger subtrees the relative overhead of task creation decreases.

If this were the only factor however, the ETC curves would be horizontal because the overhead of task creation is independent of tree depth with ETC. Since the ETC curves are clearly not horizontal other mechanisms must be at

[1] For comparison with the previous plots, the delay loop contained 303 instructions for the left-hand plot and 1003 for the right.



(a) Source granularity $g = 307$

(b) Source granularity $g = 867$

Figure 6.3: Tree depth *vs.* efficiency for `tree` benchmark, by partitioning strategy.

work; two likely possibilities are idle processors and cache effects.

During execution of the task tree, processors are idle both at the beginning (until enough breadth-first expansion has occurred) and end (until the last processor finishes). For this benchmark, measurements show that the length of these periods of idleness is largely independent of tree depth. But since small trees are executed much more quickly than large trees, the overhead due to a fixed period of idleness effects efficiency much more for small trees than for large trees.

Cache effects are also likely to degrade efficiency more for small trees than for large trees, as the cost in cache misses of migrating a subtree of tasks to another processor is amortized over a shorter period of execution.

So we see that several factors limit efficiency for small subtrees, explaining the shape of the curves in Figure 6.3. Note also that LBP is only marginally better than ETC for very small trees and that LTC performs somewhat better than LBP across the board.

This data will be important when interpreting the results of benchmarks `mst`

and `allpairs`, where tree sizes are rather small.

6.4 Effect of ETC and LBP Scheduler Changes

Section 4.1 presented three changes to the scheduler used with ETC and LBP, and claimed that the changes improved the performance of both methods despite increasing the cost of task creation. Here we consider evidence for this claim, beginning with a summary of the three changes:

1. Favor child instead of parent. With the old policy a forking processor would queue the child task and then continue executing the parent task. With the new policy a forking processor will queue the parent task and execute the child task.

2. Steal FIFO instead of LIFO. With the old policy an idle processor would remove tasks from remote queues in a LIFO manner. With the new policy an idle processor will remove tasks from remote queues in a FIFO manner.

3. Steal politely. With the old policy an idle processor i would poll starting with processor $i + 1$ (mod p, the number of processors). With the new policy an idle processor remembers which processor j was last stolen from and will poll starting with processor $j + 1$ (mod p).

The first experiment tests combinations of the first two changes. Table 6.2 shows statistics for programs `tree` (with depth 14 and $g = 307$) and `rantree`, run with and without each change using 16 processors.

For both programs the fastest time is achieved when both changes are active, under both ETC and LBP. Favoring the child task improves performance substantially because there are many fewer blocked tasks; also, fewer tasks are stolen. Using a FIFO instead of a LIFO stealing policy has a smaller but still noticeable effect; fewer tasks are created with LBP and fewer tasks are stolen with both ETC and LBP.

Evidence that the third change, polite stealing, decreases the number of tasks created with both LBP and LTC will be presented in the following section.

We now consider the cumulative effect of all 3 scheduling changes for each program in the benchmark suite. Table 6.3 compares statistics for running each program in both "old" Mul-T (without the scheduling changes) and "new" Mul-T (with the scheduling changes), with both ETC and LBP. The parenthesized figures show the percent improvement in the new system compared to the old—performance of every program is improved for both partitioning strategies, some quite dramatically.

A conclusion that the scheduling changes are beneficial seems warranted.

tree						
Partitioning Strategy	Scheduling Preference	Stealing Policy	Elapsed Time	Number of Tasks		
				Created	Blocked	Stolen
ETC	Favor Parent	LIFO	.63	16383	16165	2063
		FIFO	.66	16383	16263	488
	Favor Child	LIFO	.55	16383	49	1326
		FIFO	.53	16383	53	311
LBP ($T = 2$)	Favor Parent	LIFO	.41	3018	2909	804
		FIFO	.41	2773	2654	606
	Favor Child	LIFO	.40	3048	56	556
		FIFO	.39	2321	58	316

rantree						
Partitioning Strategy	Scheduling Preference	Stealing Policy	Elapsed Time	Number of Tasks		
				Created	Blocked	Stolen
ETC	Favor Parent	LIFO	.58	19180	17895	2842
		FIFO	.60	19180	18704	1047
	Favor Child	LIFO	.44	19180	903	2216
		FIFO	.43	19180	289	750
LBP ($T = 2$)	Favor Parent	LIFO	.19	3835	3429	1234
		FIFO	.20	3666	3309	1092
	Favor Child	LIFO	.17	3468	294	825
		FIFO	.16	2877	185	536

Table 6.2: How scheduler changes to favor the child task and adopt a FIFO stealing policy improve performance for `tree` and `rantree`.

6.5 Number of Tasks Created

Section 2.2.3 argued why lazy task creation is more effective than load-based partitioning at eliminating unnecessary tasks. Table 6.4 backs up this claim by showing the number of tasks created by each partitioning strategy for each benchmark program. The parenthesized figures give the number as a percentage of the total possible number of tasks. (For LBP, $T = 2$ gave the best time on 16 processors for all programs except `abisort`, where $T = 1$ was used.)

For all programs, LTC creates substantially fewer tasks than LBP. LTC eliminates 99% of all tasks for all programs except `mst` and `allpairs`, which have shallower trees than the other programs. (A fuller discussion of the performance of these two programs appears in the following section.)

Program	ETC Elapsed Time Old	ETC Elapsed Time New (saved)	LBP Elapsed Time Old	LBP Elapsed Time New (saved)	LBP Tasks Created Old	LBP Tasks Created New (saved)
`allpairs`	1.39	1.33 (4%)	1.35	1.30 (3%)	11616	10964 (6%)
`mst`	10.22	9.43 (8%)	7.64	7.18 (6%)	151319	154216 (-1%)
`tridiag`	1.86	1.44 (23%)	.85	.75 (12%)	6671	4637 (30%)
`abisort`	3.35	2.49 (26%)	1.08	.92 (15%)	17862	16241 (9%)
`rantree`	.55	.41 (25%)	.18	.13 (28%)	3934	2564 (35%)
`queens`	1.10	.85 (23%)	.41	.29 (28%)	6318	2673 (58%)
`fib`	3.28	2.30 (30%)	.47	.27 (43%)	11291	3021 (73%)

Table 6.3: How scheduler changes improve performance for benchmark programs.

Program	Number of Tasks Created ETC	Number of Tasks Created LBP	Number of Tasks Created LTC
`fib`	121392 (100%)	3021 (2%)	160 (0%)
`queens`	34814 (100%)	2673 (8%)	361 (1%)
`tridiag`	49150 (100%)	4637 (9%)	357 (1%)
`abisort`	106496 (100%)	10810 (10%)	704 (1%)
`rantree`	19180 (100%)	2564 (13%)	224 (1%)
`mst`	249001 (100%)	154216 (62%)	29210 (12%)
`allpairs`	13572 (100%)	10964 (81%)	4673 (34%)

Table 6.4: Number of tasks created in Mul-T benchmarks, by partitioning strategy.

Table 6.5 shows another angle on this issue, counting the number of tasks created in program `tree` as the tree height h and number of processors p are varied. The table's four blocks show this data for both LTC and LBP, with and without the polite stealing policy described in Section 4.1.3. Polite stealing sharply reduces the number of tasks created under both strategies, especially as tree size and number of processors increase.

As expected, LTC creates substantially fewer tasks than LBP across the board. For example, LTC creates fewer tasks with 16 processors than LBP creates with just 2 processors, for all tree sizes (with polite stealing).

These task counts may also be compared with the worst-case theoretical predictions from Section 2.2.3 of $O(hp^2)$ tasks for LTC and $O(h^5p^2)$ tasks for LBP with $T = 2$. While the actual task counts are certainly lower than the worst-case

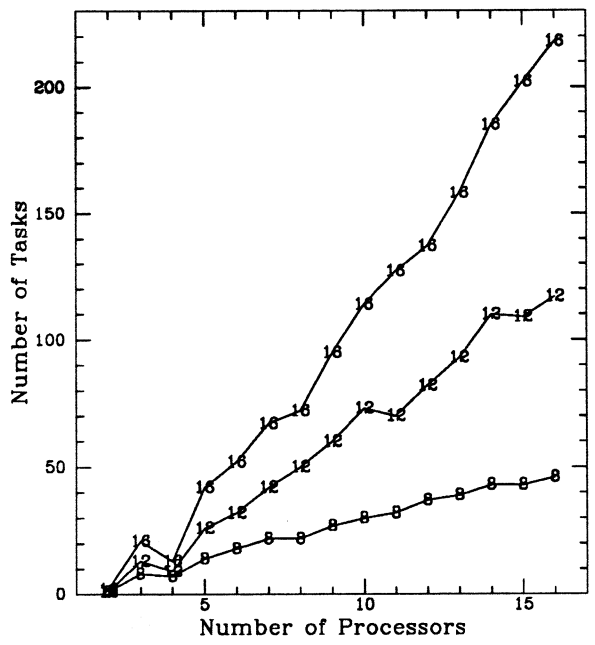
		Tree Height h											
		(with Polite Stealing)						(without Polite Stealing)					
	p	6	8	10	12	14	16	6	8	10	12	14	16
	2	1	1	2	1	1	2	1	1	1	1	2	1
	4	6	7	8	9	12	13	5	6	7	9	12	9
L	6	12	18	25	32	43	52	12	18	26	36	51	67
T	8	15	22	40	50	61	72	14	20	33	54	69	111
C	10	18	30	46	73	90	114	18	32	54	92	139	214
	12	21	37	58	82	110	137	21	36	61	102	202	415
	14	23	43	69	110	147	185	23	42	72	123	277	578
	16	25	46	77	117	155	218	24	45	84	147	275	580
	2	32	63	104	161	219	294	33	63	102	154	219	278
L	4	42	106	211	434	691	1065	43	106	215	386	664	909
B	6	47	127	298	621	1170	1998	47	128	302	682	1252	2437
P	8	51	138	334	748	1341	2438	50	140	347	750	1607	3212
	10	54	149	358	789	1553	2990	53	151	382	896	2122	4079
T	12	56	159	390	868	1718	3274	56	160	408	975	2256	5317
2	14	58	165	401	921	1882	3868	58	167	435	1023	2463	5733
	16	60	172	416	963	1997	4005	59	173	445	1091	2562	6226

Table 6.5: How task count varies with tree depth and number of processors.

