

June 1989

Report No. STAN-CS-89-1265

Thesis

2

DTIC FILE COPY

PARALLEL EXECUTION OF LISP PROGRAMS

by

Joseph Simon Weening

AD-A219 623

Department of Computer Science

Stanford University
Stanford, California 94305

DTIC
ELECTE
MAR 12 1990
S B D

90 03 12 093

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No 0704-0188	
1a REPORT SECURITY CLASSIFICATION			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION AVAILABILITY OF REPORT		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			Unlimited Distribution		
4 PERFORMING ORGANIZATION REPORT NUMBER(S) STAN-CS-89-1265			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION Computer Science Dept.		6b OFFICE SYMBOL (if applicable)	7a NAME OF MONITORING ORGANIZATION		
6c ADDRESS (City, State, and ZIP Code) Stanford University Stanford, CA 94305			7b ADDRESS (City, State, and ZIP Code)		
8a NAME OF FUNDING/SPONSORING ORGANIZATION DARPA		8b OFFICE SYMBOL (if applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-84-C-0211		
8c ADDRESS (City, State, and ZIP Code) Arlington, VA			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
			WORK UNIT ACCESSION NO		
11 TITLE (Include Security Classification) Parallel Execution of Lisp Programs					
12 PERSONAL AUTHOR(S) Joseph Simon Weening					
13a TYPE OF REPORT Thesis		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1989, June	
15 PAGE COUNT 92					
16 SUPPLEMENTARY NOTATION					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
Please see other side for abstract...					
20 DISTRIBUTION AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION		
22a NAME OF RESPONSIBLE INDIVIDUAL John McCarthy			22b TELEPHONE (Include Area Code) 723-2273 (415)		22c OFFICE SYMBOL

DD Form 1473, JUN 86

Previous editions are obsolete

S/N 0102-LF-014-6603

SECURITY CLASSIFICATION OF THIS PAGE

---Abstract.

This dissertation considers several issues in the execution of Lisp programs on shared-memory multiprocessors. An overview of constructs for explicit parallelism in Lisp is first presented. The problems of partitioning a program into processes and scheduling these processes are then described, and a number of methods for performing these are proposed. These include cutting off process creation based on properties of the computation tree of the program, and basing partitioning decisions on the state of the system at runtime instead of the program.

An experimental study of these methods has been performed using a simulator for parallel Lisp. The simulator, written in Common Lisp using a continuation-passing style, is described in detail. This is followed by a description of the experiments that were performed and an analysis of the results. Two programs are used as illustrations—a Fast Fourier Transform, which has an abundance of parallelism, and the Cocke-Younger-Kasami parsing algorithm, for which good speedup is not as easy to obtain. The difficulty of using cutoff-based partitioning methods, and the differences between various scheduling methods, are shown.

A combination of partitioning and scheduling methods which we call dynamic partitioning is analyzed in more detail. This method is based on examining the machine's runtime state; it requires that the programmer only identify parallelism in the program, without deciding which potential parallelism is actually useful. Several theorems are proved providing upper bounds on the amount of overhead produced by this method. We conclude that for programs whose computation trees have small height relative to their total size, dynamic partitioning can achieve asymptotically minimal overhead in the cost of process creation.

PARALLEL EXECUTION OF LISP PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

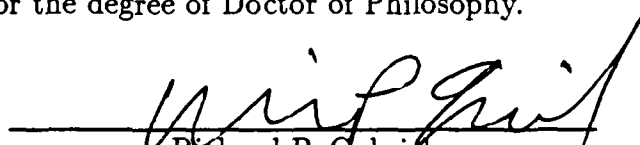
By
Joseph Simon Weening
June 1989

© Copyright 1989 by Joseph Simon Weening
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.


John McCarthy
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.


Richard P. Gabriel

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.


Jeffrey D. Ullman

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

This dissertation considers several issues in the execution of Lisp programs on shared-memory multiprocessors. An overview of constructs for explicit parallelism in Lisp is first presented. The problems of partitioning a program into processes and scheduling these processes are then described, and a number of methods for performing these are proposed. These include cutting off process creation based on properties of the computation tree of the program, and basing partitioning decisions on the state of the system at runtime instead of the program.

An experimental study of these methods has been performed using a simulator for parallel Lisp. The simulator, written in Common Lisp using a continuation-passing style, is described in detail. This is followed by a description of the experiments that were performed and an analysis of the results. Two programs are used as illustrations—a Fast Fourier Transform, which has an abundance of parallelism, and the Cocke-Younger-Kasami parsing algorithm, for which good speedup is not as easy to obtain. The difficulty of using cutoff-based partitioning methods, and the differences between various scheduling methods, are shown.

A combination of partitioning and scheduling methods which we call dynamic partitioning is analyzed in more detail. This method is based on examining the machine's runtime state; it requires that the programmer only identify parallelism in the program, without deciding which potential parallelism is actually useful. Several theorems are proved providing upper bounds on the amount of overhead produced by this method. We conclude that for programs whose computation trees have small height relative to their total size, dynamic partitioning can achieve asymptotically minimal overhead in the cost of process creation.

Acknowledgements

First I would like to thank my advisor, Professor John McCarthy, and the other members of my reading committee, Professors Richard Gabriel and Jeffrey Ullman, for their support throughout the years of researching and writing this dissertation, and their criticisms and comments on the text. Dick Gabriel, especially, provided many hours of his time listening to early versions of my ideas.

Much of the work described in Chapter 5 was done jointly with Dan Pehoushek, and the discovery of the ideas presented there would not have occurred without his dedicated persistence in fine-tuning the Qlisp system.

Other members of John McCarthy's formal reasoning and Qlisp research groups provided much encouragement and friendship, particularly Igor Rivin, Carolyn Talcott and Ramin Zabih.

Professor Robert Halstead of MIT provided me with a complete copy of his Multi-lisp system, the study of which was quite useful. Professor Anoop Gupta of Stanford suggested a number of improvements to the text of the dissertation.

Financial support for my graduate study at Stanford came from the National Science Foundation, the Fannie and John Hertz Foundation, and the Defense Advanced Research Projects Agency.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Contents

Abstract	iv
Acknowledgements	v
1 Parallel Lisp	1
1.1 Shared memory	2
1.2 Parallel argument evaluation	3
1.3 Multilisp	4
1.3.1 Futures	4
1.3.2 Delayed evaluation	6
1.3.3 Synchronization	7
1.3.4 Multischeme	7
1.4 Qlisp	7
1.4.1 Process closures	9
1.4.2 Speculative computation	10
1.5 Other versions of Parallel Lisp	10
1.5.1 PaiLisp	10
1.5.2 ZLISP	11
1.6 Conclusions on language design	11
2 Partitioning and scheduling methods	13
2.1 Computation graphs	14
2.2 Notation for describing performance	16
2.3 Partitioning methods	17
2.3.1 Height cutoff	17
2.3.2 Depth cutoff	19
2.4 Scheduling methods	21
2.5 Dynamic partitioning	22

3	A Parallel Lisp Simulator	25
3.1	A continuation passing Lisp interpreter	26
3.2	An interpreter for Common Lisp	30
3.2.1	Environments	30
3.2.2	The global environment	31
3.2.3	Function application	31
3.2.4	Special forms	33
3.2.5	Multiple values	35
3.2.6	Timing statistics	35
3.3	Parallel Lisp constructs	36
3.3.1	Scope and extent issues	37
3.4	Simulating the parallel machine	39
3.4.1	Processors	40
3.4.2	The scheduler	41
3.4.3	Processes	42
3.4.4	Process closures	43
3.5	Miscellaneous details	43
3.5.1	Use of symbols	43
3.5.2	Preprocessing of definitions	44
3.5.3	Interpreted primitives	45
3.5.4	Top level	46
3.5.5	Memory allocation and garbage collection	46
3.6	Accuracy and performance	46
3.6.1	Accuracy of simulated times	47
3.6.2	Speed of the simulator	48
3.7	A Parallel example	50
4	Experimental results and analysis	57
4.1	The Fast Fourier Transform	57
4.2	The CYK parsing algorithm	65
5	Analysis of dynamic partitioning	73
5.1	Process creation behavior	73
5.2	Extending the basic result	78
5.3	Making use of dynamic partitioning	80
6	Conclusions	81
	Bibliography	83

Chapter 1

Parallel Lisp

Multiprocessor systems have introduced a new set of challenges in the implementation of programming languages. Attempts to make use of parallelism have resulted in a variety of programming models. One way in which these approaches differ is whether they support *implicit* or *explicit* parallelism in programs.

Implicit parallelism insulates the programmer from the details of the parallel environment. The language support system, at compile time or runtime, decides how to translate the constructs of the programming language into concurrent operations on the parallel machine. This approach has been applied both to existing languages, and to new languages designed to take better advantage of a parallel environment.

Explicit parallelism lets the programmer specify what computations are to be performed in parallel. With language constructs for specifying concurrency, a programmer can make use of knowledge about parallelism in a program that an automatic system might not be able to deduce. Existing programming languages have been extended with constructs that specify parallel behavior, and new languages have been proposed that include explicit parallelism.

