THE UNIVERSITY OF CHICAGO

EFFECTIVE SCHEDULING TECHNIQUES FOR HIGH-LEVEL PARALLEL PROGRAMMING LANGUAGES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

MICHAEL ALAN RAINEY

CHICAGO, ILLINOIS

AUGUST 2010

To Beth, my friends and colleagues, my family, and most of all, my parents, without whom this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my parents, Christine and John, the rest of my family, and Beth for their constant love and support. I want to give special recognition to my parents, who are the most dedicated and loving parents a son could hope for.

My advisor, John Reppy, was instrumental in my success as a graduate student, going well beyond the call of duty to give me useful criticism, technical insights, and steadfast support. Thank you John.

Thanks to my other committee members, Matthew Fluet and Anne Rogers, for their insightful feedback and help in proofreading this dissertation.

My thanks go out to the members of the Manticore project who helped me immensely during my time as graduate student. Adam Shaw and Lars Bergstrom were always willing to listen to my ideas and ready to help me sharpen them. Matthew Fluet provided me with many insights and thoughtful criticism of my work.

To my classmates over the years, Lars Bergstrom, Matthew Hammer, Casey Klein, Jacob Matthews, Matthew Rocklin, Adam Shaw, Borja Sotomayor, and Andy Terrel: you have enriched my time in graduate school beyond measure.

I would like to thank David S. Wise for mentoring me while I was an undergraduate. His guidance gave me the confidence to continue to graduate school.

The research presented in this dissertation was supported by the NSF, under the NSF Grants CCF-0811389 and CCF-0937561.

ABSTRACT

In the not-so-distant past, parallel programming was mostly the concern of programmers specializing in high-performance computing. Nowadays, on the other hand, many of today's desktop and laptop computers come equipped with a species of shared-memory multiprocessor called a multicore processor, making parallel programming a concern for a much broader range of programmers. High-level parallel languages, such as Parallel ML (PML) and Haskell, seek to reduce the complexity of programming multicore processors by giving programmers abstract execution models, such as implicit threading, where programmers annotate their programs to suggest the parallel decomposition. Implicitly-threaded programs, however, do not specify the actual decomposition of computations or mapping from computations to processors. The annotations act simply as hints that can be ignored and safely replaced with sequential counterparts. The parallel decomposition itself is the responsibility of the language implementation and, more specifically, of the scheduling system.

Threads can take arbitrarily different amounts of time to execute, and these times are difficult to predict. Implicit threading encourages the programmer to divide the program into threads that are as small as possible because doing so increases the flexibility the scheduler in its duty to distribute work evenly across processors. The downside of such fine-grain parallelism is that if the total scheduling cost is too large, then parallelism is not worthwhile. This problem is the focus of this dissertation.

The starting point of this dissertation is work stealing, a scheduling policy well known for its scalable parallel performance, and the work-first principle, which serves as a guide for building efficient implementations of work stealing. In this dissertation, I present two techniques, Lazy Promotion and Lazy Tree Splitting, for implementing work stealing. Both techniques derive their efficiency from adhering to the work-first principle. Lazy Promotion is a strategy that improves the performance, in terms of execution time, of a work-stealing scheduler by reducing the amount of load the scheduler places on the garbage collector. Lazy Tree Splitting is a technique for auto-

matically scheduling the execution of parallel operations over trees to yield scalable performance and eliminate the need for per-application tuning. I use Manticore, PML's compiler and runtime system, and a sixteen-core NUMA machine as a testbed for these techniques.

In addition, I present two empirical studies. In the first study, I evaluate Lazy Promotion over six PML benchmarks. The results demonstrate that Lazy Promotion either outperforms or performs the same as an alternative scheme based on Eager Promotion. This study also evaluates the design of the Manticore runtime system, in particular, the split-heap memory manager, by comparing the system to an alternative system based on a unified-heap memory manager, and showing that the unified version has limited scalability due to poor locality. In the second study, I evaluate Lazy Tree Splitting over seven PML benchmarks by comparing Lazy Tree Splitting to its alternative, Eager Tree Splitting. The results show that, although the two techniques offer similar scalability, only Lazy Tree Splitting is suitable for building an effective language implementation.

CHAPTER 1

INTRODUCTION

For many years, chip manufacturers were able to regularly produce faster and faster chips by taking advantage of increases in chip clock frequency. This approach reached its limit because increasing clock frequency required extremely high levels of power and, consequently, unacceptable amounts of heat. Chip manufacturers have since turned to multicore processors in an effort to continue offering performance increases across successive generations of chips. A multicore processor is a kind of shared-memory multiprocessor in which there are multiple processors (*i.e.*, cores) contained on a single chip. Multicore processors are now commodity machines, found in laptops, desktops, and embedded computers.

One consequence of this shift is that programmers must use parallelism to take advantage of future improvements in chip technology. In this dissertation, I am interested in high-level parallel programming languages, which are languages that abstract away from the details of mapping parallel computations onto parallel hardware. High-level parallel languages have the potential to reduce the complexity of programming multiprocessors, much as garbage-collected languages reduced the complexity of application programming. They enable the programmer to focus on the algorithms and data structures for solving a problem without having to worry about aspects like task communication, which can be challenging to get right because of the potential for deadlocks and race conditions.

My work focuses on supporting *implicitly-threaded* parallelism, where the programmer provides hints about which computations might benefit from parallelism, but the details are left to the implementation. In particular, my work is in the context of Parallel ML (PML) [31, 33, 34, 35], which is a parallel functional language that supports implicit threading and nested data parallelism, a well-known style of parallel programming that originated with the NESL programming language [8]. A large part of my work is about supporting nested data parallelism effectively.

High-level parallel languages rely on language implementations, as opposed to the program-

mer, to manage parallelism effectively. Despite major advances in language-implementation technology, there is still a major obstacle: a lack of techniques for building language implementations that have predictably-good parallel performance across many programs. This dissertation represents my effort to address this issue.

The rest of this section is as follows. I give background on nested data parallelism, the problem of scheduling tasks created by nested data-parallel programs, and the well-known work-stealing policy for scheduling tasks. Then I describe the main contributions of my work, Lazy Promotion and Lazy Tree Splitting, which are techniques for increasing the effectiveness of the implementations of high-level parallel languages.

1.1 Nested Data Parallelism

Nested Data Parallelism (NDP) is a declarative style for programming irregular parallel applications. NDP languages provide language features favoring the NDP style, efficient compilation of NDP programs, and various common NDP operations like parallel maps, filters, and sum-like reductions. NDP languages include NESL [8], Data-parallel Haskell [19], Intel's Ct [38], and PML [33, 34]. NDP originated as a generalization of the data-parallel programming style found in languages such as High Performance Fortran and C*. Unlike these data-parallel forbears, NDP languages can naturally express nesting of data-parallel operations. Such nesting enables NDP to cover a wider range of algorithms, including irregular ones [7]. In NDP, irregular parallelism is achieved by the property that nested arrays do not need to have regular, or rectangular, structure; *i.e.*, subarrays may have different lengths.

Programs written in NDP express parallel computation primarily through two idioms: recursion and higher-order functions on collections (*e.g.*, maps). NDP and the style of functional programming are alike in this regard. So, it is not surprising that NDP fits naturally in functional programming languages. Purely functional languages are an especially attractive framework for NDP because these languages admit aggressive optimizations, such as fusion of NDP operations [17],

that are difficult or impossible to apply in imperative languages.

PML [31, 33, 34, 35] is a purely-functional, parallel subset of Standard ML [59], that offers a variety of features for expressing parallelism, including NDP. PML is supported by the system called Manticore [31, 32, 35], which consists of a PML compiler, a scheduling system, and a memory manager. I use Manticore as the test bed of my work.

Much past work has focused on developing efficient NDP algorithms. There is now a large body of NDP algorithms made available by the research community, and these algorithms are known to express large amounts of parallelism [78]. In this dissertation, I focus on the implementation of NDP in a functional language and discuss how various designs affect performance. Primarily, I confine my interest to execution time as a measure of performance.

1.2 Task scheduling

NDP has two characteristics that are simultaneously useful for NDP programmers and challenging for language implementers. First, NDP applications are divided into small pieces of parallel work. This characteristic gives a scheduler the flexibility it needs to distribute work evenly across processors. Second, NDP applications create large amounts of parallel work. This characteristic offers portability in the sense that programmers can write NDP programs irrespective to the number of processors, yet still be confident that applications will have sufficient work to take advantage of many or most processors.

A ***task*** is a small, independent thread of control. In order to enable parallelism, the system spawns off parallel tasks, each of which consist of one or more pieces of NDP work. These two aforementioned NDP characteristics pose the following challenge for implementing efficient scheduling: since each task inevitably involves some scheduling cost, total scheduling cost can be large. If the total scheduling costs are too large, parallelism is not worthwhile, and as such, this problem is fundamental for NDP implementations. This dissertation work focuses on this problem.

Past work on task scheduling typically used an abstract computation model based on directed,

acyclic graphs (DAGs) and an associated cost model for predicting the application run time [13, 14, 65]. Each node in the DAG corresponds to a point of parallel introduction (fork), parallel elimination (join), or a task (sequence of machine instructions that are executed serially). An edge between two nodes denotes a sequential dependency between those nodes. A task is ready to execute once all the nodes it depends on have executed. The cost model is as follows. Executing a DAG node has unit cost. The ***work*** (T_1), is the run time on a single processor, or equivalently, the total amount of time to execute all the tasks in the DAG. The ***span*** denotes the longest chain of dependencies in the DAG. The length of the span (T_∞) corresponds to the run time on an infinite number of processors,[1] which is the time to execute the tasks located on the span. The ***average parallelism*** ($T_A = T_1/T_\infty$) represents the maximum theoretical speedup for a given application. For a given application with T_A average parallelism, we can expect scalable parallel performance when $T_A \gg P$, where P is the number of processors. In this dissertation, I confine my interest to such scalable parallel applications.

This dissertation uses the notion of task scheduling given by Spoonhower [82]:

> A ***task scheduling policy*** determines the schedule, which is the order in which tasks are evaluated and the assignment of parallel tasks to processors.

An ***online task scheduling policy*** determines the schedule while the application executes, as opposed to a ***static*** scheduling policy, which determines the schedule before running the application. Online policies are preferable for use with applications in which task run times vary and are difficult or impossible to predict. Indeed, there are many applications for which this condition holds [78], and it is this kind of "irregular" parallelism that is of primary interest for this dissertation.

1. The span length is sometimes called the *depth*.

1.3 Work stealing

The starting point of this dissertation work is a particular task scheduling policy called ***work stealing***. Many parallel languages use work stealing, including Cilk++ [51], X10 [77], Fortress [58], and Intel's Threading Building Blocks [47]. The principle of work stealing is that idle workers which have no useful work to do should bear most of the scheduling costs and busy workers which have useful work to do should focus on finishing that work. In online work stealing, the system assigns a group of processors to collaborate on a given computation. Idle processors, called ***thieves***, obtain work by ***stealing*** tasks from busy processors, called ***victims*** [16, 40, 62].

The number of steals is a key performance metric because each steal operation involves communication among processors. Minimizing the total amount of communication is crucial because, if processors communicate too much, parallelism is not worthwhile. The effectiveness of work stealing follows, in large part, from the property that, for a given application, the total number of steals is small. Theoretical studies of work stealing give an upper bound on the expected number of steals and empirical studies confirm the theoretical bound, showing that the number of steals is indeed small for many parallel applications [3, 14]. Supposing a given application has sufficient parallelism (*i.e.*, $T_A \gg P$, where P is the number of processors), processors rarely need to steal and consequently spend most of their time busy doing useful work.

Some scheduling costs have a larger impact on run time than others. As a consequence of Amdahl's law, the best way to reduce the total scheduling cost is to find the subcosts that matter most and focus on reducing them. Importantly, in an implementation of work stealing, we can attribute each operation as contributing to the work overhead or span overhead.

The theoretical analysis on work stealing is crucial for finding the most significant subcosts. Because processors stay busy most of the time, we can associate the work overhead with the common case and the span overhead with the rare case. The ***work-first principle*** states that a design should focus on reducing costs associated with the work overhead, even at the expense of increasing costs associated with the span overhead [36].

1.4 Overview

This dissertation is about designing language implementations, in particular the parts of these implemenations that are relevant to work stealing, that support a variety of parallel-programming constructs in a functional language on a shared-memory multiprocessor. The goal of this dissertation is to develop techniques that increase the effectiveness of such language implementations. Specifically, I characterize the effectiveness of a language implementation as the extent to which its performance is robust, where robust performance means that performance is scalable across many applications and platforms without requiring any per-application tuning.

My thesis statement is as follows.

> The work-first principle is a useful guide for building effective implementations of high-level parallel languages.

The contributions of my dissertation work are in two parts:

- Lazy Promotion addresses scalable performance by demonstrating techniques that reduce the amount of overhead per task.
- Lazy Tree Splitting addresses scalable and robust performance by demonstrating techniques that automatically and adaptively coarsen the granularity of tasks.

1.5 Lazy Promotion

There are many studies on implementations of work stealing and implementations of parallel memory managers, but none yet which considers the two in combination. In Chapter 3, I present a study of the latter type. In this work, I present a new implementation of Manticore's work-stealing scheduler and a performance study demonstrating scalable parallel performance across several benchmarks. I use the following design methodology:

> Design the memory manager first, then adapt the scheduling policy to the memory manager (not vice versa).

My rationale for using this methodology consists of two parts. First, total memory-management costs are usually larger than costs imposed by the work stealing scheduler. Second, memory management is a less structured problem than scheduling in the following sense: memory management has no common case in general that serves as a guide for reducing communication costs; on the other hand, work stealing does have such a common case, as defined by the work-first principle. One way to reduce the communication costs is to introduce structure into the memory management scheme. For example, the approach used by Manticore relies on separation of state across processors. Given the design of an efficient memory manager, the design of an efficient work-stealing scheduler follows readily from the work-first principle.

Manticore's memory manager uses a ***split-heap architecture***, in which the heap is organized into processor-local areas and a shared global area. Processors collect their local heaps independently and asynchronously, without using any explicit inter-processor synchronization. The global heap is collected by a stop-the-world parallel collector.

Global-heap collections are expensive because they necessarily involve inter-processor synchronization and atomic operations that can saturate the memory bus. On the other hand, local-heap collections are cheap because they involve no inter-processor communication. We support independent local collections across processors by relying on a strict separation of state across processors. In particular, there can be no object in the global heap that points to an object in some local heap and there can be no object in some local heap that points to an object in another local heap.

As an consequence of these heap invariants, if we want to share a heap object between two or more processors, we must first ***promote*** that object to the global heap. Promoting some heap object x involves copying to the global heap the object graph that is rooted at x. Promotion is a major source of overhead in Manticore because promotion involves copying and promotion increases the

size of data in the global heap, which in turn increases the number of costly global collections.

The work stealing scheduler can use one of two promotion policies. In ***eager promotion***, for each task t, the scheduler promotes the object representing t just before spawning t. In ***lazy promotion***, for each task t, the scheduler delays promoting the object representing t until just before stealing t. The theoretical analysis of work stealing predicts that, for large parallel applications, steals are rare and spawns are common. Therefore, we expect a work-stealing scheduler based lazy promotion to promote fewer heap objects, on average, than a work-stealing scheduler based on eager promotion. Because promotion is expensive, the work-first principle predicts that the lazy promotion strategy will offer better performance than eager promotion.

The main results of this work come from a following performance study that compares three implementations of Manticore on a sixteen-core machine across several PML benchmark programs. Two of these implementations are based on lazy and eager promotion. The other implementation is a "flat-heap" version of Manticore, which uses a just the global heap. I compare the split- and flat-heap implementations because one might question the benefit of using split-heap design on a shared-memory machine.

The results are as follows. Eager and lazy promotion have similar performance in some cases but lazy promotion is faster for the benchmarks that exhibit high garbage-collection (GC) loads. The slowdowns of eager promotions are caused by extra time spent doing promotion and global GC. Flat heap is, in each case, much slower than split-heap versions and does not scale beyond eight processors. Furthermore, the flat-heap mutator is much slower than the split-heap mutator. The obvious culprit for the flat heap's poor performance is poor memory locality. I present a study of cache performance that supports this claim.

1.6 Lazy Tree Splitting

On its face, implementing NDP operations seems straightforward because individual array elements are natural units for creating tasks.[2] Correspondingly, a simple strategy is to spawn off one task for each array element. This strategy is unacceptable in practice, as there is a scheduling cost associated with each task (*e.g.*, the cost of placing the task on a scheduling queue) and individual tasks often perform only small amounts of work. As such, the scheduling cost of a given task might exceed the amount of computation it performs. If scheduling costs are too large, parallelism is not worthwhile.

One common way to avoid this pitfall is to group array elements into fixed-size chunks of elements and spawn a task for each chunk. ***Eager Binary Splitting*** (EBS), a variant of this strategy, is used by Intel's TBB [47, 75] and Cilk++ [51]. Choosing the right chunk size is inherently difficult, as one must find the middle ground between undesirable positions on either side. If the chunks are too small, performance is degraded by the high costs of the associated scheduling and communicating. By contrast, if the chunks are too big, some processors go unutilized because there are too few tasks to keep them all busy.

One approach to picking the right chunk size is to use static analysis to predict task execution times and pick chunk sizes accordingly [83]. But this approach is limited by the fact that tasks can run for arbitrarily different amounts of time, and these times are difficult to predict in specific cases and impossible to predict in general. Dynamic techniques for picking the chunk size have the advantage that they can base chunk sizes on runtime estimates of system load. *Lazy Binary Splitting* (LBS) is one such chunking strategy for handling parallel `do-all` loops [86]. Unlike the two aforementioned strategies, LBS determines chunks automatically and without programmer (or compiler) assistance and imposes only minor scheduling costs. LBS sticks to the work-first principle by deferring most scheduling costs to the rare cases, such as when some processors are

2. I do not address *flattening* (or *vectorizing*) [50, 52] transformations here, since the techniques of this paper apply equally well to flattened or non-flattened programs.

likely to be idle.

Chapter 4 presents an implementation of NDP that is based on my extension of LBS to binary trees, which I call ***Lazy Tree Splitting*** (LTS). LTS supports operations that produce and consume trees where tree nodes are represented as records allocated in the heap. We are interested in operations on trees because Manticore, the system that supports PML, uses *ropes* [15], a balanced binary-tree representation of sequences, as the underlying representation of parallel arrays. My implementation is purely functional in that it operates functionally over persistent data structures, although some imperative techniques are used under the hood for scheduling.

LTS exhibits performance robustness, which is a highly desirable characteristic for a parallel programming language, for obvious reasons. Prior to Manticore's adoption of LTS, Manticore used ***Eager Tree Splitting*** (ETS), a variation of EBS. My experiments demonstrate that ETS lacks performance robustness: the tuning parameters that control the decomposition of work are very sensitive to the given application and platform. Furthermore, I demonstrate that the performance of LTS compares favorably to that of (ideally-tuned) ETS across our benchmark suite.

1.7 Outline

This dissertation consists of six chapters.

Chapter 2 provides context for the rest of this dissertation, including background on the following topics: First, I describe PML, the programming language that I use as a basis for building parallel programs. Second, I present several PML benchmarks that I use for experiments appearing later in the dissertation. Third, I describe the hardware platform on which I carry out experiments. Fourth, I describe Manticore, the system that supports PML. Fifth, I describe Manticore's scheduling system, which is a collection of primitive mechanisms for building scheduling policies. Sixth, I provide background on work stealing.

Chapter 3 presents Manticore's work-stealing scheduler, which is based on the technique of Lazy Promotion. This chapter includes a description of the scheduling algorithm and its imple-

mentation. This chapter also includes an empirical study of the technique.

Chapter 4 presents Lazy Tree Splitting, a technique for scheduling the execution of parallel operations that produce and consume trees. From this technique, I build an effective implementation of nested data parallelism. This chapter includes a description of the algorithm, associated data structures, and an empirical evaluation comparing Lazy Tree Splitting to Eager Tree Splitting.

Chapter 5 presents, several low-level aspects Manticore's implementation of work-stealing, including the work-stealing scheduler loop, the task-cancellation mechanism, and the processor-interrupt mechanism.

The closing chapter summarizes these results and suggests future work.

CHAPTER 2

CONTEXT

2.1 PML

PML [31, 33, 34, 35] is a purely-functional, parallel subset of Standard ML [59], that offers a variety of features for expressing parallelism. This section gives an overview of parts of PML that are relevant to this dissertation, including the sequential part of PML, the fine-grain parallelism features, and CML.

2.1.1 Sequential programming

The sequential part of PML corresponds to a restricted subset of Standard ML (SML) [59] that provides a variety of parallel-programming mechanisms.

PML supports the functional elements of SML, including datatypes, polymorphism, type inference, and higher-order functions, and an imperative element: exceptions. PML has two main simplifications over SML:

- PML has no mutable data, *i.e.* no references or arrays.
- The PML module system is a subset of SML's module system in which there are no functors.

2.1.2 Fine-grain parallelism

In PML, fine-grain parallelism is expressed by parallel tuples and parallel arrays.

Parallel tuples

The expression

```
(| e1, ..., en |)
```

```
datatype tree
  = Lf of int
  | Nd of tree * tree

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
      (op * ) (| trProd tL, trProd tR |)
```

Figure 2.1: Parallel tree-product function

serves as a hint to the compiler and runtime that the subexpressions $e_1, \ldots, e_n$ are candidates for parallel evaluation. Up to n of these subexpressions can evaluate in parallel. There is an implicit barrier synchronization on the completion of all subexpressions. The result of the expression is an ordinary tuple value. Figure 2.1 contains an example use of a parallel tuple to compute in parallel the product of the leaves of a binary tree of integers.

Parallel thunk lists

PML provides another simple fork-join mechanism, which enables parallelism over arbitrary-length sequences of computations. The operation

```
val parN : (unit -> 'a) list -> 'a list
```

serves as a hint to the compiler that the given list of thunks are candidates for parallel evaluation. Please note that `parN` can be readily implemented in terms of parallel tuples. PML offers `parN` as a primitive because PML uses a more efficient implementation that is not visible to application programmers.

Parallel arrays

PML provides a *parallel array* type constructor (`parray`) and operations to map, filter, reduce, and scan these arrays in parallel. Like most languages that support NDP, PML includes comprehension syntax for maps and filters, but for this dissertation we omit the syntactic sugar and restrict

```
type 'a parray
val range : int * int -> int parray
val subP : 'a parray * int -> 'a
val lengthP : 'a parray -> int
val concatP : 'a parray list -> 'a parray
val mapP : ('a -> 'b) -> 'a parray -> 'b parray
val filterP : ('a -> bool) -> 'a parray
              -> 'a parray
val reduceP : ('a * 'a -> 'a) -> 'a -> 'a parray
              -> 'a
val scanP   : ('a * 'a -> 'a) -> 'a -> 'a parray
              -> 'a parray
```

Figure 2.2: Parallel arrays.

ourselves the interface shown in Figure 2.2. The function `range` generates an array of the integers between its two arguments, `subP` subscripts a value from a given position in the given parallel array, `lengthP` returns the number of elements in the given parallel array, and `concatP` returns the parallel array which is the concatenation of the given list of parallel arrays; `mapP`, `filterP`, and `reduceP` have their usual meaning, except that they can be evaluated in parallel, and `scanP` produces a prefix scan of the array. These parallel-array operations have been used to specify both SIMD parallelism that is mapped onto vector hardware (*e.g.*, Intel's SSE instructions) and SPMD parallelism where parallelism is mapped onto multiple cores; this paper focuses on exploiting the latter.

One simple example is main loop of a ray tracer, which generates an image of width `w` and height `h`.

```
fun raytrace (w, h) =
  mapP (fn y => mapP (fn x => trace (x, y))
                     (range (0,w - 1)))
       (range (0,h - 1))
```

This parallel map within a parallel map is an example of *nested data parallelism*. Note that the time to compute one pixel depends on the layout of the scene, because the ray cast from position `(x,y)` might pass through a subspace that is crowded with reflective objects or it might pass through relatively empty space. Thus, the amount of computation across the `trace(x,y)` ex-

pression (and, therefore, across the inner `mapP` expression) can differ significantly depending on the layout of the scene. A robust technique for balancing the parallel execution of this unbalanced computation is the primary contribution of this paper.

2.1.3 Concurrent ML

Concurrent ML (CML) is a language for programming concurrent applications [73]. CML provides concurrent threads. Threads synchronize and communicate via message passing over named channels.

By including CML with implicit threading, PML supports a form of two-level parallelism. A PML program consist of one or more CML threads executing concurrently, possibly on different processors. Each CML thread can, at any instant, evaluate an implicitly-threaded computation. In this model, each CML thread acts as a kind of SPMD processor for implicitly-threaded computations.

For example, one could imagine a 3-d game in PML where there is one CML thread that handles rendering, multiple CML threads that handle AI agents, and multiple CML threads that handle user interaction. There must be some limited, coarse-grain parallelism among these CML threads. The rendering and AI threads could also express parallelism by using PML's implicit-threading features.

2.2 PML benchmark programs

For benchmarking, I use six programs taken from the suite of PML benchmarks. Each program is written in a pure, functional style and was originally written by other researchers and ported to PML.

Quicksort Figure 2.3 shows the Quicksort PML program. In this program, we take a parallel array of integers called `xs` and return a brand new parallel array containing the elements

of `xs` in ascending order. The algorithm is similar to the one found in many introductory textbooks, and it works as follows. If `xs` contains one or fewer elements, we return `xs` because it is already sorted. Otherwise, we pick a pivot element `p` and partition `xs` into three parts based on `p`. The parts contain all elements of `xs` that are equal to, less than, and greater than `p` respectively. The first partition is already sorted because all of its elements are equal. The other two partitions we sort recursively. Finally, we concatenate the tree subarrays to form the sorted result.

We express parallelism in two ways: the parallel tuple lets the expression for building the first partition and two recursive calls be evaluated in parallel, and the parallel-array filters let the partitions be built in parallel. This algorithm has expected work $O(n \log n)$ and expected span length $O(\log n)$ where n is the length of the input array. It follows that the expected average parallelism is $O(n)$, which, for large n is ample parallelism to take advantage of many processors.

Our Quicksort benchmark sorts a sequence of 1,000,000 integers in parallel. This code is based on the NESL version of the algorithm [78].

Barnes Hut The Barnes Hut benchmark [4] is a classic N-body problem solver. Each iteration has two phases. In the first phase, a quadtree is constructed from a sequence of mass points. The second phase uses this quadtree to accelerate the computation of the gravitational force on the bodies in the system. Our benchmark runs 20 iterations over 200,000 particles generated in a random Plummer distribution. Our version is a translation of a Haskell program [37].

Raytracer The Raytracer benchmark renders a 256×256 image in parallel as two-dimensional sequence, which is then written to a file. The original program was written in ID [66] and is a simple ray tracer that does not use any acceleration data structures. The sequential version differs from the parallel code in that it outputs each pixel to the image file as it is computed, instead of building an intermediate data structure.

SMVM SMVM is a prime example of irregular parallelism which computes the multiplication of a sparse matrix with a dense vector. Figure 2.4 shows SMVM in PML. This program uses the *compressed row format*, a well-known representation for sparse matrices. In the matrix, each `row` is represented by a parallel array of index-value pairs where the indices specify the columns where the associated values are located.

A `matrix` is simply a parallel array of `rows`. The multiplication of the sparse matrix by the dense vector is implemented by `smvm`. For each `row`, we compute the dot product of that row with `v`. The dot product operation implemented by `dotp` multiplies elements of a given `row` with corresponding elements of the given `v`, returning the sum of all these multiplication results.

The algorithm has work $O(m)$ and span length $O(\log n)$ where m is the total number of nonzeroes and n is the maximal-length row of the sparse matrix, which means that the average parallelism is $O(m/\log n)$.

The matrix contains 1,091,362 elements and the vector 16,614. This code is based on the NESL version of the algorithm [78].

DMM The DMM benchmark is a dense-matrix by dense-matrix multiplication in which each matrix is 100×100.

Tree Rootfix The Tree Rootfix benchmark takes as input a tree structure in which each node is annotated with a value and returns, for each node, the sum of the values on the path from the root of the tree down to that node. This code is based on the NESL version of the algorithm [78] and we use it to measure the performance of the `scanP` operation.