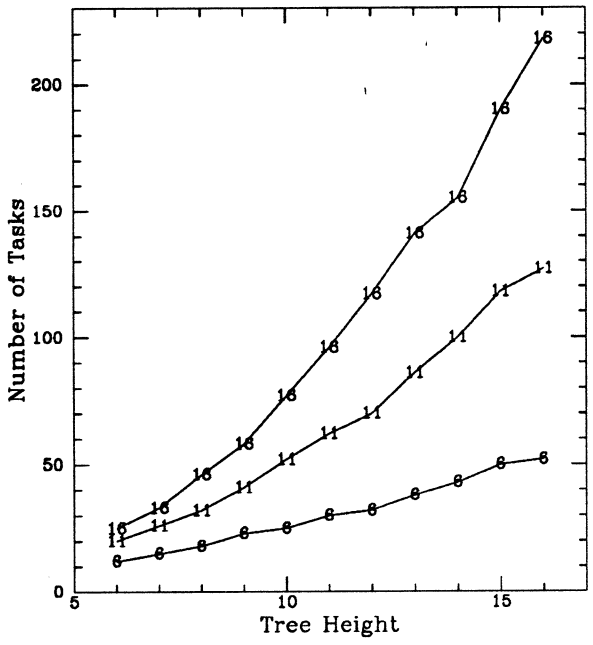
predictions, the conclusion that LBP has a higher exponent on h than LTC is supported by looking at various cross sections of the data.

Figure 6.4 shows four such cross-section plots of task counts from Table 6.5. The upper two plots show data for LTC while the lower plots show data for LBP; the left-hand plots vary p for fixed h while the right-hand plots vary h for fixed p. For example, plot (a) shows how the number of tasks created with LTC varies with the number of processors for trees of height 8, 12, and 16.

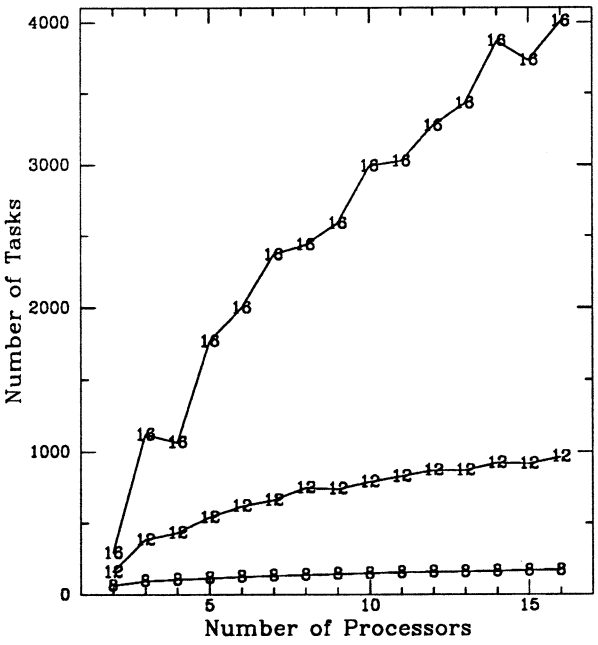
One could make a very rough estimate based on these graphs of the proper exponents on p and h for LTC and LBP. The curves for LTC are close to linear, suggesting that the number of tasks created in the `tree` benchmark is roughly $O(hp)$. With LBP the curves appear essentially linear in p (or perhaps with an exponent slightly less than 1) but at least quadratic in h, suggesting that for LBP the number of tasks created is roughly $O(h^a p)$ for some $a \geq 2$.



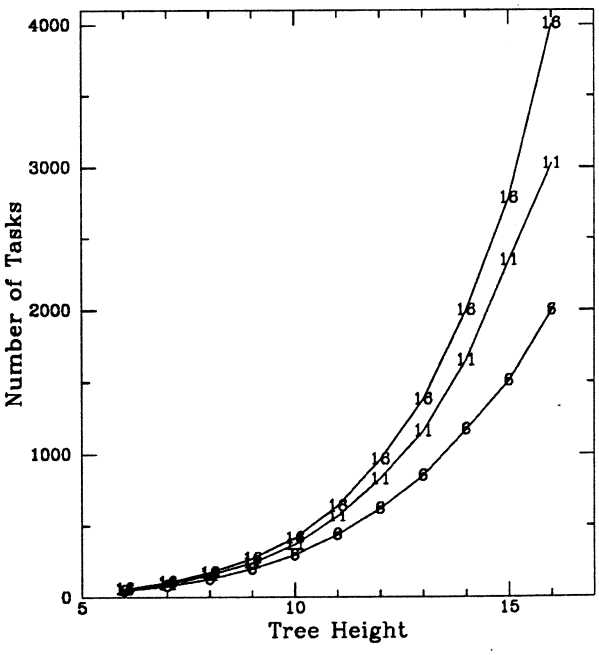
(a) LTC: p varies for $h = 8, 12, 16$.



(b) LTC: h varies for $p = 6, 11, 16$.



(c) LBP: p varies for $h = 8, 12, 16$.



(d) LBP: h varies for $p = 6, 11, 16$.

Figure 6.4: Task counts for fixed tree height or fixed processor count.

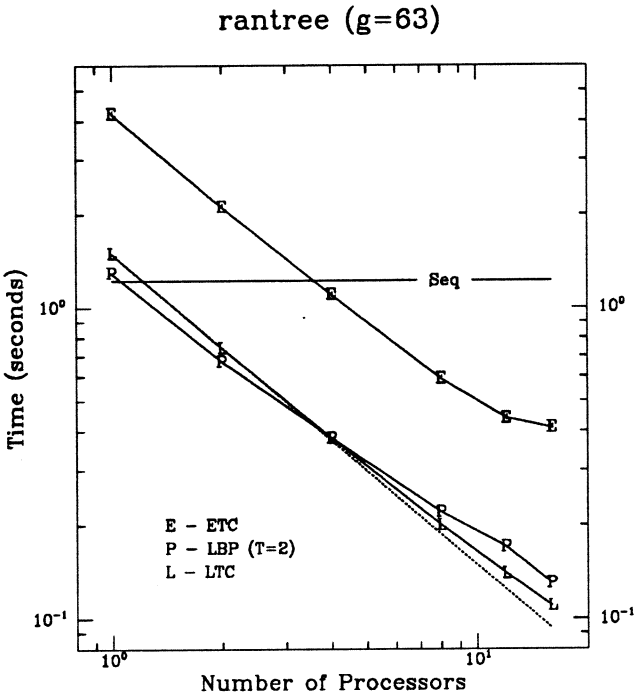


Figure 6.5: Performance of `rantree`.

6.6 Performance of Benchmark Programs

Finally we arrive at the long-awaited benchmark speedup curves. For the benchmark programs, how does performance improve as processors are added for the various partitioning strategies? Consider Figure 6.5, which shows the performance of `rantree` as well as introducing the format used in all subsequent speedup plots.[2]

A log/log format was chosen for these plots because it presents information about both absolute times and relative speedup in a single plot [Halstead 91]. The horizontal line labelled "Seq" shows the sequential time, that is, the elapsed time using one Mul-T processor for a version of `rantree` compiled with `future` and `touch` ignored. The curve labelled "E" shows times for `rantree` run using eager task creation on 1, 2, 4, 8, 12, and 16 processors; we see that the sequential time is only equalled after 4 processors are used to make up for the high overhead of eager task creation. The curves labelled "L" and "P" show times for `rantree` compiled for lazy task creation and load-based partitioning ($T = 2$) respectively.

[2]The data presented in the speedup plots also appears in Table 6.6 at the end of this section.

Both curves begin close to "Seq", showing that dynamic partitioning has much lower overhead than ETC, with LBP lower than LTC. As processors are added, LTC makes up for this initial deficit and surpasses LBP; LBP loses efficiency by creating more tasks than necessary.

A dotted line showing ideal linear speedup for LTC extends from the first "L" but is only visible at the lower right of the plot; we see that the speedup of LTC is much closer to linear than the speedup of LBP for this program. Only this single reference line is drawn to avoid cluttering the graph. A line beginning at "Seq" might seem more honest, but the point here is to judge the relative speedup of the partitioning strategies rather than to discuss issues of sequential overhead and absolute performance. These latter issues are discussed in detail elsewhere (*e.g.* Sections 5.1 and 5.1.2)

6.6.1 `fatwalk`

Next we consider program `fatwalk`, a very coarse-grained iterative program designed as a worst case to show how load-based partitioning can degrade the performance of a program. 100 coarse-grained tasks are created in an iterative loop resembling `parmap-cdrs` (see Section 4.3); the results are accumulated in a list which is walked sequentially to detect termination.

Consider Figure 6.6. All 3 curves begin right at the "Seq" line—coarse granularity means that task creation overhead is negligible for all strategies. Speedup is nearly linear with both ETC and LTC, with slight deviation due to idle processors in the "tail-off" phase. But performance is much worse with LBP; with 16 processors `fatwalk` takes 4 times longer than with ETC or LTC. Very bad load balancing due to irrevocable partitioning decisions explains the slowdown, as predicted in Section 2.1.2.

The astute reader may complain that choosing $T = 1$ for this program stacks the deck against load-based partitioning, since it presumably leads to the fewest number of actual forks and the most "welding" of parent and child. But, the point of this benchmark is to demonstrate that a value like $T = 2$ which works well for many programs will work badly for others, so only $T = 1 - -3$ were tried. In fact, $T = 1$ performed better than either $T = 2$ or $T = 3$ for `fatwalk`. The conclusion is that users of LBP must think about setting T; certainly in this program LBP would perform quite well (in fact be equivalent to ETC) if $T = 100$ were chosen.