This dissertation is about explicit parallelism applied to Lisp. Lisp is a language for symbolic data processing that has been used primarily in artificial intelligence [20]. Many problems for which Lisp is used require large amounts of computation, so executing Lisp efficiently on parallel computers is an important problem.

The remainder of this introductory chapter describes our model of computation, and presents several sets of language constructs that have been proposed in parallel Lisp systems. Chapter 2 introduces the concepts of partitioning and scheduling, which are the main focus of our research. Chapter 3 describes a simulator for parallel Lisp which was designed to test strategies for program execution, and discusses the merits and drawbacks of using simulation as an experimental tool. Experiments using the simulator are presented in Chapter 4. The main observation made from

these experiments is the success of a strategy that we call "dynamic partitioning," and an analysis of this method is done in Chapter 5. Finally, Chapter 6 summarizes the work that has been done and points out areas that require further investigation.

1.1 Shared memory

This research considers multiprocessors with shared memory, in which the access time for all memory words is uniform. In other words, there should be no penalty when a program uses "global" memory as opposed to "local" memory. This assumption frees the programmer from the burden of reorganizing data structures in order to improve the speed of memory access, and is therefore less of a departure from the programming model used for Lisp on sequential processors than distributed memory models. Our use of the term "shared memory" will always include this assumption of uniform access time.

While shared memory with uniform access time is useful as a programming model, it may not be possible to build multiprocessors having this characteristic with more than a limited number of processors, using current techniques in hardware design. (The limit seems to be in the range of tens to hundreds of processors.) To study parallel machines with a large number of processors therefore, we may have to allow a non-uniform memory model. On the other hand, several shared-memory machines with a small number of processors have been built and are now commercially available. The methods for programming these machines are still in an early stage of development, and this area is where we hope to make a contribution.

The usual methods of Lisp implementation lead us to prefer a shared-memory approach. Lisp is a call-by-value language, but the values that are passed are often pointers to data structures of arbitrary size. One of the things that has made Lisp such a success is that the management of these data structures (allocation and reclamation by garbage collection) is completely automatic. We do not want to remove this strength of the language by introducing programming tasks that are extraneous to the actual problems being solved, such as the explicit distribution of data and processes to particular parts of memory in order to improve performance. Automating these tasks is a possibility for future research.

Garbage collection is an integral part of Lisp systems, but we do not discuss it in this dissertation. Implementing garbage collection on a multiprocessor is a challenging problem and the subject of much ongoing research.

1.2 Parallel argument evaluation

The most obvious place in which to introduce parallelism in Lisp programs is in the evaluation of arguments to functions. Especially in code written in a "functional" style, most computation in Lisp programs consists of functions calling other functions in order to perform subcomputations, and often no specific execution order is required for correct execution. For example, in an arithmetic expression such as

$$(+ (\log x) (\log y)),$$

the calls to `(log x)` and `(log y)` may be performed in either order, or in parallel.

The main problem with parallel argument evaluation arises when we allow side effects in programs. In some versions of Lisp, the order of evaluation of function arguments is defined by the language (e.g., in Common Lisp [24] it is left-to-right), so side effects in the first function call must happen before those in the second. In other versions of Lisp (e.g., Scheme [1]), the order is unspecified, but it is still assumed that the arguments are evaluated one at a time. For instance, if two arguments to a function are calls to `cons`, then the usual sequential implementation, in which `cons` performs side effects to a list of free cells, is not correct for parallel execution. There will be a critical region in the code for `cons` between the time when the old free-list pointer is read from memory and the time when the new free-list pointer is stored. If several processors execute the code in this region simultaneously, they will all end up allocating the same cell. Correct behavior of `cons` can be ensured in several ways: adding synchronization code around the critical region to prevent simultaneous execution; using special atomic hardware instructions that remove the need for a critical region; or having `cons` allocate from a separate free list for each processor, which also eliminates the critical region.

It is not too hard to avoid the problem of side effects when implementing the pre-defined Lisp functions such as `cons`, so that they may always be called concurrently, but synchronization problems may remain in user-written functions that perform side effects. Therefore, parallel argument evaluation is dangerous when the forms evaluated in parallel may be calls to arbitrary functions. Some Lisp systems avoid this danger by restricting the language to be side-effect free. We feel that side effects and parallelism can coexist, by not making parallel argument evaluation the default behavior of the system, but providing language constructs by which a programmer can explicitly state that concurrency is allowable.

Another important reason for requiring parallel argument evaluation to be explicit is the cost associated with starting a computation on another processor. On most multiprocessors, it is several times the cost of a function call, and adding this cost

to each function call would multiply the total running time by a significant constant factor. We will examine several techniques that avoid much of this potential overhead cost.

1.3 Multilisp

Multilisp [13] is a system begun in 1980 by Halstead at the Massachusetts Institute of Technology. Most of its sequential Lisp constructs come from Scheme [1] [26], although some features of Scheme such as continuations are not supported. Along with language constructs that add parallelism to the basic Lisp, Halstead and his colleagues have designed and implemented a prototype shared-memory multiprocessor, Concert [4], on which Multilisp runs.

One of Multilisp's forms for specifying parallel execution is `pcall`, which implements parallel argument evaluation as described in the previous section. The form

```
(pcall fun arg1 ... argn)
```

causes the expressions `fun` and `arg1, ..., argn` to be evaluated in separate processes;¹ once these processes finish, the value of `fun` is applied to the values of `arg1, ..., argn`. (Although `fun` is usually just a symbol that names a function, Scheme allows an arbitrary expression in this position, and Multilisp evaluates it in parallel with the function arguments.) Using `pcall`, our example from the previous section would be written as

```
(pcall + (log x) (log y)).
```

1.3.1 Futures

Multilisp's other form for creating parallel processes is `future`. When `(future expr)` is evaluated in a program, a new process is created to evaluate `expr`. Unlike `pcall`, which waits for the processes that it creates to finish and then uses the values that they return, `future` allows the original process to continue execution. In place of the value that `expr` will produce, `future` returns an object called a *future*, which represents that value. In the tagged-pointer data representation used by Multilisp, a future is a pointer with a special tag indicating that it is not an ordinary data object, and pointing to the process that is computing the actual object's value. We will call this the future's "associated process."

For most purposes, a future is not distinguishable from the data object that will be returned by its associated process, because Multilisp ensures that whenever an

¹Multilisp calls these "tasks", but we use the term "processes" throughout this dissertation.

operation needs the value of the object represented by a future. If the associated process has not yet finished, then the process requiring the value is suspended until that value is available. For example, the expression

(+ (future (log x)) (log y))

will start a new process to evaluate (log x) and continue in the original process to evaluate (log y). Some concurrency is therefore achieved by performing these calls in parallel. After the call to (log y) returns, however, the '+' function is called; its arguments are the future that was created for (log x) and the value of (log y). The '+' function cannot add a future to a number; it requires the actual value associated with the future. It therefore performs an operation that Multilisp calls "touching" on both of its arguments, to ensure that they are normal Lisp objects, not futures. If an object that is touched is a future, then if its associated process is finished, the value that the process returned is used; if the associated process is still executing, then the touching process is suspended, and is awakened when the value has been computed. In either case, the touching operation eventually returns the value associated with the future.

Aside from the suspension and resumption of the process, there is no semantic difference between what happens when `future` is used and what would have happened had the original expression been (+ (log x) (log y)). As in parallel argument evaluation, the presence of side effects may require a specific execution order and therefore make the use of `future` inappropriate. Several of Halstead's students have studied methods for automatically and safely inserting `future` into programs, in both the side-effect free case [12] (where adding `future` is always safe, but should be avoided when it does not yield any parallelism), and in some programs that perform side effects [30].

One important benefit of futures is the way in which they are treated by forms that do not require the values of their arguments. These are forms which will perform the same operations no matter what arguments they are given. A good example is `cons`, which allocates a cons cell and stores its two arguments into the cell, without examining the arguments. If a future is passed as an argument to `cons`, Multilisp does not wait for the object represented by the future to be computed; instead, it stores the future into the cons cell just like any other object. The future's associated process may therefore execute in parallel with other processes that pass around this value, as long as they do not perform operations that "touch" the value as described above.

Lisp operations that manipulate values without touching them include `cons` and other functions that construct data structures, assignment operations such as `setq`,

and the operations of passing argument values to functions and returning their results. That is to say, a future can be returned from a function and passed to another, all without waiting for the object that it represents to be determined.