I also use two synthetic benchmarks to test various aspects of the system.

Nested Sums Figure 2.5 shows the Nested Sums benchmark, which is a synthetic benchmark that exhibits irregular parallelism.

```
val quicksort : int parray -> int parray
fun quicksort xs =
 if lengthP xs <= 1 then xs
 else let
 (* select some pivot element p from xs *)
   val p = subP (xs, lengthP xs div 2)
   val (eq, lt, gt) = (| filterP (fn x => x = p) xs,
                         filterP (fn x => x < p) xs,
                         filterP (fn x => x > p) xs |)
   in
     concatP [lt, eq, gt]
   end
```

Figure 2.3: Quicksort in PML

```
type row = (int * float) parray
type matrix = row parray

val dotp : row * float parray -> float
fun dotp (row, v) = sumP (mapP (fn (i, x) => x * subP(v, i)) row)

val smvm : matrix * float parray -> float parray
fun smvm (m, v) = mapP (fn row => dotP (row, v)) m
```

Figure 2.4: SMVM in PML

Fib Figure 2.6 shows a PML program that computes the 29^{th} fibonaci number. The program uses a naïve algorithm that runs in exponential time.

2.3 Test machine

Experiments reported in this dissertation were performed on a test machine with the following specifications: The machine has four quad-core AMD Opteron 8380 processors running at 2.5GHz. Each core has a 512Kb L2 cache and shares 6Mb cache with the other cores of the processor. The

```
let fun upTo i = range (0, i)
in mapP sumP (mapP upTo (range (0, 5999)))
end
```

Figure 2.5: Nested Sums in PML

```
fun fib n =
  (case n
    of 0 => 0
     | 1 => 1
     | n => (op +) (| fib (n-1), fib (n-2) |))

val x = fib 29
```

Figure 2.6: Fib in PML

system has 32Gb of RAM and is running Debian Linux (kernel version 2.6.31.6-amd64). I ran each experiment 10 times and report the average performance results. For most of these experiments the standard deviation was below 1%, but in 10 (out of 144 experiments) the deviation was over 4%. The worst-case deviation was 6.66%.[1] For this reason, I omit error bars from plots.

2.4 Manticore

Manticore serves as a bridge from PML programs to shared-memory multiprocessors, such as our test machine. Manticore consists of a PML compiler, parallel memory manager, and a scheduling system for the various types of parallelism that can be expressed in PML. This section outlines the main aspects of the system, including the process model, the memory-management scheme, and the language called BOM on top of which all scheduling and synchronization code is written. In the sequel, I describe the Manticore scheduling system in detail.

2.4.1 Process model

Manticore could use a process model in which each CML thread or task is hosted by a different operating-system thread. This model has the advantage that scheduling is provided by the operating system, which would obviate the need for scheduling in Manticore, thus greatly simplifying the Manticore implementation. This technique does not scale to large numbers of threads or tasks

1. Specifically, the worst case was a deviation of 0.19 seconds on an average of 2.85 seconds for the lazy-promotion version of Quicksort on 16-processors.

because the cost of managing operating-system threads is too high in general. Furthermore, an operating-system often lacks scheduling policies that are sufficient for managing fine-grain computations.

Manticore has a user-level process model that sits on top of a more primitive process model, such as operating-system threads or hardware processors. Manticore's model includes two types of processes: A ***fiber*** is a lightweight thread of control. A ***virtual processor*** (vproc) is a processing element that executes fibers. There are a fixed number of vprocs and an unbounded number of fibers. At any instant during execution, a given vproc is either idle or executing a fiber.

Manticore assigns vprocs to hardware threads or operating-system threads, such as POSIX threads (pthreads), as is the case in this dissertation. This design, although portable across many systems, has the disadvantage that Manticore has little control over the scheduling of pthreads. Many operating systems allow the operating-system scheduler to deschedule pthreads at any time for an arbitrary duration, and even worse, to migrate pthreads to different hardware processors. Some operating systems, such as Linux, offer an affinity extension to bind pthreads to distinct processors. The current Manticore implementation relies on this feature. But this feature, however, does not prevent vprocs from being preempted by the operating system scheduler. Systems with threading libraries, such as Manticore, would benefit from a mechanism that provides some control over operating-system scheduling, such as a mechanism to dedicate a hardware processor to a pthread. Investigating the design of such a mechanism is beyond the scope of this dissertation.

2.4.2 Memory management

Functional languages tend to have high allocation rates and require efficient garbage collectors. The Manticore heap architecture is designed to maximize locality and to minimize synchronization between processors. This design is based on a combination of the approach of Doligez, Leroy, and Gonthier (DLG) [25, 26] and Appel's semi-generational collector [1].

The heap is organized into a local heap for each vproc and a global heap shared by all vprocs.

Following Appel [1], each local heap is divided into a nursery where new objects are allocated and an old-object region. The local heaps all have fixed size, whereas the global heap consists of an unbounded number of fixed-size heap chunks. Each vproc "owns" several global-heap chunks. The set of global-heap chunks are partitioned among vprocs.

Figure 2.7 illustrates the heap organization for a three-processor system. Like the DLG collector, Manticore's collector maintains two heap invariants (in Figure 2.7, the invariants are indicated by crossed-out lines): First, there is no object in the global heap that contains a reference to an object in some local heap. Second, there is no object in some local heap that contains a reference to an object in some other local heap. These invariants let us implement a local-heap collector in which processors collect their local heaps independently and without synchronization.

The Manticore runtime system uses four different kinds of garbage collection:

Minor GC is used to collect the nursery by copying live data into the old region of the local heap. After a minor GC, the remaining free space in the local heap is divided into half and the upper half is used as the new nursery.

Major GC is used to collect the old data in the local heap. The major collector is invoked at the end of a minor collection when the amount of local free space is below some threshold. The major collector copies the live old data into the global heap (except for the data that was just copied by the minor collector; it is left in the local heap).

Promotion is used to copy objects from the local heap to the global heap when they might become visible to other vprocs. For example, if a thread is going to send a message, then the message must be promoted first, since the receiver can be running on a remote vproc.

Global GC is used to collect the global heap. The global collector is invoked when a major collection detects that the amount of data allocated in the global heap has exceeded a threshold. The global collector is a stop-the-world parallel collector. We currently do not attempt to balance the load from global collector; each vproc traces the from-space data that is reachable

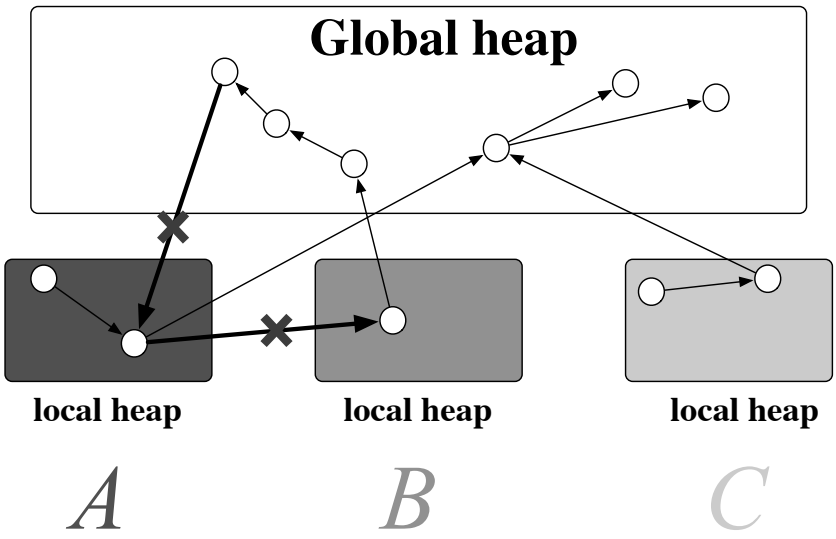


Figure 2.7: The Manticore heap

from its roots, stopping when it hits a forward pointer.

There are two important consequences of this heap design. On the positive side, most garbage-collection activity is asynchronous. The minor collections require no synchronization with other vprocs, and major collections and promotions only require synchronization when the vproc's global chunk becomes full and a new chunk must be acquired. The major drawback of this design is that any data that might be shared across vprocs must be promoted into the global heap.

2.4.3 Ropes

Manticore uses ropes as the underlying representation of parallel arrays. Ropes, originally proposed as an alternative to strings, are persistent balanced binary trees with `seq`s, contiguous arrays of data, at their leaves [15]. For the purposes of this dissertation, we view the rope type as having the following definition:

```
datatype 'a rope
  = Leaf of 'a seq
  | Cat of 'a rope * 'a rope
```

although in Manticore's actual implementation there is extra information in the `Cat` nodes to support balancing. Read from left to right, the data elements at the leaves of a rope constitute the data of a parallel array it represents.

Since ropes are physically dispersed in memory, they are well-suited to being built in parallel, with different processors simultaneously working on different parts of the whole. Furthermore, the rope data structure is persistent, which provides, in addition to the usual advantages of persistence, two special advantages related to memory management. First, the system can avoid the cost of store-list operations [1], which would be necessary for maintaining an ephemeral data structure. Second, a parallel memory manager, such as the one used by Manticore [32], can avoid making memory management a sequential bottleneck by letting processors allocate and reclaim ropes independently.

As a parallel-array representation, ropes have several weaknesses as opposed to contiguous arrays of, say, unboxed doubles. First, rope random access requires logarithmic time. Second, keeping ropes balanced requires extra computation. Third, mapping over multiple ropes is more complicated than mapping over multiple arrays, since the ropes may have different shape. As I show in performance studies in Chapters 3 and 4 these weaknesses are not crippling by themselves, yet I am aware of no study in which NDP implementations based on ropes are compared side by side with implementations based on alternative representations, such as contiguous arrays.

The maximum length of the linear sequence at each leaf of a rope is controlled by a compile-time constant M. At run-time, a leaf contains a number of elements n such that $0 \leq n \leq M$. In general, rope operations try to keep the size each leaf as close to M as possible, although some leaves will necessarily be smaller. We do *not* demand that a rope maximize the size of its leaves, as doing so would involve lots of extra computation with little gain.

Our rope-balancing policy is a relaxed, parallel version of the sequential policy used by Boehm, *et al.* [15]. The policy of Boehm, *et al.* is as follows. For a given rope r of depth d and length n, the balancing goal is $d \leq \lceil \log_2 n \rceil + 2$. This property is enforced by the function

```
val balance : 'a rope -> 'a rope
```

which takes a rope r and returns a balanced rope equivalent to r (returning r itself if it is already balanced).

In my rope-balancing policy, only those ropes that are built serially are balanced by `balance`, *i.e.*, the serial balancing process only ever takes place within a given chunk. There is no explicit guarantee on the balance of a rope containing subropes that are built by different processors. For such a rope, the amount of rope imbalance is proportional to the distribution of work across processors rather than the size of the rope itself. As I discuss in the performance study in Chapter 4, across all benchmarking results, balancing has minimal impact on performance.

As noted above, rope operations try to keep the size of each leaf as close to M as possible. In building ropes, rather than using the `Cat` constructor directly, we define a smart constructor:

```
val cat2 : 'a rope * 'a rope -> 'a rope
```

If `cat2` is applied to two small leaves, it may coalesce them into a single larger leaf. Note that `cat2` does not guarantee balance, although it will maintain balance if applied to two balanced ropes of equal size. We also define a similar function

```
val catN : 'a rope list -> 'a rope
```

which returns the smart concatenation of its argument ropes.

We sometimes need a fast, cheap operation for splitting a rope into multiple subropes. For this reason, we provide

```
val split2 : 'a rope -> 'a rope * 'a rope
```

which splits its rope parameter into two subropes such that the size of these ropes differs by at most one. We also define

```
val splitN : 'a rope * int -> 'a rope list
```

which splits its parameter into n subropes, where each subrope has the *same* size, except for one subrope that might be smaller than the others.

We sometimes use

```
val length : 'a rope -> int
```

which takes a rope r and returns the number of elements stored in the leaves of r.[2]

The operation

```
val mapSequential : ('a -> 'b) -> 'a rope -> 'b rope
```

is the obvious sequential implementation of the rope-map operation.

The various parallel-array operations described in Section 2.1.2 are implemented by analogous operations on ropes. Chapter 4 describes the implementation of these rope-processing operations in detail.

2.4.4 BOM

BOM is a normalized functional language extended with a variety of low-level features, such as mutable memory, atomic memory operations, and first-class continuations. BOM serves two different roles: First, BOM is an intermediate language: the compiler maps PML programs to BOM programs. Second, BOM is the programming language used by Manticore to express scheduling policies and synchronization protocols. This section describes several features provided by BOM that I use later to program scheduling policies.

For simplicity, in this dissertation, I present BOM programs in SML syntax. The actual BOM syntax bears resemblance to a normalized version of SML with a few additional constructs. The description of these constructs is below.

2. In Manticore's actual implementation, this operation takes constant time, as we cache lengths in `Cat` nodes.

Continuations

A ***continuation*** reifies an instance of a computational process (*e.g.*, a fiber) at a given step in the execution of the process. The continuation data structure encapsulates the state of a process at a given step of evaluation. This structure includes the values of machine registers and the process stack.

A ***first-class continuation*** is a language construct that lets a program capture the current state at any point and later resume execution at that point. First-class continuations are a well-known mechanism for expressing concurrency [41, 70, 68, 63, 80, 89]. In BOM, the state of a suspended fiber is represented by a continuation.

BOM continuations have the type `'a cont`, where the type variable `'a` ranges over the argument types of continuations. BOM provides the **`cont`** binding form for introducing continuations and the **`throw`** form for applying continuations. The **`cont`** binding:

$$\texttt{\textbf{let cont}}\ k\ arg\ \texttt{=}\ exp\ \texttt{\textbf{in}}\ body\ \texttt{\textbf{end}}$$

binds k to the first-class continuation

$$\lambda arg.(\texttt{\textbf{throw}}\ k'\ (exp))$$

where k' is the continuation of the whole expression. The scope of k includes both the expression $body$ and the expression exp (*i.e.*, k can be recursive). Continuations have indefinite extent and can be used multiple times.[3]

Example: `callcc` The traditional `callcc` function can be defined as

```
fun callcc f = let cont k x = x in f k end
```

3. The syntax of Manticore's continuation mechanism is taken from the Moby compiler's BOL IR [69], but Manticore's continuations are first-class, whereas BOL's continuations are a restricted form of one-shot continuations known as *escaping continuations*.

The **cont** binding reifies the return continuation of `callcc`. Manticore uses **cont** because it is more convenient than `callcc`, as programming with **cont** avoids the need to nest `callcc`'s in many places.

Memory management

BOM provides immutable and mutable tuples. An immutable tuple has the type

$$\texttt{[}bty_1\texttt{, ..., }bty_n\texttt{]}$$

where the types $bty_1 \ldots bty_n$ specify the types of elements of the tuple elements. The operation

$$\texttt{\#}i\texttt{(}x\texttt{)}$$

extracts out the i^{th} element of the tuple x.

A mutable tuple is denoted similarly, but is prefixed by `!`.

$$\texttt{![}bty_1\texttt{, ..., }bty_n\texttt{]}$$

In-place update is provided by the operation

$$\texttt{\#}i\texttt{(}x\texttt{) := }y$$

which stores y into the i^{th} element of x.

Data placement is made explicit by two operations:

- (**alloc** `(`x_1`, ..., `x_n`)`) allocates tuple `(`x_1`, ..., `x_n`)` in the vproc-local heap.
- (**promote** x) takes x, a reference to a heap object, and returns the promoted reference, *i.e.*, the reference that points to a copy of the heap object x that is located in the global heap. If x has been promoted before, then the operation just returns the promoted reference. Otherwise, the operation recursively promotes all objects reachable from x and then copies x to the global heap.

```
fun atomicTestAndSet (lock : ![bool]) = let
  val initial = #0(lock)
  in
    #0(lock) := true;
    initial
  end

fun atomicCompareAndSwap (
        x : ![word],
        old : word,
        new : word) : word = let
  val oldVal = #0(x)  (* select out word stored in x *)
  in
    if oldVal = old then #0(x) := new else ();
    oldVal
  end
```

Figure 2.8: Atomic operations

There is one additional property that helps reduce promotion costs. When promoting some object x, the memory manager needs only to return the new pointer to x. Observe that "stale" references to the leftover local version of x can still exist after the promotion. The absence of mutable objects in the local heap means that other objects in the local heap can safely continue to point-to and use the leftover local object.

Of course, these leftover local objects will eventually become a space leak if not collected. The trick used by Manticore is to let the memory manager recognize and reclaim these leftover objects subsequently during minor collections. These leftover local objects are apparent to the minor collector, as promoting some object x involves installing a forwarding pointer into the header word of x, indicating that leftover local x can be reclaimed. This technique is one important way in which Manticore leverages the functional purity of PML programs.

Atomic operations

BOM provides the two atomic operations shown in Figure 2.8.

The atomic test-and-set operation takes a memory location, writes to it, and returns the old

value. If multiple processors are accessing the same location and a processor is executing the operation, then no other processors are allowed to execute their test-and-set operations until the first one is finished.

The atomic compare-and-swap operation takes a memory location and two parameters, and if the value stored at the location is the same as the first parameter, then the operation writes the second parameter to the location. The return value is the value stored at the location just before the operation, which can be used to indicate whether a substitution was performed.

Example

As an example, consider the following BOM fragment in which two fibers, `k1` and `k2`, modify a shared state variable called `x`.

```
let
  val x = promote (alloc false)
  cont k1 () = (
    f1 ();
    if not (atomicTestAndSet x) then h () else exit ())
  cont k2 () = (
    f2 ();
    if not (atomicTestAndSet x) then h () else exit ())
in
  ... start k1 and k2 running on two different vprocs ...
end
```

Fibers `k1` and `k2` perform `f1 ()` and `f2 ()` respectively and then first of these fibers to complete applies `h ()`. The race condition is resolved by using the `atomicTestAndSet` operation. This operation takes a memory cell containing a boolean value and then atomically reads the value and writes `true` into the cell. The result is the value read from the cell.

2.5 The Manticore scheduling system

PML programs can express parallelism using different language mechanisms, *e.g.*, by using NDP and CML threads. These different mechanisms have fundamentally different objectives and therefore use different scheduling policies. For instance, a CML thread might need a scheduling policy

that offers fairness, one of many real-time scheduling policies that offers responsiveness, or a resource-aware policy [87] that tries to maximize throughput of IO devices. NDP requires a work-distribution policy such as work stealing to balance load effectively among system processors. The work stealing policy and many other policies used for NDP are inherently unfair, and would not be suitable in general for CML threads. Other special-purpose threading mechanisms have been proposed, including clocks and phasers [79], and PML's `pcase` [33], that require special scheduling policies. No single scheduling policy will be suitable in general.

In order to support various parallelism mechanisms effectively, Manticore provides a scheduling system that supports various different scheduling policies, in particular those policies that are useful for PML, including a CML-thread scheduler and a work-stealing scheduler. In work that originally appeared in ICFP 2008 [32], I presented the design and implementation of Manticore's scheduling system. This section gives an overview of parts of the design that are relevant to this dissertation. First, I present the scheduling primitives exposed by the scheduling system, including scheduler actions, and the interface to vprocs. Second, as an example, I present a CML thread scheduler written in BOM.

2.5.1 Scheduling primitives

This scheduling system consists of a few abstractions and primitive operations for programming scheduling policies. The goal of the design is to be simple, yet provide a basis for programming the various scheduling policies used by PML programs. The contribution of this design is a style, following from the primitives, for building scheduling policies out of small scheduling components. The organizing feature of the design is the *scheduler action*, a component based on first-class continuations for building schedulers.

The scheduling system exposes a small collection of primitive operations, on top of which more complex scheduling code can be written. This section presents the abstractions provided by the runtime system and gives an informal description of the scheduler operations with some simple

examples.

Manticore's scheduling system uses a cooperative model for scheduling in which a scheduler and the fiber that it schedules act like coroutines. The scheduler hands off its time to the fiber and the fiber relinquishes its time by passing a ***signal*** to the scheduler. The signal denotes one of several events that require the scheduler to act.

The scheduling system defines three signals:

- `STOP` denotes the termination of the current fiber.
- `(PREEMPT` k`)` denotes an asynchronous interrupt. The signal is paired with the value k, the continuation of the fiber interrupted by the signal.
- `(SLEEP` $nsec$`)` denotes a request made by the current fiber to put itself to sleep for $nsec$ nanoseconds.

A ***scheduler action*** is a function that consumes a signal and performs some scheduling activity based on the signal. Each vproc has its own stack of scheduler actions, and the top of this stack is called the ***current scheduler action***. When a signal arrives at a vproc, the vproc handles it by popping the current action from the stack, and passing the signal to the current action.

There are two basic scheduling operations: `run` starts executing a given fiber and `forward` passes a signal to the current scheduler action.

Figures 2.9 and 2.10 show the state transitions for these operations.

- The expression `run(`K_1`,` K_2`)` pushes the scheduler action K_1 onto the action stack and invokes the fiber K_2.
- The expression `forward` X pops the current action from the stack and applies the action to the signal X.

The `run` operation requires that signals be masked; the `forward` operation does not.

Let us derive some basic scheduling operations from the above primitives.

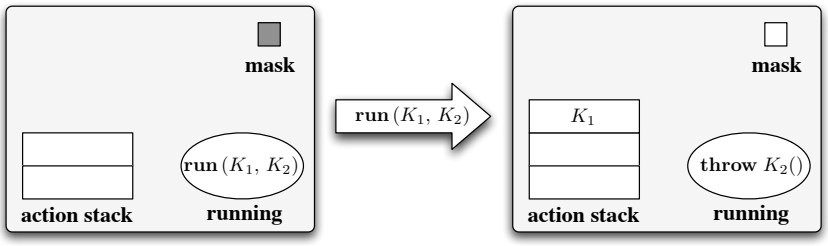


Figure 2.9: The `run` operation

Terminating The operation

```
fun stop () = forward STOP
```

informs the current scheduler that the executing fiber is terminating.

Creating a fiber The operation

```
fun fiber (f : unit -> unit) = let
    cont k () = ( f (); stop () )
    in k end
```

takes a given function and returns a fiber. When scheduled, the fiber applies the function and `stop`s. For example, when executed by a scheduler, the fiber

```
fiber (fn => print "hello")
```

will print `hello` and terminate.

Yielding The operation

```
fun yield () = let
    cont k () = ()
    in forward (PREEMPT k) end
```

has the calling fiber release to the current scheduler the host vproc. The scheduler receives a preempt signal, which contains the continuation of the calling fiber.

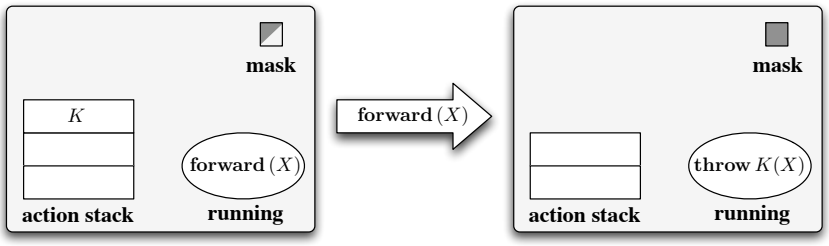


Figure 2.10: The `forward` operation

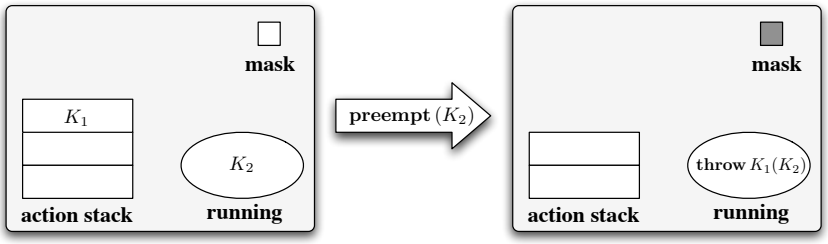


Figure 2.11: VProc preemption

Preemption

Figure 2.11 shows the vproc state transition for the preemption signal. A preemption signal can arrive at a vproc at any instant in which the vproc has signals unmasked and the scheduler-action stack is nonempty. The fiber K_2 is running at the instant that the preemption arrives. The vproc creates a preemption signal and passes the signal to K_1, the current scheduler action.

2.5.2 Fiber-local state

Often, a fiber needs to access local state. For example, the fiber might need to know

- the unique identifier of the current CML thread;
- scheduling data structures, such as queues, associated with a given thread

Fiber-local state (FLS) provides such local state.

Each vproc stores an FLS record, and the scheduler maintains this record. Figure 2.12 shows some of the FLS interface, including the `fls` type, an operation to create FLS, one to set the

```
type fls
val newFls : unit -> fls
val setFls : fls -> unit
val getFls : unit -> fls
```

Figure 2.12: Fiber-local state

current FLS, and one to access the current FLS.

2.5.3 VProcs

If a fiber is being executed by a vproc, then we say that vproc is the ***host vproc*** for that fiber. The operation below returns the reference to the host vproc.

```
val host : unit -> vproc
```

The operation below returns the list of all vprocs.

```
val vprocs : unit -> vproc list
```

Each vproc maintains a local FIFO queue of threads, and access to this queue is provided by the operations below.

```
val enq : fiber * fls -> unit
val deq : unit -> (fiber * fls) option
```

The operations implicitly select the queue of the host vproc.

Each vproc has a "landing pad," which is a list of incoming threads. The operation

```
val enqOnVP : (vproc * fiber * fls) -> unit
```

pushes the given thread on the landing pad of the given vproc. Each vprocs regularly moves threads from its landing pad onto its local scheduling queue.

```
cont roundRobin (STOP) = throw dispatch()
   | roundRobin (PREEMPT k) = let
     val fls = getFls ()
     cont k' () = (
          setFls fls;
          throw k () )
     in
        enq k';
        dispatch ()
     end

cont dispatch () = run (roundRobin, deq ())
```

Figure 2.13: The basic round-robin thread scheduler

Example: migrating The operation

```
fun migrateTo vp = let
    val fls = getFls ()
    cont k (x) = ( setFls fls; x )
    in
       enqOnVP (vp, k);
       stop ()
    end
```

causes the calling thread to migrate to the given vproc.

2.5.4 The CML *thread scheduler*

Let us build a basic CML thread scheduler using the scheduling primitives from above. The scheduler uses a round-robin policy in which each thread gets to execute for one quantum. The quantum begins when the thread executes and ends when either a preemption arrives or the thread terminates. Figure 2.13 shows the implementation of the scheduler. The continuation `roundRobin` is the scheduler action that schedules threads. Each vproc executes an independent instance of this scheduler action.

To complete this scheduler, let us implement the thread-spawn operation. This operation, shown below, creates a fiber to execute the body of the thread, and enqueues the fiber on the host vproc.

```
fun spawn f =
    enq (fiber (fn () => (setFls (newFls ()); f ())))
```

The remote-spawn operation spawns a given thread on a given vproc.

```
fun spawnOn (f, vp) = spawn (fn () => (migrateTo vp; spawn f))
```

2.6 Work stealing

The principle of work stealing states that idle processors should bear the responsibility of finding work to do and that busy processors should focus on completing their own local work before participating in scheduling. This strategy works well because, for applications with a sufficient parallelism, processors rarely run out of work to do. The idle processors bear the brunt of the scheduling costs, while the busy processors greedily complete their local work.

Let us consider the schedule produced by work stealing for a given parallel execution graph. For now, we assume that the whole parallel execution graph is provided to the scheduler in advance (*i.e.*, static scheduling), and later we consider online scheduling. In work stealing, each processor runs an independent instance of the scheduler and all processors proceed ahead one instant at a time. At a given instant, a processor is either busy doing local work or idle and in search of work. A busy processor owns a subgraph of the parallel execution graph from which the processor greedily executes a node. The execution of nodes proceeds in depth-first, left-to-right order over the subgraph. Whenever a busy processor executes a fork node, the right child of the fork node is made available to be stolen by an idle processor, or thief. We call an idle processor a *thief* because at each instant it tries to steal a subgraph from another processor, which we call the *victim*. Each steal attempt, the thief picks a victim. Supposing that the victim has a subgraph available to steal, the steal attempt succeeds and the execution graph owned by the victim is split at the oldest (unexecuted) fork node. The thief takes for itself the subgraph rooted at the right child of this fork node and proceeds to execute it, while the victim continues executing where it left off in its own subgraph. This process repeats until the entire parallel execution graph has been executed.