6.6.2 Four Programs with Excellent Speedup

That dynamic partitioning allows efficient execution of fine-grained programs is clearly shown by the four plots in Figure 6.7. ETC performs poorly for these four benchmarks due to high task creation overhead—in `abisort` for example, creat-

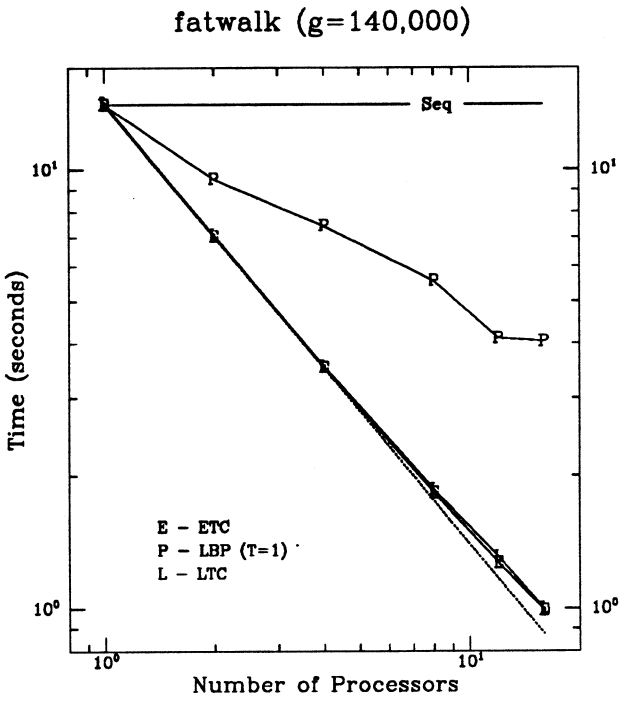


Figure 6.6: Performance of `fatwalk`.

ing tasks eagerly more than triples the sequential running time. The overhead of dynamic partitioning is much smaller, and LTC gives nearly linear speedup since task creation costs rise very slowly as processors are added. LBP shows greater variation, but its performance is essentially comparable to LTC's for these programs.

In `fib`, the large tree height caused by using $n = 25$ allows LBP to eliminate 98% of the possible tasks and come out ahead of LTC with 16 processors. In `abisort` and `queens` the speedup gained with LBP becomes less linear as processors are added. In general, LTC shows better relative speedup than LBP, suggesting that LTC will scale better to larger systems.

For `abisort` and `tridiag` the "Best" line shows the time to run a good sequential algorithm using one Mul-T processor, `mergesort` and `tribest` respectively. As mentioned earlier, speedup of both `abisort` and `tridiag` is close to linear with dynamic partitioning so they outperform the sequential competition when more than 4 processors are used.

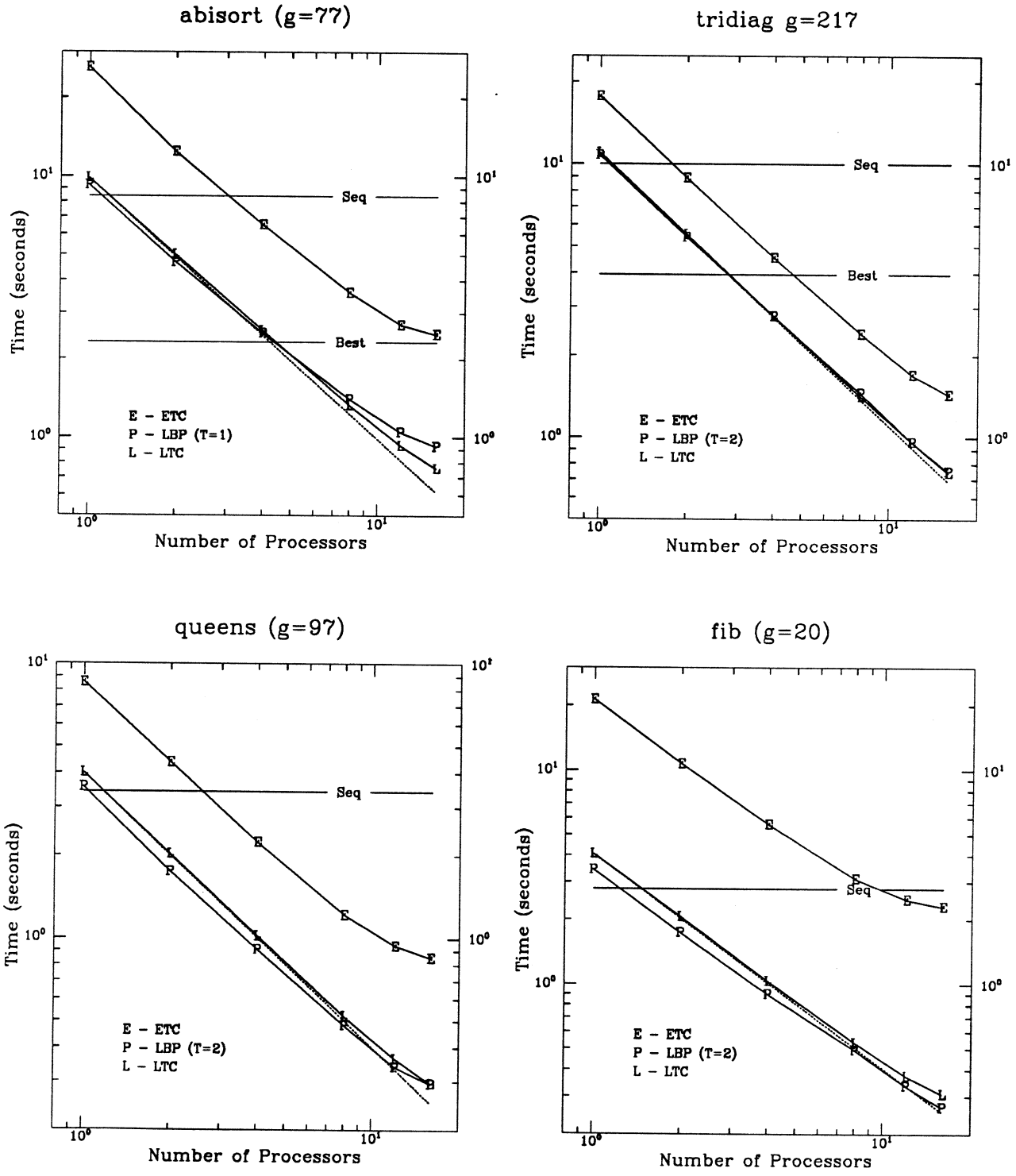


Figure 6.7: Performance of `abisort`, `tridiag`, `queens`, and `fib`.

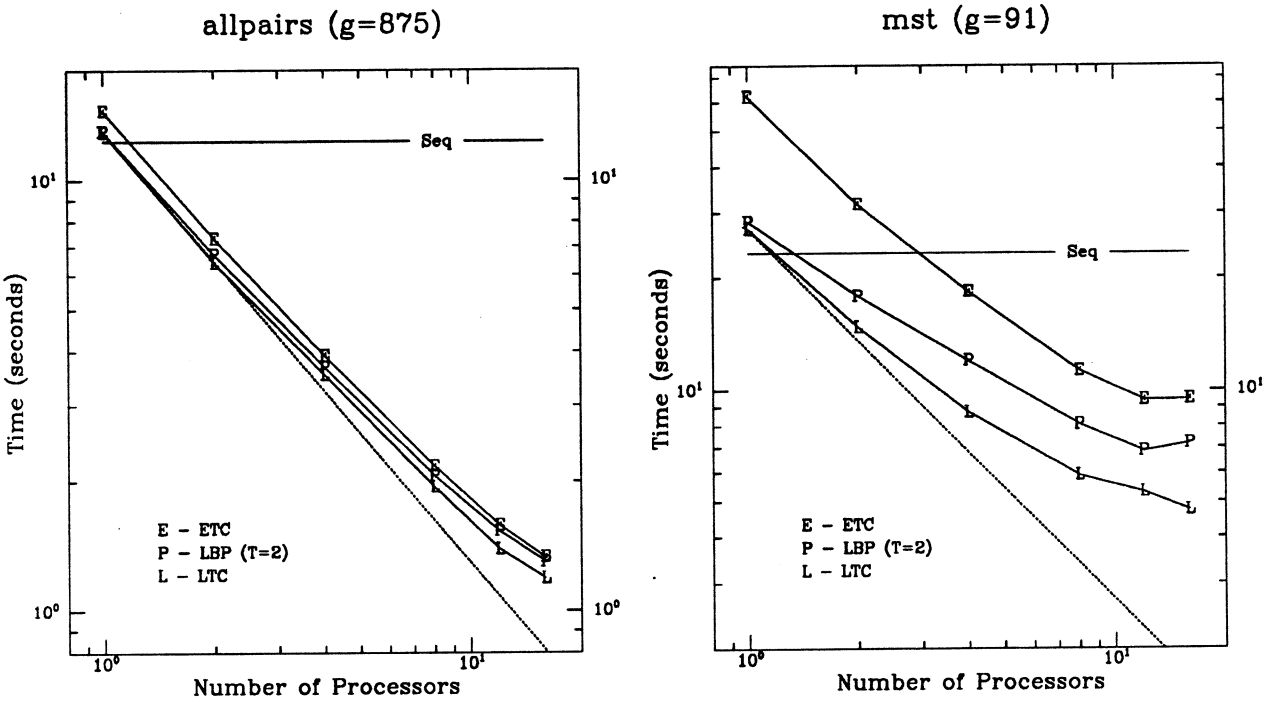


Figure 6.8: Performance of `allpairs` and `mst`.

6.6.3 Two Programs with Not Quite as Good Speedup

It is hard to find fault with the performance of dynamic partitioning for the programs in Figure 6.7. Next we consider two programs where the picture is not quite as rosy, `allpairs` and `mst`, shown in Figure 6.8.