1.3.2 Delayed evaluation

The concept of futures is related to that of “delayed” or “lazy” evaluation, which was first proposed for Lisp by Henderson and Morris [15], and is discussed as an aspect of parallel programming by Friedman and Wise [6]. In delayed evaluation, functions like `cons`, that do not require the values of their arguments, do not evaluate their arguments at all. In place of each argument value, they use an object that represents the computation of that value. These objects are recognized by functions that do require an actual value, and at that point, computation of the value is performed. Lazy evaluation can be used to define “infinite data structures.” For example, the following program represents the infinite set of natural numbers by a list, which is only computed as far as any part of the program examines it:

```
(defun natnum (n)
  (cons n (natnum (+ n 1))))

(setq natural-numbers (natnum 0))
```

In Multilisp, `cons` does evaluate its arguments, but delayed evaluation can be indicated by a `delay` expression. The above definition would be written in Multilisp as:

```
(defun natnum (n)
  (cons n (delay (natnum (+ n 1)))))

(setq natural-numbers (natnum 0))
```

The same tagged pointer that is used to represent a future can be used to implement delayed evaluation. The difference between lazy evaluation and the use of futures, also known as “eager” evaluation, is that the lazy method waits to compute a value until it is required, while the eager method computes it in parallel with the rest of the program, on the assumption that the value will be needed. This assumption may be false; the object might be stored in a data structure and never examined, or may be passed as an argument to a function but ignored. Eager evaluation can not always be substituted for delayed evaluation; in the example of the infinite data structure above, substituting `future` for `delay` might cause an infinite sequence of processes to be created. (The method used for scheduling processes would determine whether this would actually occur.)

1.3.3 Synchronization

Futures provide implicit synchronization of processes because the consumer of a data value is forced to wait until the value has been produced. Such synchronization is done by Multilisp and need not be specified by the programmer, so the code for parallel programs is often very clean and concise—it may be the same as the code for a sequential version of the program, except for `future` added wherever concurrency is desired. However, sometimes a programmer wishes to use side effects or destructive operations on data structures, and in these situations the synchronization provided by futures is not enough.

To allow this style of programming, Multilisp provides locks. These are objects on which processes may perform `wait` and `signal` operations. If a process waits on a lock that is busy, it is suspended using the same mechanism that causes processes to wait for futures, and it is resumed when a `signal` causes the lock to be made free.

Multilisp also has two atomic memory operations, `replace` and `replace-if-eq`, which can be used to perform synchronization.

1.3.4 Multischeme

Miller, a student of Halstead, has implemented a system called Multischeme [21]. Miller's work pays particular attention to implementation strategies for various aspects of the system, and is the basis for a parallel Lisp system that has been implemented on the BBN Butterfly.

Multischeme introduces a data type called *placeholders* that are a generalization of Multilisp's futures. Placeholders can be used for both the delayed and eager evaluation methods described above, and they are "invisible" in the sense that a reference to a placeholder automatically returns the associated data object, or suspends the process making the reference until the placeholder is determined. In addition, placeholders can be *mutable*, meaning that the object they refer to may be changed even after its initial value is determined. If there are several references to such a placeholder, all of them will automatically refer to the new object.

Using placeholders, Multischeme can implement logic variables (as in Prolog), as well as a variety of other programming constructs.

1.4 Qlisp

In 1984, Gabriel and McCarthy proposed the Qlisp language [8]. The name arises from the queue of processes kept in shared memory, to hold work that is available for

idle processors. As modified in later documents [9], Qlisp is an extension of Common Lisp with several forms for the creation and control of parallel processes.

One of the principal points of the Qlisp philosophy is that process creation can be made *conditional* on arbitrary propositions that may be evaluated during the execution of a program. This is important for several reasons:

- The creation of a process may involve overhead that can be avoided when it does not contribute to increased parallelism.
- Creating too many processes results in excessive use of memory, and for some computations not enough memory will be available unless process creation is controlled at runtime.

Instead of specifying the creation of processes at a point where a function is called, as in the `pcall` and `future` forms of Multilisp, Qlisp's primary construct for creating parallel processes is `qlet`, a parallel form of the `let` form. The expression

```
(qlet prop
  ((var1 form1)
   ...
   (varn formn))
  body)
```

first causes the evaluation of the proposition *prop* to occur, and then either evaluates the forms *form*₁, ..., *form*_n sequentially, if *prop*'s value is `nil`; or evaluates them in separate parallel processes, if it is non-`nil`. Each of these processes returns the value of the form that it has evaluated, and these values are bound to the corresponding variables. When all of the processes have finished, the body of the `qlet` form is evaluated in the context established by these bindings.

Sometimes, additional concurrency may be gained by starting evaluation of the `qlet` body before all of these processes have returned their values. This is the case when, for example, the body performs some initialization before using any of the values, or if a value is used by a function like `cons` that does not need to "touch" it, as discussed above. Evaluation of the body in parallel with the new processes created by `qlet` is specified by having *prop* evaluate to the symbol `:eager`. In this case, Qlisp creates a future, with the same properties as Multilisp's futures, for each of the bound variables, and associates it with the process computing that variable's value. Evaluation of the body then occurs with the variables bound to these futures. Note that it is possible for a process associated with a future to continue executing after the `qlet` form that created it has finished. Therefore, Qlisp must accept the

possibility of accesses to futures occurring anywhere in a program, not just inside the bodies of `qlet` forms.

Although the syntax is different, Qlisp's `qlet` is semantically equivalent to Multilisp's `pcall` and `future` constructs. Each can be written in terms of the others by a simple source transformation.

1.4.1 Process closures

Qlisp introduces several new concepts that are not found in other versions of parallel Lisp. One of these is the *process closure*, defined by the `qlambda` special form. It provides three important features:

1. Encapsulation of a lexical environment. This is done in the same way as ordinary closures provided by Common Lisp's `lambda`.
2. Mutual exclusion. A process closure, which can be called as a function, ensures that its body is not executing in more than one process at a time. Qlisp saves the arguments of calls to process closures that occur while a previous call is still in progress, and evaluates the calls sequentially. The code inside a process closure may therefore perform side effects without the problems of non-determinacy.
3. Parallel execution. When a process closure is called as a function, it immediately returns a future. The caller may then proceed with other work, using the future to represent the value that will eventually be returned by the call to the process closure. If the calling process needs to wait for the process closure to finish, it can pass the future as an argument to a value-requiring form, and thus suspend until the process closure returns a value for the future.

The syntax of `qlambda` is the same as that of `lambda`, except that a propositional parameter is added, as with `qlet`, to allow a runtime decision about whether to use parallelism. If this proposition evaluates to `nil`, then property (3) described above does not hold; the process closure still maintains mutual exclusion, but when it is called, the calling process executes the body of the `qlambda` form. If the proposition is non-`nil`, a new process is created along with the closure for the `qlambda`, and the bodies of calls to the process closure are executed in this process.

`Qlambda` is a higher-level synchronization construct than the locks provided by Multilisp. (Qlisp also provides low-level locks, for applications in which their efficiency is crucial.) The semantics of `qlambda` is very similar to that of monitors [16], combined with the object-oriented features that the encapsulation of a lexical environment provides.

1.4.2 Speculative computation

Another feature that Qlisp provides is the ability to perform *speculative computation*, by starting processes when it is possible but not certain that their results will be needed, and thus making use of processors that would otherwise have nothing to do.

Speculative computation can be used in a problem that has several strategies for solution, if it is not known which one will be fastest on a given problem instance. We may start separate processes to try each of these strategies and use the result of the one that returns first.

When one of these processes finishes, the results of the other processes are not needed, and it is desirable to terminate those processes rather than allow them to continue using resources. The parallel Lisp implementation must provide a mechanism for doing this, and a well-defined way of determining which processes are to be terminated when a given process finishes.

Qlisp accomplishes this by extending the meaning of the `catch` and `throw` constructs of Common Lisp. Some such extension is necessary in any case, because we need to define what it means when a `throw` occurs in one process but the corresponding `catch` was performed in a different process. To resolve this problem, Qlisp applies the principle that (in Common Lisp) each `catch` can be returned from only once, either by a `throw` or by a normal function return. Therefore, any other processes that might cause the same `catch` to return are terminated.

The exact definition of which processes are terminated turns out to be quite complicated, and is explored in [9]. (Along with `catch` and `throw`, the semantics of `unwind-protect` is defined there.) Especially in the presence of side effects and futures, deciding what is the “right thing” to do is not always clear. This area of language design is still undergoing research.

1.5 Other versions of Parallel Lisp

Several other projects have researched or are now working on various aspects of parallel Lisp. These include the following.

1.5.1 PaiLisp

Ito and Matsui, at Tohoku University in Sendai, Japan, have defined a language called PaiLisp, based on the constructs of Multilisp and Qlisp, and implemented a simulator for PaiLisp in Common Lisp [18]. PaiLisp uses Scheme as a base language and adds a “kernel” of four basic forms from which the other parallel Lisp constructs can be defined. The forms in the kernel are:

1. A `spawn` form to spawn a new process;
2. A version of Scheme's `call-with-current-continuation` that works as follows: When a continuation that was created by process *P* is called by process *Q*, process *P* calls the continuation while *Q* continues with its normal continuation;
3. An exclusive function closure (like the `nil` version of Qlisp's `qlambda` form);
4. A `suspend` function to suspend a process. The suspended process can be resumed by calling its continuation.