Let us turn our attention to some issues involved in implementing the online work stealing scheduler. Implementations of work stealing typically maintain stealable nodes (*i.e.*, the right children of fork nodes) in double-ended queues (deques). Each processor maintains its own deque. A processor executes from its own deque in much the way a sequential processor executes from its stack. When a processor executes a fork node, the processor pushes the right child of the fork node onto the bottom of its deque. Then the processor proceeds to execute the left-child node. Upon completion of the left subgraph, the processor pops a task off the bottom of its deque and executes it. If the deque is not empty, then the task is necessarily the most recently pushed task; otherwise all of the local tasks have been stolen by other processors and the processor must steal a task from the top of some victim processor's deque. The strategy for choosing the victim varies from implementation to implementation, but typically, on a shared-memory multiprocessor, the victim is chosen from all processors uniformly at random.

The work-first principle has been shown to be an effective guide for building an efficient implementation of work stealing [36]. The principle states that a design should focus on reducing costs carried by the work of the computation, even at the expense of increasing costs carried by the span.

The justification for the work-first principle can be derived from three assumptions.

Assumption 1 In practice, the work-stealing scheduler generates schedules that are similar to the ones predicted by the abstract analysis of work stealing [11, 14]. In particular, we assume that the scheduler executes a given computation on P processors in expected time

$$T_P = T_1/P + O(T_\infty) \tag{2.1}$$

The left-hand side is called the ***work term*** and the right-hand side is the ***span term***. We use these terms to distinguish the common and rare case of work stealing respectively. We define the ***span***

overhead to be the smallest constant c_∞ such that

$$T_P \leq T_1/P + c_\infty T_\infty \tag{2.2}$$

Assumption 2 The average parallelism T_A is sufficiently large to obtain linear speedup. That is, we have $T_A/P \gg c_\infty$. Following from Equation 2.1, we have $T_P \approx T_1/P$, a linear speedup.

Assumption 3 For a given program, there is a corresponding ***sequential elision***. The sequential elision is a version of the program in which all parallel constructs are replaced with their sequential counterparts. For instance, a parallel tuple becomes an ordinary tuple. The sequential elision provides a baseline for measuring the performance of parallel evaluation. We can thus define the work overhead $c_1 = T_1/T_S$ where T_S is the execution time of the sequential elision. Following from our previous assumptions we have

$$\begin{aligned} T_P &\leq c_1 T_S/P + c_\infty T_\infty \\ &\approx c_1 T_S/P \end{aligned} \tag{2.3}$$

which suggests shows the work overhead as the common case.

Now we can state the work-first principle precisely [36]:

> Minimize c_1 even at the expense of increasing c_∞.

The key performance metric for work stealing is the number of steals required to schedule a given parallel execution graph. Blumeofe *et al.* proved that, for a given program scheduled by work stealing, the expected number of steals is $O(PT_\infty)$, and, in particular, the expected number of steals per processor is $O(T_\infty)$ [14]. Consequently, work stealing has the crucial property that all stealing-related overheads are contained in c_∞. This property is the key in applying the principle to an implementation of work stealing. It suggests moving most of the costs to when tasks are stolen and reducing costs elsewhere, such as when tasks are spawned.

For intuition on why the bound on the number of steals holds, recall that the thief steals the oldest ready task that is available in the subgraph owned by the victim. The subgraph rooted at this oldest task often contains the more nodes than the subgraphs rooted at the other ready nodes of the victim. This oldest task has the greatest potential in the sense that it contains the largest amount of work and is thus likely to keep the thief busy for the longest time. Arora *et al.* quantify this notion of potential for an individual steal and extend the notion to the whole program execution [3]. Using this notion of potential, they are able to bound the number of steals. The total potential of the program never increases, and each steal decreases this potential by a constant fraction with constant probability. Therefore, the expected total number of steals is limited by how quickly this potential vanishes.

On a shared-memory machine, there are several mechanisms one might use to provide task stealing. I categorize such a mechanism as having either public or private access. In public access, each processor shares a pointer to its own deque with the other processors. A thief steals a task from a victim by modifying in place the deque of the victim. The race condition between the thief and victim is avoided by using a shared-memory synchronization protocol. In private access, each processor maintains exclusive access to its own deque. To steal a task, a thief signals a victim asynchronously and awaits a response that contains either a message indicating failure or a stolen task. Signaling is typically implemented by software polling or OS interrupts.

CHAPTER 3

LAZY PROMOTION

In this chapter, I present WS LP, a work-stealing policy that uses lazy promotion. The design of WS LP is informed by the work-first principle [36]. The work-first principle states that a scheduler design should seek to minimize the scheduling costs imposed on the work of the computation. In Manticore, the most significant amount of work overhead is imposed by memory management, and in particular, promotion. The design of WS LP stays faithful to the work-first principle by setting its highest priority to be the reduction of promotion costs.

3.1 Motivation

Recall the description of the Manticore memory manager from Chapter 2.4. In order to share a heap object between processors, that heap object must first be promoted. Promotion is the process of copying a heap object (and transitively copying each object reachable from that object) from a processor-local heap to the global heap. Reducing the amount of promoted data reduces both the cost of copying objects to the global heap and the amount of data that enters the global heap. Promoting a *task* involves promoting all the heap objects that are reachable from that task. The scheduler plays an important role in determining when and how many tasks are promoted.

Manticore's first implementation of work stealing used eager promotion, which means that, for each task t, the scheduler promotes t just before pushing t on the deque. Through some early analysis I found that the scheduler overhead was high. A parallel function call was at least seventeen times more expensive than the corresponding serial function call. By comparison, in Cilk, this factor is between four and eight depending on the architecture. Figure 3.1 shows the breakdown of the serial overhead of the Fib benchmark. The breakdown reveals that about half of the overhead of work stealing can be attributed to promotion.

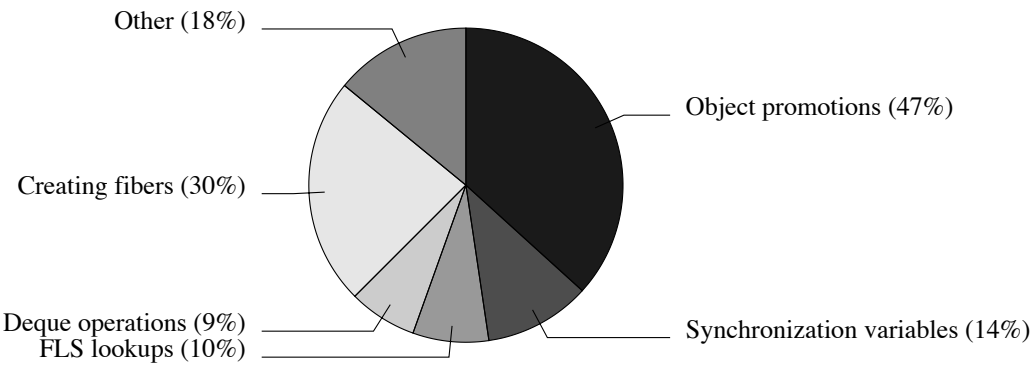


Figure 3.1: Breakdown of work-stealing overheads

Methodology The methodology I used to obtain this result is similar to the methodology used by Frigo *et al.* in their evaluation of Cilk's work-stealing scheduler [36]. The methodology is as follows. Because the `fib` function spends nearly all of its time scheduling and very little time computing, its execution time gives a reasonably accurate estimate of scheduling overhead. I start with the sequential elision `fib` and measure the time to execute it on a single processor. Then, to this program, I manually add on each source of overhead and time the resulting code. Doing so, I was able to isolate the five main components of the total scheduling cost. These components are

Object promotions (47%) This accounts for promoting the task being pushed on the deque.

Synchronization variables (14%) This accounts for the non-synchronization overhead associated with synchronization variables.

FLS lookups (10%) This accounts for processor-local storage lookups, which are used to access scheduling state, such as the deque.

Deque operations (9%) This accounts for deque operations.

Creating fibers (3%) This accounts for creating the fibers that go on the deque.

In addition, 18% of the overhead is unaccounted for.

In addition to the high overheads of eager promotion, there is another, stronger motivation to use a different strategy. Global heap collections require barrier synchronizations and use atomic memory operations that have the potential to saturate the memory bus. Promotion incurs such a high cost because it increases the amount of data in the global heap, and more data in the global heap means more global collections. Communication costs such as these limit the scalability of the whole system.

Using lazy promotion can help to lower the work overhead and reduce the amount of promoted data. In lazy promotion, for each task t, the scheduler delays promoting the t until just before t is stolen, and if t is never stolen, then t is never promoted. Recall that the expected number of steals in randomized work stealing is $O(PT_\infty)$. Since lazy promotion promotes two tasks per steal, one for the right branch and one for the join task, the expected number of promoted tasks promoted by lazy promotion is $O(PT_\infty)$, and as such lazy promotion moves the cost of promotion onto the span of the computation. Lazy promotion effectively moves the promotion overhead off the work overhead and onto the span, and since promotion is costly, doing so is likely to be worthwhile.

In order to support lazy promotion, either Manticore's heap invariant must be relaxed or the scheduler implementation must use a private-access model in which deques are hidden from other processors. To understand why, recall that Manticore's memory manager relies on two heap invariants to reduce the cost of communication (Figure 2.7). A pointer outside a per-processor heap cannot point to an object inside a per-processor heap. Under such a heap invariant, a public-access implementation must promote each task entering a deque because a thief processor could steal the task at any moment. A private-access implementation can readily support lazy promotion because, in private-access, a deque is hidden from thief processors.

3.2 The WS LP policy

Figure 3.2 shows the pseudocode for the per-worker loop of WS LP. WS LP bears resemblance to the work stealing policies proposed by Blumeofe and Leiserson [14] and Arora *et al.* [3]. As is

common, in WS LP, each processor i has a deque Q_i of tasks and an assigned task t_i that records the task that the processor is executing. But unlike the other policies, in WS LP, a processor has sole access to its own deque (*i.e.*, it uses the private-access model described above). To steal from another processor's deque, a thief must send a steal request to its victim, and the victim must reply. WS LP guarantees lazy promotion by delaying the promotion of a task until that task is stolen. If a task is executed locally (*i.e.*, not stolen), the task will never be promoted.

Let us consider the operation of an individual processor. Each round of scheduling, a processor checks whether it has an assigned task (line 4). If it does not, the processor becomes a thief. Otherwise, the processor executes the assigned task. When the assigned task spawns two tasks, called r and s, the processor pushes s on the local deque and makes r the assigned task (line 9). When the assigned task joins two tasks, called r and s, the processor has to determine which node to execute next (line 13). Figure 3.3 illustrates the structure of the parallel execution graph at this point. In particular, the task f spawns r and s, the task t_i joins r and s, and the task k is the continuation of the join. Without loss of generality, we may put t_i on the left- or right-hand side of this graph. If s was not stolen, then the processor pops s from the deque proceeds to execute s locally. If s was stolen, then the processor promotes the continuation task k and schedules it if necessary (line 24).

Figure 3.4 gives the pseudocode for initializing the scheduler. The scheduler starts with an one unexecuted task t_r, which represents the root task of the parallel execution graph. Processor zero gets assigned t_r while the other processors are assigned no tasks (they immediately become thieves). The per-processor scheduling instances are spawned after the state is initialized. Note that the per-processor loops never terminate. In an implementation, it is desirable to have the workers terminate once all tasks in the parallel execution graph have been executed. I elide this detail here because the termination protocol is fairly subtle, but note that Herlihy and Shavit describe a readily-applied solution [43].

WS LP uses the following protocol to handle task stealing. The thief, shown in Figure 3.5, first

```
1: procedure WORKER(i):
2: {each vproc maintains an assigned task t_i and a deque Q_i}
3: loop
4:    if t_i has already been executed then
5:       yield to parent scheduler {gives a thief a chance execute}
6:       call THIEF(i)
7:    else
8:       execute t_i
9:       if t_i spawns two tasks then
10:         {call these tasks r and s}
11:         push s on top of Q_i
12:         t_i ← r
13:      else if t_i joins two tasks then
14:         {call these tasks r and s}
15:         {let r be the ancestor of t_i (see Figure 3.3)}
16:         if Q_i is nonempty then
17:            {task s was not stolen}
18:            pop the task u from top of Q_i {it must be the case that u = s}
19:            t_i ← u
20:            push the continuation task k on top of Q_i
21:         else
22:            {task s was stolen}
23:            promote the continuation task k
24:            if k is ready to execute then
25:               t_i ← the continuation task k
26:            else
27:               continue to next round
28:            end if
29:         end if
30:      else if t_i is interrupted by an asynchronous signal then
31:         yield to parent scheduler {gives a thief a chance execute}
32:      end if
33:   end if
34: end loop
```

Figure 3.2: WS LP per-processor scheduling loop.

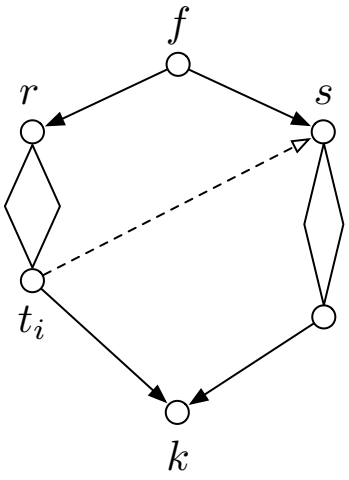


Figure 3.3: The structure of the parallel execution graph

1: **procedure** MAIN(t_r): {root task t_r}
2: $t_0 \leftarrow t_r$
3: $\forall_{i \neq 0}\ t_i \leftarrow$ dummy task
4: $\forall_i\ Q_i \leftarrow$ empty deque
5: **call** WORKER(i) in parallel for all vprocs i

Figure 3.4: WS LP initialization.

```
1: procedure THIEF(i):
2: {the thief subroutine is called by vproc i}
3: select a victim vproc j uniformly at random
4: spawn VICTIM(i, j) on vproc j
5: wait for notification from the victim vproc j
6: if the steal succeeded then
7:     t_i ← the stolen task
8: end if
9: return
```

Figure 3.5: WS LP thief subroutine.

selects a victim uniformly at random and then spawns a liaison fiber on the victim vproc. I describe the liaison fiber below. The thief then awaits a response from the liaison fiber. If the response is a negative acknowledgement, the thief returns and soon after, the scheduler loop will try to steal again. When the response is a stolen task, the thief pushes the task on its deque and returns.

The liaison fiber executes the subroutine shown in Figure 3.6 on the victim processor. If the victim's deque has fewer than one task, the liaison fiber replies with a negative acknowledgement, and otherwise, it pops the task from the bottom of the deque, promotes the task to the global heap, and replies by sharing a reference to the stolen task. Once the subroutine returns, the victim processor can then resume what it was doing previously.

This task-stealing mechanism bears some similarity to *active messages* [88]. The portion of the thief code that runs on the victim vproc is an independent computational agent (it is represented as a fiber). This approach has the advantage that there is no need for the vproc or the scheduler to "interpret" a steal request. The fiber carrying the thief code is itself the message. We can use this flexibility to experiment with alternative steal policies, such as one that reduces latency by sending out multiple thieves at once.

```
1: procedure VICTIM(t, v): {thief t, victim v}
2: if there are at least two tasks on Q_v then
3:     pop the oldest task s from the bottom of Q_v
4:     promote s
5:     notify the thief vproc t that it has stolen s
6: else
7:     notify the thief vproc t that the steal attempt failed
8: end if
```

Figure 3.6: WS LP victim subroutine.

3.3 Implementing WS LP in Manticore

The pseudocode in Figures 3.2, 3.4, 3.6, and 3.5 contain the specification for WS LP. In this section, I present the implementation that is used by Manticore. The implementation consists of three parts. The first two, the implementation of deques and clone compilation, provide optimizations that are crucial for providing reasonably-low scheduling costs. The third part is the mechanism used by WS LP to facilitate steal requests. This mechanism is based on a technique in which each processor polls regularly for steal requests. I show how to balance between overloading processors with steal requests and keeping them responsive enough to maintain high processor utilization.

The new implementation offers improved performance over Manticore's original eager-promotion implementation. The `fib` benchmark in Figure 2.6 runs almost twice as fast as the version under eager-promotion. This improvement is largely due to the reduced promotion costs and faster private-access deques used by WS LP. Furthermore, Section 3.4 shows that WS LP also offers superior scalability over the eager-promotion implementation.

One contribution of this implementation is its overall robustness. To achieve robustness, the implementation addresses three issues:

- Deques do not overflow.
- Clone compilation avoids two potential sources of blowup in the size of its generated code.
- The steal-request protocol avoids flooding busy processors with excessive steal requests but

still polls actively enough to keep processors busy most of the time.

Several alternative techniques for reducing work overheads have been used in other implementations of work stealing [27, 39, 62]. These techniques rely on extending a compiler to support special calling conventions and stack layouts. Such extensions complicate the compiler implementation and make it hard to provide flexibility (*e.g.*, it is harder to switch to a different stack layout or support multiple layouts). In this sense, these techniques are not as modular as techniques used by WS LP.

The implementation of WS LP does not require special calling conventions or stack layouts. The only special compiler support the implementation is a clone-compilation pass, which is a lightweight source-to-source transformation. Otherwise, the Manticore implementation is a standalone library.

3.3.1 Deques

With regard to Manticore's implementation, the purpose of the deque is to provide the victim processor with constant-time access to its oldest stealable task s. Observe, however, that if we relax the constant-time access requirement, the deque becomes superfluous. The victim could instead access s by either parsing its own call stack or by backtracking. Both of these strategies would require $O(T_\infty)$ time per steal. Supposing the number of steals is $O(T_\infty P)$, as predicted by the theoretical analysis, the scheduler would spend $O({T_\infty}^2 P)$ time traversing stacks, which would be unacceptable for large T_∞. The parsing strategy could perhaps reduce this cost in practice by walking the stack from the oldest frame to the newest, but in general, the length of the walk would still be unbounded. Nevertheless, it may be profitable to investigate techniques to improve on this worst-case behavior. In Section 6, I discuss the issue further.

The deque supports just the three operations shown in Figure 3.7. The first two are used by the local worker. Recall that a local worker treats its deque as a LIFO stack. These operations provide exactly this function to the local worker. The other operation is used by the thief to steal a task.

```
(* pushTop f *)
(* create a task t corresponding to thunk f and push t on  *)
(* the top of the local deque *)
val pushTop  : (unit -> unit) -> task

(* popTop () *)
(* pop a task from the top of the local deque. *)
(* returns either SOME t where t is the popped task  *)
(* or NONE if the deque is empty. *)
val popTop   : unit -> task option

(* popBot () *)
(* pop a task from the bottom of a local deque *)
(* returns either SOME t where t is the popped task  *)
(* or NONE if the deque is empty. *)
val popBot   : unit -> task option
```

Figure 3.7: Three deque operations.

The task returned by this operation is the oldest stealable task local to the processor. Therefore, the task typically represents a large chunk of work. The thief is likely to stay busy executing this task for a long while before it needs to steal again.

The two local operations are executed frequently during the execution of a program, once for each fork-join. An efficient deque implementation can reduce scheduling costs significantly. The deque representation that I use is a simple circular buffer. The buffer uses the "always leave one byte open" policy in order to distinguish between the cases when the buffer is full or empty. Although a few bytes are wasted as a result, this policy is preferable because it requires a smaller number of memory accesses than some alternative policies, such as read/write counts and fill counts. One disadvantage of using a circular buffer is the cost of the conditional branch to handle pointer wraparound. But this cost is negligible on processors, such as the x86-64, that provide a conditional-move instruction.

Deque overflow is an issue that any robust implementation must address. If a processor tries to push a task on a full deque, the scheduler allocates a new deque that is twice the size of the original deque and copies the tasks from the old to the new deque. Doing so is straightforward

because the deque is processor local, so there is no need for synchronization. The race condition with the thief is prevented by masking interrupts temporarily. It would be a waste, however, for each deque to remain at its maximum size, so the implementation provides a shrinking policy. If, when a processor tries to pop from its deque, the processor finds that the number of tasks in the deque is smaller than some faction of the deque size, the processor shrinks the deque. Shrinking entails allocating a new deque and copying the tasks to the new deque.

As an added benefit, WS LP may choose a small initial size for each deque under the assumption that the the number of tasks that are stored in the deque at a given instant tends to be small and nearly constant, even for large problem sizes. The deque size was set to 64 for every experiment reported in this chapter, and there was never a need to resize the deque.

3.3.2 Compiling clones

Several implementations of work stealing employ specialized calling conventions and stack layouts with the goal of reducing scheduling overheads [28, 39, 45]. Frigo *et al.*'s Cilk-5 implementation pioneered an alternative approach called clone compilation [36]. Clone compilation is a compiler pass that transforms a program with fork-join parallelism into a version that is readily optimized by the compiler. Clone compilation requires no special calling conventions or stack layouts and offers competitive performance.

The idea is to compile each parallel branching point to a fast and slow clone. The fast clone is executed in the common case in which both branches execute locally (*i.e.*, the right arm of the branch branch is not stolen). The fast clone executes with a minimal amount of synchronization (or none at all depending on the implementation). The slow clone is executed when the right arm of the branch is stolen. In the slow clone, because both branches run concurrently, the join point between the two branches must employ synchronization to determine which of the processors will execute the join continuation. The slow clone is more expensive because it uses costly synchronization. Clone compilation derives its efficiency from the property that the slow clone is only ever taken in

the rare case of a steal.

This section presents the clone compilation techniques I have developed for Manticore. Manticore offers clone compilation as an option for compiling PML's parallel tuple construct. Let us examine the clone compilation by example. Figure 2.1 shows the function to which we apply the translation. The `trProd` function takes a binary tree with integers at its leaves and returns the product of the leaves. It exploits parallelism at each branch of the tree. I show the compiled version in Figure 3.8.[1]

This code shows the fast and slow clone intermingled. The fast clone is taken if the second branch of the two parallel branches (*i.e.*, `trProd tr`) is not stolen. Observe that the test for this condition is based on whether the deque is empty or not. Consider why this check works. When the deque is empty, clearly all the tasks on the local processor were stolen, and so was the right branch. If the deque is not empty, then the right branch must be sitting at the top of the deque where the processor left it. After popping this task, the processor can simply discard it and proceed to with the fast path.

The slow clone is taken if the right branch is stolen. The computation splits into two parallel tasks, one local and one on the thief. When each task completes, it joins with the other one by calling `join ()`. The first processor to join calls `stop ()`, thus yielding control to its parent scheduler; the second processor resumes the return continuation `retK`. The tie between the two processors is broken by performing an atomic test-and-set operation on the `resume` variable. The results of the two branches are maintained in `xL` and `xR`.

By calling `stop ()`, the joining processor hands control off to the scheduler action sitting on the top of the scheduler-action stack. The implementation WS LP is structured so that that this scheduler action will be the part of WS LP which handles stealing. When activated, this scheduler activation immediately makes the joining processor a thief processor. Because it is not crucial for understanding the performance of WS LP, I describe this part of the implementation in Chapter 5.

1. Note that, in Manticore, clone compilation is an AST-to-BOM pass because the AST lacks support for mutable state and first-class continuations. Such mechanisms are present in BOM.

Observe that initially all of these state variables are allocated in the local heap. Only when the slow clone executes do these variables get promoted to the global heap. This property is enough to ensure lazy promotion.

Figure 3.9 shows the clone compilation for the general case. This translation is mostly a straightforward generalization of our example, but with the complication of supporting exceptions and avoiding code blowup for large tuples. The rest of this section describes three issues I address in this general case that have not been addressed elsewhere:

- Supporting lazy promotion.
- Supporting the semantics for exceptions defined by PML [33].
- Avoiding code blowup.

Supporting exceptions

The semantics of PML requires that exceptions be delivered in the order in which they would be delivered under a sequential evaluation [33]. In the case of a parallel tuple, an exception raised during the evaluation of the element e_i trumps any exception raised by an element in the tuple that is to the right of e_i. For example, the expression

```
(| raise A, raise B |)
```

always raises `A`.

Below are two necessary modifications to the basic clone translation.

1. If, during evaluation of the tuple element e_i, an exception is raised, the raising processor cancels evaluations of tuple elements right of e_i. The cancellation operation

```
val cancelTask : task -> unit
```

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = callcc (fn retK => let
    val resume = alloc false
    val xL = alloc NONE
    val xR = alloc NONE
    fun join () = if atomicTestAndSet (promote resume)
      then
        throw retK (valOf (!(promote xL)) * valOf (!(promote xR)))
      else
        stop () (* this worker now becomes a thief *)
    val _ = pushTop (fn _ =>
              (* slow clone (executed by the thief) *)
              (xR := promote (SOME (trProd tR));
               join ()))
    val vL = trProd tL
    in
      if popTop () <> NONE
        then
          (* fast clone *)
          vL * trProd tR
        else
          (* slow clone (executed by the victim) *)
          (xL := promote (SOME vL); join ())
    end)
```

Figure 3.8: Parallel tree-product function after clone compilation.

takes a task and frees up its resources quickly. The `cancelTask` operation is derived from the implementation of cancellable fibers, which I describe later in Chapter 5.

2. If, during evaluation of the tuple element e_i, an exception is raised, the raising processor waits for one of two events to occur:

   (a) At least one of the tuple elements left of e_i raises an exception.

   (b) Every tuple element left of e_i finishes evaluating.

   In case (a), the processor becomes a thief and in (b), the processor propagates the exception to the next level up in the call chain.

One remaining issue is that, in handling case 1, our clone translation has the raising processor busy wait by calling `waitOn`. Let e_i be the tuple element raising the exception. Busy waiting can be avoided, however, by suspending the task t_i corresponding to e_i. We have two cases to address.

1. An element to the left of e_i raises an exception, in which case, the suspended task gets thrown away and reclaimed by the GC.

2. All elements to the left of e_i finish evaluating, in which case, the last of these elements to finish resumes e_i.

The suspend and resume protocol can be implemented by using a few shared state variables and protecting them with a mutex lock. Since the this protocol only gets invoked in the slow clone, the rare case, a lock is an acceptable cost.

Avoiding code blowup

Manticore's clone compilation scheme avoids two potential sources of code blowup.

Exponential code blowup Recall that each subexpression except for the first one is compiled to a fast and a slow clone version. It is naïve to simply copy these versions because doing so would lead to an exponential increase in program size. For example, the naïve compilation of

$$\texttt{(|}\ e_1\texttt{,}\ \texttt{(|}\ e_2\texttt{,}\ \texttt{(|}\ e_3\texttt{,}\ e_4\ \texttt{|)}\ \texttt{|)}\ \texttt{|)}$$

would make eight copies of e_4. To prevent such blowup, I lift each subexpression (except the first one) into its own function, which is shared by the fast and slow clone.

Quadratic code blowup In the body of the let expression, there are a series in-place updates performed on the x_i variables. For each element j, there are $n - j$ updates, which leads to n^2 such updates overall. I avoid this quadratic blowup by using the following strategy. For parallel tuples with greater than four elements, I break the construction into two phases, one that builds an intermediate tuple and one that builds a flattened copy of that has the desired structure. I limit the size of each intermediate tuple to a small constant so as to avoid the quadratic blowup. Note that this strategy is implemented in Manticore but not reflected in the translation in Figure 3.9.

3.3.3 Steal requests

The work-first principle suggests that, because steals are rare, lowering the overheads imposed by the thief should be of secondary concern to lowering work overheads. But there is a limit to the latency that is tolerable for steal requests, as this latency sets a lower bound on the amount of parallelism necessary to obtain a speedup. The Manticore implementation described in this chapter uses a lightweight communication mechanism based on software polling. In software polling, each processor polls regularly for incoming messages, such as steal requests, and reacts to new messages quickly. Manticore amortizes much of the work overhead costs by integrating the polling with GC checks.

There are some reasonable alternatives to software polling, such as hardware interrupts. Later,