Because `allpairs` has relatively coarse granularity the overhead of ETC is not a large factor; dynamic partitioning does improve performance modestly, but the curves for the three strategies are quite similar. The small margin between LTC and LBP widens slightly as more processors are added so that LTC is marginally better on 16 processors. Perhaps more significantly though, speedup is not quite as good for this program as previously observed with dynamic partitioning. Here we see the effect of the choice to have each (potential) task handle all n vertex pair tests in a single matrix row. Since there are 117 rows, the divide-and-conquer task tree is fairly shallow. As we saw in Section 6.3, dynamic partitioning is less efficient for shallow trees both because a higher percentage of fork points lead to task creation and because processor idleness at start-up and tail-off is more significant. Another interpretation of the latter factor is that processors spend more time waiting at barriers between the sequential steps.

These factors could be avoided by using a finer-grained parallelization where each potential task handles just one vertex test. The problem with this approach is that adding divide-and-conquer control structure to the inner loop would introduce significant runtime overhead. Here we begin to get into issues of how to deal with fine-grained iterative parallelism; let us postpone this discussion until Section 8.1. Of course, each potential task could handle several vertex tests, but that would require introducing explicit parameterization into the source code.

Like `allpairs`, `mst` is structured as a sequence of parallel steps separated by barrier synchronizations. The problem size of $n = 1000$ used here means that the average step handles 500 items (recall that one item is eliminated in each step) so the average task tree is somewhat larger in `mst` than in `allpairs` (500 *vs.* 117). But `mst` is also substantially finer-grained than `allpairs`, so the effects of shallow trees are more pronounced. The result, seen in the right-hand plot of Figure 6.8, is that speedup tails off sharply with all partitioning strategies. Still, dynamic partitioning performs better than ETC, and LTC is significantly better than LBP.

`mst` is a fine-grained iterative program, about which more will be said in Section 8.1.

6.6.4 Numerical Data

For completeness, performance statistics for all programs in the benchmark suite appear in Table 6.6. For each program and each partitioning strategy the elapsed time and number of tasks created are shown as the number of processors p increases. An additional quantity is added, namely the number of "steals", or tasks transferred from one processor's queue to another. With LTC this column is omitted since it is the same as the number of tasks created (every task creation involves a steal). The number of steals is roughly comparable among partitioning strategies, except that LTC always has fewer than both ETC and LBP.

With LBP, $T = 2$ was used for all programs except `abisort`, where $T = 1$ was used.

The rows labelled "seq" and "best"show the same quantities as the lines labelled "Seq" and "Best" in the plots—"seq" gives the elapsed time using one Mul-T processor for a version of the program compiled with `future` and `touch` ignored, and "best" gives the Mul-T time for running the best sequential version of the benchmark.

If one had to pick a single strategy to use for all programs—referring to the data in this chapter as a guide—LTC would clearly be the best choice. After considering all of the different angles, the cumulative performance of LTC is superior to the performance of LBP.

Program	p	ETC time	ETC tasks	ETC steals	LBP time	LBP tasks	LBP steals	LTC time	LTC tasks
`abisort`	best	2.19			2.19			2.21	
($g = 77$)	seq	8.23			8.23			8.27	
	1	26.15	106496	0	9.12	104	0	9.83	0
	2	12.37	106496	8	4.58	535	10	5.00	9
	4	6.51	106496	106	2.40	2215	127	2.57	72
	8	3.58	106496	395	1.43	7761	1075	1.33	259
	16	2.49	106496	1115	.87	10810	2105	.76	704
`allpairs`	seq	11.88			11.88			11.93	
($g = 875$)	1	14.50	13572	0	12.99	2691	0	12.94	0
	2	7.31	13572	205	6.71	4433	232	6.42	120
	4	3.91	13572	1520	3.68	6861	1305	3.53	924
	8	2.17	13572	3442	2.06	8809	3398	1.94	2606
	16	1.33	13572	5629	1.30	10964	5509	1.19	4673
`fib`	seq	2.37			2.37			2.48	
($g = 20$)	1	21.24	121392	0	3.42	90	0	4.06	0
	2	10.72	121392	9	1.74	471	9	2.07	6
	4	5.58	121392	37	.90	1209	56	1.04	23
	8	3.10	121392	.94	.49	2518	192	.54	70
	16	2.30	121392	268	.27	3021	286	.31	160
`mst`	seq	22.68			22.68			22.75	
($g = 63$)	1	61.99	249001	0	28.37	29746	0	27.13	0
	2	31.59	249001	3331	17.85	59557	3156	14.74	2344
	4	18.33	249001	11218	11.98	92539	10413	8.71	7384
	8	11.27	249001	25175	8.08	122215	23065	5.89	15942
	16	9.43	249001	48085	7.18	154216	39439	4.74	29210
`queens`	seq	3.37			3.37			3.39	
($g = 91$)	1	8.54	34814	0	3.56	82	0	4.01	0
	2	4.38	34814	21	1.75	340	18	2.03	14
	4	2.23	34814	76	.91	830	71	1.02	62
	8	1.21	34814	194	.48	1498	181	.52	169
	16	.85	34814	432	.30	2673	455	.30	361
`rantree`	seq	1.21			1.21			1.22	
($g = 97$)	1	4.21	19180	0	1.29	40	0	1.49	0
	2	2.11	19180	10	.67	258	8	.74	6
	4	1.10	19180	48	.38	1106	96	.38	34
	8	.59	19180	124	.22	1939	244	.20	90
	16	.41	19180	328	.13	2564	424	.11	224
`tridiag`	best	3.91			3.91			3.94	
($g = 217$)	seq	10.69			10.69			10.13	
	1	17.71	49150	0	10.76	225	0	10.98	0
	2	8.89	49150	8	5.40	454	7	5.53	5
	4	4.54	49150	54	2.78	1651	55	2.78	29
	8	2.39	49150	167	1.45	3042	177	1.41	127
	16	1.44	49150	394	.75	4637	360	.74	357

Table 6.6: Performance of Mul-T benchmark programs.

A summary of important conclusions will appear after the following discussions of salient related work and future work.

Chapter 7

Related Work

Several categories of research in dynamic partitioning deserve consideration in comparison to the dynamic methods explored here. First, some systems implement methods very similar to load-based partitioning. Second, there are several methods resembling lazy task creation in that some task creation overhead is postponed until it is clear that a task will be executed remotely. Third, two other researchers have implemented full lazy task creation as presented here. Finally, some other more distantly related dynamic methods are considered.

7.1 Methods Resembling Load-Based Partitioning

Methods like load-based partitioning have often been proposed as an easy way to reduce overhead in task-based parallel runtime systems. In many cases the method is mentioned only in passing; here we shall consider two works where LBP-like methods receive more detailed attention. In both, LBP is referred to as simply "dynamic partitioning".

Weening made a detailed study of load-based partitioning in Qlisp [Weening 89, Pehoushek & Weening 89], finding that it gave better performance while requiring less work from the programmer than the cutoff-based methods originally proposed for that language. His insightful analysis of the number of tasks created with LBP points out some of the difficulties with that method, and motivated my analysis of the number of tasks created with LTC (Section 2.2.3). However, his conclusion that task creation overhead becomes asymptotically minimal as problem size increases must be set against actual performance results; the graphs of Chapter 6 show that program speedup with LBP suffers when more processors are used and the number of tasks created increases.

Pehoushek and Weening studied the performance of three programs (`boyer`, `fib`, and `tak`) in an implementation of Qlisp with LBP running on an 8-processor Alliant FX [Pehoushek & Weening 89]. They point out that the need to set the threshold parameter T is a drawback of LBP. However, although the dan-

gers of deadlock and performance degradation put forth here (in Section 2.1) could definitely arise with LBP in Qlisp, neither [Pehoushek & Weening 89] nor [Weening 89] mentions these drawbacks.

Another system implementing load-based partitioning is VMMP [Gabber 90]. Gabber makes a distinction between "tree" algorithms and "crowd" algorithms similar to the distinction made here between bushy and spindly task trees. In VMMP load-based partitioning is only used with the `Vcall` primitive, which is intended for use only with tree algorithms. As the dangers of deadlock or performance degradation do not generally arise in tree algorithms, Gabber may perhaps be excused for not discussing these issues.

VMMP is intended as a portable parallel platform for both shared-memory and distributed-memory machines, although [Gabber 90] reports only shared-memory implementations using a maximum of 6 processors. These implementations maintain both a global count of busy processors and a global task queue, allowing a better estimate of system workload than when only local information is used. This may allow LBP to be more effective in reducing the number of tasks created; however, the overhead due to contention in maintaining this global information is likely to become an issue as the number of processors increases. The reported speedup of benchmark programs is excellent, although evaluation is difficult because no "sequential" time appears as a check on system overhead and only 6 processors are used.

The investigation of dynamic partitioning in this work grew out of Halstead's "optimization of `future`" discussion for Mul-T in [Kranz *et al* 89], which did raise the possibilities of performance degradation and deadlock with load-based partitioning (referred to there as "inlining"). One intent of the present work was to provide compelling examples (such as `find-primes` and `fatwalk`) of these and other drawbacks of LBP. [Kranz *et al* 89] also introduced the idea of "lazy futures", which led to the development of lazy task creation as published in [Mohr *et al* 90, Mohr *et al* 91].

7.2 Methods Resembling Lazy Task Creation

Lazy task creation takes advantage of two observations. First, executing a task remotely requires a large number of bookkeeping operations but executing a task locally needn't require any bookkeeping operations (as for example with inlining). Second, when parallelism is abundant most tasks can be executed locally. The essence of lazy task creation is to do as few of the bookkeeping operations as possible when processing a potential fork point while preserving the option to do them later if executing the task remotely becomes desirable.

Several other researchers have derived a similar philosophy based on similar observations and have pared down somewhat the set of bookkeeping operations

performed at potential fork points, but none has reduced the set quite as far as has LTC. The four systems described here all have a flavor of lazy task creation and succeed in reducing task creation overhead somewhat; however, all require more bookkeeping operations than LTC for the common case when tasks are executed locally.