1.5.2 ZLISP

Dimitrovsky, working with the Ultracomputer Group at New York University, has created a parallel Lisp system called ZLISP that makes effective use of synchronization constructs such as the fetch-and-add instruction of the Ultracomputer. His dissertation [5] describes the implementation of the system as well parallel algorithms and data structures used in ZLISP programs and in ZLISP itself.

1.6 Conclusions on language design

This chapter has been a summary of the work to date in parallel Lisp systems, focusing on extensions to the Lisp language and how they are used in parallel programs. We believe that the various systems described provide a reasonable foundation for experimental work.

Some of the language constructs, such as parallel argument evaluation, futures, and process closures, have gained acceptance and seem appropriate tools for use in parallel Lisp programs. Others, such as the support of speculative computation, are still unsettled.

Experience with parallel programming is beginning to show that while these language constructs are sufficiently powerful, they do not help much in solving the conceptual problems that programmers face when trying to write parallel Lisp programs. Much work needs to be done in discovering "high-level primitives" that remove the programmer from low-level details of process creation and synchronization, while not sacrificing too much efficiency, since fast execution is the reason that most programmers are interested in parallel programming.

Chapter 2

Partitioning and scheduling methods

There are several stages in the preparation and execution of a parallel program:

- *Algorithm design.* The overall design of a program and its data structures will help to determine whether it will run well on a parallel computer, and it remains the programmer's job to specify this aspect of the computation. A parallel algorithm can be translated into a program using the language constructs described in the previous chapter, or using higher-level constructs based on those.
- *Partitioning.* Not all of the possible parallelism in a program may contribute to an increase in its performance. Some process creations may in fact reduce performance, because of the additional overhead. We view the parallel algorithm as specifying *potential processes*, which are computations that may be run in parallel when there is some benefit to doing so. Partitioning is the problem of choosing which potential processes to make into actual processes.
- *Scheduling.* We must also address the problem of which processes to execute at any given time in the computation. Since all of the processors are assumed to be identical in speed and memory access time, it does not generally matter where a given process is run; but the choice of which processes wait while others run may have an effect on the overall execution time.

We view algorithm design and the identification of parallelism as activities that take place before the program is run. These may be done entirely by the programmer or with the help of tools that partially automate the process, including compiler algorithms that detect parallelism. Our main focus is on what happens after the

parallelism in the program is specified. (Some research has considered identifying parallelism at runtime as well, for example the ParaTran system described in [28].)

In our model of parallel execution, both partitioning and scheduling happen at runtime. Others have considered doing one or both of these at compile time [14] [23], and that approach has advantages for programs that have predictable behavior. But for the kinds of problems that Lisp is typically used to solve, important parameters about the input data are often not available until runtime and are necessary for good partitioning and scheduling decisions.

Partitioning and scheduling need not be independent. A decision about whether to create a process may depend on the current state of the system as well as properties of the program and the input data. We will investigate several methods that have this property, as well as methods that separate partitioning and scheduling into independent problems.

Several of the language features described in Chapter 1 make the performance of partitioning and scheduling methods harder to analyze. We can sometimes make stronger statements about programs that avoid these features, so for this reason we define the following categories of parallel Lisp programs.

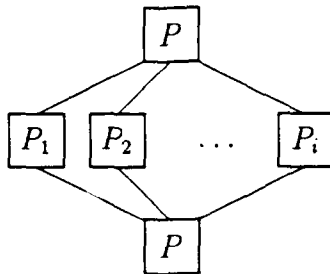
- *Non-speculative programs* are those that require each process that is started to run to completion. The computation is not considered finished until all processes are done, even if the top-level process has returned a value, which may happen when futures are used.
- *Non-eager programs*, in addition to the above restriction, do not use futures. Non-eager programs use only the basic fork/join style of parallelism provided by `pcall` in Multilisp or `qlet` in Qlisp. Programs with futures are harder to analyze because not all of the work done by a process needs to be finished when that process returns a value, and the point at which synchronization occurs may be difficult or impossible to discover by static analysis of the program.

2.1 Computation graphs

A graphical representation of a computation will help in describing some events that take place during its execution, and in analyzing partitioning and scheduling methods. For a given execution of a program, we define a *computation graph*, a directed graph where the nodes are processes or portions of sequential execution within a process, and the edges are precedence relations between them. An edge from node *A* to node *B* indicates that execution of *B* may begin after execution of node *A* is complete.

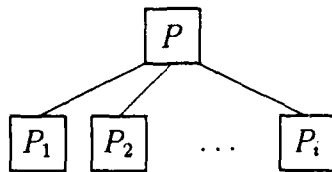
Labels on the nodes in a computation graph are used to identify the process in which they occur (a process's computation might consist of several nodes in the graph), and when necessary to show what parts of the program they correspond to.

We construct the graph of a computation as follows. A potential process that cannot create any subprocesses and that never needs to wait for other processes to finish is a single node. If process P can create processes $P_1, P_2, \dots, P_i$ running in parallel, and then will wait for them to finish (using Multilisp's `pcall` or Qlisp's non-eager `qlet` forms), the corresponding portion of the graph is



Edges in such a diagram implicitly point downward, i.e., the lower node is assumed to depend on the completion of the higher node.

The two nodes labeled P represent different portions of sequential execution within process P . Process P may create subprocesses at several points during its execution, and these result in concatenation of such graphs. Therefore, the computation graphs of programs of this type (non-eager programs) are series-parallel graphs in general. If, however, each potential process has at most one point at which it can create a set of subprocesses, the computation graph will be symmetric—it will consist of an out-tree of process creations (forks) and an in-tree of synchronization points (joins). We will often show just the tree of process creation in this case, and call this a *computation tree* and draw it as follows.



When eager programming constructs are used (Multilisp's `future` and Qlisp's eager `qlet` forms), the computation graphs may become arbitrary DAGs (directed acyclic graphs). The creator of a process need no longer wait for that process to finish; on the other hand, by passing the future as a data value it is possible for more than one node in the graph to require the completion of a given node.

2.2 Notation for describing performance

Here we define some notation that we will use in making statements about program performance. A *program* is a set of Lisp functions that are used to perform some computation. We are generally interested in the behavior of a program over a range of input values and a variety of configurations of a multiprocessor. We express these parameters by variables n , a “size” function of the input, and p , the number of processors.

For given values of n and p , let T_{seq} be the sequential running time (i.e., the time on a non-parallel machine), and T_{par} be the running time on a parallel machine with p processors. We define the *overhead time* to be

$$T_{overhead} = p \cdot T_{par} - T_{seq}.$$

The overhead time summarizes all of the extra time that is spent because of parallelism.

The *speedup* resulting from the use of a multiprocessor is defined as

$$S = \frac{T_{seq}}{T_{par}},$$

and the *efficiency* is

$$E = \frac{T_{seq}}{p \cdot T_{par}}.$$

Perfect speedup on p processors is achieved when $S = p$, or equivalently $E = 1$.

Based on the parallel execution model outlined in Chapter 1, we can divide overhead time into the following categories.

- *Partitioning* contributes to overhead when it is performed at runtime. Let $T_{partition}$ be the total time spent evaluating runtime propositions that decide when to create a process.
- *Process creation* requires a certain amount of additional time for each process that is created, in order to allocate and initialize data structures. If $nproc$ is the number of processes created (as a function of n and p) and T_c is the time required to create a single process, then $T_{create} = nproc \cdot T_c$.
- *Scheduling* requires some time both when a process is created and when a processor becomes idle. Some of this is a fixed cost that we can associate with the creation of a process; we include this in T_c . There is also some variation in scheduling time because of contention for data structures that are shared between the processors. Let $T_{schedule}$ represent this portion of the overhead.

- *Futures* introduce a source of overhead. When the value of a future is required but is not available, a process is suspended and later resumed. The extra work required to perform these operations will be called T_{future} .
- *Idle time* is the amount of time spent waiting for a process to become available when there are none ready to run. T_{idle} is the sum of all of the processors' idle times.

There are no other sources of overhead time because of the simplifying assumptions that we made in our parallel execution model. Therefore we can write the equation

$$T_{overhead} = T_{partition} + T_{create} + T_{schedule} + T_{future} + T_{idle}.$$

For non-eager programs, T_{future} is always zero, but we expect the other forms of overhead to have positive values whenever a program is run on a parallel machine. We make the distinction between running a program sequentially, which involves no overhead, and running it in parallel on a one-processor machine, which might involve all of the above sources of overhead except idle time. In our experiments, the sequential programs are versions of the parallel programs with all runtime partitioning tests and creation of processes removed.

We could go one step further, and insist that a program expressing a given parallel algorithm be compared with the best sequential algorithm for the same problem, even if the sequential algorithm cannot be sped up as much as the parallel algorithm. This would provide a better measure of the true performance gain that parallel execution provides. However, we will not make this comparison because it is not relevant to our main goal, which is to provide the best possible performance for a given parallel program. For this, the appropriate base measure is the sequential version of the same program.

2.3 Partitioning methods

This section describes several partitioning methods that have been proposed for parallel Lisp. We will examine the performance of these methods in Chapter 4.