```
⟦ (| e1, ..., en |) ⟧
                                        ⟹
callcc(fn k =>
  let val nDone = alloc 1
      val (x1, ..., xn) = (alloc NONE, ..., alloc NONE)
      fun join i =
          if atomicFetchAndAdd (promote nDone, i) = n - i + 1
          then throw k(valOf (!(promote x1)),
                       ...,
                       valOf (!(promote xn)))
          else stop()
      fun waitOn xi = if (!xi) = NONE then waitOn xi else ()
      fun fn () = ⟦ en ⟧
      val wn = pushTop (fn () => (
                  xn := SOME (fn()
                     handle e => (waitOn xn-1; ... waitOn x1;
                                  raise e)));
                  join 1))
           ...
      fun f2 () = ⟦ e2 ⟧ handle e => (cancelTask w3; ... cancelTask wn;
                                      raise e)
      val w2 = pushTop (fn () => (
                  x2 := SOME (f2() handle e => (waitOn x1;
                                                raise e));
                  join 1))
      val v1 = ⟦ e1 ⟧
               handle e => (cancelTask w2; ... cancelTask wn; raise e)
  in
      if popTop() <> NONE then
          ...
          let val vn-1 = fn-1()
          in
            if popTop() <> NONE then
                (v1, ..., fn())
            else
                (x1 := promote (SOME v1);
                 ...
                 xn-1 := promote (SOME vn-1);
                 join(n - 1))
          end
          ...
      else
          (x1 := promote (SOME v)1; join 1)
  end)
```

Figure 3.9: The full clone translation for parallel tuples.

in Section 5.3, I discuss mechanism used by Manticore in detail and evaluate it compared to hardware interrupts.

On a multicore machine, using software polling, the round-trip latency of a steal request is in the tens of microseconds for a steal and in the hundreds of nanoseconds for a failed steal attempt. By comparison, a public-access implementation of work stealing typically offers latency in the hundreds of nanoseconds in both cases. The higher latency has not been a problem so far, but reducing this latency is still a worthwhile goal, as it is quite high compared to the public-access approach.

There are two techniques that can help reduce latency in software polling. The first is to have each processor maintain a flag indicating whether the processor is busy or idle. The thief always checks this flag before sending a steal request. The second technique allows the thief to send multiple steal requests concurrently, much like a broadcast mechanism. This latter mechanism remains future work.

One disadvantage of such a signaling approach is that processors that are busy with useful work must bear some of the scheduling overhead by responding to steal requests, including those requests that are unsuccessful. The scheduler must find a balance between sending too few and too many steal requests. If there are too few, processors will go underutilized. If there are too many, performance will be degraded because busy processors will spend much of their time servicing steal requests. This latter issue is especially problematic when a processor is executing a short sequential section and is about expose parallelism. It is best to let the busy processor execute uninterrupted until it exposes parallelism.

WS LP operates under the assumption that the longer a thief goes without stealing, the longer it will be until stealable tasks become available. On the other hand, stealable tasks are likely to be exposed soon after a processor starts emitting steal requests. WS LP therefore uses the following back-off strategy for steal requests. The thief first tries to steal every processor. If the attempts fail, the thief spins for a few microseconds before repeating. The next round, the thief spins twice

as long. After reaching some maximum of spin cycles, the thief sleeps for a few milliseconds by calling `pthread_cond_timedwait()`. This way, the thief finds new work quickly if it is available, but does not overburden the system if work is scarce.

3.4 Empirical evaluation

In this section, I present three empirical studies using the six benchmarks from Chapter 2. The first study looks at the performance of Manticore's work-stealing system as compared to a state-of-the art sequential implementation. The second study examines the performance benefits of lazy-promotion as compared with Manticore's original eager-promotion implementation. Since one might question the split-heap design choice, the last study compares the split heap with a flat-heap implementation.

3.4.1 Scalability

Figure 3.10 contains speedup curves for our six benchmarks, where the sequential baseline times are collected from the sequential elisions of the benchmarks compiled by MLton. MLton is a sequential whole-program optimizing compiler for Standard ML [90, 60], which is known for the high performance of its generated code.[2] Considering MLton's suite of aggressive optimizations and its overall maturity over our implementation in terms of sequential performance, the speedup curves are encouraging.

The Quicksort benchmark scales linearly to only four times as fast as MLton. I attribute this imperfect speedup to three factors. First, the MLton compiler aggressively flattens data representations, but Manticore does not. For example, the MLton version sorts vectors of integers, whereas Manticore sorts vectors of boxed (*i.e.*, heap-allocated) integers. Second, the MLton compiler specializes the higher-order filter function, which is used to partition the array, whereas our version

2. We used MLton version 20070826 for the AMD64.

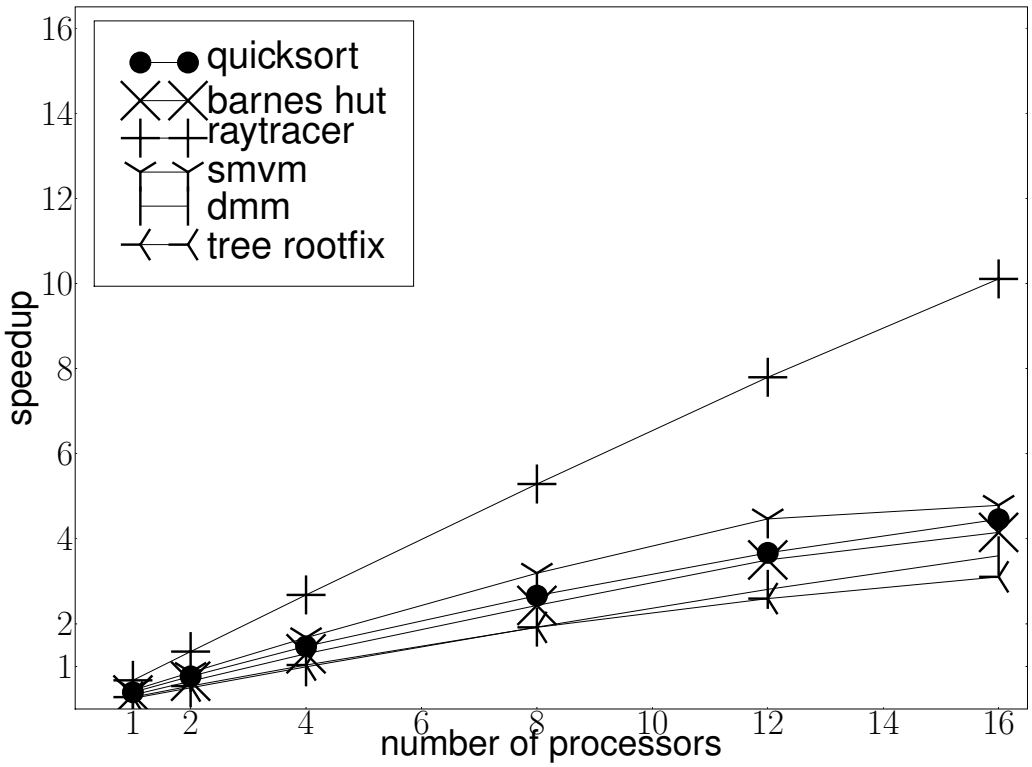


Figure 3.10: Parallel speedup of our lazy-promotion work-stealing system over sequential MLton

is building and passing a closure for the predicate functions. Third, as discussed below, there is higher global-GC overhead for this benchmark. The first two of these issues can be addressed by improving Manticore's sequential performance.

Performance is slightly weaker for Barnes-Hut. Our single-processor performance is about twice as slow as MLton, which gives us a flatter curve. This program's lack of scaling beyond twelve processors stems from the limited average parallelism, since we saw better speedups when we increased the problem size to 400,000 particles.

The Raytracer benchmark is embarrassingly parallel and scales linearly, on sixteen cores running about ten times faster than MLton. We note that the Raytracer's single-processor performance in our system is nearly the same as in MLton, which is partly the reason Raytracer has a steeper curve than in the other benchmarks.

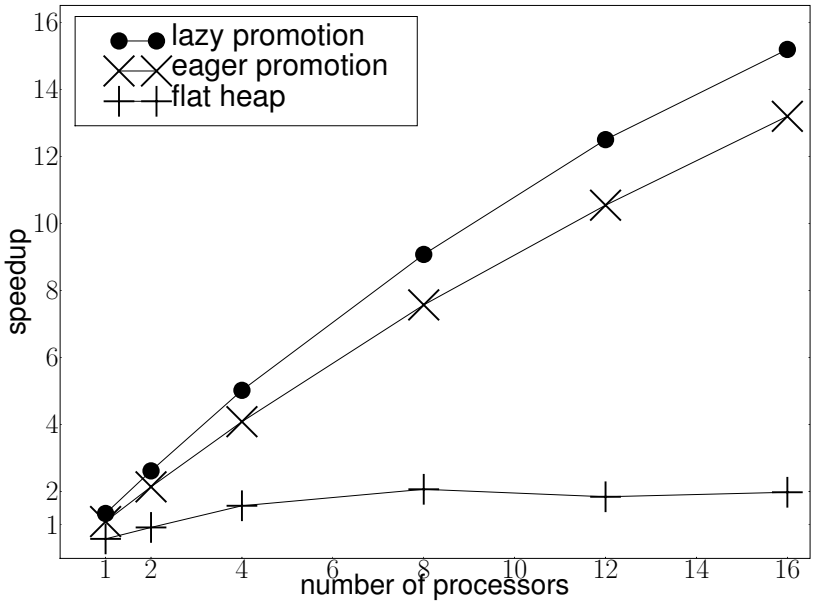
Performance for SMVM is similar to Barnes Hut. The lack of scaling beyond twelve processors stems from the limited average parallelism for the chosen problem size (1 million nonzeroes).

DMM has a smooth linear speedup, but the speedup curve is flat for the same reason as with Quicksort and Barnes Hut: Manticore offers weaker sequential sequential performance than MLton.

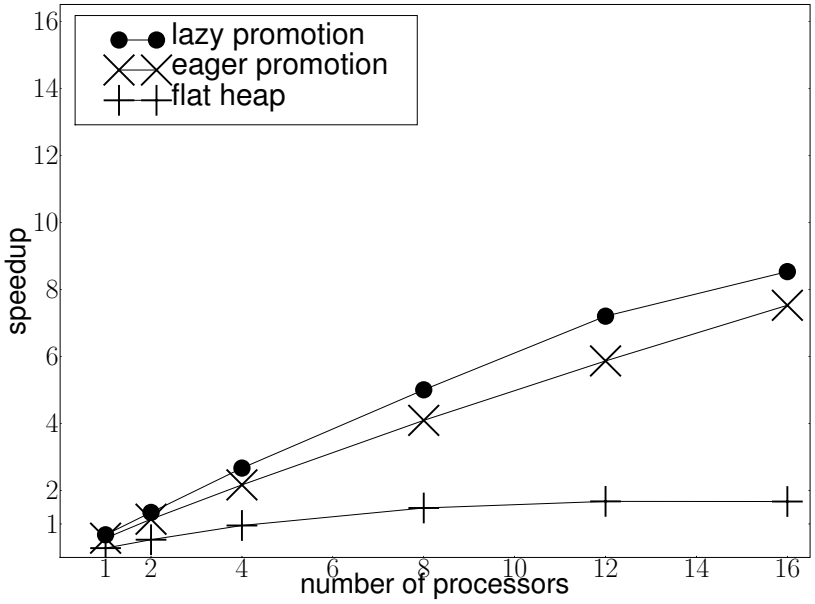
The Tree Rootfix benchmark scales linearly up to sixteen cores, where the speedup is about three times MLton.

3.4.2 Lazy vs. *eager promotion*

Across the six benchmarks, lazy-promotion often outperforms eager promotion. Quicksort, Barnes Hut, and DMM show significant benefits from lazy promotion, and only SMVM shows a penalty from lazy promotion. By breaking down where these programs spend their time, we can understand why and when lazy promotion helps performance. In Tables 3.1 and 3.2, I show the break down of the execution time into non-GC (*i.e.*, mutator) and GC components. This data demonstrates that the benefit of lazy promotion over eager promotion comes from reducing the time spent in the

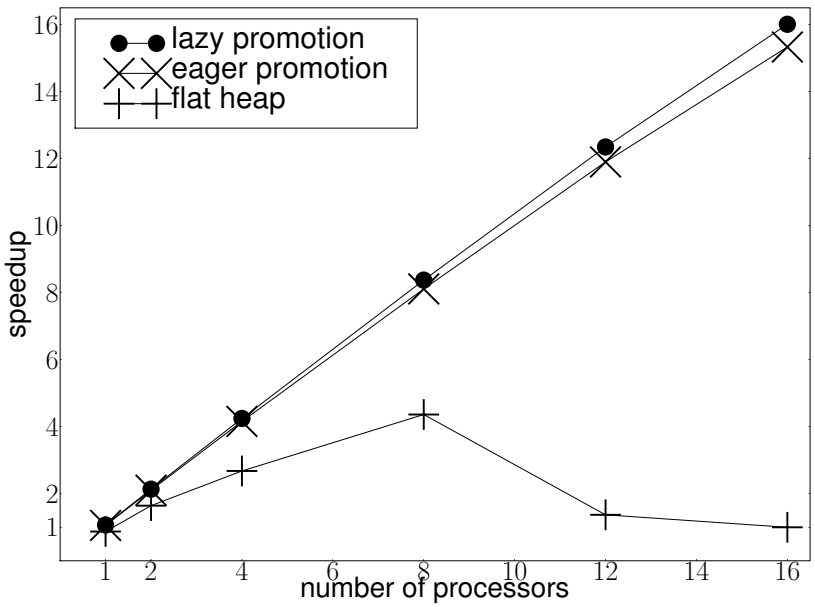


(a) Quicksort

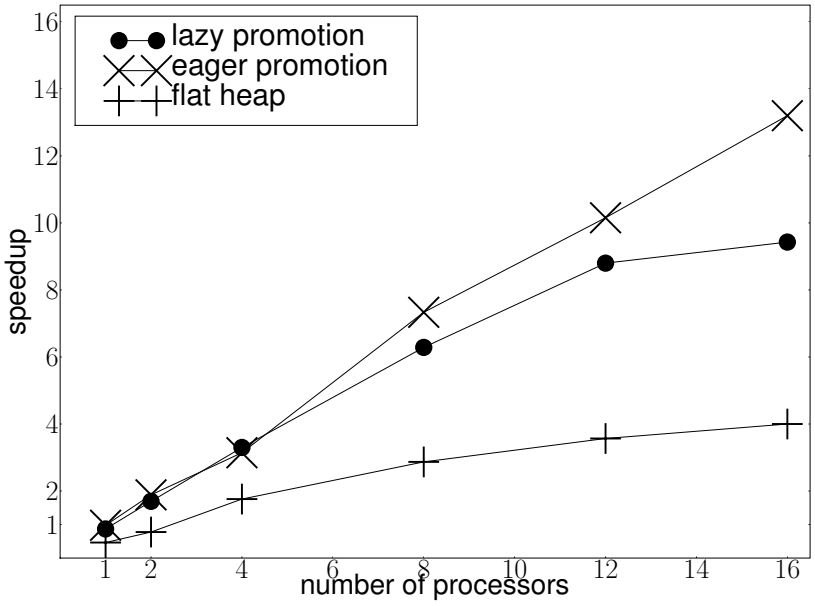


(b) Barnes-Hut

Figure 3.11: Comparative speedup plots for the three versions of our system.

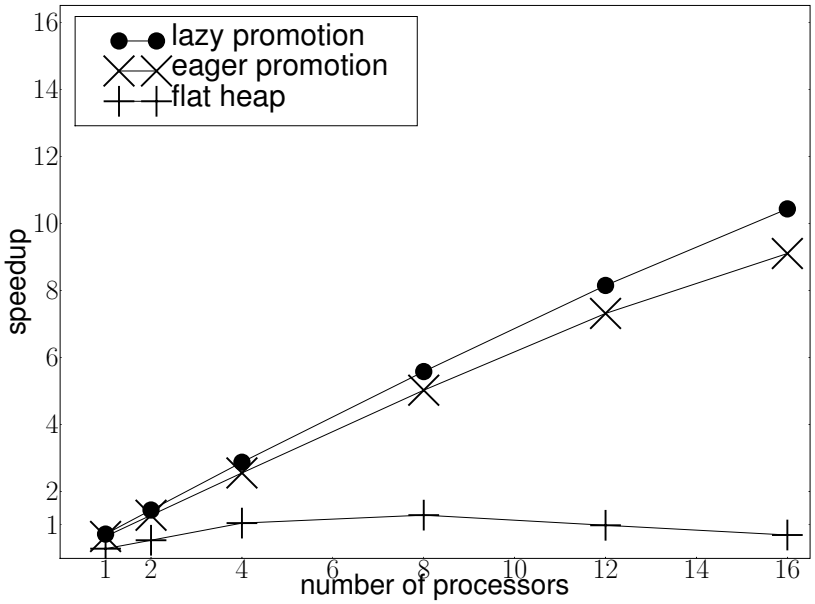


(c) Raytracer

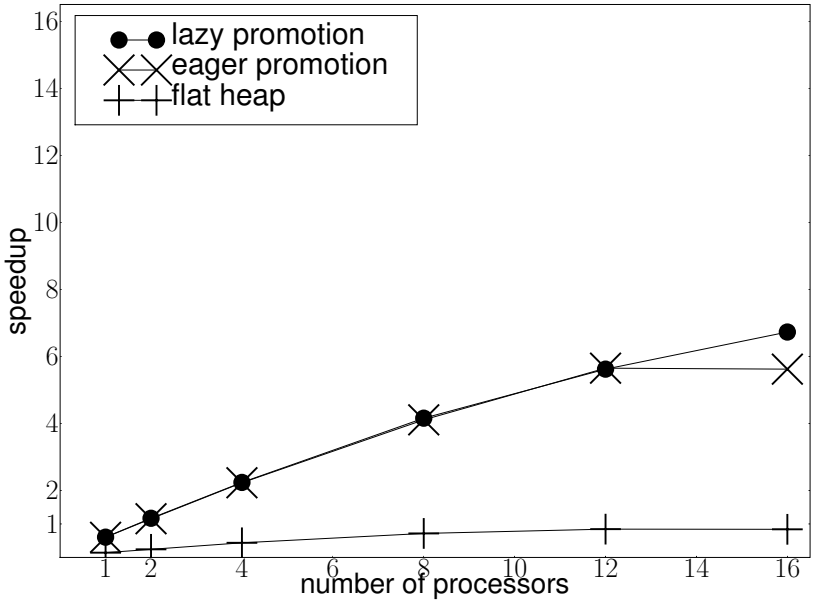


(d) SMVM

Figure 3.12: Comparative speedup plots for the three versions of our system.



(e) DMM



(f) Tree Rootfix

Figure 3.13: Comparative speedup plots for the three versions of our system.

garbage collector. It also explains why we see no improvement for the Raytracer and a degradation in SMVM, since both benchmarks spend virtually no time in GC.

We can further refine this data by dividing the GC time into three categories: local GC (*i.e.*, minor and major collections), promotion, and global GC as is shown in Tables 3.3 and 3.4. This data shows that extra time spent promoting objects and extra time spent in global collections accounts for the bulk of the extra GC time in the eager-promotion version. Note, however, that promotion is more significant for Barnes-Hut, whereas global GC is more significant for Tree Rootfix. I attribute much of this difference to the fact that Tree Rootfix maintains a larger live-data set than Barnes-Hut.

3.4.3 Split-heap vs. flat-heap

The last set of experiments are intended to test the design choice of splitting the heap. In particular, we want to know if the cost of promotion outweighs the benefit of local heaps. For these experiments, I used a version of Manticore combined with a flat (or unified) heap.[3] In the flat-heap implementation, each vproc allocates from its own chunk of memory. When this chunk is full, it is added to the the global to-space and another chunk is acquired. Eventually the global to-space exceeds a threshold triggering a parallel collection, which is essentially the global GC used by the other versions. The threshold is determined in the same way as in the split heap. Because the heap is flat, there is no need to promote objects that are globally visible. As can be seen from Figures 3.11, 3.12, and 3.13, the flat-heap version is significantly slower than either the eager or lazy-promotion versions.

Tables 3.1 and 3.2 show the execution time broken down into non-GC and GC components. It might be argued that a better flat-heap GC (*e.g.*, a generational GC) might improve the performance of the flat-heap version sufficiently to better our split-heap performance, but there is data that weighs against this conclusion. Specifically, even though the GC overhead in the flat-heap

3. The flat-heap version of Manticore was programmed by John Reppy with some assistance from the author.

Quicksort

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
1	4.44	4.05+0.39	6.07	4.57+1.50	10.10	8.96+1.14
2	2.82	2.67+0.15	3.14	2.38+0.77	6.40	5.53+0.86
4	1.19	1.10+0.10	1.60	1.27+0.33	3.77	3.21+0.55
6	0.83	0.76+0.08	1.07	0.78+0.29	3.34	2.76+0.58
8	0.65	0.59+0.06	0.81	0.62+0.19	2.78	2.53+0.25
10	0.55	0.51+0.04	0.72	0.43+0.29	2.97	2.70+0.27
12	0.46	0.42+0.04	0.56	0.45+0.11	3.19	2.97+0.21
14	0.42	0.38+0.04	0.63	0.54+0.09	3.19	2.87+0.32
16	0.40	0.38+0.03	0.58	0.45+0.13	3.65	3.46+0.19

Barnes-Hut

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
1	15.66	10.90+4.76	17.87	11.55+6.32	29.67	25.98+3.69
2	13.45	10.89+2.57	8.97	5.85+3.12	25.03	22.29+2.74
4	11.41	10.06+1.35	4.55	3.00+1.55	23.52	21.10+2.41
6	10.77	10.24+0.53	3.28	2.16+1.12	22.66	20.88+1.78
8	5.05	4.29+0.76	2.64	1.77+0.86	21.36	19.79+1.57
10	2.43	1.77+0.66	2.23	1.57+0.66	21.80	20.47+1.33
12	2.43	1.91+0.51	1.96	1.24+0.72	21.25	19.94+1.30
14	1.84	1.18+0.65	1.91	1.21+0.71	13.59	12.70+0.88
16	1.82	1.09+0.72	1.83	1.05+0.78	13.35	12.54+0.80

Raytracer

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
1	3.39	3.34+0.05	3.59	3.51+0.08	4.33	4.28+0.05
2	1.69	1.67+0.02	1.80	1.76+0.04	2.33	2.29+0.03
4	0.85	0.84+0.01	0.91	0.90+0.02	1.31	1.27+0.04
6	0.57	0.56+0.01	0.61	0.60+0.01	0.97	0.96+0.01
8	0.43	0.43+0.01	0.47	0.46+0.01	0.84	0.82+0.02
10	0.35	0.34+0.01	0.38	0.37+0.01	1.62	1.60+0.02
12	0.29	0.29+0.00	0.32	0.31+0.01	2.54	2.53+0.00
14	0.25	0.25+0.00	0.28	0.27+0.01	3.00	3.00+0.00
16	0.22	0.22+0.01	0.24	0.24+0.00	3.60	3.60+0.00

Table 3.1: Execution-time breakdown (in seconds)

SMVM

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
	0.16	0.16+0.00	0.14	0.14+0.00	0.28	0.28+0.00
2	0.08	0.08+0.00	0.07	0.07+0.00	0.17	0.17+0.00
4	0.04	0.04+0.00	0.04	0.04+0.00	0.08	0.08+0.00
6	0.03	0.03+0.00	0.02	0.02+0.00	0.06	0.06+0.00
8	0.02	0.02+0.00	0.02	0.02+0.00	0.04	0.04+0.00
10	0.02	0.02+0.00	0.02	0.02+0.00	0.04	0.04+0.00
12	0.02	0.02+0.00	0.01	0.01+0.00	0.04	0.04+0.00
14	0.01	0.01+0.00	0.01	0.01+0.00	0.04	0.04+0.00
16	0.01	0.01+0.00	0.01	0.01+0.00	0.03	0.03+0.00

DMM

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
1	6.04	5.41+0.63	6.94	5.33+1.61	14.74	14.37+0.36
2	3.02	2.71+0.32	3.48	2.65+0.83	7.57	7.35+0.22
4	1.52	1.36+0.16	1.78	1.36+0.42	4.05	3.90+0.16
6	1.03	0.91+0.11	1.20	0.90+0.30	3.16	3.03+0.14
8	0.78	0.70+0.09	0.90	0.68+0.23	3.33	3.23+0.11
10	0.65	0.57+0.08	0.73	0.54+0.19	3.74	3.65+0.09
12	0.53	0.47+0.06	0.62	0.46+0.16	4.34	4.27+0.07
14	0.47	0.41+0.06	0.54	0.39+0.15	5.14	5.07+0.07
16	0.43	0.37+0.06	0.49	0.35+0.14	6.29	6.22+0.07

Tree Rootfix

	lazy		eager		flat	
n-procs	time	non-GC + GC	time	non-GC + GC	time	non-GC + GC
1	10.68	8.45+2.23	10.77	7.88+2.89	27.40	13.57+13.83
2	7.01	6.46+0.55	5.92	4.19+1.74	17.63	8.07+9.56
4	4.65	4.16+0.49	3.69	2.51+1.19	13.46	5.95+7.51
6	3.46	2.98+0.48	2.62	1.63+0.98	10.11	5.22+4.90
8	3.49	3.18+0.31	2.21	1.41+0.80	8.51	4.70+3.80
10	2.18	1.85+0.33	1.97	1.12+0.86	7.59	4.72+2.87
12	1.43	1.21+0.22	2.15	1.10+1.05	6.97	4.60+2.37
14	1.36	1.15+0.22	1.87	1.11+0.75	6.46	3.68+2.78
16	1.36	1.17+0.19	1.74	0.61+1.13	6.34	4.42+1.92

Table 3.2: Execution-time breakdown (in seconds)

Quicksort

	lazy promotion			eager promotion		
procs	local	prom.	global	local	prom.	global
1	0.38	0.00	0.00	0.21	1.29	0.00
2	0.19	0.00	0.00	0.10	0.66	0.00
4	0.07	0.00	0.00	0.03	0.33	0.00
6	0.10	0.00	0.00	0.03	0.20	0.00
8	0.04	0.00	0.00	0.03	0.14	0.00
10	0.04	0.00	0.00	0.00	0.17	0.00
12	0.03	0.00	0.00	0.00	0.14	0.00
14	0.03	0.00	0.00	0.02	0.12	0.00
16	0.02	0.00	0.00	0.01	0.12	0.00

Barnes-Hut

	lazy promotion			eager promotion		
procs	local	prom.	global	local	prom.	global
1	3.85	0.00	0.89	2.73	1.72	2.03
2	1.54	0.01	0.52	1.44	0.91	0.98
4	1.10	0.03	0.33	0.68	0.46	0.47
6	0.65	0.04	0.20	0.49	0.33	0.32
8	0.53	0.07	0.14	0.35	0.26	0.29
10	0.37	0.12	0.10	0.30	0.23	0.19
12	0.34	0.13	0.09	0.27	0.21	0.11
14	0.36	0.19	0.10	0.29	0.28	0.17
16	0.39	0.26	0.07	0.28	0.35	0.17

Table 3.3: Garbage collection time statistics (in seconds)

Tree Rootfix

	lazy promotion			eager promotion		
procs	local	prom.	global	local	prom.	global
1	1.41	0.72	0.00	0.82	1.41	0.69
2	0.97	0.32	0.00	0.39	0.72	0.64
4	0.33	0.27	0.00	0.22	0.37	0.59
6	0.34	0.09	0.00	0.13	0.21	0.49
8	0.17	0.11	0.00	0.12	0.16	0.53
10	0.16	0.12	0.00	0.32	0.14	0.48
12	0.13	0.08	0.00	0.06	0.16	0.57
14	0.12	0.07	0.00	0.07	0.14	0.52
16	0.14	0.09	0.00	0.07	0.13	0.50

DMM

	lazy promotion			eager promotion		
procs	local	prom.	global	local	prom.	global
1	0.63	0.00	0.00	0.33	1.27	0.00
2	0.32	0.00	0.00	0.17	0.68	0.00
4	0.16	0.00	0.00	0.08	0.34	0.00
6	0.11	0.00	0.00	0.06	0.24	0.00
8	0.08	0.02	0.00	0.05	0.18	0.00
10	0.06	0.01	0.00	0.04	0.15	0.00
12	0.05	0.01	0.00	0.03	0.16	0.00
14	0.05	0.01	0.00	0.03	0.12	0.00
16	0.04	0.01	0.00	0.02	0.12	0.00

Table 3.4: Garbage collection time statistics (in seconds).

implementation is higher, so is the non-GC time. In fact, the flat-heap non-GC time is higher than the total time of our split-heap versions.

The obvious explanation is that the split-heap implementations make better use of the L3 cache. Table 3.5 shows the L3 cache read- and write-miss counts across Quicksort, Barnes Hut, and Ray-tracer.[4] I report the average miss counts across ten runs of the benchmarks and across several processor configurations and input data sizes. Cache misses were read from the processor hardware counter, not from simulation. Due to overhead of rapidly and simultaneously reading the L3 cache performance counters from multiple vprocs, the global garbage collections had to be tracked separately. For this reason, the measurements include only global collections for the lazy and eager versions, but all collections for the flat-heap version.

Even including minor and major GC activity, the L3 cache read- and write-miss levels are significantly lower for the split-heap architecture than the flat-heap. Across all benchmarks, there are roughly an order of magnitude more L3 cache misses when using the flat-heap. These results suggest that the better performance of split heap can be attributed, in large part, better cache usage.

3.5 Related work

There are three major groups of previous work related to WS LP:

- older implementations of work stealing that are based on the idea of lazy task creation
- implementations of parallel Haskell
- implementations of imperative parallel languages

I discuss each of these in turn.

4. Lars Bergstrom instrumented Manticore with the necessary performance counters.

Barnes-Hut

		lazy		eager		flat	
n-procs	size	non-GC	gc	non-GC	gc	non-GC	gc
2	10K	0.853	0.0	2.27	0.0	25.5	0.109
2	50K	6.11	0.0	13.5	0.0	144.	2.10
2	100K	14.9	0.0	30.3	3.67	289.	8.12
8	10K	1.99	0.0	2.75	0.0	27.2	0.0101
8	50K	10.2	0.0	15.0	0.0	154.	0.763
8	100K	22.4	0.0	33.8	0.570	311.	2.12
16	10K	2.91	0.0	3.11	0.0	32.7	0.0266
16	50K	16.2	0.0	16.8	0.0	17.5	0.207
16	100K	35.6	0.109	38.1	0.183	362.	0.927

Raytracer

		lazy		eager		flat	
n-procs	size	non-GC	gc	non-GC	gc	non-GC	gc
2	100	0.439	0.0	0.645	0.0	7.28	0.0266
2	250	1.54	0.0	1.48	0.0	44.9	0.308
2	500	6.08	0.0	5.77	0.0	181.	2.47
8	100	0.701	0.0	2.32	0.0	11.2	0.0186
8	250	2.48	0.0	5.32	0.0	52.4	0.0813
8	500	6.18	0.0	8.04	0.0	192.	0.804
16	100	0.739	0.0	1.99	0.0	23.4	0.0192
16	250	2.41	0.0	3.97	0.0	69.9	0.0509
16	500	5.75	0.0	10.3	0.0	221.	0.451

Quicksort

		lazy		eager		flat	
n-procs	size	non-GC	gc	non-GC	gc	non-GC	gc
2	1M	7.32	0.0	10.6	0.0	148.	11.2
2	2M	16.0	0.0	22.1	0.0	286.	43.0
2	3M	26.7	0.0	32.0	16.6	420.	114.
8	1M	10.1	0.0	12.9	0.0	164.	4.68
8	2M	21.5	0.0	26.9	0.0	317.	15.6
8	3M	36.0	0.0	38.2	6.44	469.	42.4
16	1M	14.3	0.0	15.4	0.0	190.	3.61
16	2M	29.0	0.0	31.4	0.211	341.	8.25
16	3M	49.7	0.0	46.7	2.52	562.	20.2

Table 3.5: L3 Read+Write Cache Misses (in millions).

3.5.1 *Lazy task creation*

There are several early scheduler implementations for functional languages with similarities to the implementation of WS LP. These implementations use *lazy task creation* [62] to schedule computation, which results in schedules that are similar to those of WS LP, but they require special calling conventions and stack layouts, and do not support multiprogramming.

Lazy task creation was introduced by Mohr for the Mul-T language [62]. Other early examples include TAM [22] and Lazy Threads [39], which were implementations for non-strict languages.

Of the various implementation of work stealing, Feeley's design is most similar to Manticore's [27]. Like WS LP, Feeley's design has the thief and victim communicate via inter-processor signals, and the inter-processor communication is implemented by software polling. The primary motivation of Feeley's design is to improve the memory locality on a shared-memory, NUMA multiprocessor. His work does not address the interaction of the garbage collector and the scheduler, wheras this issue is the primary motivation and focus of this chapter.

3.5.2 *Parallel-Haskell implementations*

GUM is an implementation of parallel Haskell on clusters [85]. GUM uses a distributed-heap GC in which each processor can collect its own heap independently. Unlike Manticore's heap architecture GUM has no global heap. Instead, the GUM runtime distinguishes between global and local addresses and maintains information about references to and from other processors.

Like Feeley's system, GUM uses a special message (called a FISH) to request work from other processors. In both designs, the notion of "steal request" is a part of the notion of virtual processor. In contrast, the Manticore implementation carries out steal requests by sending "thief fibers" to take work directly from victim vprocs. The notion vproc is independent of the notion of steal request. Because of this separation, the notion of vproc is both simpler and more general. Although the thief-fiber mechanism is more expensive, the work-first principle suggests that the cost is negligible because of the infrequency of steals.

The Glasgow Haskell Compiler (GHC) is the compiler and runtime system for parallel Haskell. GHC supports fine-grain tasks (called sparks) that are similar to the tasks created by parallel tuples. Sparks are scheduled by a work stealing scheduler. The scheduler implementation is based on the public-access model. GHC offers parallel speedups ranging from 3 to 6.6 on seven cores, but it is unclear how far beyond seven cores GHC system will scale before its GC will become a sequential bottleneck [56].

GHC's GC is a sophisticated parallel generational collector. But its primary disadvantage is its stop-the-world requirement. Each processor has an local allocation area, and when the local allocation area fills up, all processors stop together to perform a parallel collection. In this regard, the GHC collector is similar to the flat-heap GC used in the performance study above.

3.5.3 Imperative parallel-language implementations

Cilk is an extension to the C programming language that supports fork-join parallelism [12]. The work stealing algorithm used by Cilk is similar to WS LP, although WS LP extends work stealing to support lazy promotion and a distributed-memory style protocol for stealing tasks. In my design, I use the common case analysis of work stealing, or work-first principle, that was used for the Cilk-5 implementation [36]. Since Cilk is designed for C with manual memory management, there have been, to my knowledge, no studies carried out on garbage collection performance related to the Cilk scheduler.

Deque overflow is a well-known problem for implementations of work stealing that use the public-access model. Hendler *et al.* [42] and Chase and Lev [20] present two solutions to the overflow problem, though each of their stealing protocols rely on subtle lock-free synchronization algorithms to protect the state of deques. Such complications do not apply to WS LP because it uses a private access model. In WS LP, the deque is protected by temporarily postponing signals while the deque is being resized.

Backtracking-based Load Balancing (BLB) is a distributed-memory implementation of work

stealing [45]. BLB supports a C-like language with manual memory management, and as such, BLB does not address garbage collection.

Like WS LP, BLB uses an implementation of work stealing in which the thief and victim communicate via inter-processor signals. But instead of using deques to maintain ready tasks, as WS LP does, BLB uses a backtracking mechanism. When a thief receives a steal request, the thief pauses and backtracks to find its oldest stealable task. Backtracking is attractive because it obviates the need for deques, but as I discussed in Section 3.3.1, it implies that each steal takes unbounded time.

3.6 Summary

This chapter presented the design and implementation of Manticore's work-stealing scheduler and split-heap memory manager. The memory manager uses a split-heap architecture where each processor has a local heap that is independent from other processors' local heaps. While this architecture provides better locality and reduced garbage collection synchronization, it suffers from the requirement that any heap object that is made visible to other processors must be promoted (*i.e.*, copied) to the global heap. An early analysis shows that this process accounts for almost 50% of the scheduling overhead in our work-stealing implementation.

To address the cost of promotion, I have developed a new work-stealing scheduler based on *lazy promotion*. I have analyzed the performance of our systems and presented results from three representative benchmarks. These results show that lazy promotion is often beneficial and does no harm. The data shows that lazy promotion imposes a lower GC load, largely because less data is promoted to the global heap. I also present experiments comparing Manticore's split-heap implementation to a flat-heap version. These experiments demonstrate the benefits of the split-heap design. In particular, the split-heap design has much better L3 cache behavior, as well as lower GC overhead.

CHAPTER 4

LAZY TREE SPLITTING

The Goldilocks problem In NDP programs, computations are divided into chunks, and chunks of work are spawned in parallel. Those chunks might be defined by subsequences (of arrays, for example, or, in our case, ropes) or iteration spaces (say, k to some $k + n$). The choice of chunk size influences performance crucially. If the chunks are too small, there will be too much overhead in managing them; in extreme cases, the benefits of parallelism will be obliterated. On the other hand, if they are too large, there will not be enough parallelism, and some processors may run out of work. An ideal chunking strategy apportions chunks that are neither too large nor too small, but are, like Goldilocks's third bowl of porridge, "just right." Some different chunking strategies are considered in the sequel.

4.0.1 Fragile chunking strategies

A fragile chunking strategy is prone either to creating an excessive number of tasks or to missing significant opportunities for parallelism. Let us consider two simple strategies, ***T-ary decomposition*** and ***structural decomposition***, and the reasons that they are fragile. In T-ary decomposition, we split the input rope into $T = \min(n, J \times P)$ chunks, where n is the size of the input rope, J is a fixed compile-time constant, and P is the number processors, and spawn a task for each chunk. For example, in Figure 4.1(a), I show the T-ary decomposition version of the rope-map operation.

In computations where all rope elements take the same time to process, such as those performed by regular affine (dense-matrix) scientific codes, the T-ary decomposition will balance the work load evenly across all processors because all chunks will take about the same amount of time. On the other hand, when rope elements correspond to varying amounts of work, performance will be fragile because some processors will get overloaded and others underutilized. Excessive splitting is also a problem. Observe that for i levels of nesting and sufficiently-large ropes, the T-ary