The key difference between LTC and these other systems lies in which branch of a potential fork (the parent or the child) is continued immediately and which is packaged up for possible migration—with LTC the parent is packaged while with the other systems the child is packaged. When the child is packaged, certain bookkeeping operations just cannot be eliminated. First, a closure (or similar object) must be allocated and initialized to contain the code pointer and free variables (or procedure arguments) for the child task so that a stealing processor will have the information necessary to run it. Next, the existence of the child must be publicized by placing some "task" object (usually also newly allocated and initialized) on a queue. Finally, even when the child is ultimately executed locally the task must still be dequeued and the information necessary to run the child must be extracted from the closure.

Actually, similar operations are required when packaging the parent. But the key difference is that in most cases, even if the potential fork point were completely ignored, the parent would have to be packaged up anyway. In the kinds of programs studied here as well as by these other researchers, potential fork points invariably occur at calls where the parent's return address and live registers must be stored on the stack. In other words, a stack-allocated closure for the parent already exists so only the queuing and dequeuing bookkeeping remain. With LTC as implemented in Encore Mul-T this is accomplished by storing a single pointer, resulting in very low overhead when tasks are executed locally.

Each of the following four systems have a form of lazy task creation in the sense described above. At potential fork points the child task is packaged up; it will be executed either remotely if a processor becomes idle or locally otherwise.

- WorkCrews [Vandevoorde & Roberts 88] is built on top of Modula-2+, a version of Wirth's Modula-2 supporting lightweight threads. Here much of the mechanics of lazy task creation must be specified directly in the source code. To initiate lazy task creation at a certain procedure call point the programmer builds a data block containing the call's arguments and calls `RequestHelp(`*proc*`,`*data*`)`. The programmer must also determine later whether the resulting task was stolen using a call to `GotHelp` and call *proc(data)* directly if not; if so, the programmer must supply code to synchronize with and retrieve a value from the stolen task. In Mul-T the programmer's job is much simpler (just inserting `future`) because LTC is integrated directly with the compiler and runtime system; in addition, LTC

has lower overhead for the reasons given above.

Still, although the mechanics of the two systems are rather different their underlying philosophies are quite similar. The authors briefly consider and reject a method resembling load-based partitioning as well as arguing in favor of oldest-first scheduling.

- SOS [Jagannathan & Philbin 91] is a parallel Scheme system also based on T's Orbit compiler, allowing different styles of parallel constructs to be built using a basic thread object and associated operations. The authors show how `future` and `touch` can be built and explain how efficiency is improved by "lazy thread creation". Essentially they have pared down thread creation to the set of bookkeeping operations described above. The system is still under development, but early measurements suggest that lazy thread creation eliminates about 40% of the overhead of task creation [Jagannathan 91]. This reduction in overhead is not as dramatic as that achieved with lazy task creation in Mul-T.

- GRIP is a multiprocessor built to execute functional programs by parallel graph reduction; [Peyton Jones *et al* 89] describes a parallel graph reduction strategy for GRIP where tasks are represented by closures. As with the other systems discussed here a task will be evaluated locally if it is touched by its parent before it has been stolen; this local evaluation saves some bookkeeping operations but the cost of building a closure for the task is still incurred. Realizing the significance of this cost the authors propose to augment their strategy with a method resembling load-based partitioning, where no closure will be built if the system's load is above a certain threshold.

- The final method, discussed in [Pehoushek & Weening 89] and built into an implementation of Qlisp, is restricted to programs programs with a fork/join style of parallelism. The authors explain how in such programs the closure for the child task can be stack-allocated, although they do not explain how a stealing processor gains access to such a task. The mechanism could very well resemble that used for LTC in Encore Mul-T, with the producer posting a queue of stack locations of stealable child tasks and consumers copying out the relevant information. Still, as expected from the argument above, LTC appears to be the more efficient method; also, LTC can be used safely on non-fork/join programs.

7.3 Other Implementations of Lazy Task Creation

In addition to the Encore Mul-T implementation described here, two other implementations (built respectively by David Kranz and Marc Feeley) were motivated by the idea of "lazy futures" first published in [Kranz *et al* 89].

In Kranz's Mul-T implementation of LTC (described in [Mohr *et al* 90, Mohr *et al* 91]) for the impending ALEWIFE machine, a stack is represented as a doubly linked list of stack frames. This stack representation avoids the need to copy frames during a steal operation—since adjacent stack frames are linked by pointers, splitting a stack during a retroactive fork requires only pointer manipulation.

The linked-frame stack representation, together with judicious use of ALEWIFE's special machine instructions which trap when the full/empty bit of a memory location is not as expected, allows lazy task queue operations to be implemented somewhat more efficiently than in Encore Mul-T. This is not the whole story however, since the cost of stack operations in sequential sections of code is different with the linked-frame stack representation than with a conventional contiguous-memory representation. The ALEWIFE Mul-T implementors have been careful to minimize the cost of such operations, but no direct comparison with conventional stacks has yet been performed.

Direct comparison is also difficult because as yet ALEWIFE Mul-T has only run on a detailed simulator. However, it appears that both stack representations allow quite viable implementations of lazy task creation. The presence of full/empty bits and hardware traps in ALEWIFE make the linked-frame representation perhaps more attractive on that platform than on machines like the Encore Multimax without such hardware support.

Feeley has also implemented lazy task creation, in his parallel "Gambit" Scheme system which runs on the BBN Butterfly [Feeley 91]. The greater cost of accesses to shared memory on the Butterfly motivated some different design choices, but in rough outline Feeley's system and Encore Mul-T are rather similar.

Feeley discovered independently a lockless synchronization algorithm similar to the one presented here (Section 3.3.2) as well as an alternative method where consumers interrupt producers to steal tasks. This second method achieves very low overhead at potential fork points, although there is some hidden overhead in polling for interrupts. In addition, his parallel Lisp system provides an efficient integration of LTC with dynamic binding, a strong concept of fairness in scheduling, and coexistence of `future` and `call-with-current-continuation`.

Although there is some overlap between Feeley's work and my own, there is a difference in emphasis. Feeley's emphasis is on providing a "general" implementation of futures so that many other features (such as those listed above) can coexist and on addressing the issues of large-scale shared memory machines. My work emphasizes the development, analysis, and evaluation of dynamic par-

titioning methods in the context of parallel Lisp.

7.4 Other Related Methods

For logic programming languages Debray *et al* present a strategy of static partitioning based on compile-time cost estimates augmented with dynamic runtime tests of quantities such as the size of input to a procedure [Debray *et al* 90]. The work is primarily devoted to describing cost-estimation techniques, and includes a good set of references for other work in that domain. The authors state the goal of minimizing runtime overhead, and the runtime tests in the examples shown appear to be fairly cheap. Performance statistics are given using 4 processors on a Sequent Symmetry. Although some speedup is shown there is not enough data to judge how good a partition is created by their method. Also it appears that the underlying Prolog implementation contains substantial overhead so it is difficult to verify their claim that runtime tests have low cost.

"Throttling" (as pursued in systems based on dataflow [Culler 89, Ruggiero & Sargeant 87] or graph reduction [Greenberg & Woods 90] techniques) has some similarities with load-based partitioning, as system workload information is consulted at runtime as a basis for scheduling decisions. However, the purpose of these scheduling decisions is not to change the partition of operations to tasks, as task granularity is fixed at compile time in these systems (at a very fine granularity in the case of dataflow). Rather, these scheduling decisions improve load balancing (by directing tasks to less-loaded processors), improve data locality (by resisting task migration), or control memory usage (by deciding whether to evaluate the branches of a fork sequentially or in parallel). For the latter purpose BUSD execution is best; after expanding a call tree breadth-first to provide work for all processors, processors expand local trees depth-first so as to avoid swamping task queues and exhausting available memory [Greenberg & Woods 90].

A promising method not directly related to dynamic partitioning is the work of Vivek Sarkar, where information gleaned from execution profiles is used to inform compile-time partitioning decisions [Sarkar 87, Sarkar & Hennessy 86].

Finally, "Guided Self-Scheduling" [Polychronopoulos & Kuck 87] is a dynamic partitioning method developed specifically for programs with loop parallelism, identified in this work as programs with spindly task trees. Here blocks of loop iterations are doled out to requesting processors; the size of each block is determined dynamically by the number of remaining iterations and the number of processors. This method could be used as an alternative "back end" for the spindly task tree strategies discussed in Section 8.1.

Chapter 8

Future Work

Four categories of future work are considered: handling a larger class of programs (those with fine-grained iterative parallelism), using lazy task creation with other parallel languages, using dynamic partitioning on larger parallel machines, and demonstrating that parallel Lisp is useful.

One topic not considered is the possibility of using some global information to reduce the number of tasks created with load-based partitioning. While exploring the tradeoff between contention and task creation overheads might prove interesting, gaining a minor speedup of LBP seems moot because the important problems of deadlock and performance degradation make the method unattractive.

8.1 Handling Fine-grained Iterative Parallelism

For programs with bushy call trees the programmer can use `future` to identify parallelism, effectively ignoring granularity considerations. A remaining challenge are fine-grained programs with spindly call trees, such as those with data-level parallelism expressed iteratively

For example, consider a procedure `doall` which performs a fine-grained operation on all elements of an array. The obvious parallel version of `doall` would create one task per element using an iterative loop, but this version would not execute efficiently in parallel unless its granularity were increased so that tasks handled several array elements instead of just one. Unfortunately, dynamic methods alone cannot partition such a program effectively because they are unable to change program structure. If the iterative structure of this program is obeyed, parallelism is inherently limited.

If instead of using iteration this program were restructured to perform a divide-and-conquer division of the array's index set, we know that lazy task creation could achieve an efficient partition. But such a restructuring has two problems: divide-and-conquer divisions are both more complex to program (and

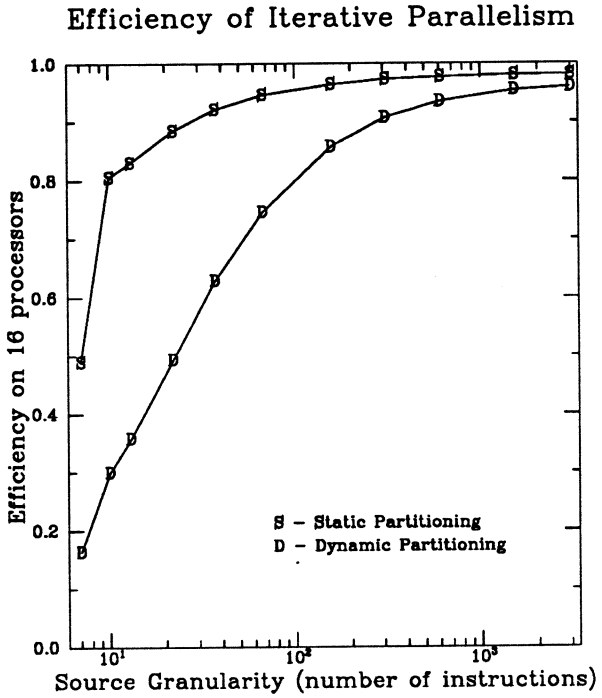
understand) and less efficient to execute than iterative loops.