2.3.1 Height cutoff

Sometimes a potential process is so small that creating a process to run it takes more time than performing its computation sequentially. If an estimate of the running time for each potential process is available, it can be used to avoid creating such processes.

Such an estimate may be derived from information available at compile time as well as at runtime. Compile time information is most useful for potential processes that do not contain recursive function calls or calls to functions whose running time is unknown.

In non-eager programs, this method ensures a bound on the overall cost of process creation. The time T_{create} spent in process creation will be less than the sequential running time T_{seq} , because each process creation, taking time T_c , corresponds to at least time T_c performing useful work in the created process.

In a recursive function, one of the argument values often provides a bound on the running time. We may decide not to create a potential process when this value falls below some threshold. This value corresponds, directly or indirectly, to the height of the computation graph for this potential process, so we call this partitioning method a "height cutoff."

With an appropriately chosen height cutoff, it is possible to bound T_{create} to $\epsilon \cdot T_{seq}$, for arbitrary values of ϵ . The method described above gives $\epsilon = 1$; if instead we require that a process must run for at least time $k \cdot T_c$, then $\epsilon = 1/k$.

However, we cannot simply raise the cutoff level and expect the parallel runtime to decrease for any given input size n . While T_{create} is reduced, T_{idle} will often increase for two reasons. One is that there may not be enough parallel processes to keep the processors busy. The other is that at the end of the computation, there is always some idle time while the "last process" is finishing and other processors are waiting. Increasing the granularity of processes will increase this time, in general.

For a given input data size and number of processors, therefore, there is an "optimal" height cutoff that balances the costs of process creation and idle time, and results in the lowest total overhead. Our experiments will determine this value for some sample programs. One question of importance is whether the cutoff can be made independent of either the input size or the number of processors, so that it can be determined more easily and can be used on a wider range of problem instances.

This analysis does not always apply to programs that use futures. In such programs, creating a process may serve to prevent the parent process from waiting for a value that is being computed, and therefore contribute to increased parallelism. For example, consider the expression

`(cons (times c1 c2) (plus e1 e2)).`

(This expression might occur in the symbolic multiplication of polynomials where $c1$ and $c2$ are coefficients of terms and $e1$ and $e2$ are the corresponding exponents.) If $c1$, $c2$, $e1$ and $e2$ are all small integers then creating a process for `(times c1 c2)` or for `(plus e1 e2)` will probably take more time than the sequential computation

would. But if one of the values is an undetermined future, then doing the computation sequentially will cause us to wait for the future to be determined. We can avoid this waiting by always creating new processes:

```
(cons (future (times c1 c2)) (future (plus e1 e2))).
```

If the values of the arguments to `cons` are not needed immediately, the performance of the program as a whole may improve due to additional parallelism, in spite of the extra process creation cost.

Choosing an appropriate height cutoff is not always easy. The following factors must be taken into account.

- The size of a subcomputation is not always known. In some cases, modifying data structures may solve this. For example, lists that are represented in the usual way by cons cells can be changed to record their length as well. However, this adds overhead cost that we must consider when computing the speedup, i.e., perfect speedup on the program using such data structures is not as good as perfect speedup on the original program, but is the best we can hope to achieve.
- For a given number of processors, as the problem size increases, a height cutoff results in the creation of more processes. The partitioning method does not address the issue of limiting parallelism to conserve resources; its only effect is to put a bound on the overhead of process creation relative to the total amount of work done by the program.

One optimization that we can make when using a height cutoff is to avoid making the partitioning test on potential processes that are below the cutoff level. That is, once we decide to execute a potential process sequentially, we can call a sequential version of its code instead of the parallel version that includes partitioning tests. This change to the program improves its performance by a constant factor, but it can be worthwhile if potential processes become arbitrarily small, as they do in some programs.

We could take this idea further and switch to a better sequential algorithm when a potential process is executed sequentially. This may speed up the whole computation significantly. However, it then becomes harder to determine a meaningful speedup value, so in our examples this optimization will not be made.

2.3.2 Depth cutoff

The height cutoff method decides whether to create a process by determining how far it is from the leaves of a computation tree. Let us consider instead looking the

distance of a potential process from the root of the tree, which we will call the depth of the process. A “depth cutoff” partitioning method will create a process only if this value is less than some threshold.

This method has several benefits. First, the depth is always easy to compute, because it depends on events that have already happened by the time of a partitioning test, rather than events in the future. Second, the structure of the program often determines the number of potential processes at each depth, so we can decide directly how many processes to create by limiting creation to the appropriate depth.

The “optimal” depth cutoff value for a program, unlike the height cutoff value, may be fairly independent of the size of the problem instance, though it will still depend on the number of processors. Our analysis and experiments will show that it is often possible to choose a value that gives good performance for a wide range of problem sizes.

As the problem size n increases, the depth cutoff method does not create more processes. Therefore, the process creation cost T_{create} becomes asymptotically negligible. So do some other components of overhead: $T_{partition}$ because we can switch to sequential code after deciding not to create a potential process, and $T_{schedule}$ because the number of processes scheduled is independent of the problem size. If we can keep T_{idle} from growing as fast as T_{seq} , then we can achieve close to perfect speedup on sufficiently large problems.

The depth cutoff method does have some disadvantages. One is that since we do not consider the size of a subcomputation, only its distance from the root of the computation tree, we may end up creating processes that are too small to be worthwhile. We could use both a depth and height cutoff in the partitioning decision to avoid this, if the information for a height cutoff is available.

Also, the depth cutoff might create a set of processes that is difficult to schedule. This could result in sufficiently large idle time to affect the computation’s speedup noticeably.

Finally, while it is possible to compute a good depth cutoff for a single function, or a small set of functions that call each other to solve a problem, it is not always easy to combine these values into a good partitioning method for a program that contains a collection of functions. Such a program will have several points at which a depth cutoff value is needed, and the appropriate value for each will depend on the rest of the program. The analysis needed to compute these values is often not feasible.

2.4 Scheduling methods

The parallel Lisp systems described in Chapter 1 have been implemented using several scheduling methods. In this section we describe these methods and comment on their advantages and disadvantages.

From the programmer's point of view, there is a single pool of runnable processes, to which new processes can be added and from which idle processors can find work to do. The simplest scheduling methods create a global resource in the system corresponding to this pool of processes. Some of the different ways in which the pool may be handled are:

- A first-in, first-out (FIFO) queue. The process that is removed from the pool by an idle processor is the one that has been in the pool for the longest time.
- A last-in, first-out (LIFO) queue, or stack. The most recently added process is the first one to be removed.
- A priority queue. Each process entered into the pool is given a priority value. The highest-valued process is removed whenever a processor becomes idle. FIFO and LIFO queues are a special case of this, if the priority of the previously entered process is remembered.

There is a problem with all of these methods when the process pool is a single queue: the operations on the queue typically require critical regions of code that cannot be executed concurrently, and introduce the possibility of contention. Certain types of hardware support may reduce or even eliminate these critical regions (see [11]), so single-queue methods may be appropriate for these machines, but we would also like to consider systems where contention is a real problem.

Thus we are led to consider more "distributed" scheduling algorithms, even though shared memory is used. Such methods have been implemented on many of the parallel Lisp systems described in Chapter 1. The most successful of these have a separate queue for each processor, with each processor adding processes that it creates to its own queue and removing processes from its own queue whenever possible. If a processor becomes idle and its own queue is empty, it may remove a process from another processor's queue.

A variety of implementation decisions (FIFO versus LIFO in each queue, which process to take from another's queue, and in what order to examine other queues) can affect the performance of such a scheduling method. It is not clear that a single scheduling method will work well for all programs, so most systems have a default scheduler and provide a way for the programmer to modify it.

In Multilisp, the scheduler maintains a separate queue per processor, using LIFO ordering for processes added to the queues, but taking the oldest process when a processor's own queue is empty and it removes a process from another processor's queue. In addition, Multilisp has a `dfuture` form that behaves like `future` with one difference: using `future`, when a new process is created, the continuation of the parent process is added to the queue and the new process is run, while with `dfuture`, the new process is added to the queue and the parent is continued.

The initial implementation of Qlisp provided only a global FIFO queue. The results of this dissertation and further experiments with Qlisp [22] has resulted in the addition of LIFO scheduling and the ability for the programmer to replace the scheduler with an arbitrary method, to implement multiple queues or experiment with other algorithms.

2.5 Dynamic partitioning

Information from the scheduler can be used to help make partitioning decisions, providing an alternative to the height cutoff and depth cutoff methods described above. We call the use of such information "dynamic partitioning" because the decision to create a process depends on events that occur during the execution of the program; events that can not easily be predicted ahead of time even if the input data is known.

The information that is available to make dynamic partitioning decisions depends on the scheduling method. With a single global queue of processes, we can use the length of the queue as an indication of how busy the system is. As long as processors are idle, the queue length will stay close to zero, because any newly created process will very quickly be removed from the queue. If the queue length is non-zero for a sufficiently long time, we can conclude that all of the processors are busy.