```
fun mapTary J f rp = let
  fun g chunk = fn () => mapSequential f chunk
  val chunks = splitN (rp, J * numProcs ())
  in
    catN (parN (map g chunks))
  end
```

(a) T-ary decomposition

```
fun mapStructural f rp = (case rp
  of Leaf s => mapSequential f rp
   | Cat(l, r) =>
       Cat(| mapStructural f l,
             mapStructural f r |))
```

(b) structural decomposition

Figure 4.1: Two fragile implementations of the rope-map operation.

decomposition creates $(J \times P)^i$ tasks overall, which can be excessive when either i or P get large.

To remedy the imbalance problem, we might try ***structural decomposition***, in which both children of a `Cat` node are processed in parallel and the elements of a `Leaf` node are processed sequentially. I show the structural version of the rope-map operation in Figure 4.1(b). Recall that the maximum size of a leaf is determined by a fixed, compile-time constant called M and that rope-producing operations tend to keep the size of each leaf close to M. But by choosing an $M > 1$, some opportunities for parallelism will always be lost and by choosing $M = 1$, an excessive number of threads may be created, particularly in the case of nested loops.

4.0.2 Eager binary splitting

EBS is a well-known approach that is used by many parallel libraries and languages, including Thread Building Blocks [75, 47] and Cilk++ [51]. In EBS (and, by extension, eager tree splitting (ETS)), we group elements into fixed-size chunks and spawn a task for each chunk. This grouping is determined by the following recursive process. Initially, we group all elements into a single chunk. If the chunk size is less than the stop-splitting threshold (SST), evaluate the elements

```
fun mapETS SST f rp =
  if length rp <= SST then mapSequential f rp
  else let
    val (l, r) = split2 rp
    in
      cat2 (| mapETS SST f l,
              mapETS SST f r |)
    end
```

Figure 4.2: The ETS implementation of the rope-map operation.

sequentially.[1] Otherwise, we create two chunks by dividing the elements in half and recursively apply the same process to the two new chunks. For example, in Figure 4.2, I show the ETS version of the rope-map operation.

EBS has greater flexibility than the T-ary or structural decompositions because EBS allows chunk sizes to be picked manually. But this flexibility is not much of an improvement, because, as is well known [47, 75, 86], finding a satisfactory SST can be difficult. This difficulty is due, in part, to the fact that parallel speedup is very sensitive to SST. I ran an experiment that demonstrates some of the extent of this sensitivity. Figure 4.3 shows, for seven PML benchmarks (see Section 4.2 for benchmark descriptions), *parallel efficiency* as a function of SST. The parallel efficiency is the sixteen-processor speedup divided by sixteen times 100, where the baseline time for the speedup is taken from the sequential evaluation. For example, 100% parallel efficiency represents perfect linear speedup and 6.25% parallel efficiency represents almost no speedup. The results demonstrate that there is no SST that is optimal for every program and furthermore that a poor SST is far from optimal.

The Raytracer benchmark demonstrates, in particular, how fragile ETS can be with respect to nesting and to relatively small ropes. Raytracer loses 80% of its speedup as SST is changed from 32 to 128. The two-dimensional output of the program is a 256×256 rope of ropes, representing the pixels of a square image. When $SST = 128$, Raytracer has just two chunks it can process in parallel: the first 128 rows and the second. We could address this problem by transforming the

1. In TBB, if SST is unspecified, the default is $SST = 1$, whereas Cilk++ only uses $SST = 1$.

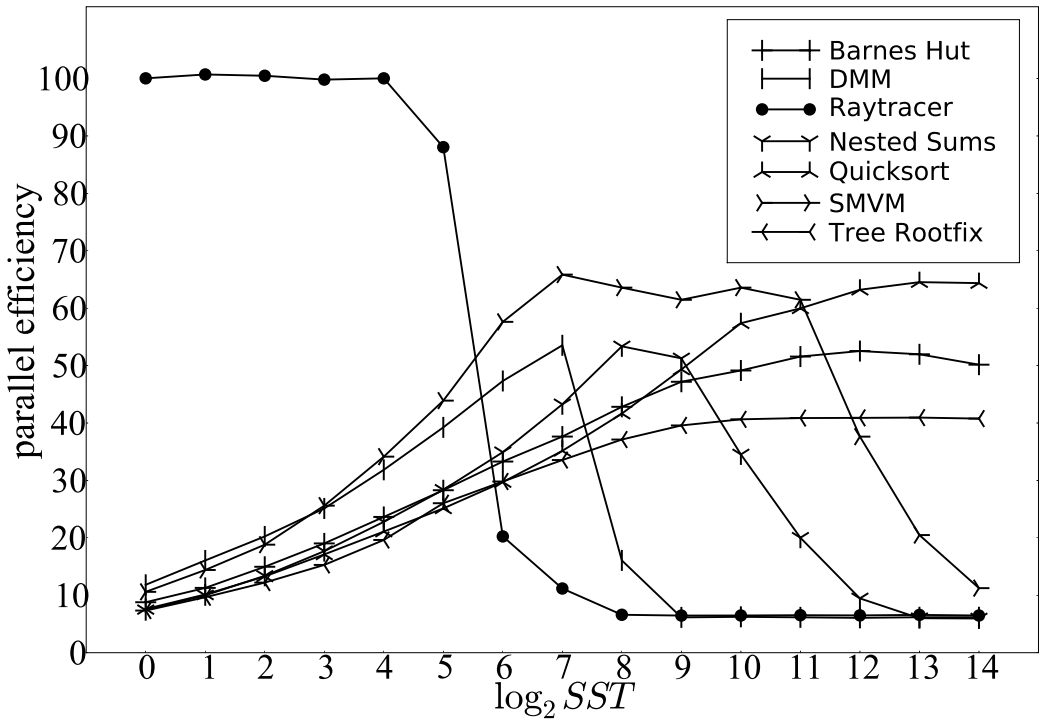


Figure 4.3: Parallel efficiency is sensitive to SST (16 processors).

two-dimensional representation into a single flat rope, but then the clarity of the code would be compromised, as we would have to use index arithmetic to extract any pixel, thereby breaking with the nested-data-parallel programming style.

Recall that task execution times can vary unpredictably. Chunking strategies that are based solely on fixed thresholds, such as EBS and ETS, are bound to be fragile because they rely on accurately predicting execution times. A superior chunking strategy would be able to adapt dynamically to the current state of load balance across processors.

4.0.3 Lazy binary splitting

The LBS strategy of Tzannes, *et al.* [86] is a promising alternative to the other strategies because it has good adaptivity to dynamic load balance. Tzannes, *et al.* show that LBS is capable of

performing as well or better than each configuration of eager binary splitting, and does so without tuning.

LBS is similar to eager binary splitting but with one key difference. In LBS, we base each splitting decision entirely on a dynamic estimation of load balance. Let us consider the main insight behind LBS. We call a processor ***hungry*** if it is idle and ready to take on new work, and ***busy*** otherwise. It is better for a given processor to delay splitting a chunk and to continue processing local iterations while remote processors remain busy. Splitting can only be profitable when a remote processor is hungry.

Although this insight is sound, it is still unclear whether it is useful. A naïve hungry-processor check would require inter-processor communication, and the cost of such a check would hardly be an improvement over the cost of spawning a thread. For now, let us assume that we have a good approximate hungry-processor check

```
val hungryProcs : unit -> bool
```

which returns `true` if there is probably a remote hungry processor and `false` otherwise. Later, we consider how to implement such a check.

LBS works as follows. The scheduler maintains a current chunk c and a pointer i that points at the next iteration in the chunk to process. Initially, the chunk contains all iterations and $i = 0$. To process an iteration i, the scheduler first checks for a remote hungry processor. If the check returns false, then all of the other processors are likely to be busy, and the scheduler greedily executes the body of iteration i. If the check returns true, then some of the other processors are likely to be hungry, and the scheduler splits the chunk in half and spawns a recursive instance to process the second half.

Tzannes, *et al.* [86] show how to implement an efficient and accurate hungry-processor check. Their idea is to derive such a check from the work stealing policy. Recall that, in work stealing, each processor has a deque, which records the set of tasks created by that processor. The hungry-processor check bases its approximation on the size of the local deque. If the deque of a given

processor contains some existing tasks, then these tasks have not yet been stolen, and therefore it is unlikely to be profitable to add to these tasks by splitting the current chunk. On the other hand, if the deque is empty, then it is a strong indication that there is a remote hungry processor, and it is probably worth splitting the current chunk. This heuristic works surprisingly well considering its simplicity. It is cheap because the check itself requires two local memory accesses and a compare instruction, and it provides an accurate estimate of whether splitting is profitable.

Let us consider how LBS behaves with respect to loop nesting. Suppose our computation has the form of a doubly-nested loop, one processor is executing an iteration of the inner loop, and all other processors are hungry. Consequently, the remainder of the inner loop will be split (possibly multiple times, as work is stolen from the busy processor and further split), generating relatively small chunks of work for the other processors. Because the parallelism is fork-join, the only way for the computation to proceed to the next iteration of the outer loop is for all of the work from the inner loop to be completed. At this point, all processors are hungry, except for the one processor that completed the last bit of inner-loop work. This processor has an empty deque; hence, when it starts to execute the next iteration of the outer loop, it will split the remainder of the outer loop.

Because there is one hungry-processor check per loop iteration, and because loops are nested, most hungry-processor checks occur during the processing of the innermost loops. Thus, the general pattern is clear: splits tend to start during inner loops and then move outward quickly.

4.1 Lazy tree splitting for ropes

LTS operations are not as easy to implement as ETS operations, because, during the execution of a given LTS operation, a split can occur while processing *any* rope element. This section presents implementations of five important LTS operations. The technique I use is based on Huet's zipper technique [46] and a new technique we call *splitting a context*. Let us first look in detail at the LTS version of map (`mapLTS`) because its implementation offers a simple survey of the relevant techniques. Then I summarize implementations of the additional operations.

4.1.1 *Implementing* `mapLTS`

Structural recursion, on its own, offers no straightforward way to implement `mapLTS`. Consider the case in which `mapLTS` detects that another processor is hungry. How can `mapLTS` be ready to halve the as-yet-unprocessed part of the rope, keeping in mind that, at the halving moment, the focus might be on a mid-leaf element deeply nested within a number of `Cat` nodes? In a typical structurally recursive traversal (*e.g.*, Figure 4.1(b)), the code has no handle on either the processed portion of the rope or the unprocessed remainder of the rope; it can only see the current substructure. We need to be able to "step through" a traversal in such a way that we can, at any moment, pause the traversal, reconstruct processed results, divide the unprocessed remainder in half, and resume processing at the pause point.

A key piece of our implementation is an internal operation called `mapUntil`. The `mapUntil` operation is capable of pausing its traversal based on a runtime predicate:

```
val mapUntil :
  (unit -> bool) -> ('a -> 'b)
  -> 'a rope
  -> ('a rope * 'b rope, 'b rope) progress
```

The first argument to `mapUntil` is a polling function (*e.g.*, `hungryProcs`); the second argument is the function to be applied to the individual data elements; and the third argument is the input rope. Instead of returning a fully processed rope, `mapUntil` returns a value of type `('a rope * 'b rope, 'b rope) progress`, where the type constructor `progress` is defined as

```
datatype ('a, 'b) progress
  = More of 'a
  | Done of 'b
```

In the result of `mapUntil`, a value `More (u, p)` represents a partially processed rope where `u` is the unprocessed part and `p` is the processed part; a value `Done p` represents a fully processed rope. The evaluation of `mapUntil cond f rp` proceeds by applying `f` to the elements of `rp` from left to right until either `cond ()` returns `true` or the whole rope is processed. Before we consider the implementation of `mapUntil`, we examine how `mapUntil` is used to implement `mapLTS`.

```
fun mapLTS f rp =
  if length rp <= 1 then
    mapSequential f rp
  else (case mapUntil hungryProcs f rp
    of More(u, p) => let
         val (u1, u2) = split2 u
         in
           catN (parN [fn () => balance p,
                       fn () => mapLTS f u1,
                       fn () => mapLTS f u2])
         end
     | Done p => balance p)
```

Figure 4.4: The LTS implementation of the rope-map operation.

The `mapLTS` algorithm, shown in Figure 4.4, starts by checking the length of the input rope. When the rope length is greater than one (the interesting case), the algorithm calls `mapUntil` to start processing elements. If this call returns a partial result (`More (u, p)`), then `mapLTS` splits the unprocessed subrope `u` and schedules the parallel evaluation of the balancing (if necessary) of the processed subrope `p` and the recursive mapping of the halves of the unprocessed subrope `u`. At the join of the parallel computation, the three now processed subropes are concatenated and returned. Note that because this algorithm is recursive, splitting may continue until a single rope element is reached. If the call to `mapUntil` returns a complete result (`Done p`), then `p` is balanced (if necessary) and returned. Balancing `p` (in either the `More` or `Done` cases) may be profitable here because the ropes returned by `mapUntil` may be unbalanced.

It remains to implement the `mapUntil` operation. The crucial property of the `mapUntil` operation is that during the traversal of the input rope, it must maintain sufficient information to, at any moment, pause the traversal and reconstruct both the processed portion of the rope and the unprocessed remainder of the rope. Huet's zipper technique [46] provides the insight necessary to derive a persistent data structure, and functional operations over it, which enable this "pausable" traversal. A zipper is a representation of an aggregate data structure that factors the data structure into a distinguished substructure under focus and a one-hole context; plugging the substructure into

the context yields the original structure. Zippers allow efficient navigation through and modification of a data structure. With a customized zipper representation, some basic navigation operations, and our novel context-splitting technique, we arrive at an elegant implementation of `mapUntil`.

To represent the rope-map traversal, we use a context representation similar to Huet's single-hole contexts [46], but with different types of elements on either side of the hole, as in McBride's contexts [57]. Thus, our context representation is defined as

```
datatype ('a, 'b) ctx
  = Top
  | CatL of 'a rope * ('a, 'b) ctx
  | CatR of 'b rope * ('a, 'b) ctx
```

where `Top` represents an empty context, `CatL(r, c)` represents the context surrounding the left branch of a `Cat` node where `r` is the right branch and `c` is the context surrounding the `Cat` node, and `CatR(l, c)` represents the context surrounding the right branch of a `Cat` node where `l` is the left branch and `c` is the context surrounding the `Cat` node. Note that, for a rope-map traversal, all subropes to the left of the context's hole are processed (`'b rope`) and all subropes to the right of the context's hole are unprocessed (`'a rope`).

The implementation of `mapUntil` will require a number of operations to manipulate a context. The `leftmost (rp, c) ⇒ (s', c')` operation plugs the (unprocessed) rope `rp` into the context `c`, then navigates to the leftmost leaf of `rp`, returning the sequence `s'` at that leaf and the context `c'` surrounding that leaf:

```
val leftmost : 'a rope * ('a, 'b) ctx
               -> 'a seq * ('a, 'b) ctx

fun leftmost (rp, c) = (case rp
  of Leaf s => (s, c)
   | Cat(l, r) => leftmost (l, CatL(r, c)))
```

The `start` operation simply specializes `leftmost` to the case of the whole unprocessed rope in the empty context:

```
val start : 'a rope -> 'a seq * ('a, 'b) ctx

fun start rp = leftmost (rp, Top)
```

It is used to initialize the `mapUntil` traversal. See Figure 4.5(a) for a pictorial example of this operation. In the figure, a right-facing leaf node denotes a processed node and facing the left an unprocessed node. The `next (rp, c)` operation plugs the (processed) rope `rp` into the context `c`, then attempts to navigate to the next unprocessed leaf.