To address the complexity problem such parallel operations on data aggregates could be expressed at a higher level and converted to appropriate divide-and-conquer divisions at compile time. Ideas for how to express such high-level operations appear in [Waters 90, Steele & Hillis 86, Sabot 88]. One effort assuming such a high-level expression of parallelism [Chatterjee *et al* 91] uses a *static* compiler-based approach to the partitioning problem. In order to use the more flexible *dynamic* partitioning methods developed here the efficiency problem of divide-and-conquer divisions must be solved. This problem arises because the execution overhead of a divide-and-conquer division is large compared to the low overhead of an iterative loop. When loops are fine-grained this additional overhead is unacceptably high.

For example, let us compare the sequential overhead of an iterative *vs.* a divide-and-conquer implementation of `doall`. In Encore Mul-T a simple iterative loop is compiled optimally to 3 instructions; executing these instructions costs 1.4 microseconds per array element on the Yale Multimax. By contrast, the obvious divide-and-conquer division (where the index range is repeatedly divided in half) introduces 19.3 microseconds of overhead per element. However, this overhead can be reduced by half to 9.7 microseconds by using recursive doubling, where a task handling an array element i spawns children to handle elements $2i$ and $2i+1$. And, in the proposed system where such traversals are generated by compiler from high-level operations in the source code, significant further optimizations are possible.

As an experiment, a hand-written assembler version simulating such compiler output for `doall` was written using lazy task creation. This version, highly optimized but still general, achieves an average overhead per element of only 12 instructions. (Lazy future calls account for 4 of these instructions; there are half as many lazy future calls as array elements). This program, `dynamic-doall`, was timed using 16 processors on a 100,000 element array and compared with `static-doall`, which makes a static partition of array elements to 16 tasks. In both versions, each array element was incremented by one after executing a delay loop of a specified length (to allow granularity control).

The following plot shows an efficiency profile for each program. As with the similar plots of Section 6.2, source granularity is plotted along the x axis (in a log scale) with efficiency along the y axis. The curves show how efficiency varies as granularity is increased from 7 to 3007 instructions to process an array element and execute the delay loop. The sequential time used in the efficiency calculation measures a sequential iterative version of `doall` with no `futures`.



No program could be better suited to static partitioning than `doall`, so this comparison shows dynamic partitioning in the worst possible light. Although dynamic partitioning performs decently, static partitioning performs better at all granularities because it still has less runtime overhead. But, other programs might be more difficult to partition statically than `doall`, and might not perform as well if task execution costs were data-dependent. Dynamic partitioning would handle such programs well.

Still, `dynamic-doall` does not perform well at finer granularities, with efficiency below 50% when task granularity is below 20 instructions. How can overhead be reduced at the finest granularities? One observation is that a fine-grained inner loop is very likely to contain straight-line code rather than additional loops or calls to unknown procedures, so estimating its cost at compile time should be straightforward. When such estimates indicate a very fine-grained loop the compiler could unroll a few loop iterations to raise the granularity to a satisfactory level. Such an optimization meshes easily with the recursive doubling algorithm for divide-and-conquer partitioning.

To summarize then, fine-grained data-level parallelism could be handled by specifying parallelism in terms of high-level operations on data aggregates and compiling these specifications into efficient divide-and-conquer traversals. For very fine-grained operations the compiler can ameliorate the the overhead of such traversals by a small amount of loop unrolling.

8.2 Implementing LTC in Support of Other Parallel Languages

From handling a wider class of programs we proceed to handling a wider class of languages. Load-based partitioning places few constraints on a parallel runtime system and has been implemented in support of several types of parallel language. In contrast, fully general implementations of lazy task creation have been built only for Lisp systems based on `future`. Here we explore how well LTC fits in with other parallel languages and runtime systems using a dynamic task or thread creation model.

First, LTC should fit in easily to any parallel Lisp system based on dynamic task creation. LTC does not depend on the unlimited task lifetimes provided by `future`, and could be implemented as described here to support constructs with implicit join points such as Multilisp's `pcall` or Qlisp's `qlet`.

8.2.1 The Algol Family

Languages in the Algol family (such as Pascal, C, and Modula) provide greater challenges, as their runtime systems do not typically offer as much functionality as Lisp runtime systems. Can fully general lazy task creation be implemented for languages in the Algol family? WorkCrews (discussed in Section 7.2) provides a start in this direction, but would have to be improved in several ways to approach the flexibility and efficiency of full LTC. Four changes would be required to upgrade WorkCrews to full LTC:

1. Automate the mechanics of lazy task control which must currently be specified by the programmer.

2. Allow child tasks to execute arbitrary expressions in addition to procedure calls.

3. Package up the parent task for stealing instead of the child task.

4. Lift the fork/join restriction to allow unlimited task lifetimes.

The first change could be made in a fairly straightforward manner by adding compiler support for task control primitives. The compiler could generate code to allocate and initialize a data block using an argument list, to call the child task "in-line" at the join point if it had not been stolen, and to deallocate the data block after the join point.

The last change on the other hand would require fairly extensive modifications to a conventional compiler and runtime system by introducing the need for garbage collection.

If the last restriction (fork/join) were retained however, changes 2 and 3 would be feasible with moderate compiler and runtime system modifications. The need for modification arises because conventional compilers use stack-allocated storage for procedure arguments and local variables. Allowing arbitrary expressions in the child task would introduce complications if both parent and child needed to modify a stack-allocated variable. And packaging up the parent for stealing instead of the child could lead to overwhelming copying costs in the steal operation if for example there were a large stack-allocated array.

These difficulties don't arise in Mul-T because nested lexical environments and mutable storage are always heap-allocated, and because procedure arguments and local variables are stored in registers (actual machine registers or pseudo-registers in a memory block). So mutable variables shared between parent and child have a single heap location; also, the only variables which are stack-allocated and thus copied in a steal operations are live variables needed to continue executing the parent.

To implement changes 2 and 3 efficiently in the Algol world, local variables and procedure arguments accessed by both parent and child must reside in a single location.

This could be accomplished by the following method which involves a a compiler change of moderate complexity and assumes that a task's entire address space resides in shared memory, making its stack and heap accessible to other tasks. Since there is a known join point, a stolen parent task could continue to reference stack-allocated variables in their original locations on the producer's stack. The producer executing the child task would refer to the same locations and would not deallocate them until after the join point.

A compiler change would be necessary because code generated for the parent task would have to alter the way local variables and procedure arguments were referenced. Instead of using the current stack pointer as a base from which to apply offsets, the parent would have to use a pointer to the appropriate frame in the producer's stack. This would add moderately to the complexity of the compiler and possibly increase the cost of executing the parent somewhat.

Several advantages would accrue, however. The child task could contain arbitrary expressions because it would be executed locally by the producer, referencing all variables in their usual locations. This would eliminate the need for a separate data block or closure for the child task. Also, the overhead of creating a task lazily could be substantially reduced, by using a lazy task queue of pointers into the stack as in Encore Mul-T and analogous operations to lazy future call and return. Because of the fork/join restriction the amount of copying required in a steal operation would decrease significantly, since only the live variables needed to continue executing the parent would need to be copied.

In the resulting system lazy task creation would have comparable overhead to LTC in Encore Mul-T, although there might be additional overhead due to

the changed method of accessing variables in the parent task. The flexibility of LTC would be almost as good as in Mul-T, as programs with bushy call trees often obey the fork/join restriction.

8.2.2 Lazy Functional Languages

Finally let us consider lazy functional languages. Traditionally implementations of these languages have used either *graph reduction* [Peyton Jones 87] or else conventional compilation optimized to support lazy evaluation using *delays* [Bloss *et al* 88], though in recent years the distinction between these methods has become somewhat blurred. As suggested in Section 7.4, pure graph reduction is not amenable to dynamic partitioning since building the program graph at compile time effectively establishes a fixed partition of the program's operations. The task structure is built into the program graph so there is no way to increase granularity by combining tasks dynamically at runtime. Runtime decisions can in fact be made to decide whether to reduce a node locally or remotely, but the savings of local execution arise from data locality rather than from a difference in task creation costs—with both remote and local execution the "task object" (a closure, perhaps) has already been built and must be placed on a queue.

On the other hand, implementations using delays are quite amenable to dynamic partitioning and their runtime systems often contain the same features as Lisp runtime systems. Delays implement lazy evaluation by building a "package" containing all information necessary to evaluate an expression; the expression is only evaluated when its value is needed by a *strict* operation like +. There is a very close analogy between delays and futures; packaging up an expression for remote execution requires the same operations as packaging up an expression for delayed evaluation. In the parlance of this thesis a delay may be seen as a packaged-up child task.

Thus an implementation based on delays could easily be extended to use futures for parallel execution, as the "packaging-up" mechanism would already exist. ("Modern" graph reduction systems resemble rather closely systems using delays; [Peyton Jones *et al* 89] describes one such system which does implement dynamic partitioning.) Such a parallel implementation could be used in support of both languages where parallel tasks are identified by the compiler and where they are identified by the programmer. Dynamic partitioning (both LTC and LBP) could be implemented as in Encore Mul-T to decrease the overhead of task creation. But, efficiency issues must be carefully considered given the presence of delays.