When this happens, an effective strategy may be to let the queue fill up to a certain level and then stop process creation. The exact level at which process creation should stop may depend on the program and the number of processors in the system.

As mentioned above, however, systems based on a global queue of processes may suffer from contention. Contention is avoided with a separate queue per processor, and this scheduling method also has a corresponding dynamic partitioning method. Each processor creates processes and adds them to its own queue, but stops when the length of its queue exceeds a certain threshold. Since there are as many queues as processors in the system, the threshold can be much lower than for a single queue. In fact, if the threshold is zero, i.e., a processor creates a process only if its queue is empty, there may still be as many as p processes available to run in the system as a whole. We will see that this is often enough to satisfy the needs of idle processors.

When using a dynamic partitioning method, the program does not switch to sequential code when it decides not to create a potential process. It is necessary to keep running the parallel version, because although a processor's queue may be non-empty at a time when a partitioning test is made, it may later be emptied by the action of a different processor. After this happens, the dynamic partitioning method will create a new process to refill the queue. We must allow such processes to be created in order for the method to be effective, as we will show later.

There is an obvious objection to dynamic partitioning methods. It is that the partitioning decision is not based on any properties of the potential process being considered, such its size (as in height cutoff) or its position in the computation tree (as in depth cutoff). A process is created only when there is an indication that the system needs additional work to keep busy.

In the worst case, instead of creating a small number of large-granularity processes and performing most of the work inside them sequentially, a dynamic partitioning method may create many small processes. Such an execution of the program will have little idle time, but much of the work done will be wasted overhead.

On the other hand, dynamic partitioning has an obvious benefit. This is that the programmer does not need to determine cutoff values as in the height and depth cutoff methods, which generally requires running the program with test data of various sizes. Performing such tests is time-consuming and not a productive use of programmers' time. In some cases, no useful cutoff values may be available.

Chapter 5 treats dynamic partitioning in more detail, and shows that there are programs, including many that arise in practice, for which dynamic partitioning avoids creating an excess of small processes, and is competitive with the height and depth cutoff methods. In some cases its performance is actually better.

Chapter 3

A Parallel Lisp Simulator

To perform experiments that test the ideas presented in the previous chapter, we wrote a simulator for parallel Lisp, called CSIM. The “C” stands for continuation passing, which is the basic programming technique that our simulator uses to model multiprocessing. CSIM is written in Common Lisp and runs on several systems. It provides the following facilities:

- The user can investigate the effects of varying parameters in a parallel environment, such as number of processors, cost of process creation, and contention for resources. Using CSIM, one can modify these parameters beyond the ranges in currently available hardware.
- CSIM allows metering and performance debugging of programs without modifying them or changing their execution environment. This is easier to do with a simulator than on a real machine. Results on the simulator are also completely reproducible, which is often not the case on actual parallel machines.
- In the absence of an actual multiprocessing system, CSIM can be used as a testbed for parallel Lisp programs.

CSIM was used extensively by the Qlisp project at Stanford until an initial implementation of Qlisp became available, and continues to be a valuable tool in our study of parallel Lisp programming.

The detailed description of CSIM presented in this chapter is not necessary for an understanding of the later chapters, so the reader uninterested in these details may wish to skip ahead to Chapter 4.

The principal disadvantage of using a simulator is that it does not take into account certain aspects of a real system, such as non-uniform memory access time and the cost of garbage collection. CSIM is also much slower than a system running compiled

code. (The slowdown is in the range of 200 to 1000 for most programs.) This limits the size of examples that we can run, but it is sufficient to see many important effects.

3.1 A continuation passing Lisp interpreter

As an introduction to the style in which CSIM is written, we describe here a simple continuation passing interpreter for a subset of Common Lisp. Readers familiar with the continuation passing style of programming may wish to skip this section.

Writing a Lisp interpreter in Lisp is easier than the equivalent task in most other languages, for several reasons. First, the representation of Lisp programs as Lisp data greatly simplifies syntactic analysis. More importantly, the interpreter can be “metacircular,” using parts of the environment in which it runs to simulate the same constructs in the language being interpreted. This lets us focus on the parts of the evaluation process that are of interest. (See [2] for a discussion of metacircular interpreters in Scheme, a simple dialect of Lisp. Our examples will all be based on Common Lisp.)

The main function of the interpreter is `eval`, which takes a form (a Lisp expression representing a program) and an environment (a data structure representing the values of variables), and returns the value of the form in the environment. It usually looks something like this:

```
(defun eval (form env)
  (cond ((symbolp form)
        (lookup-variable form env))
        ((atom form)
         form)
        ((special-form-p form)
         ...)
        (t (apply (first form)
                    (eval-list (rest form))))))

(defun eval-list (formlist env)
  (if (null formlist)
      nil
      (cons (eval (first form) env)
              (eval-list (rest form) env))))
```

This program is not yet complete. In place of the ‘...’ must be inserted code to handle all of Lisp’s special forms. We also need an implementation of environments,

and we need to define the functions `lookup-variable` and `apply`. These involve details that are unimportant at this point.

The above interpreter is a *functional* program, and its runtime behavior follows the pattern of function calls and returns in the program being interpreted. For a subset of Lisp restricted to functional constructs, such an interpreter is fine. However, it becomes increasingly hard to maintain the simple structure of the interpreter as we add Common Lisp's special forms for sequencing (`progn`), iteration (`tagbody/go` or `do`), and non-local return (`catch/throw` and `block/return-from`), as well as the parallel constructs that we will introduce.

Using *continuations* allows us to expand the range of constructs that the interpreter can handle with a manageable increase in the complexity of the program. Continuations, which were originally invented to define the semantics of sequential programming constructs (see [10] and [27]), were shown in [25] and related papers to be a very convenient programming tool as well.

A continuation is a function that represents "the rest of the program" as the interpreter progresses. The interpreter's job changes from "evaluate a form in an environment and return the result" to "evaluate a form in an environment and call a continuation with the result." Using continuation passing style,¹ our example becomes:

```
(defun eval (form env cont)
  (cond ((symbolp form)
        (funcall cont (lookup-variable form env)))
        ((atom form)
         (funcall cont form))
        ((special-form-p form)
         ...)
        (t (eval-list (rest form) env
                       #'(lambda (args)
                           (apply (first form) args cont))))))

(defun eval-list (formlist env cont)
  (if (null formlist)
      (funcall cont nil)
      (eval (first form) env
            #'(lambda (first-value)
                (eval-list (rest formlist) env
```

¹The reader familiar with continuation passing style will notice that some parts of this code do not pass continuations; for instance the `lookup-variable` function. We do this to improve the performance of the interpreter by creating fewer unnecessary closures.

```

#'(lambda (rest-values)
      (funcall cont (cons first-value
                           rest-values))))))

```

It is important to notice that the functions defined above by `lambda` expressions are closures; they contain free references to variables that are lexically bound outside the `lambda` expressions.

In a continuation passing program such as this one, each function that is called with a continuation as an argument ends by calling another function, passing it a new continuation. If the interpreter is run using an ordinary stack-based Lisp system, the stack will grow quite large, and any program doing a non-trivial amount of work will cause the system to run out of memory. To avoid this, the Lisp system in which the interpreter is run must detect *tail recursion* and cause stack space to be reused whenever such a call is encountered. While coding the interpreter, the programmer must ensure that all functions called with continuations are tail-recursive.

Let us go through a simple example to illustrate how the continuation passing interpreter works. Suppose we want to evaluate the expression `(+ x 3)` and print the result. Previously, we would have said

```
(print (eval '(+ x 3) *top-level-env*))
```

where `*top-level-env*` is used to hold the "top-level" environment of values assigned to global variables. Let us assume that it associates `x` with the value 4. With the continuation passing interpreter, we say

```
(eval '(+ x 3) *top-level-env* #'print)
```

This call to `eval` examines the form `(+ x 3)`. It is not an atom or a special form, so it results in a call to

```
(eval-list '(x 3) *top-level-env*
  #'(lambda (args) (apply #'+ args #'print)))
```

The quoted expressions in the above call and the rest of this example are used to represent the values that will actually be passed. The original continuation `print` has become part of a new continuation (the `lambda` expression above). `Eval-list` now calls

```
(eval 'x *top-level-env*
  #'(lambda (first-value)
        (eval-list '(3) *top-level-env*
          #'(lambda (rest-values)
```

```
(funcall #'(lambda (args)
              (apply #' + args #'print))
  (cons first-value
        rest-values))))))
```

which has constructed a new continuation that contains the old one buried inside two levels of closures! But now we have called `eval` with an atom, and it calls

```
(lookup-value 'x *top-level-env*)
```

to find the value associated with `x` in `*top-level-env*`. This will return `4`. Then `eval` will invoke

```
(funcall #'(lambda (first-value)
              (eval-list '(3) *top-level-env*
                #'(lambda (rest-values)
                    (funcall #'(lambda (args)
                                (apply #' + args #'print))
                      (cons first-value rest-values))))))
  4)
```

This becomes

```
(eval-list '(3) *top-level-env*
  #'(lambda (rest-values)
      (funcall #'(lambda (args) (apply #' + args #'print))
        (cons 4 rest-values))))
```

so we are making some progress. After several more steps similar to those above, the interpreter will invoke

```
(funcall #'(lambda (args) (apply #' + args #'print))
  (cons 4 '(3)))
```

and finally

```
(apply #' + '(4 3) #'print)
```

The continuation passing version of `apply` (which we haven't yet defined) will call the continuation `#'print` with the result of applying the function `#' +` to the argument list `'(4 3)`, so it will finally call `(print 7)` and display the answer.