```
val next :
  'b rope * ('a, 'b) ctx
  -> ('a seq * ('a, 'b) ctx, 'b rope) progress

fun next (rp, c) = (case c
  of Top => Done rp
   | CatL(r, c') =>
       More(leftmost (r, CatR(rp, c')))
   | CatR(l, c') => next (cat2 (l, rp), c'))
```

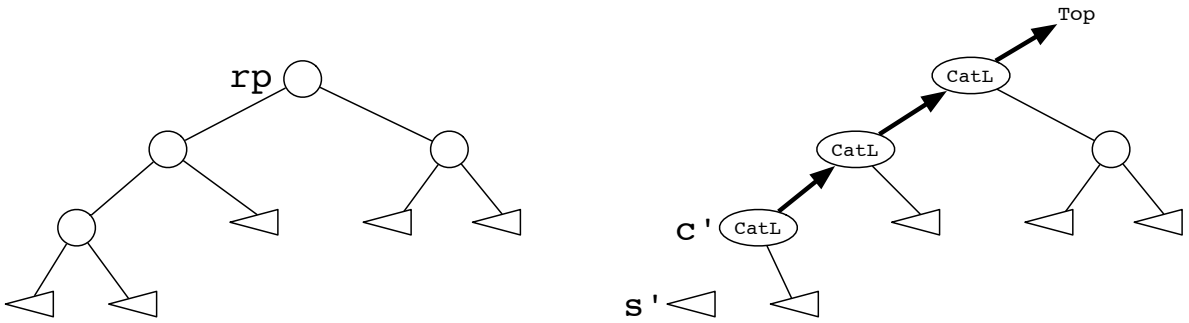
This navigation can either succeed, in which case `next` returns `More (s', c')` (see Figure 4.6(c)), where `s'` is the sequence at the next leaf and `c'` is the context surrounding that leaf, or fail, in which case `next` returns `Done rp'` (see Figure 4.5(b)), where `rp'` is the whole processed rope.

The final operation on contexts is an operation to split a context into a pair of ropes — the unprocessed subrope that occurs to the right of the hole and the processed subrope that occurs to the left of the hole. It is convenient for the `splitCtx` operation to additionally take an unprocessed rope and a processed rope meant to fill the hole, which are incorporated into the result ropes (see Figure 4.6(d)):

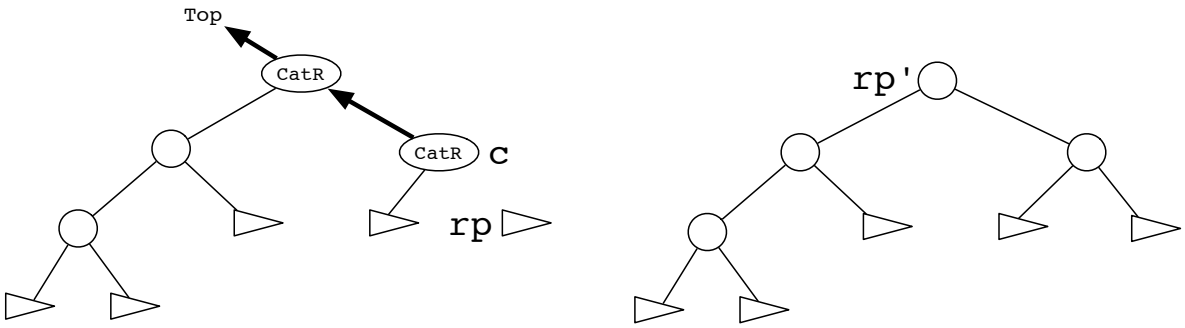
```
val splitCtx : 'a rope * 'b rope * ('a, 'b) ctx
               -> 'a rope * 'b rope

fun splitCtx (u, p, c) = (case c
  of Top => (u, p)
   | CatL(u', c') =>
       splitCtx (cat2 (u, u'), p, c')
   | CatR(p', c') =>
       splitCtx (u, cat2 (p', p), c'))
```

With these context operations, we give the implementation of `mapUntil` in Figure 4.7. The traversal of `mapUntil` is performed by the function `lp`. The argument `s` represents the sequence of the leftmost unprocessed leaf of the rope and the argument `c` represents the context surrounding that leaf.

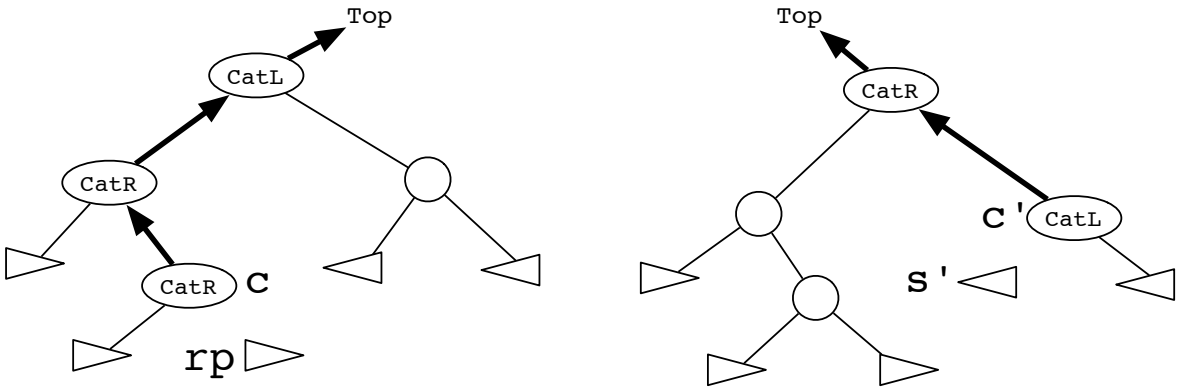


(a) `start rp ⇒ (s′,c′)`

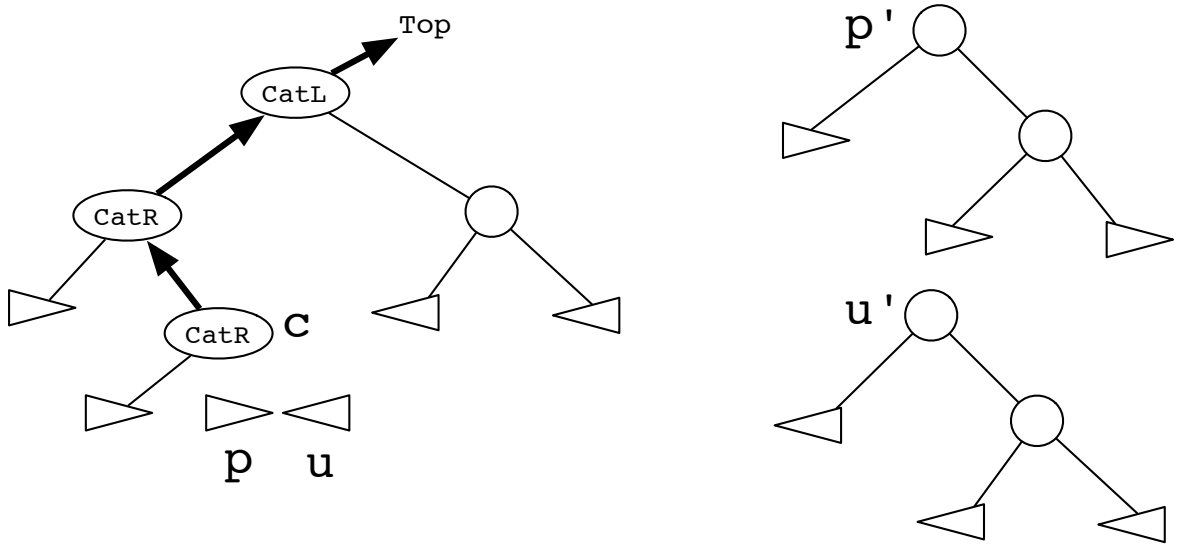


(b) `next (rp,c) ⇒ Done rp′`

Figure 4.5: Operations on contexts



(c) `next (rp,c) ⇒ More (s',c')`



(d) `splitCtx (u,p,c) ⇒ (u',p')`

Figure 4.6: Operations on contexts

```
fun mapUntil cond f rp = let
  fun lp (s, c) = (case mapSeqUntil cond f s
    of More(us, ps) =>
         More(splitCtx (Leaf us, Leaf ps, c))
     | Done ps => (case next (Leaf ps, c)
         of Done p' => Done p'
          | More(s', c') => lp (s', c')))
  in
    lp (start rp)
  end
```

Figure 4.7: The `mapUntil` operation.

The processing of the sequence is performed by `mapSeqUntil`, a function with similar behavior to `mapUntil`, but implemented over linear sequences. It is `mapSeqUntil` that actually calls `cond` and applies the function `f`. Note that `mapSeqUntil` must also maintain a context with processed elements to the left and unprocessed elements to the right, but doing so is trivial for a linear sequence. (Recall the standard accumulate-with-reverse implementation of `map` for lists.)

If `mapSeqUntil` returns a partial result (`More (us, ps)`), then the traversal pauses and returns its intermediate results by splitting its context. (This pause and return gives `mapLTS` the opportunity to split the unprocessed elements and push the parallel mapping of these halves of the unprocessed elements onto the work-stealing deque.) If `mapSeqUntil` returns a complete result (`Done ps`), then the traversal plugs the context with this completed leaf sequence and attempts to navigate to the next unprocessed leaf by calling `next (Leaf ps, c)`. If `next` returns `Done p'`, then the rope traversal is complete and the whole processed rope is returned. Otherwise, `next` returns `More (s', c')` and the traversal loops to process the next leaf sequence (`s'`) with the new context (`c'`).

4.1.2 Implementing other operations

The implementation of `filterLTS` is very similar to that of `mapLTS`. Indeed, `filterLTS` uses the same context representation and operations as `mapLTS`, simply instantiated with unprocessed

and processed elements having the same type:

```
val filterLTS : ('a -> bool)
                -> 'a rope -> 'a rope

type 'a filter_ctx = ('a, 'a) ctx
```

As with `mapLTS`, where the mapping operation was applied by the `mapSeqUntil` operation, the actual filtering of elements is performed by the `filterSeqUntil` operation.

The `reduceLTS` operation takes an associative operator and its zero and a rope and returns the rope's reduction under the operator.

```
val reduceLTS : ('a * 'a -> 'a) -> 'a
                -> 'a rope -> 'a
```

Thus, the `reduceLTS` operation may be seen as a generalized sum operation. The implementation of `reduceLTS` is again similar to that of `mapLTS`, but uses a simpler context:

```
datatype 'a reduce_ctx
  = Top
  | CatL of 'a rope * 'a reduce_ctx
  | CatR of 'a * 'a reduce_ctx
```

where `CatR (z, c)` represents the context surrounding the right branch of a `Cat` node in which `z` is the *reduction* of the left branch and `c` is the context surrounding the reduction of the `Cat` node.

The `scanLTS` operation, also known as *prefix sums*, is an important building block of a data-parallel programming language. Like `reduceLTS`, the `scanLTS` operation takes an associative operator and its zero and a rope and returns a rope of the reductions of the prefixes of the input rope.

```
val scanLTS : ('a * 'a -> 'a) -> 'a
              -> 'a rope -> 'a
```

For example,

```
scanLTS (op +) 0 (Cat (Leaf [1, 2], Leaf [3, 4]))
  ⇒ Cat (Leaf [1, 3], Leaf [6, 10])
```

In a survey on prefix sums, Blelloch describes classes of important parallel algorithms that use this operation and gives an efficient parallel implementation of prefix sums [6], on which our

implementation of `scanLTS` is based. The algorithm takes two passes over the rope. The first performs a parallel reduction over the input rope, constructing an intermediate rope in which partial reduction results are recorded at each internal node. The second pass builds the result rope in parallel by processing the intermediate rope. The efficiency of this second pass is derived from having constant-time access to the cached sums while it builds the result.

The result of this first pass is called a *monoid-cached tree* [44], specialized in the current case to *monoid-cached rope*. In a monoid-cached rope,

```
datatype 'a crope
  = CLeaf of 'a * 'a seq
  | CCat of 'a * 'a crope * 'a crope
```

each internal node caches the reduction of its children nodes. For example, supposing the scanning operator is integer addition, one such monoid-cached rope is

```
CCat (10, CLeaf (3, [1, 2]), CLeaf (7, [3, 4]))
```

Our implementation of Blelloch's algorithm is again similar to that of `mapLTS`, except that we use a context in which there are `rope`s to the right of the hole and `cached_rope`s to the left of the hole. Aside from some minor complexity involving the propagation of partial sums, the operations on this context are similar to those on the context used by `mapLTS`.

The `map2LTS` operation maps a binary function over a pair of ropes (of the same length).

```
val map2LTS : ('a * 'b -> 'c)
              -> 'a rope * 'b rope -> 'b rope
```

For example, the pointwise addition of the ropes `rp1` and `rp2` can be implemented as

```
map2LTS (op +) (rp1, rp2)
```

Note that `rp1` and `rp2` may have completely different branching structures, which would complicate any structural-recursive implementation. The zipper technique provides a clean alternative: we maintain a pair of contexts and advance them together in lock step during execution. The result rope is accumulated in one of these contexts.

Contexts and partial results nicely handle the processing of leaves of unequal length. When the `map2SeqUntil` function is applied to two leaves of unequal length, it simply returns a partial result that includes the remaining elements from the longer sequence. The `map2Until` function need only step the context of the shorter linear sequence to find the next leaf with which to resume the `map2SeqUntil` processing. Note that we do need to distinguish `map2SeqUntil` returning with a partial result due to the polling function, in which case `map2Until` should also return a partial result (signaling that a task should be pushed to the work-stealing deque), from `map2SeqUntil` returning with a partial result do to exhausting one of the leaves, in which case `map2Until` should not return a partial result. The implementation straightforwardly extends to maps of arbitrary arity.

4.1.3 Rebalancing

In our implementation, there are two circumstances in which we need to do balancing. The first is in `filterLTS`, because the filtering predicate may drop elements at arbitrary positions inside the rope. The second is in operations like `mapLTS`, because such operations may split at an arbitrary rope leaf.

4.2 Evaluation

I have already presented data that shows the performance of ETS is sensitive to the SST parameter. In this section, I present the results of additional experiments that demonstrate that LTS performs as well or better than ETS over a range of benchmarks. Furthermore, it demonstrates scalable performance without any application-specific tuning.

4.2.1 Benchmarks

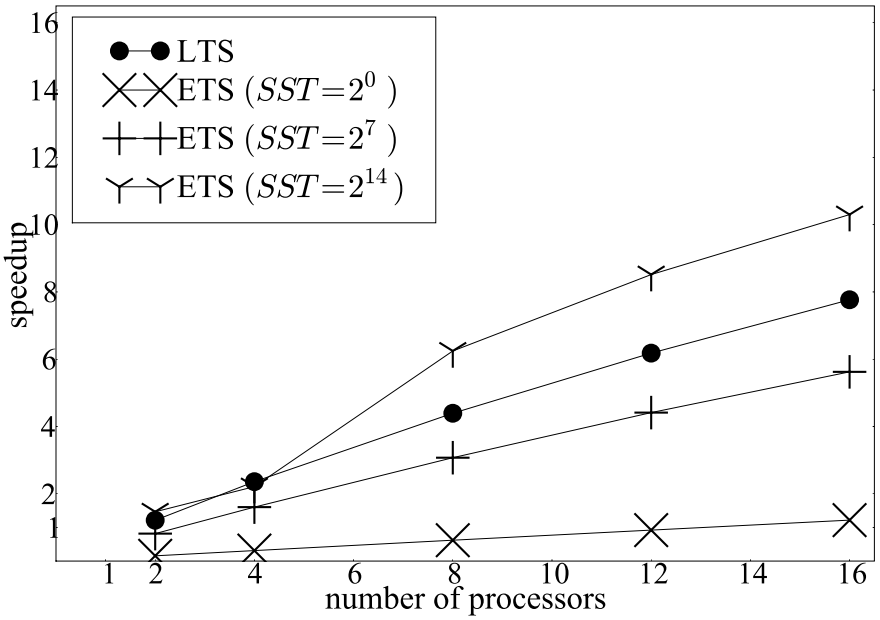
For this empirical evaluation, I use six benchmark programs from the benchmark suite and one synthetic benchmark. All benchmarks use the same max leaf size ($M = 256$), which provides the best average performance over the programs in the benchmark suite.

4.2.2 *Lazy* vs. *eager tree splitting*

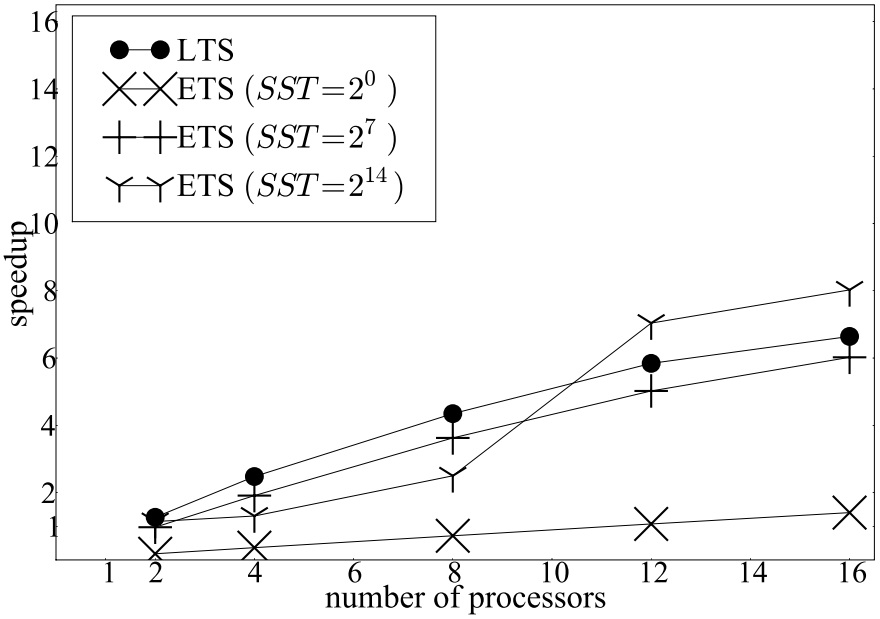
My most important experimental results come from a comparing LTS to ETS side by side. Figures 4.8, 4.9, and 4.10 show speedup curves for all seven of the benchmarks. For each graph, I plot the speedup curve (over sequential PML performance) of ETS with SST values of 1, 128, and 16384 and of LTS. I have argued that one of the main advantages of LTS over ETS is that LTS does not require tuning for each benchmark. These graphs show that LTS is better than most configurations of ETS, and that the downside of picking a poor SST value for ETS can be quite severe (*e.g.*, Figure 4.8*(b)* with an SST of 128). They also show that not only is the best choice of SST for ETS dependent on the particular benchmark, but in some cases it is also dependent on the number of processors (*e.g.*, Figure 4.8*(a)* and Figure 4.11 textit(f)).

With an optimal pick of SST value, ETS can outperform LTS, because of lower overhead. In my experiments, I collected data for every $SST \in \{2^i \mid 0 \leq i \leq 14\}$ and compared the best ETS performance against LTS for each benchmark on 16 processors. I found that even with always choosing the best SST value for the given benchmark and number of processors, ETS was never more than 20% faster than LTS. In practice, it is impossible to make such precise and specialized tuning decisions *a priori*, since workloads and compute resources are unpredictable. Therefore, I believe that LTS provides a much better solution to the Goldilocks problem.

To address the question of why optimal ETS is faster than LTS, I collected profiling data for the benchmarks. This data shows that the per-processor utilization for ETS is never more than 3% greater than that of LTS, which is almost within the 2% of sampling error mentioned above. Thus, I believe that the performance gap has to do with increased overhead, rather than poorer

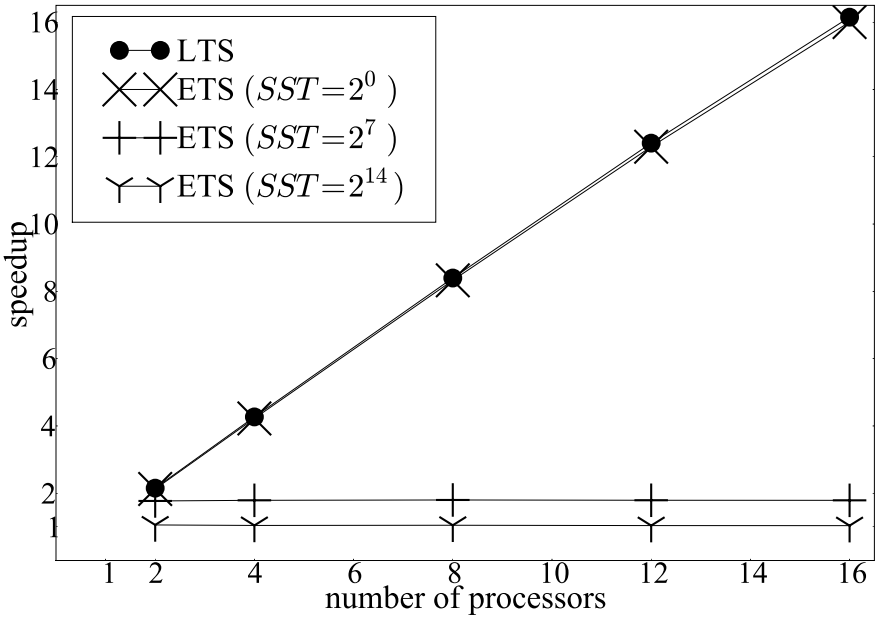


(a) Quicksort

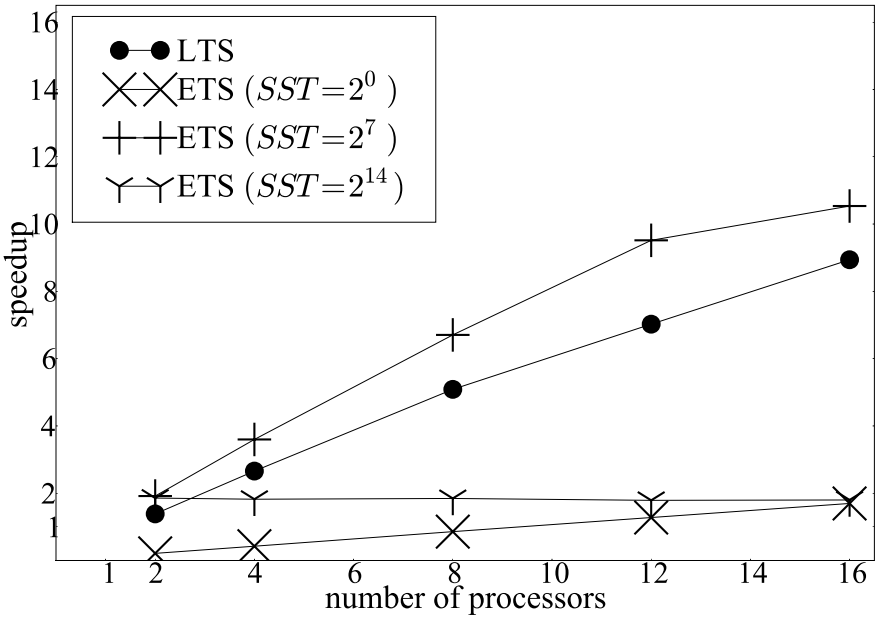


(b) Barnes-Hut

Figure 4.8: Comparison of lazy tree splitting (LTS) to eager tree splitting with ETS.

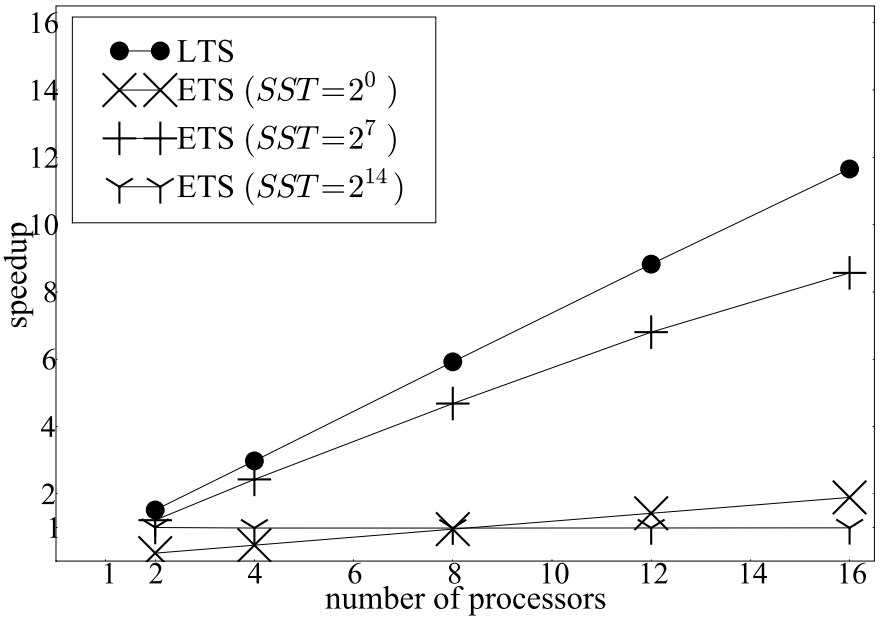


(c) Raytracer

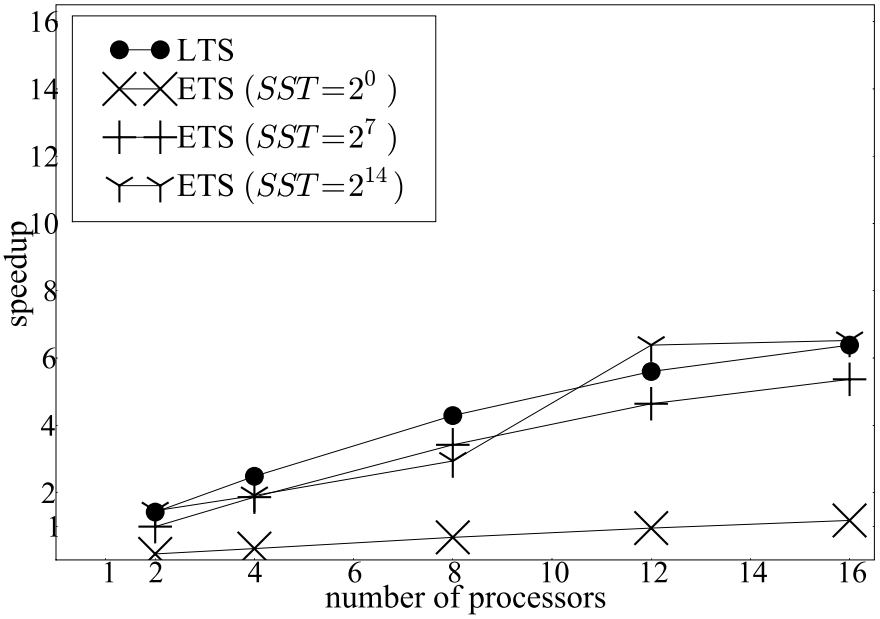


(d) SMVM

Figure 4.9: Comparison of lazy tree splitting (LTS) to eager tree splitting with ETS.

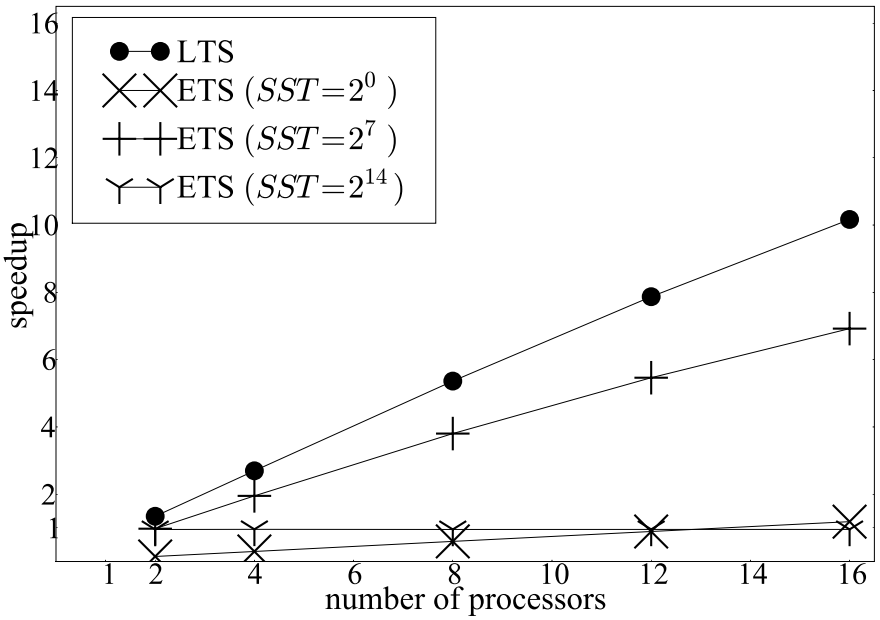


(c) DMM



(d) Tree Rootfix

Figure 4.10: Comparison of lazy tree splitting (LTS) to eager tree splitting with ETS.



(f) Nested Sums

Figure 4.11: Comparison of lazy tree splitting (LTS) to eager tree splitting with ETS.

scheduling. I also considered the possibility that rebalancing was the source of the performance gap, but my profiling data showed that the total time spent rebalancing is an insignificant fraction of the the total program's run time. Thus, I believe that the main source of this performance gap is the overhead of using a zipper to implement LTS (this point is discussed in further detail below).

In Table 4.1, I present performance measurements for the seven benchmarks run in several different sequential configurations, as well as on 16 processors.

The first column of data presents timing results for MLton. MLton is a sequential whole-program optimizing compiler for Standard ML [60, 90], which is the "gold standard" for ML performance. The second data column gives the baseline performance of the natural sequential PML versions of the benchmarks (*i.e.*, parallel operations are replaced with their natural sequential equivalents). Manticore I about a factor of two slower than MLton for all of the benchmarks except DMM and Nested Sums. Considering MLton's suite of aggressive optimizations and maturity, the sequential performance of PML is encouraging. The slower performance here can be attributed to

		PML			
Benchmark	MLton	Seq.	LTS	Par. 16	Speedup
Quicksort	1.36s	3.93s	5.61s	0.51s	7.77
Barnes Hut	7.71s	14.63s	20.62s	2.20s	6.64
Raytracer	2.29s	3.58s	3.54s	0.22s	16.15
SMVM	0.07s	0.15s	0.19s	0.02s	8.94
DMM	0.84s	3.49s	4.12s	0.30s	11.65
Tree Rootfix	3.79s	8.43s	10.44s	1.32s	6.38
Nested Sums	0.21s	1.46s	1.80s	0.14s	10.17

Table 4.1: The performance of LTS for seven benchmarks.

at least two factors. First, the MLton compiler monomorphizes the program and then aggressively flattens the resulting monomorphic data representations. Since ropes are polymorphic, Manticore used a boxed representation for the array elements, instead of an unboxed representation. Second, my profiling shows higher GC overheads in Manticore. These issues can be addressed by improving Manticore's sequential performance.

The third data column reports the execution time of the benchmarks using the LTS runtime mechanisms (*e.g.*, zippers), but without parallelism. By comparing these numbers with the natural sequential measurements, we get a measure of the overhead of the LTS mechanisms. On average, the LTS version is about 24% slower. I have determined through profiling that the main source of this overhead is *not* from calls to `hungryProcs` or rebalancing. Instead, the primary source of the overhead comes from maintaining the traversal state via the zipper context. Such a strategy is less efficient than implicitly maintaining the state via the run-time call stack in a natural structural recursion.[2]

The last two columns report the parallel execution time and speedup on sixteen processors. Overall, the speedups are quite good. The super-linear speedup of the Raytracer is explained by a reduction in GC load per processor. This reduction happened because each processor has its own local heap, so the total size of the available heap increases with the number of processors.

2. Manticore uses heap-allocated continuations to represent the call stack [2, 31].

The Barnes-Hut benchmark achieves a modest speedup, which I believe stems from a limit on the amount of parallelism in the program. This hypothesis is supported by the fact that increasing the problem size to 400,000 particles improves the speedup results. Because matrix-matrix multiplication exposes a large amount of average parallelism, on the order of 150,000 tasks, we expect DMM to have a near-perfect linear speedup. However, DMM is 25-27% slower than a perfect speedup. I attribute the slower performance on DMM to an increase in overheads incurred by the LTS Zipper traversal. Observe that the sequential version of DMM that uses a LTS is 20% slower than a similar version that does not.

There is still a question of whether my technique trades one tuning parameter (SST) for another, the max leaf size (M). I address this concern in two ways. First, observe that even if performance is sensitive to M, this problem is specific to ropes, but neither ETS nor LTS. Second, consider Figure 4.12 which shows, for each of the benchmark programs, the parallel efficiency as a function of M (the parallel efficiency has the same meaning as it does in Figure 4.3). The results show all benchmarks performing well for $M \in \{512, 1024, 2048\}$. One concern is DMM, which is sensitive to M because it does many random access operations on its two input ropes. One can reduce this sensitivity by using an alternative rope representation that provides more efficient random access.

4.3 Related work

Adaptive parallel loop scheduling The original work on lazy binary splitting presents a dynamic scheduling approach for parallel `do-all` loops [86]. Their work addresses splitting ranges of indices, whereas ours addresses splitting trees where tree nodes are represented as records allocated on the heap.

In the original LBS work, they use a *profitable parallelism threshold* (PPT) to reduce the number of hungry-processor checks. The PPT is an integer which determines how many iterations a given loop can process before a doing hungry-processor check. Our performance study has

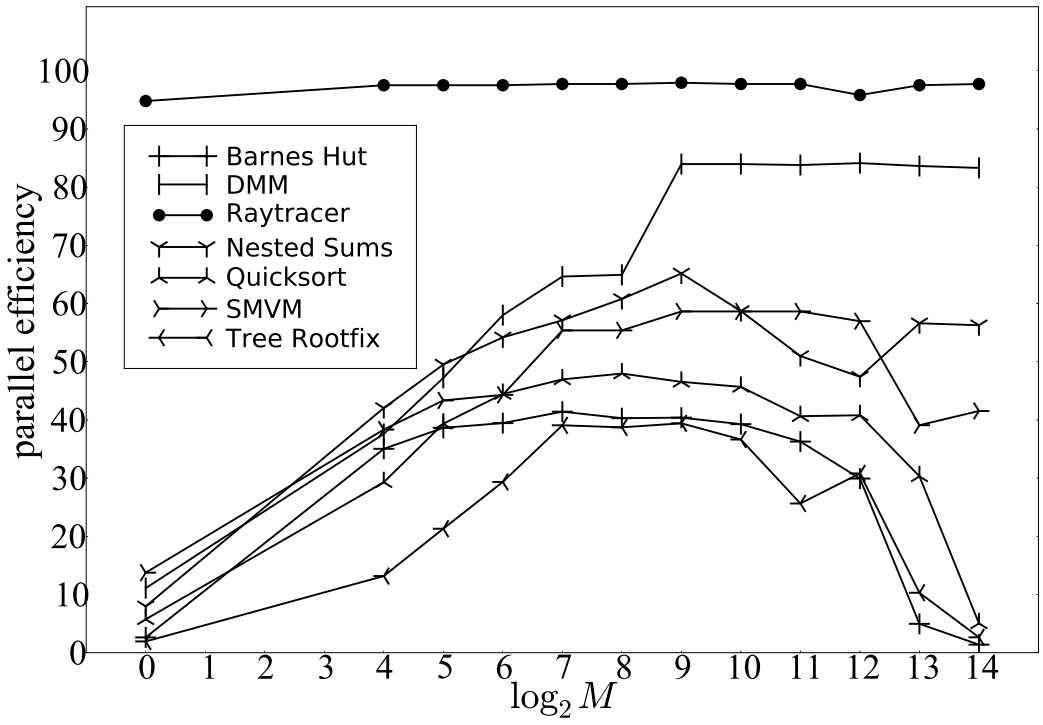


Figure 4.12: The effect of varying max leaf size M (16 processors)

$PPT = 1$ (*i.e.*, one hungry-processor check per iteration) because we have not implemented the necessary compiler mechanisms to do otherwise.

Robison *et al.* propose a variant of EBS called ***auto partitioning*** [75], which offers good performance for many programs and does not require tuning.[3] Auto partitioning derives some limited adaptivity by employing the heuristic that when a task detects it has been migrated it splits its chunk into at least some fixed number of subchunks. The assumption is that if a steal occurs, there are probably other processors that need work, and it is worthwhile to split a chunk further. As discussed by Tzannes, *et al.* [86], auto partitioning has two limitations. First, for i levels of loop nesting, P processors, and a small, constant parameter K, it creates $(K \times P)^i$ chunks, which is excessive if the number of processors is large. Second, although it has some limited adaptivity, auto

3. Auto partitioning is currently the default chunking strategy of TBB [47].

partitioning lacks performance portability with respect to the context of the loop, which limits its effectiveness for scheduling programs written in the liberal loop-nesting style of an NDP language.

Granularity control Early work by Loidl and Hammond in the context of Haskell compared three strategies for deciding whether to create a thread for parallel work or continue in sequence [55]. In simulation, they found that using a simple cut-off generated more speedup than more complicated strategies that dynamically determine whether to create a thread and which thread to run based on a priority associated with the function to run. This cut-off is a value based on a granularity estimation function provided to the parallel primitives. They found, as we did, that speedup was highly dependent upon the cut-off value. Their approach differs from ours in that the cut-off value is statically provided to the runtime; they require a function that can report a granularity metric of the work to perform based on the function being called and the data computed upon. Notably, their work handles any divide-and-conquer algorithm, whereas our solution specifically addresses parallel map operations.

Tick and Zhong presented an approach using compile-time granularity analysis in concurrent logic programs [83]. Their compiler creates a call graph,[4] collapses all strongly-connected components (mutually-recursive functions), and then walks up the collapsed graph creating recurrence equations representing cost estimates. These recurrence questions are solved at compile time and used at run time for cost estimation of functions based on their dynamic inputs. This work does not discuss how these cost metrics are integrated into their scheduler, but does provide an 85–91% accurate estimator of runtime costs for arbitrary functions across their suite of benchmarks. Their static analysis takes advantage of logic programming language features, but demonstrates a potentially more effective approach to determining a satisfactory PPT.

Data parallelism NESL is a nested data-parallel dialect of ML [8]. The NESL compiler uses a program transformation called *flattening*, which transforms nested parallelism into a form of

4. This language is not higher-order, which greatly simplifies the construction of the call graph.

data parallelism that maps well onto SIMD architectures. Note that SIMD operations typically require arrays elements to have a contiguous layout in memory. Flattened code maps well onto SIMD architectures because the elements of flattened arrays are readily stored in adjacent memory locations. In contrast, LTS is a dynamic technique that has the goal of scheduling nested parallelism effectively on MIMD architectures. A flattened program may still use LBS (or LTS) to schedule the execution of array operations on MIMD architectures, so in that sense, flattening and LTS are orthogonal.

There is, of yet, no direct comparison between an NDP implementation based on LTS and an implementation based on flattening. One major difference is that LTS uses a tree representation whereas flattening uses contiguous arrays. As such, the LTS representation has two major disadvantages. First, tree random access is costlier, for a rope it is $O(\log n)$ time, where n is the length of a given rope. Second, there is a large constant factor overhead imposed by maintaining tree nodes. One way to reduce these costs is to use a "bushy" representation that is similar to ropes but where the branching factor is greater than two and child pointers are stored in contiguous arrays.

The NESL backend written by Chatterjee [21] and Data Parallel Haskell [18] performs fusion of parallel operations in order to increase granularity. We do not currently implement such transformations. While fusion reduces overall work for data-parallel operations, it reduces the work per element but does not affect the coarsening of the iterations within a data-parallel operation. Such fusion techniques are orthogonal to LTS.

Narlikar and Blelloch present a parallel depth-first (PDF) scheduler that is designed to minimize space usage [65]. Later work by Greiner and Blelloch on proposes an implementation of NDP based on the PDF scheduler [10]. The PDF schedule is a greedy schedule that is based on the depth-first traversal of the parallel execution graph. The PDF schedule is as close to the sequential schedule as possible in the sense that the scheduler only ever goes ahead of the sequential schedule when the scheduler is limited by data dependencies. In contrast, the work stealing approach used by LTS has each processor doing an independent depth-first traversal of that processor's own

portion of the parallel execution graph.

The work on space efficient scheduling does not address the issue of building an automatic chunking strategy, which is the main contribution of LTS. Narlikar and Blelloch coarsen loops manually in order to obtain scalable parallel performance in their performance study. LTS finds good chunk sizes automatically, without programmer assistance.

Ct is an NDP extension to C++ [38]. So *et al.* describe a fusion technique for Ct that is similar to the fusion technique of DPH [81]. The fusion technique used by Ct is orthogonal to LTS for the same reasons as for the fusion technique of DPH. The work on Ct does not directly address the issue of building an automatic chunking strategy, which is the main contribution of LTS.

GpH GpH introduced the notion of an "evaluation strategy," [84] which is a part of a program that is dedicated to controlling some aspects of parallel execution. Strategies have been used to implement eager-splitting-like chunking for parallel computations. We believe that a mechanism like an evaluation strategy could be used to build a clean implementation of lazy tree splitting in a lazy functional language.

Cilk Cilk is a parallel dialect of the C language extended with linguistic constructs for expressing fork-join parallelism [36]. Cilk is designed for parallel function calls but not loops, whereas our approach addresses both.

4.4 Discussion

The main idea of lazy splitting is to maintain some extra information so it is always possible to spawn off half of the remaining work. This paper presents an instantiation of this idea for operations that produce and consume ropes. Although the main idea has potential to be adapted to a larger class of divide-and-conquer programs, I believe at least three substantial challenges must be met before this goal can be achieved. The first challenge is to support other tree representations,

such as, for example, red-black trees. Specifically, one must derive efficient traversal patterns that preserve the invariants of such structures. Second, LTS programs involve zippers, which are an implementation detail. Are there general techniques to derive LTS specifications automatically from more natural specifications? For example, is there a mechanical process for deriving LTS programs (*e.g.*, `mapLTS`) from structural-recursive programs (*e.g.*, `mapStructural`)? One possible approach is to use a static analysis to identify divide-and-conquer recursive functions, then apply a program transformation to generate analogous lazy-splitting versions. Third, there is a need for general techniques to aggregate work for small problem sizes (rope leaves effectively provide this mechanism in the system described here). Failure to provide such techniques will result in excessive overhead and limited scalability.

The splitting strategy used by LBS and my LTS can cause unnecessary splitting. To understand why, observe that splitting is prone to start at the innermost loops and then work its way to the outer loops, as discussed at the end of Section 4.0.3. Having the thief worker split the *outermost* loops is more efficient because the outer iterations usually contain the most work.

The implementation here uses innermost splitting for two reasons. First, to support outermost splitting would involve special support from the language implementation, as splitting the outermost loop would involve modifying a part of the whole continuation, not just a part of the continuation of the current loop. Second, in the empirical study, for each benchmark, I observed that total number of splits stayed under the low hundreds. Since, steals are extremely fast in the test machine, having a few extra steals made little difference. I expect that an implementation based on outermost stealing would be superior for larger machines.

4.5 Summary

I have described the implementation of NDP features in the Manticore system. I have also presented a new technique for parallel decomposition, lazy tree splitting, inspired by the lazy binary splitting technique for parallel loops. I presented an efficient implementation of LTS over ropes,

making novel use of the zipper technique to enable the necessary traversals. My techniques can be readily adapted to tree data structures other than ropes and is not limited to functional languages. A work-stealing thread scheduler is the only special requirement of my technique.

LTS compares favorably to ETS, requiring no application-specific or machine-specific tuning. For any given benchmark, LTS outperforms most or all configurations of ETS, and is, at worst, only 20% slower than the optimally tuned ETS configuration. Since, in general, optimal tuning of ETS for arbitrary programs and computational resources is not possible, I believe that LTS is a superior implementation technique. The ability of LTS to enable good parallel performance without requiring application-specific tuning is very promising.

CHAPTER 5

IMPLEMENTING WORK STEALING IN MANTICORE

This chapter discusses several parts of Manticore that are important for work stealing, including task cancellation, the BOM implementation of WS LP's scheduler loop , and the vproc-interrupt mechanism.

5.1 Task cancellation

Task cancellation is a mechanism that provides a crucial optimization: when a speculative task is known to be unecessary for the rest of the computation, that task can be terminated in order to free up system resources. In PML, task cancellation is used by parallel tuples as a means to clean up after an exception is raised, and it is used by the speculative constructs **`pval`** and **`pcase`** [33, 34].

Task cancellation involves a subtle synchronization protocol. To cancel a given task, that task must be terminated and an all of its descendants in the spawning tree must be canceled as well. At the time of cancellation, these descendants are spread across vprocs, and can themselves be creating new tasks. Arriving at a correct protocol is challenging.

This chapter presents the design for the task-cancellation mechanism used by Manticore. The mechanism that I propose is implemented entirely in terms of the runtime primitives presented in Chapter 5. The design allows for cancellation to be integrated into an existing scheduling policy with a modest programming effort. The implementation consists of a single cancellation library that is shared by all task schedulers.

This design makes a step toward a larger goal: to structure the Manticore scheduling system as a collection of small components. From these small components, the system can provide a wide variety of policies and support various forms of implicit threads.

The rest of this chapter is as follows. First, I present the design of the mechanism. Then, I describe the implementation of the "parallel-or" operation, which uses cancellation to implement