Creating a delay for every expression in a functional program leads to unacceptable overhead for the same reason as with eager task creation in a fine-grained program—the lion's share of execution time is spent creating tasks rather than executing them. For this reason researchers have developed methods for *strict-*

ness analysis or annotations, to identify which expressions can be safely evaluated immediately rather than packaged up for delayed evaluation. Although optimizations based on strictness analysis significantly reduce the cost of using lazy evaluation, analysis of programs containing data structures such as lists is difficult and the cost of creating delays remains a major source of overhead in the execution of lazy functional programs.

The goal of both LTC and LBP is very similar to the goal of strictness optimizations: to eliminate the overhead of packaging up an expression by evaluating it immediately instead. But, *both* optimizations (strictness and dynamic partitioning) are necessary if a fine-grained functional program is to run efficiently in parallel. Dynamic partitioning will not increase efficiency much if delays are present in fine-grained tasks—most of the cost of packaging up an expression for remote execution has already been paid if the expression is delayed, and little is left to be saved by dynamic partitioning. At best the cost of executing a delayed expression locally is about half the cost of executing it remotely; in a fine-grained program this overhead would still be unacceptable. However, if the fine-grained tasks of a functional program are free of delays both LTC and LBP could be very effective at decreasing task creation overhead.

To summarize, dynamic partitioning is quite compatible with delay-based parallel implementations of lazy functional languages, but its effectiveness is greatly hindered if delays are present in fine-grained parallel tasks. Since most real programs involve traversals of data structures where strictness analysis has limited effectiveness, my opinion is that efficient parallel (and sequential!) implementations of lazy functional languages will require language support, perhaps in the form of strictness annotations and/or strict versions of data constructors. The para-functional approach [Hudak 86, Hudak 91], based on programmer annotations to control evaluation order, may supply the needed control. However, control of evaluation order in a lazy language is a slippery subject and delays may be created far from the place where their parallel evaluation is directed.

8.3 Dynamic Partitioning on Larger Parallel Machines

Machines like the Encore Multimax (or its newer workstation cousins made by Silicon Graphics and Digital) will probably not disappear; architectures where several processors share a common bus and i/o facilities will continue to occupy an attractive price/performance niche. The dynamic partitioning techniques described here have been shown to be effective for such machines and thus will always have some utility. Still, scalability is a prime consideration for any parallel strategy.

Because of the access bottleneck inherent with physically shared memory, larger machines tend to have distributed memory. Such memory can still be

logically shared however, as with the BBN Butterfly, IBM RP3, and the experimental ALEWIFE machine [Agarwal *et al* 91]. ALEWIFE's directory-based cache-coherency scheme [Chaiken *et al* 91] promises to provide a shared address space more cheaply than existing machines. But as Feeley has shown, lazy task creation can be effectively implemented even on the Butterfly [Feeley 91].

Issues of data locality become more important on distributed-memory machines because of the increased cost of remote memory references. Lazy task creation can be helpful in this regard, improving data locality by executing an entire subtree of tasks on a single processor. For example, in `tridiag` when a processor handles a subtree of tasks it localizes access to a contiguous section of the original array. The dynamic nature of the partition could also hurt locality though. During the backsubstitution phase of `tridiag` a second partition is created; chances are good that a given data element will be handled by different processors in the two phases.

To combat such problems the ALEWIFE researchers are experimenting with `future-on`, which allows a task to be assigned to a specific processor. Using this construct is likely to improve efficiency but also represents a step away from declarative programming and toward explicitly-specified static partitioning. Preserving data locality across a distributed memory using a language without a concept of locality remains a definite challenge.

Still, the locality offered by subtree grouping may suffice for many programs. And, the advantages of lazy task creation over load-based partitioning will likely become more pronounced on larger machines since adding additional processors causes the number of tasks created (and therefore the overhead due to task creation) to increase less with LTC than with LBP.

8.4 Demonstrating that Parallel Lisp is Useful

Having considered three rather specific areas for future work let us consider a rather general one. Much of the driving force behind the development of parallel hardware and software has come from the scientific computing community, where numerous applications exist which take too long to execute on even the fastest sequential processors. Parallel versions of many such applications have been built and executed, showing real speedup over sequential versions on uniprocessors. In some cases parallel machines are now being used for everyday execution of such problems.

In contrast, demanding symbolic applications have been much less visible. Usable, efficient parallel Lisp implementations have existed for at least a few years, yet I know of no examples where application developers have used them successfully for real problems. People tend to sniff out systems that fill a need they have; my conclusion is that parallel Lisp systems don't yet fill any existing

needs.

One explanation is that the small-scale multiprocessors used thus far to implement parallel Lisp systems don't offer enough advantages over personal workstations. Because designing a multiprocessor takes longer than designing a uniprocessor, available uniprocessors tend to have more recent (and thus more powerful) processors than available multiprocessors. As a personal example of this, I was approached by a researcher who wanted to use Encore Mul-T to speed up a program which had to interrogate a large database. When we compared the speed of the Multimax with the speed of his Sun Sparcstation we decided that the small potential speedup didn't justify the work of porting his program. Mul-T's features for easing parallel programming didn't even enter the debate because a viable sequential alternative was handy.

The Encore's multi-user nature also influenced that decision. The researcher would have to compete with other users for the Encore processors, while he was guaranteed exclusive access to his workstation. This factor could be eliminated by the advent of personal multiprocessor workstations, but it remains to be seen whether such machines will see widespread use.

One potential platform which avoids the above problems is a network of workstations. With a distributed operating system providing friendly and flexible access to the available processing resources and a shared virtual memory system [Li 86, Li & Hudak 89] providing a shared address space, a parallel Lisp system could make the processing resources easily exploitable.

But unless parallel Lisp is to be a solution looking for a problem, a real need must be demonstrated. A crucial challenge for future parallel Lisp researchers is to demonstrate useful speedup on an important problem. Specifically, can we find a compute-intensive symbolic application (or 5? or 10?) which is important enough that speeding it up would make a qualitative improvement in the ability to solve some problem? Many such scientific applications exist. If so, can we write a program in parallel Lisp which executes 20-50 times faster on a multiprocessor than an optimized sequential version on a top-of-the-line workstation?

Assuming that such an application can be found (and I consider this to be a very important prerequisite), achieving large speedups will involve several challenges. First, obviously, the underlying system must execute on a multiprocessor larger than the Encore Multimax. Some of the challenges involved in that endeavor were outlined in the previous section.

Another very important question is whether "real-world" symbolic applications are amenable to large-scale parallelism. The programs described in Section 5.3 are a step more realistic than the usual `fib/tak/boyer` group, but probably still do not reflect all the demands of real applications. For example, symbolic applications often use large DAGs to represent data, *e.g.* representing program code in compilers or representing relationships between objects in artificial intelligence programs. Typically sequential programs will repeatedly traverse

the graphs, destructively modifying the nodes during each traversal. Performing such traversals in parallel requires the careful addition of explicit inter-task synchronization, making program development and debugging much more difficult. Also, the dynamic methods discussed here work much less well when programs contain explicit synchronization. A possible alternative is to write the traversal functionally so that new nodes are created rather than old ones modified, but the copying required by such a method could add significant overhead. Also, a parallel functional traversal of a graph with shared nodes is complex at best.

On small-scale machines such complications may sometimes be avoided by coarse-grained partitions, as for example by compiling several files in parallel or compiling all procedures within a file in parallel, but on larger machines the partition must become finer-grained and the complications will be less easily avoidable.

A specific example of these problems is the **speech** benchmark used to measure Mul-T performance in [Mohr *et al* 90, Mohr *et al* 91]. The initial version of this program had a lot of parallelism but ended up doing much more work than necessary. Subsequent re-implementations were able to reduce the work required but also resulted in a more complex control structure requiring synchronization which was not at all amenable to dynamic partitioning.

So it appears that to achieve good performance of real applications on large-scale multiprocessors the programmer will tend to need code for explicit partitioning and explicit synchronization. If the parallel Lisp version of our hypothetical application can in fact be made to execute 20-50 times faster than its sequential competition, we must then evaluate what price has been paid in coding it.

I believe that the challenges outlined in this section should be the primary focus of future research in parallel symbolic computing.

Chapter 9

Conclusions

These final paragraphs highlight the important results gleaned from studying dynamic partitioning in parallel Lisp.

The benchmark results show that when fine-grained programs are partitioned in a straightforward way—using `future` to identify parallelism without grouping operations together explicitly in the source code—performance is not acceptable. This is true even though the overhead of (eager) task creation in the underlying system (Encore Mul-T) has been carefully minimized.

But, with dynamic partitioning the same programs exhibit virtually linear speedup with very acceptable runtime overhead. A satisfactory partition can be created dynamically at runtime using either load-based partitioning or lazy task creation, without complicating the source code by explicit repartitioning.

The performance of the two methods is fairly close for many programs. LBP has less overhead per potential fork point and thus is faster when few processors are used; however, LTC creates fewer tasks and thus exhibits more nearly linear speedup.

But a close examination of load-based partitioning reveals unexpected defects, most notably that it degrades the performance of some programs and introduces deadlock in others. One could perhaps live with these defects by being careful to use LBP only where it is safe, accepting the extra work of specifying on a case-by-case basis whether LBP should be applied, and the responsibility of making a correct decision. But fortunately taking on this burden is not necessary because LTC repairs all of the defects of LBP, meanwhile performing as well or better than LBP and scaling better as processors are added.

The advantage of LTC over LBP arises because LTC delays all partitioning decisions and thus can recover flexibly in situations where LBP has made an unfortunate and irrevocable partitioning decision. Also very important is the fact that LTC performs only a minimal set of operations when delaying a partitioning decision, allowing runtime overhead to be low enough that LTC performs competitively with LBP. The set is minimal because of the key design decision to

package up the parent and continue the child at potential fork points rather than the other way around. Other systems which delay partitioning decisions haven't made this key change and have paid the price by performing a much larger set of operations at potential fork points, as described in Section 7.2.

It would be quite difficult to pare down the set of operations any further than LTC has while retaining the important property that deadlock will never be introduced. However, there is room for improvement in how efficiently the operations are performed; machines with better hardware support for synchronization and cache management would allow LTC to be implemented with less overhead than on the Encore Multimax.