3.2 An interpreter for Common Lisp

We now extend the simple continuation passing interpreter to one that accepts almost all of Common Lisp. This will be the basis of our parallel Lisp simulator. To avoid discussing various unimportant details, the code described in the next few sections is often a simplification of what actually appears in CSIM.

3.2.1 Environments

Symbols in Common Lisp programs refer to values based on the rules of *scope* and *extent* as described in [24], ch. 5. While it would be possible to pass in a single *env* variable all of the information needed to resolve any symbol reference, CSIM divides the kinds of references into two classes, *lexical* and *dynamic*, and uses variables *lex-env* and *dyn-env* to store different parts of the environment. The pragmatic reason for this separation is that a call to a new function defined at “top level,” which is a frequent occurrence, uses none of the lexical information present in its calling environment, but retains all of the dynamic environment.

Lexical environments are represented by structures with four components:

- **variables** that are lexically bound, for example as function parameters or by *let*. What is actually stored is an association list (*alist*) of (*variable* . *value*) pairs. Since lexical binding is the default in Common Lisp, most variable references will be found here.
- **functions** defined by *flet* or *labels*. This slot contains an alist that associates each name with a lexical closure (see definition below), since lexically bound functions can have free variable references.
- **blocks** defined by *block*. Also contains an alist, which is described in more detail in Section 3.2.4.
- **tagbodies** defined by *tagbody* (or implicitly by *prog*, *do*, etc.) This slot contains a list each of whose members is the entire body of a *tagbody* form.

Lexical closures are represented by structures with two components:

- **function**, represented by a *lambda* expression.
- **environment**, a lexical environment.

Dynamic environments are represented by structures with three components:

- **variables** that are "special," and hence dynamically bound (an alist).
- **catches**, information about **catch** forms that have been entered and not yet exited.
- **unwinds**, representing **unwind-protect** forms that are pending.

A new environment is created whenever there is a new piece of information to add to an existing environment. For example, to interpret a **let** form that binds lexical variables, we create a new lexical environment structure, copy the slots that have not changed from the existing environment (**functions**, **blocks**, **tagbodies**), and store in the **variables** slot an alist that begins with the variables being bound and eventually shares the list structure of the variables in the original environment. We create a new environment, rather than change the slots in the existing environment structure, because the extent of each binding in Lisp is finite and the binding must at some point be "undone;" the best way to do this is to preserve the environment existing before the binding.

Sometimes we modify the data structures contained in an environment without changing the environment itself. For example, to interpret **setq** we find a (*variable . value*) pair in an environment and destructively modify the value part of this cons cell.

3.2.2 The global environment

CSIM does not use the environment structures just described to implement Common Lisp's "global environment," consisting of values and functions assigned to unbound special variables (**symbol-value** and **symbol-function**). When simulating a reference or assignment to an unbound symbol's value, we use **symbol-value**, which lets the simulated program share the global environment of the simulator.

This makes using CSIM more convenient, because assignments to global variables can be made in the ordinary Lisp environment and then be seen by simulated code, or vice versa. Doing this for function definitions would cause difficulties, however (since CSIM provides interpreted definitions for many of the predefined Common Lisp functions), so these are stored on the symbol's property list.

3.2.3 Function application

Let us now look further into CSIM's **apply** function, which has been mentioned several times but not yet defined. The role of **apply** is to take a function object, a list of

argument values, a lexical and dynamic environment, and a continuation, and to call the continuation with the result of the function applied to the arguments.

The function objects that `apply` allows as its first argument fall into the following classes:

1. Symbols naming primitive Common Lisp functions. These functions are called directly by the simulator.²
2. Symbols naming Common Lisp functions that must be treated specially. For example, an instance of `eval` in code being simulated should result in a call to CSIM's `eval`, not the `eval` in the underlying Common Lisp.
3. Symbols naming functions whose definitions should be interpreted. CSIM finds the definition for such a function on the symbol's property list, where it will have been stored as a `lambda` expression by CSIM's version of `defun`, and applies it in a null lexical environment and the current dynamic environment.
4. Explicit `lambda` expressions. These are applied in the current lexical and dynamic environment.
5. Closures. These are represented by structures containing both a `lambda` expression and a lexical environment in which to apply it. The current dynamic environment is used.

Applying a `lambda` expression is fairly straightforward. We create new environments to contain the bindings of the `lambda` expression's variables. (Since some of them may be special variables we may create both a new lexical environment and a new dynamic environment.) In the new environments, we associate the variables with the corresponding values taken from the argument list to `apply`. Finally, we evaluate the body of the `lambda` expression in the new environments. Since its value should be passed to the continuation that was given to `apply`, we use this continuation in the call to `eval` for the body. A skeleton of the code for this is:

```
(let ((new-lex-env ...)
      (new-dyn-env ...))
  (eval <body-of-lambda-expr> new-lex-env new-dyn-env cont))
```

In the actual simulator, the application of `lambda` expressions is more complicated because we interpret Common Lisp's `&optional`, `&rest` and `&aux` parameters, and avoid creating new lexical or dynamic environments when not necessary.

²The dynamic environment in which these calls take place will not correspond to the simulated dynamic environment. The user of CSIM must expect dynamic binding of variables to affect only references that are interpreted by the simulator.

3.2.4 Special forms

As an example of how continuations simplify the simulation of Common Lisp special forms, let us look at the implementation of `block` and `return-from`. In a program such as

```
(block b1
  (foo (block b2
    (if p (return-from b1 7) 3))))
```

if the value of `p` is `nil`, the inner block will return 3 and the outer block will compute `(foo 3)`. But if `p` is not `nil`, the `return-from` form will cause 7 to be immediately returned from the outer block and `foo` will not be called. The symbol `b1` in the `return-from` matches the name of the outer block because it is lexically contained within that block, but if the inner block were also named `b1` then the `return-from` would match the inner block's name.

Each lexical environment includes a `blocks` slot. To interpret a block form, we create a new lexical environment; in the `blocks` slot of this environment we put a list whose first element represents the block we are interpreting; the rest of the list is the `blocks` slot from the previous environment. With the block name we associate the continuation for the block, because this represents what we want to do with the value returned by the block, whether it comes from the last form in the block or is supplied by a `return-from`.

The code to interpret a block form is therefore

```
(defun eval-block (form lex-env dyn-env cont)
  (let ((new-lex-env (copy-lex-env lex-env)))
    (push (cons (block-name form) cont)
          (lex-env-blocks new-lex-env))
    (eval (block-body form) new-lex-env dyn-env cont)))
```

and the code to interpret a `return-from` form is³

```
(defun eval-return-from (form lex-env-dyn-env cont)
  (let ((find-block (assoc (return-block-name form)
                           (lex-env-blocks lex-env))))
    (if find-block
        (eval (return-expr form) lex-env dyn-env
              (cdr find-block))
        (error "No block for ~S" form))))
```

³The code as shown here is incomplete, because it doesn't handle `unwind-protect` forms that may have to be evaluated as a result of a `return-from`. CSIM does handle this case.

When `return-from` is seen, the interpreter looks through the list of blocks in the current lexical environment, which will have the innermost blocks listed first. It examines block names (using `assoc`) until it finds one matching the name in the `return-from`. The continuation that is associated with this block name is the one to which we want to pass the return value. Therefore we end with a (tail-recursive) call to `eval` using this continuation. Note that the `cont` argument to `eval-return-from` is ignored. This is because `return-from` never returns a value to its caller; it always passes a value to some other continuation.

If no `return-from` is encountered in the course of evaluating the body of a block, then the evaluation of `(block-body block)` will eventually call `cont` with a value, as expected.

`Catch` and `throw` are simulated in a very similar way. `Catch` saves its tag and continuation in a new dynamic environment, and `throw` looks for the appropriate continuation by matching its tag to those saved in its dynamic environment.

`Unwind-protect` is not hard to handle, although it must be coded quite carefully. The main idea is that every time an `unwind-protect` form is evaluated, a new dynamic environment is created; its `unwinds` slot contains a list with the cleanup forms and the lexical and dynamic environments in which they must be executed. Upon normal return through an `unwind-protect` these forms are evaluated in a straightforward way. A non-local exit (caused by `throw`, `go` or `return-from`) causes a change from the current dynamic environment to a previous dynamic environment. When this happens, we evaluate all of the cleanup forms associated with environments between the one we are leaving and the one we are returning to, in the proper order.