```
datatype cancelable = CANCELABLE of {
    canceled : ![bool],
    running  : ![vproc option],
    children : ![cancelable list],
    parent   : cancelable option
  }
```

Figure 5.1: The type of `cancelable`.

a powerful form of speculative parallelism. Then I evaluate the design through a few experiments.

5.1.1 The design

Let us build a cancellation mechanism for fibers, the most basic form of process in Manticore. The ***cancelable tree*** is a tree that records parent-to-child relationships among executing fibers. The scheduler uses this tree to locate at runtime subtrees of a computation that are executing on remote vprocs.

A `cancelable` is a channel by which cancellation signals are communicated to a fiber. Each channel is paired with exactly one fiber at runtime. Figure 5.1 shows the type of the channel. The channel has four states: running and not canceled, running and canceled, terminating and not canceled, and terminating and canceled. The field `running` tracks on which vproc the associated fiber is running (it is set to `NONE` if the fiber is not running). In addition, the channel maintains all the node information for the cancelable tree. The parent node records the parent of the cancelable or a nil value if the node is the root node. The child list tracks all the nodes spawned by the fiber corresponding to the channel.

Figure 5.2 shows the interface for cancelable fibers.

- `makeCancelable ()` returns a new cancelable.
- `cancel` c cancels the fiber associated with cancelable c. This operation is synchronous – it waits for its canceled processes and their children to terminate if they are running.

```
val makeCancelable : unit -> cancelable
val cancel : cancelable -> unit
val cancelWrapFiber : (cancelable * fiber) -> fiber
val cancelWrapFun : (cancelable * (unit -> unit))
                                    -> (unit -> unit)
```

Figure 5.2: Operations over cancelables.

- `cancelWrapFiber (`c`, `k`)` returns a new cancelable version of fiber k that has the same behavior as k.

- `cancelWrapFun (`c`, `f`)` returns a new cancelable version of function f that has the same behavior as f.

Below is an example of cancelable fibers. The program initiates a long-lived parallel computation and then cancels the computation, along with any fibers that it spawned in the meantime.

```
let
val k = fiber (fn () => largeComputation ())
val c = makeCancelable ()
val cFn = cancelWrapFiber (c, k)
in
  enqOnVP (vproc, fiber cFn, getFLS ());
  cancel c
end
```

Let us turn to the implementation of `cancelWrapFiber`, which is shown in Figure 5.3.[1] The operation uses the scheduler action, `wrapper`, to sit on the scheduler-action stack while the fiber executes and poll for cancellation.

- The `setInactive` function moves focus up one level in the cancelable tree and marks the cancelable as terminating. Canceling a terminating fiber is a no-op.

- The `setActive` function sets the current cancelable to the given cancelable and marks the fiber as active.

1. Neal Glew identified a race condition in the original implementation. This original implementation appeared in ICFP'08 [32]. The implementation reported here uses the fix that he suggested.

```
fun cancelWrapFiber (c, k) = let
  val CANCELABLE{canceled, ...} = c
  fun terminate () = (setInactive c; stop ())
  fun dispatch (wrapper, k) = (
    setActive c;
    if (#0(promote canceled)) then
        terminate ()
    else
        run (wrapper, k))
  cont wrapper (sgn) = (case sgn
    of STOP => terminate ()
     | PREEMPT k =>  (
       setInactive c;
       atomicYield ();
       dispatch (wrapper, k) )
    (* end case *))
  in
    fn () => (maskPreemption (); dispatch (wrapper, k))
  end
```

Figure 5.3: Cancel wrapper.

The `cancel` operator marks a channel as cancelled and waits the fiber associated with the channel to terminate. The behavior of `cancel` c is as follows:

- The calling vproc marks c as canceled.
- The calling vproc preempts the host vproc of c.
- The the calling vproc waits for the cancelable to become inactive. Once the associated fiber is both inactive and canceled, the fiber can no longer spawn child fibers, so it must be the case that the cancelable's children list can no longer change.
- The calling vproc clears the child-pointer list and recursively applies `cancel` to each child of c.

Let us consider the modifications necessary to add cancellation support to a given scheduling policy.

1. When spawning a fiber, the scheduler needs to wrap the new fiber in a cancelable.

2. When blocking a fiber, the scheduler must always re-wrap the resumption fiber.

5.1.2 Implementing parallel-or

The "parallel-or" operation is a well-known form of speculative computation that dates back to Multilisp [67]. We define

```
val por : ((unit -> 'a option) *
           (unit -> 'a option)) -> 'a option
```

as the parallel-or operation which nondeterministically returns the result of one of two function applications. The expression `por (`f_1`,` f_2`)` has the following behavior: An implementation adheres to the following five criteria:

1. It creates a task to evaluate each f_i in parallel.
2. It returns the first `SOME v` value.
3. It returns `NONE` if both f_i evaluate to `NONE`.
4. It cancels useless tasks after the first `SOME v` value is returned.
5. It schedules the execution of subtasks so as to minimize the time to return a result.

Let us implement `por` in terms of cancelables and work stealing.

A "`por` cell" maintains the intermediate state of a `por` call. Each of the two branches of a `por` call use this state to determine whether to post a result or to terminate. Figure 5.4 shows the implementation of `por` cells, which is encapsulated by the `porCell` function. This function maintains a local state variable called `done`, which is in one of three states:

- If `EMPTY`, neither branch has completed.
- If `PARTIAL`, the result of exactly one branch is `NONE`.
- If `FULL`, the result of one or both branches is non-`NONE`.

```
fun porCell () = let
    datatype por_cell = EMPTY | PARTIAL | FULL
    val done = alloc EMPTY
    fun markFull () =
        if (atomicCompareAndSwap (promote done, EMPTY, FULL)
              orelse
            atomicCompareAndSwap (promote done, PARTIAL,  FULL))
         then ()
         else stop ()
    fun markEmpty () =
         if (atomicCompareAndSwap (promote done, EMPTY, PARTIAL))
           then stop ()
         else if (#0(promote done) = FULL)
           then stop ()
         else ()
    in
      {markFull=markFull, markEmpty=markEmpty}
    end
```

Figure 5.4: `por` cells

The function exports two functions:

- `markEmpty` is called by a branch in order to record that the branch has computed `NONE`. If this branch is the second branch to complete and the other branch computed `NONE`, then `markEmpty` returns. Otherwise, `markEmpty` terminates the branch by calling `stop ()`.

- `markFull` is called by a branch in order to record that the branch has computed a non-`NONE` value. If this branch is the first branch to compute a non-`NONE` value, then `markFull` returns. Otherwise, `markFull` terminates the branch by calling `stop ()`.

Figure 5.5 contains the definition of `por`. The implementation is based on the work stealing strategy similar to the one described in Chapter 3. Initially, the operation pushes on the deque a task for the second branch and then starts executing the first branch. There are two fast paths and two slow paths. In a fast path, the operation executes locally, whereas in the slow path, the first branch executes locally and the second executes in parallel on a thief processor.

```
fun por (f1, f2) = let
  cont retK (x) = x
  val {markFull, markEmpty} = porCell ()
  val c1 = makeCancelable ()
  val c2 = makeCancelable ()
  fun slowClone () =
      (case f2 ()
        of SOME v => (markFull (); cancel c1; throw retK (SOME v))
         | NONE => (markEmpty (); throw retK NONE))
  fun body () =
    (case f1 ()
      of SOME v =>
         if popTl () = NONE then
           (* slow path *)
           (markFull (); cancel c2; throw retK (SOME v))
         else
           (* fast path *)
           SOME v
       | NONE =>
         if popTl () = NONE then
           (* slow path *)
           (markEmpty (); throw retK NONE)
         else
           (* fast path *)
           f2 ())
  in
     pushTl (cancelWrapFiber (c2, slowClone));
     cancelWrapFun (c1, body) ()
  end
```

Figure 5.5: The `por` function.

Associated with the branches are cancelables `c1` and `c2`. In the slow path, the first branch to return a non-`NONE` value cancels its sibling. In the fast path, there is no cancellation.

5.1.3 Evaluation

Experiment setup The test machine has four dual-core AMD Opteron 870 processors running at 2GHz. Each core has a 1Mb L2 cache. The system has 8Gb of RAM and is running Debian Linux (kernel version 2.6.18-6-amd64).

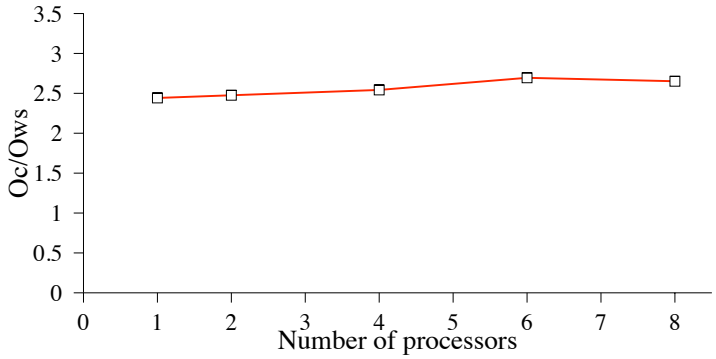


Figure 5.6: Ratio of cancellation overhead (O_c) to work-stealing overhead (O_{ws}).

I performed two experiments to measure the costs of cancellation. The first experiment measures the extra scheduling overhead imposed by cancellation. I measured execution times for versions of the synthetic fibonacci benchmark (Figure 3.1) compiled with and without support for cancellation. The quantity O_{ws} represents the overhead of work stealing scheduler and O_{ws} the overhead of work stealing and cancellation combined.

Figure 5.6 plots the O_c/O_{ws} as a function of number of processors. On a single processor, scheduling costs with cancellation are 2.5 times more expensive than without. The cancellation overhead remains nearly constant up to eight processors.

The second experiment measures the latency of cancellation using n-Queens as a benchmark. The benchmark solves the n-Queens puzzle for $n = 20$, and is written in terms of `por`.

Figure 5.7 shows the average time over 20 runs that it took to cancel the outstanding computations (i.e., all of the fibers that were either running or on the deque waiting to be stolen). The results show (1) that the latency of cancellation is largely independent of the number of processors, and (2) that the standard deviation is small, except for the six-processor configuration, where two outliers skewed the results. These outliers are likely due to the OS swapping out a vproc at the instant in which a cancellation is occurring. Similar anomalies have been reported in the context of JCilk [23], although on a different architecture and OS.

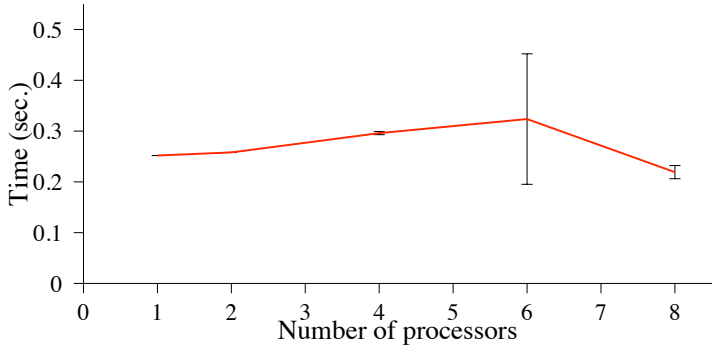


Figure 5.7: Cancellation time for n-Queens benchmark.

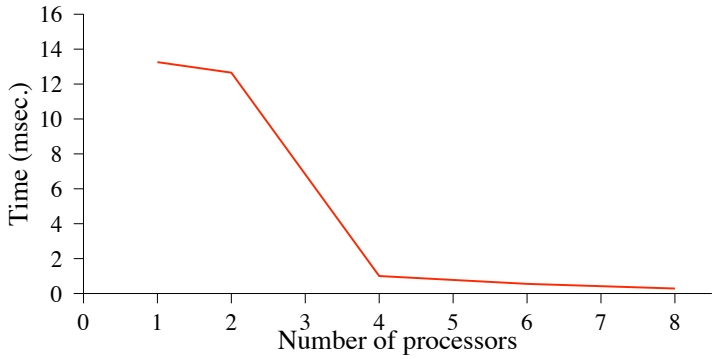


Figure 5.8: Time per canceled fiber.

For the second experiment, I plotted the average time in milliseconds to cancel a computation in Figure 5.8. The graph shows that the cancellation phase is benefiting from parallelism.

These two experiments show that the overheads from supporting cancellation are not dependent on the number of processors, which suggests that cancellation will scale well when applied to more realistic workloads.

5.2 The WS LP scheduler loop

In Chapter 3, I described WS LP's clone compilation. This section presents WS LP's scheduler loop, which completes our description of Manticore's WS LP implementation The implementation of the scheduler loop demonstrates how WS LP fits into the larger Manticore scheduling system. In

particular, this implementation makes explicit the relationship between WS LP and CML threads.

Recall that a PML program consists of one or more CML threads. Furthermore, there can be many more CML threads than the number of processors. As with other implementations of CML [72, 74], the Manticore scheduler multiplexes CML threads on a given processor.

Because each CML thread can launch its own independent instance of WS LP, we allow multiple instances of WS LP to execute concurrently. An instance of WS LP consists of P workers where P is the number of processors. The workers collaborate on behalf of the CML thread that created the corresponding instance. At a given instant, a worker is either idle and attempting to steal a task or busy working on part of the computation. Each instance of the work stealing scheduler is identified by a `wsInstanceUID`, a unique integer identifier.

Let us consider the implementation of the worker loop. The type

```
datatype worker
  = WORKER of {
      assignedVP    : vproc,
      wsInstanceUID : int,
      terminate     : ![bool],
      nIdle         : ![int],
      idleWorkers   : bool array,
      fls           : fls
    }
```

represents the local state of a given worker, including the vproc to which the worker is assigned, the instance identifier, the termination flag, the number of idle workers, the idle-worker array, and the FLS of the calling CML thread.

The behavior of a given worker is specified by the WS LP loop, shown in Figure 5.9. When a task terminates, the loop either executes a task from the local deque or, if the local deque is empty, becomes idle and tries to steal. Stealing is accomplished by the operation

```
val steal : worker -> task option
```

which takes a worker and repeatedly tries to steal from another worker, returning either `SOME` t where t is the stolen or `NONE` if the number of failed steal attempts exceeds some threshold value.

The thief uses the array `idleWorkers`, which maintains flags indicating which workers are currently idle. If the `idleWorkers` flag of a given worker is set to true, then the thief delays sending the potential victim a thief request. This policy helps reduce the latency of steals by reducing time wasted sending useless thief requests.

Recall that WS LP uses the private access model described in Section 2.6, and therefore cannot share deque references across processors. A thief locates the deque of its victim by looking up the deque in a vproc-local table. The thief uses the `wsInstanceUID` as a key into the table.

Multiplexing The WS LP loop also handles preemption events, which cause the scheduler to relinquish the processor for a while. By relinquishing the processor on preemption, the worker allows other threads or workers from other instances of WS LP get access to the relinquished vproc. In this way, the scheduler makes it possible for the system to multiplex a vproc among multiple WS LP instances.

Termination A worker is terminated once the CML thread that created the worker is itself terminated. Termination occurs exactly when all workers are idle. Worker termination is determined by the flag `terminate`, is set to true once the CML thread that created the worker is determined to be terminated. Workers determine when termination occurs by maintaining `nIdle`, a count of the number of workers that are idle. Once this variable is equal to the number of workers, the `terminate` flag is set to true and the workers terminate.

5.3 The vproc interrupt mechanism

One important part of the system is the mechanism for interrupting the execution vprocs. The system uses this interrupt mechanism to perform several critical duties, including global garbage collector initiation, delivering preemptions, and handling steal requests generated by the work-stealing scheduler, *etc.*. Consequently, properties of a given interrupt mechanism, such as latency,

```
(* runs task t under scheduler action a *)
(* built on top of the "run" operation *)
val runTask : (signal cont * task) cont
fun runTask (a, t) = ...