Still, the benchmark results show that the overhead of LTC in Encore Mul-T is low enough to give excellent performance. If LTC fails to improve the performance of a program the reason is usually something other than overhead introduced at potential fork points. Another result of this study is increased knowledge about the kinds of programs where dynamic partitioning is and is not effective. It is not effective when task trees are spindly because it can only combine tasks rather than restructure trees; also it is less effective when task trees are shallow because fewer tasks can be inlined and processor idleness is higher.

But we must keep in mind that the methods used to increase the efficiency of such programs in other parallel languages may also be used in Mul-T. Dynamic partitioning successfully shrinks the set of parallel programs which require fine-tuning for good performance, but the remaining programs in the set are still amenable to other methods. Dynamic partitioning works very well when task trees are bushy and moderately deep; programs like `abisort` and `tridiag` may be coded much more simply for execution with dynamic partitioning. Static partitioning would make these programs more complex and the partition achieved might not be as satisfactory.

In addition to the points made above, a few other important results of this work deserve highlighting. Not least is the existence of a robust and efficient implementation of lazy task creation in Encore Mul-T. Also useful are the ETC scheduling improvements resulting from insights gained in studying LTC; future implementors of task-based systems may refer to some hard evidence advocating polite stealing, favoring the child task, and using double-ended task queues with a FIFO stealing policy. Finally, I have added a few programs to the rather small set of benchmarks usually seen in parallel Lisp studies.

The dynamic partitioning strategies developed here enlarge the set of parallel Lisp programs which can be handled efficiently and safely. LTC is preferred over LBP because it has better overall performance, eliminates the potential for deadlock, and needn't be applied selectively.

I look forward to the day when complex symbolic applications run with satisfying speedups on large parallel processors.

Bibliography

[Agarwal *et al* 91] A. Agarwal, et. al., "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," in M. DuBois and S. Thakkar, eds., *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991.

[Agarwal *et al* 90] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *17th Annual Int'l. Symp. on Computer Architecture*, Seattle, Wa., May 1990, pp. 104–114.

[Aho *et al* 83] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, pp. 208-212.

[Arvind & Culler 86] Arvind and D. Culler, "Dataflow Architectures," *Annual Reviews in Computer Science*, Annual Reviews, Inc., Palo Alto, Ca., 1986, pp. 225–253.

[Bilardi & Nicolau 89] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines," *SIAM J. Comput.*, 18:2, April 1989, pp. 216–228.

[Bloss *et al* 88] A. Bloss, P. Hudak, and J. Young, "An Optimising Compiler for a Modern Functional Language," *The Computer Journal*, 31:6, 1988, pp. 152–161.

[Chaiken *et al* 91] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, Ca., April 1991.

[Chatterjee *et al* 91] S. Chatterjee, G. Blelloch, and A. Fisher "Size and Access Inference for Data-Parallel Programs,"

ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991, pp. 130–144.

[Culler 89] D. E. Culler, "Managing Parallelism and Resources in Scientific Dataflow Programs," Ph.D. thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, Cambridge, Mass., June 1989.

[Debray *et al* 90] S. Debray, N. Lin, and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *1990 ACM Conf. on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 174–188.

[Feeley 91] M. Feeley, "Efficient and General Implementation Strategies for Futures on Large Shared Memory MIMD Computers" Ph.D. thesis, Brandeis University (in preparation).

[Gabriel 84] R. P. Gabriel and J. McCarthy, "Queue-based Multi-processing Lisp," *1984 ACM Symp. on Lisp and Functional Programming*, Austin, Tex., Aug. 1984, pp. 25–44.

[Gabber 90] E. Gabber, "VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems* 1:3, July 1990, pp. 304–317.

[Goldberg 88] B. Goldberg, "Multiprocessor Execution of Functional Programs," *Int'l. J. of Parallel Programming* 17:5, Oct. 1988, pp. 425–473.

[Goldman 88] R. Goldman and R. P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 143–152.

[Goldman *et al* 89] R. Goldman, R. Gabriel, and C. Sexton, "Qlisp: An Interim Report," in T. Ito and R. Halstead, eds., *Proceedings of U.S./Japan Workshop on Parallel Lisp* (Springer-Verlag Lecture Notes in Computer Science 441), Sendai, Japan, June 1989, pp. 161–181.

[Greenberg & Woods 90] M. Greenberg, and V. Woods, "FLAGSHIP—A Parallel Reduction Machine for Declarative Programming," *Computing and Control Engineering Journal*, 1:2, March 1990.

[Gurd *et al* 85] J. Gurd, C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Comm. ACM* 28:1, January 1985, pp. 34–52.

[Halstead 85] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. on Prog. Languages and Systems* 7:4, October 1985, pp. 501–538.

[Halstead 86] R. Halstead, "An Assessment of Multilisp: Lessons from Experience," *Int'l. J. of Parallel Programming* 15:6, Dec. 1986, pp. 459–501.

[Halstead 89] R. Halstead, New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools," in T. Ito and R. Halstead, eds., *Proceedings of U.S./Japan Workshop on Parallel Lisp* (Springer-Verlag Lecture Notes in Computer Science 441), Sendai, Japan, June 1989, pp. 2–57.

[Halstead 91] R. Halstead, Religious Preference, February, 1991. But obviously correct.

[Hockney & Jesshope 88] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, Adam Hilger, Bristol and Philadelphia, 1988, pp. 475–83.

[Hudak 86] P. Hudak, "Para-Functional Programming," *Computer*, 19(8):60–71, August 1986.

[Hudak 91] P. Hudak, "Para-Functional Programming in Haskell," in B. Szymanski, ed., *Parallel Functional Languages and Compilers*, ACM Press, 1991.

[Hudak & Goldberg 85] P. Hudak and B. Goldberg, "Serial Combinators: 'Optimal' Grains of Parallelism," *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 201, September 1985, pp. 382–388.

[Jagannathan 91] S. Jagannathan, Personal Communication, August 1991.

[Jagannathan & Philbin 91] S. Jagannathan and J. Philbin, "A Foundation for an Efficient Multi-Threaded Scheme System," NEC Research Institute Technical Report 91-009-3-0050-2, March 1991.

[Kranz 88] D. Kranz, "ORBIT: An Optimizing Compiler for Scheme," Ph.D. Thesis, Yale University Technical Report YALEU/DCS/RR-632, February 1988.

[Kranz *et al* 89] D. Kranz, R. Halstead, and E. Mohr, "Mul-T, A High-Performance Parallel Lisp", *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 81–90.

[Kranz *et al* 86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "Orbit: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, June 1986, pp. 219–233.

[Larus 89] J. Larus, "Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors," Ph.D. Thesis, Univ. of California at Berkeley, Report UCB/CSD 89/502, May 1989.

[Li 86] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Yale University Technical Report YALEU/DCS/RR-492, September 1986.

[Li & Hudak 89] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems* 7:4, November 1989, pp. 321–359.

[Miura 88] K. Miura, "Tradeoffs in granularity and parallelization for a Monte Carlo shower simulation code," *Parallel Computing* 8:1–3, 1988, pp. 91–100.

[Mohr *et al* 90] E. Mohr, D. Kranz, and R. Halstead, "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs," *Proceedings of ACM*

Symposium on Lisp and Functional Programming, June 1990, pp. 185–197.

[Mohr *et al* 91] E. Mohr, D. Kranz, and R. Halstead, "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs," *IEEE Trans. on Parallel and Distributed Systems* 2:3, July 1991, pp. 264–280.

[Pehoushek & Weening 89] J. Pehoushek and J. Weening, "Low-cost process creation and dynamic partitioning in Qlisp," in T. Ito and R. Halstead, eds., *Proceedings of U.S./Japan Workshop on Parallel Lisp* (Springer-Verlag Lecture Notes in Computer Science 441), Sendai, Japan, June 1989, pp. 182–199.

[Peyton Jones 87] S. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[Peyton Jones *et al* 89] S. Peyton Jones, C. Clack, and J. Salkild, "High-performance parallel graph reduction," Springer-Verlag LNCS 365, *PARLE: Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, June 1989, pp. 193-206.

[Polychronopoulos & Kuck 87] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers*, C-36:12, December 1987, pp. 1425–1439.

[Rees *et al* 86] J. Rees, W. Clinger, *et. al.*, "Revised^3 Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices*, 21:12, December 1986.

[Ruggiero & Sargeant 87] C. A. Ruggiero and J. Sargeant, "Control of Parallelism in the Manchester Dataflow Machine," Springer-Verlag LNCS 274, *Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987, pp. 1–15.

[Sabot 88] G. Sabot, *The Paralation Model*, M.I.T. Press, 1988.

[Sarkar 87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," London: Pitman,

and Cambridge, Mass.: MIT Press, 1989. This is a revised version of the author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.

[Sarkar & Hennessy 86] V. Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," *SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pp. 17-26.

[Steele & Hillis 86] G. L. Steele, Jr. and W. D. Hillis, "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing," *1986 ACM Symp. on Lisp and Functional Programming*, Cambridge, MA, August 1986, pp. 279-297.

[Vandevoorde & Roberts 88] M. Vandevoorde and E. Roberts, "WorkCrews: An Abstraction for Controlling Parallelism," *Int'l. J. of Parallel Programming* 17:4, August 1988, pp. 347–366.

[Waters 90] R. C. Waters, "Series", in G. Steele, Jr., *Common Lisp: the Language*, Second Edition, Digital Press, Maynard MA, 1990, pp. 923-955.

[Weening 89] J. Weening, "Parallel Execution of Lisp Programs," Ph.D. Thesis, Stanford Computer Science Report STAN-CS-89-1265, June 1989.

[Wilson 91] D. Wilson, Encore Computer Corporation, Personal Communication, March 1991.

Trademarks

Multimax and UMAX are trademarks of Encore Computer Corporation.

Series 32000 is a trademark of National Semiconductor Corporation.

Sparcstation is a trademark of Sun Microsystems, Inc.

UNIX is a trademark of AT&T Bell Laboratories.