(* builds a task record corresponding to fiber k *)
val fiberToTask : fiber -> task
fun fiberToTask k = ...

(* similar to pushTop and popTop, but instead take deques *)
(* and tasks explicitly as parameters *)
val pushTop' : deque -> task -> unit
val popTop'  : deque -> task option

(* spawn a worker instance w on its assigned vproc vp *)
val spawnWorker : vproc * worker -> unit
fun spawnWorker (vp, w as WORKER{deque, fls, ...}) = let
  fun launch () = let
    val deque = newDeque ()
    in
      cont ws STOP =
          (case popTop' deque
            of NONE =>
               (case steal w
                 of NONE => (
                    sleep ();
                    throw ws STOP)
                  | SOME t => runTask (ws, t))
             | SOME t => runTask (ws, t))
        | ws (PREEMPT k) = (
          pushTop' deque (fiberToTask k);
          yield ();
          throw ws STOP);
      run (ws, fiber (fn _ => stop ())
    end
  in
    enqOnVP (vp, fls, fiber launch)
  end
```

Figure 5.9: WS LP loop implemented in BOM

affect the performance of the system as a whole, making it important to invest some effort into designing the mechanism carefully.

Manticore's interrupt mechanism is based on a technique called software polling in which each processor polls its own memory location to detect interrupts. A software-polling system must strike a balance between polling frequently enough so that response time is low but not so much so that application performance is degraded by overhead expenses.

Feeley describes a technique to address this issue in which the compiler inserts polling operations into application code. [29]. The frequency of polling checks can be controlled to a large degree by adjusting several tuning parameters.

Manticore uses a different software-polling technique that is based on "zeroing the limit pointer." This techniques forces the target vproc to perform a garbage collection. Once in the garbage collector, the system makes a preemption transition like the one shown in Figure 2.11. There is a potential problem, however, that the program can run for a long time without checking its heap-limit pointer and during that time be unresponsive to interrupts. The Manticore compiler prevents unbounded waits by inserting heap-limit checks into non-allocating loops. The technique is similar to the one used by SML/NJ [71].

An analysis carried out by Mogul suggests that software polling is preferable for handling events that are frequent and spaced out regularly and that hardware interrupts are preferable for events that are infrequent and spurious [61]. Here, I argue that Manticore's choice of software polling does not contradict this result.

Primarily, Manticore uses its form of software polling because the system already pays a cost for frequent heap-limit checks and extending the mechanism to support software polling increases this cost only slightly. The software-polling mechanism provides reasonably low latency. The round-trip time for an interrupt, *i.e.*, the time it takes for one vproc to interrupt another and receive a response interrupt, is in the tens of microseconds.

An earlier version of Manticore used a zero-cost interrupt mechanism based on hardware in-

terrupts. In this version, an interrupt is delivered to a vproc by first triggering a hardware interrupt on that vproc. The interrupt handler then sets the limit pointer to zero.

One might wonder how the latency of software polling compares to hardware interrupts. So, I ran an experiment on the sixteen-core test machine in which two vprocs alternate passing interrupts back and forth for one million iterations. Software polling averages at 10 microseconds and hardware interrupts averages at 90 microseconds.

There is another Manticore-specific reason for not using hardware interrupts: building a robust implementation is much harder because interrupts arrive asynchronously, whereas interrupts detected by polling are handled synchronously. The polling mechanism rules out the possibility of an interrupt handler getting invoked during sensitive points, such as in between transfers between Manticore code and the GC when the stack is in an inconsistent state.

5.4 Summary and related work

The Manticore scheduling system presented in Section 2.5.1 is an extension of Shivers' proposal for exposing hardware concurrency using continuations [80]. The Manticore system extends Shivers' proposal to support nesting scheduler actions and multiple processors.

Morrisett and Tolmach designed MP, a extension to SML/NJ, for running SML programs on multiprocessors [63]. MP's concurrency features are built on top of first-class continuations and implemented in SML. In this regard, MP is similar to Manticore's scheduling system. But unlike MP, Manticore's scheduling policies are written in a more primitive language, BOM. The split between BOM and PML avoids polluting PML with implementation-specific details, such as promotion or atomic operations.

Fisher and Reppy designed a multiprocessing system for the Moby programming language [30]. Their design uses a compiler intermediate representation called BOL as the language for programming scheduling and synchronization policies. In their design, much of the runtime is still implemented in C and assembly, including parts of the thread scheduler. Primitives written at this

level might be faster than their counterparts written in BOL, but this style of programming introduces unwanted complexity. For example, there is complexity in sharing data structures between C and BOL. Both languages need to know the exact layout in memory in order to share the data, so changing a data structure involves changing both the C and BOL runtime. Another source of complexity is related to garbage collection. Scheduling code often needs to allocate objects in the heap, but heap allocation is complex because it might involve invoking the garbage collector. Heap allocating in a BOL (or BOM) program is easier than heap allocating in a C program.

STING is a system that supports a concurrent dialect of SCHEME [48, 49]. Like Manticore, STING offers flexibility in scheduling, synchronization, and memory management. STING's process model consists of threads and virtual processors. This thread data type is more complex than the type of continuation used to represent a fiber. For example, a STING thread consists of a thread control block with associated stack, heap segment, saved registers, *etc.*. STING's notion of virtual processor is different than Manticore's. STING virtual processors can be created and destroyed dynamically by the application. Because the number of STING virtual processors is unbounded, the virtual processors are scheduled among the actual hardware processors. vprocs corresponds to actual hardware processors and therefore require no such scheduling. Furthermore, the dynamic behavior of a STING virtual processor is specified by a rich data type. For example, the data structure of a STING virtual processor records preemption and migration policies. The scheduler-action stack plays a similar role by determining the scheduling policy for a given vproc at a given instant. In general, the Manticore design favors simplicity and portability, whereas the STING design favors rich abstractions and fast execution.

Li *et al.* propose a new version of the GHC runtime system designed for shared-memory multiprocessors [53, 54]. The GHC runtime supports Haskell, a lazy functional language. As in the Manticore runtime, a goal of the design is to enable programmable scheduling policies in Haskell.

One problem faced by GHC is that locking synchronization interacts poorly with lazy evalua-

tion. In a lazy setting, it is difficult to guarantee that a thread holding a lock will release the lock in a bounded amount of time because that thread might have to evaluate a long-running thunk. To address this issue, the GHC runtime avoids locking and instead relies on a notion of transactional memory, which is a concurrency-control mechanism analogous to database transactions. The Manticore runtime does not have to address this issue because BOM programs are strict. Therefore, BOM synchronization can be performed safely by using atomic-memory instructions, such as compare-and-swap.

GHC uses a software-polling mechanism that is similar to the one I describe above in Section 5.3. Polling is handled by piggy backing off heap-limit pointer checks. GHC uses this mechanism not for stealing (GHC handles steals by public-deque access) but for forcing processors to join in a parallel GC. Marlow *et al.* did a performance study to compare this mechanism to an alternative one in which the heap-limit pointer is in memory and GCs are initiated via OS interrupts [56]. They find that the polling mechanism speeds up most benchmarks by reducing the latency involved in stopping all processors for a parallel GC.

In his dissertation, Spoonhower presents a parallel implementation of MLton [82]. The implementation consists of some low-level mechanisms for taking advantage of multiprocessors and an SML library of scheduling primitives, which supports multiple task-scheduling policies. The library interface is similar to the interface the Manticore scheduling system provides for task-scheduling policies. He demonstrates implementations of breadth-first, depth-first, and work-stealing scheduling policies. Unlike Manticore, Spoonhower's system has sophisticated mechanisms for profiling memory usage. Interestingly, by relying on MLton's aggressive suite, Spoonhower is able to achieve the effect of the clone translation through just library functions.

The JCilk language [23] is a version of Java that is extended with fork-join parallelism and is based on Cilk's work-stealing scheduler. The primary novelty of JCilk is speculative computation. In JCilk, speculative computation is expressed by using a mechanism similar to Java exceptions. JCilk and PML have different ways of expressing speculative computations, and these differences

are described elsewhere [34]. But both languages have similar implementation requirements with respect to the cancellation of unecessary tasks. Specifically, canceling some task t involves terminating t and recursively cancelling all of t's spawned tasks. The JCilk implementation is written in Java, using a control mechanism similar to first-class continuations and a few data structures to track parent-child relationships in the computation. JCilk's implementation is specialized to the work stealing scheduler, whereas our Manticore's cancellation is not. Our cancellation mechanism is independent of the work-stealing scheduler, and as such, our mechanism has potential to be integrated with any task scheduler that maps down to fibers. This feature is useful if the language supports different task-scheduling strategies, such as PDF [65].

CHAPTER 6

CONCLUSION

High-level parallel languages are expressive frameworks for building parallel programs. In such languages, programmers are able to specify concisely what they want to compute by specifying a little about how to decompose parallel computation. The example programs shown in Chapter 2 demonstrate the simplicity and clarity that is possible with such languages, in the particular case of PML. This expressiveness is made possible, in large part, by abstracting away scheduling and memory management. The complexity inherent in implementing these duties falls to the language implementation.

A robust language implementation offers scalable parallel performance across many applications and platforms. As this dissertation has demonstrated, if the language implementation is not robust, programmers (or compilers) will be faced with unrealistic for most application demands to tune programs (*e.g.*, by manually coarsening the granularity of recursions) to perform well under particular configurations, such as the number of processors or input-data size. Such tuning is unrealistic because the number of possible configurations is enormous.

High-level parallel languages are typically paired with simple cost models, such as the work-span model. But the ability of these cost models to predict actual performance relies crucially on the language implementation. Robust language implementations are important because they increase the likelihood that programs will perform according to the language cost models.

This dissertation attacks these problems by starting from within the well-established framework of work stealing, a policy for scheduling the execution of large numbers of tasks generated by NDP constructs across processors. Seminal work on the Cilk-5 system demonstrated that the work-first principle is an essential tool for building efficient implementations of work stealing [36]. Using the work-first principle as a guide, I have designed two techniques, Lazy Promotion and Lazy Tree Splitting, for scheduling the execution of applications written in NDP. In the next four sections, I summarize my work on Lazy Promotion, Lazy Tree Splitting, and the Manticore implementation,

and then I propose some ideas for building on these techniques.

6.1 Lazy Promotion

A simple example demonstrates why high-level parallel languages rely crucially on efficient memory management. Suppose that, for a given application, memory management takes 10% or more of total execution time, which is a reasonable assumption for many functional applications. If this 10% is sequential, then Amdahl's law predicts that the *best* possible speedup on sixteen processors is 6.4. We can avoid such sequential bottlenecks by using memory managers that offer scalable parallel performance.

Manticore has a split-heap memory manager that offers scalable performance up to at least sixteen processors. Promotion is the main cost imposed by Manticore's memory manager, and is a significant cost for overall performance. In addition to copying overhead, promotion increases the amount of data allocated in the global heap, thereby increasing the frequency of global collections. Reducing the amount of promoted data reduces the frequency of global garbage collections, and thus can help to reduce communication overall.

This dissertation presented a work-stealing policy called WS LP that is efficient with respect to promotion. The design of WS LP was inspired by the work-first principle. Crucially, WS LP delays promoting task data until the rare case in which a task is stolen, limiting the number of overall promoted tasks to $O(PT_\infty)$. The implementation of WS LP is based on three techniques: an efficient deque data structure, a clone-compilation scheme adapted from the Cilk-5 compiler, and a steal-request mechanism. An engineering advantage of the implementation is that minimal support is required from the compiler (just a simple source-to-source translation at an early phase in the compilation process).

To evaluate the Manticore design, I have performed an empirical study of six PML benchmark programs. The study shows that lazy promotion often outperforms eager promotion and never does worse. The study also evaluates Manticore's split-heap design by comparing the system with

a modified version based on a flat-heap architecture. Our split heap outperforms the flat heap by a wide margin across all benchmarks. The results show that the flat heap has a much higher miss rate in the L3 cache than do the other versions of the system. The poor locality of a flat heap is a major factor in its poorer mutator performance.

The main lesson that can be drawn from this study is that, in garbage-collected languages, the performance of memory management is crucial for scalable parallel performance and by using the principle of separating local state across processors, we can readily design a memory manager that offers scalable parallel performance. We can make this design simple by relying on strict invariants on data placement, which are readily enforced in the setting of a pure, functional language, such as PML. With this memory-manager design in hand, we can adapt our work-stealing policy so that it performs well with the memory manager and the system as a whole.

6.2 Lazy Tree Splitting

I have presented an efficient and robust implementation of nested-data parallelism in Manticore. The implementation is based on a new technique called Lazy Tree Splitting, which is a generalization of Lazy Binary Splitting. Crucially, LTS provides programs with both robust performance and a means to process hierarchical structures that are distributed in memory. I have evaluated the technique by comparing it to an alternative ETS approach. LTS outperforms most configurations of ETS, and is at most 20% worse than the optimal ETS configuration. I have also presented evidence that tuning ETS to achieve decent performance, let alone optimal performance, is unrealistic.

Although the zipper style used by LTS is a style of functional programming, LTS is a general technique that can be beneficial for a large class of languages, including imperative or object-oriented languages. The only special requirement of LTS is a work-stealing scheduler, which is now part of many parallel languages. LTS also has potential to be implemented in pure languages, such as Haskell and Concurrent Clean, although such an implementation would require special measures to avoid violating referential transparency.

6.3 Implementing work stealing in Manticore

Manticore's implementation of work stealing is built from many parts. In Chapter 5, I gave in-depth discussion on several important parts, including the task-cancellation mechanism, the work-stealing scheduler loop, and the vproc-interrupt mechanism.

Task cancellation is a mechanism which task schedulers, such as work stealing, use to provide efficient speculative parallelism. The cancellation protocol is subtle because it involves tracking down nodes in the task graph as nodes are being scheduled across different processors. I presented an implementation of task cancellation and demonstrated its efficiency on an eight-processor system. An advantage of the design of the mechanism is that task cancellation is implemented as a small scheduling component that can be reasoned about in isolation.

The work-stealing scheduler loop is the final part of this implementation. Its implementation shows how I address work stealing for PML programs, where there are multiple CML threads and each of which can spawn off parallel tasks. I use a strategy in which each CML thread has its own work-stealing scheduler, which is similar to Blumofe's multiprogrammed work stealing [3].

WS LP uses the vproc-interrupt mechanism interrupts to accomplish low-latency steals. The main lesson from this study is that software polling is a viable mechanism for interrupts, which can be readily implemented in a system like Manticore where heap-limit checks are frequent. The secondary lesson is that hardware interrupts are also viable in terms of performance but much harder to implement without concurrency bugs.

6.4 Future directions

Tree data structures, such as ropes, give the memory manager two useful types of flexibility:

- Because trees can be distributed in memory, subtrees of the same tree can be allocated and reclaimed concurrently by different processors.
- When trees are persistent, as is the case with our ropes, the memory manager has the freedom

to replicate subtrees across processors.

The Manticore memory manager takes advantage of both features, but uses replication to a very limited extent. An interesting piece of future work is to extend LTS to support a memory manager with an even more aggressive heap-splitting strategy, such as the memory manager offered by GUM [85], in which there is no global heap. In such a memory manager, effective replication is crucial for achieving scalable parallel performance.

LTS has potential to be used for more types of balanced trees than just ropes, *e.g.*, red-black trees. An interesting question is whether one can derive traversal patterns that are both efficient and preserve the invariants of such tree structures.

Unlike the flattening approach to NDP, LTS is a purely dynamic approach and therefore can be ported to an existing language implementation without need for sophisticated compiler transformations. Nevertheless, as discussed in Section 4.3, LTS can benefit from compiler optimizations such as fusion [18] and compile-time granularity analysis [83].

The lazy splitting idea is general enough to be applied to the class of divide and conquer algorithms. There are certain technical difficulties that must be overcome to make it effective for general divide and conquer algorithms. These difficulties include identifying recursive functions that exhibit the divide-and-conquer pattern and developing techniques to aggregate work for small problem sizes (rope leaves effectively provide this mechanism). Failure to do the latter is likely to result in excessive overhead that will limit scalability.

6.4.1 A domain-specific language for scheduling

Research on task scheduling has suggested that no single scheduling policy is the best for every application and, furthermore, that there are many distinct policies, each with its own strengths and weaknesses. For instance, Blelloch *et al.* have shown that, for NDP programs, the PDF scheduling policy offers lower memory use at run time than many other policies [9]. But such a space-efficient scheduler typically has a higher communication overhead than alternative work-stealing

policies. There exists a trade-off between the two policies and supporting both approaches is the most flexible approach. Furthermore, with respect to work stealing, certain applications are known to benefit from different variants of the stealing policy [24, 76]. For example, Sanchez, *et al.*, show cases where applications executed on NUMA architectures perform better with *hierarchical* work stealing [76]. In hierarchical work stealing, as opposed to fully-randomized work stealing, thief processors prefer to steal from processors that are closer in memory. These results suggest that a language implementation can broaden the range of applications it supports well by supporting multiple scheduling policies and easy extensions to those policies. An even more flexible approach would be to support a programmable scheduling system, such as the Manticore scheduling system that I presented in Chapter 2.

The way that scheduling policies are written in a language implementation (or operating system) leads to unclear and incorrect code. Because of performance reasons, the scheduling code is often split into small pieces and the pieces are spread across several functions or files. The more natural specification of the scheduler, however, often consists of a single scheduling loop. Such a specification is simpler and easier to test in isolation. A domain-specific language offers the potential to write code that is closer the the original specification, and if the domain-specific language employs an optimizing compiler, it also offers the potential to generate efficient scheduling code. For example, a domain specific language would use a specification of WS LP that is closer to the pseudocode I presented in Chapter 3 than the BOM code I presented across Chapters 3 and 5, yet the using domain-specific language would ideally provide much the same performance as using BOM directly.

As I argued above, it is desirable to support a mix of scheduling policies. A domain-specific language can assist in this goal by admitting a high-level specification of a policy that is liberated from complexities of a particular language implementation. There are situations in which an application writer who is not an expert in the language implementation may desire a custom scheduling policy for his or her application. A domain-specific language would allow such a programmer to

write a custom policy.

These issues have motivated other researchers to explore the domain-specific language called Bossa [5, 64]. Bossa, however, is limited to uniprocessor scheduling, and multiprocessor scheduling would require a more general model than Bossa provides. A multiprocessor policy, such as WS LP, is a *distributed program* in which a whole scheduler consists of multiple instances that execute concurrently. Often these instances need to communicate to balance workload. I am unaware of recent work that addresses a domain-specific scheduling language targeting multiprocessors. I believe this line of work has the potential to both standardize the notation in multiprocessor scheduling policies and provide more flexible and efficient multiprocessor scheduling for language implementations.

REFERENCES

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *SP&E*, 19(2):171–183, 1989.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors, 1998.

[4] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, Dec. 1986.

[5] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *RTS '02*, pages 19–31, Mar. 2002.

[6] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[7] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, Mar. 1996.

[8] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, 21(1):4–14, 1994.

[9] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *JACM*, 46(2):281–321, 1999.

[10] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96*, pages 213–225, New York, NY, May 1996. ACM.

[11] R. D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Cambridge, MA, USA, 1995.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95*, pages 207–216, New York, NY, July 1995. ACM.

[13] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *STOC '93*, pages 362–371, New York, NY, 1993. ACM.

[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[15] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *SP&E*, 25(12):1315–1330, Dec. 1995.

[16] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, New York, NY, Oct. 1981. ACM.

[17] M. M. T. Chakravarty and G. Keller. Functional array fusion. In *ICFP '01*, pages 205–216, New York, NY, 2001. ACM.

[18] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial Vectorisation of Haskell Programs. In *DAMP '08*, New York, NY, Jan. 2008. ACM.

[19] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, pages 10–18, New York, NY, Jan. 2007. ACM.

[20] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05*, pages 21–28, New York, NY, 2005. ACM.

[21] S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM TOPLAS*, 15(3):400–462, July 1993.

[22] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *ASPLOS*, pages 164–175, New York, NY, USA, 1991. ACM.

[23] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with Exceptions in JCilk. *SCP*, 63(2):147–171, 2006.

[24] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC '09*, pages 1–11, New York, NY, 2009. ACM.

[25] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, pages 70–83, New York, NY, Jan. 1994. ACM.

[26] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, pages 113–123, New York, NY, Jan. 1993. ACM.

[27] M. Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Brandeis University, Waltham, MA, USA, 1993.

[28] M. Feeley. A message passing implementation of lazy task creation. In *Proceedings of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, pages 94–107, London, UK, 1993. Springer-Verlag.

[29] M. Feeley. Polling efficiently on stock hardware. In *FPCA '93*, pages 179–187, New York, NY, June 1993. ACM.

[30] K. Fisher and J. Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, Mar. 2002. Available from `http://moby.cs.uchicago.edu/`.

[31] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *ML '07*, pages 15–24, New York, NY, Oct. 2007. ACM.

[32] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, pages 241–252, New York, NY, Sept. 2008. ACM.

[33] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, pages 119–130, New York, NY, Sept. 2008. ACM.

[34] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, 2010. Accepted.

[35] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*, pages 37–44, New York, NY, Jan. 2007. ACM.

[36] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, New York, NY, June 1998.

[37] GHC. Barnes Hut benchmark written in Haskell. Available from `http://darcs.haskell.org/packages/ndp/examples/barnesHut/`.

[38] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A flexible parallel programming model for tera-scale architectures. Technical report, Intel, Oct. 2007. Available at `http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf`.

[39] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: implementing a fast parallel call. In *JPDC*, pages 37(1):5–20, 1996.

[40] R. H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*, pages 9–17, New York, NY, Aug. 1984. ACM.

[41] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84*, pages 293–298, New York, NY, Aug. 1984. ACM.

[42] D. Hendler, Y. Lev, and N. Shavit. Dynamic memory ABP work-stealing. In *DISC '04*, volume 3274 of *LNCS*, pages 188–200, New York, NY, 2004. Springer-Verlag.

[43] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, New York, NY, 2008.

[44] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *JFP*, 16(2):197–217, 2006.

[45] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa. Backtracking-based load balancing. *PPoPP '09*, 44(4):55–64, Feb. 2009.

[46] G. Huet. The zipper. *JFP*, 7(5):549–554, 1997.

[47] Intel. *Intel Threading Building Blocks Reference Manual*, 2008.

[48] S. Jagannathan and J. Philbin. A customizable substrate for concurrent languages. In *PLDI '92*, pages 55–81, New York, NY, June 1992. ACM.

[49] S. Jagannathan and J. Philbin. A foundation for an efficient multi-threaded scheme system. In *LFP '92*, pages 345–357, New York, NY, June 1992. ACM.

[50] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 1999.

[51] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09*, pages 522–527, New York, NY, 2009. ACM.

[52] R. Leshchinskiy. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 2005.

[53] P. Li. *Programmable concurrency in a pure and lazy language*. PhD thesis, University of Pensylvania, 2008.

[54] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for GHC. In *HASKELL '07*, pages 107–118, New York, NY, Sept. 2007. ACM.

[55] H. W. Loidl and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *GWFP '95*, pages 8–10. Springer-Verlag, 1995.

[56] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP '09*, pages 65–77, New York, NY, August–September 2009. ACM.

[57] C. McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL '08*, pages 287–295, New York, NY, Jan. 2008. ACM.

[58] S. Microsystems. Fortress Programming Language.

[59] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[60] MLton. The MLton Standard ML compiler. Available at `http://mlton.org`.

[61] J. C. Mogul. Network locality at the scale of processes. *ACM TOCS*, 10(2):81–109, 1992.

[62] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90*, pages 185–197, New York, NY, June 1990. ACM.

[63] J. G. Morrisett and A. Tolmach. Procs and locks: A portable multiprocessing platform for Standard ML of New Jersey. In *PPoPP '93*, pages 198–207, Apr. 1993. An earlier version is available as CMU report CMU-CS-92-155.

[64] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE '05*, pages 56–65, Oct. 2005.

[65] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM TOPLAS*, 21(1):138–173, 1999.

[66] R. S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[67] R. B. Osborne. Speculative computation in multilisp. In *LFP '90*, pages 198–208, New York, NY, June 1990. ACM.

[68] N. Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Dept. of C.S., Princeton University, Apr. 1990.

[69] J. Reppy. Optimizing nested loops using local CPS conversion. *HOSC*, 15:161–180, 2002.

[70] J. H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Dept. of CS, Cornell University, Dec. 1989.

[71] J. H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Dept. of CS, Cornell University, Ithaca, NY, Aug. 1990.

[72] J. H. Reppy. CML: A higher-order concurrent language. In *PLDI '91*, pages 293–305, New York, NY, June 1991. ACM.

[73] J. H. Reppy. *Higher-order concurrency*. PhD thesis, Dept. of CS, Cornell University, Ithaca, NY, Jan. 1992. Available as Technical Report TR 92-1285.

[74] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[75] A. Robison, M. Voss, and A. Kukanov. Optimization via Reflection on Work Stealing in TBB. In *IPDPS '08*, Los Alamitos, CA, 2008. IEEE Computer Society Press.

[76] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. *ASPLOS*, 38(1):311–322, 2010.

[77] V. Saraswat. Report on the experimental language X10. Technical Report DRAFT v 0.41, IBM Thomas J. Watson Research Lab, Yorktown Heights, NY, Feb. 2006.

[78] Scandal Project. A library of parallel algorithms written NESL. Available from `http://www.cs.cmu.edu/~scandal/nesl/algorithms.html`.

[79] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICSA '08*, pages 277–288, New York, NY, 2008. ACM.

[80] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *CW '97*, New York, NY, Jan. 1997. ACM.

[81] B. So, A. Ghuloum, and Y. Wu. Optimizing data parallel operations on many-core platforms. In *STMCS '06*, 2006.

[82] D. Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburg, PA, USA, 2010.

[83] E. Tick and X. Zhong. A compile-time granularity analysis algorithm and its performance evaluation. In *FGCS '92*, pages 271–295, New York, NY, USA, 1993. Springer-Verlag.

[84] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, Jan. 1998.

[85] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. Gum: a portable parallel implementation of haskell. *PLDI '96*, 31(5):79–88, 1996.

[86] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*, pages 179–190, New York, NY, Jan. 2010. ACM.

[87] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03*, pages 268–281, Dec. 2003.

[88] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *ICSA '09*, pages 430–440, New York, NY, 1998. ACM.

[89] M. Wand. Continuation-based multiprocessing. In *LFP '80*, pages 19–28, New York, NY, Aug. 1980. ACM.

[90] S. Weeks. Whole program compilation in MLton. Invited talk at ML '06 Workshop, Sept. 2006. Invited talk; slides available at `http://mlton.org/pages/References/attachments/060916-mlton.pdf`.