Another important special form is `setq`. For the moment, the following code will suffice to simulate `(setq var value)`:

```
(defun eval-setq (form lex-env dyn-env cont)
  (eval (third form) lex-env dyn-env
        #'(lambda (value)
              (modify-binding (second form) value lex-env dyn-env)
              (funcall cont value))))
```

`Modify-binding` finds the association-list pair for the variable in the appropriate environment and changes the value. In Section 3.4 we will make some additions to this code.

Most of the remaining special forms of Common Lisp perform various operations on environments; they are straightforward to implement so we omit them from the description here.

3.2.5 Multiple values

Common Lisp's multiple values are supported by CSIM. We have previously defined a continuation to be a function of one variable, and simulated returning a value from a function call by calling a continuation with the value that is returned. To allow multiple values to be returned, we let a continuation be called with any number of arguments.

Instead of a function with one parameter, we let each continuation be a function with a `&rest` parameter. When the continuation is called, the `&rest` parameter variable is bound to a list of the argument values. It is then straightforward either to use just the first element of this list when only one value is expected, or to use the whole list in the places that allow multiple values.

The initial implementation of CSIM was done without supporting multiple values. When it came time to add this feature (because some programs that we wanted to simulate used multiple values), it took very little effort to do so.

We will not mention multiple values in the remainder of this chapter since in general they are not relevant to issues of parallelism.

3.2.6 Timing statistics

Up to now, we have not made CSIM do anything more than the Lisp system that it is built on. The first feature that we will add is the ability to measure and record "simulated" execution time. This meets one of our initial goals, which is to reflect the timing of computation on an actual or hypothetical machine.

We use a global variable `*time*`, which is initialized to 0 at the start of each "top-level" call to the interpreter. Whenever CSIM performs an operation that reflects work in the simulated machine, it adds an appropriate amount to `*time*`. (Section 3.6.1 explains how the basic timings are chosen.) When the computation is done, we can see how much work our simulation corresponded to.

A benefit of the simulator is that we can gather some statistics that would be hard to obtain in a real machine without affecting the timings. For example, we keep track of how much work is spent in each function, in addition to the total work done. This cannot generally be done on standard hardware without, for instance, having the compiler generate additional code at each function entry and exit; this extra code will affect the statistics. Worse, from our point of view, it will affect the relative timing of activity in a parallel processor and possibly change the amount of speedup for the program.

CSIM keeps track of time spent in functions in three different ways. The first is the time spent in each function exclusive of the functions that it calls. These timings

will add up to the total time spent in the program.

A more useful statistic is obtained by counting all of the computation done in a function, including functions that it calls. When a function recursively calls itself (either directly or with calls to other functions intervening), we must decide whether to charge it only once, or once for each call. CSIM actually does both, because a different useful measure is obtained each way. These are the second and third sets of function timings.

The information needed to compute these timings is stored in extra slots in each dynamic environment structure. One slot contains the name of the current function being simulated; it is used to charge time to just that function. The second slot contains a list of all function calls currently in progress. The third slot contains such a list, but with each function appearing only once.

When a basic operation is simulated, CSIM adds its simulated time to the time for the current function, and the times for functions in the two lists. For each of the three statistics there is a hash table indexed by the function's name and containing its accumulated time.

After a top-level form has been simulated, CSIM optionally prints the timings. The timings for functions in which recursive calls are counted more than once are not useful by themselves (some may be more than 100% of the total simulated time), but when divided by the number of calls to the function, they give the average time spent in that function.

For example, suppose we have the sequence of calls

$$FOO \rightarrow FOO \rightarrow FOO \rightarrow BAR,$$

in which each call to `FOO` takes 10 steps before calling the next `FOO` or calling `BAR`, and the call to `BAR` takes 40 steps. Thus the total computation takes 70 steps. The first statistic would show 30 steps spent in `FOO` and 40 steps spent in `BAR`.

The second statistic would show 70 steps spent in `FOO` (since all the work is done within the toplevel call to `FOO`), and 40 steps spent in `BAR`. The third statistic would show an average of 60 steps spent in `FOO`, since 70 steps are charged to the first call, 60 to the second, and 50 to the third. It would also show 40 steps spent in `BAR`.

Section 3.7 will describe how we use these statistics.

3.3 Parallel Lisp constructs

Of the parallel Lisp constructs described in Chapter 1, we have implemented the following in CSIM:

1. Qlisp's `qlet` (both regular and `eager` forms) and `qlambda`.
2. Multilisp's `future`, `dfuture` and `touch`.
3. Simple test-and-set locks (busy waiting).

We do not yet support the extensions to `catch` and `throw` defined by Qlisp.

Locks are provided as a low-level synchronization primitive for two reasons: first, they are better suited for certain parallel algorithms than futures (particularly for "in-place" algorithms that destructively modify data structures); and they are needed to write the scheduler, as described in Section 3.4.2.

3.3.1 Scope and extent issues

The definitions of scope and extent for variables and other objects in Common Lisp require some reinterpretation in parallel Lisp. This was foreseen in [24, p. 38], where Steele writes:

Behind the assertion that dynamic extents nest properly is the assumption that there is only one program or process. Common Lisp does not address the problems of multiprogramming (timesharing) or multiprocessing (more than one active processor) within a single Lisp environment.

We have chosen the following policies:

- Lexical variable references behave the same as in Common Lisp, even if the binding of a variable is in a different process from the reference. Thus, in

```
(qlet t ((x (let ((v 5)) (foo v)))
        (y (let ((v 4)) (bar v))))
...)
```

there is no relation between the binding of `v` in the two processes created by `qlet`, while in

```
(let ((v 5))
  (qlet t ((x (foo v))
          (y (bar v)))
    ...))
```

the two references to *v* are both to the binding established by the `let`. If one of the processes used `setq` to change the value of *v*, the new value would be seen in the other process (and in the body of the `qlet`).

If the parameter *t* in `qlet` is changed to `'eager`, then the process computing the body may return from the `qlet` even though the processes computing the bindings are still running. In this case, the variable *v* must remain accessible to these processes. (The same situation can occur if `future` is used.)

CSIM has no problem implementing this, because it uses list structure to store lexical environments and never explicitly deallocates them. (They are garbage collected once they are no longer needed.) An efficient parallel Lisp implementation might avoid allocating environments when possible, but will have to use a lexical closure to allow the passing of bindings from a parent process to a child in this manner.

- The dynamic environment of a process cannot be changed by other processes, even when a binding is undone in a process. If we change our first example to

```
(defvar v)
(qlet t ((x (let ((v 5)) (foo v)))
        (y (let ((v 4)) (bar v))))
...)
```

then the two bindings of *v* are independent, even though they may occur concurrently. The “shallow binding” technique used by many Lisp implementations does not do the right thing in this case; each process would try to store its new value for *v* into a shared global value cell. Deep binding, on the other hand, does work correctly if each process is provided with its own stack for bindings, and inherits the bindings of its parent process. However, in the case

```
(defvar v)
(let ((v 5))
  (qlet 'eager ((x (foo v))
                (y (bar v)))
...))
```

we want the binding of *v* to be accessible to the processes created by the `qlet` even after the `qlet` returns. This is a problem, since the process that established the binding now will undo it. In a stack-based implementation of dynamic binding, even with deep binding, this will not work. CSIM uses list structure to implement its dynamic environments, just as with lexical environments, and hence does what we want.

3.4 Simulating the parallel machine

Our main concern in simulating a multiprocessor is that we accurately model the order of reads and writes to the shared memory. Although parallel programs that share data generally use synchronization constructs such as futures or locks, we want to produce realistic results for programs that make unsynchronized memory references. (Among other benefits, this will help us find bugs in programs that do not use correct synchronization.)

In Sections 3.1 and 3.2 we described how our single-processor interpreter keeps track of its progress using continuations. This takes the place of the “control stack” in an ordinary interpreter, and consequently it is very easy to capture the interpreter’s state. This design lets us have an interpreter for each processor in the simulated machine, and switch between them whenever we want.

We do this by introducing a new kind of continuation, which we call a *process continuation*. Process continuations are closures with no parameters; their purpose is solely to capture the lexical environment of the interpreter at a point where we wish to switch the simulation to a new processor, so that we can later resume the current processor’s simulation. (In [29], continuations created by `catch` in the then-current version of Scheme were used for much the same purposes as our process continuations.)

For example, the code to handle `setq` that was presented in Section 3.2.4 is modified in the parallel simulator to

```
(defun eval-setq (form lex-env dyn-env cont)
  (eval (third form) lex-env dyn-env
        #'(lambda (value)
              (switch-processors
               #'(lambda ()
                    (modify-binding (second form) value
                                   lex-env dyn-env)
                    (funcall cont value)))))))
```

where `switch-processors` is a function that does what we have been describing. Its argument is a process continuation that captures the necessary parts of the simulator’s state in its free variables. Calling the process continuation will resume the interpretation of the `setq` form, but the `switch-processors` function can defer this call until the appropriate time to do so.

Process waiting is also simulated using process continuations. When a process needs to wait for an event (such as a future’s value being determined, or to call a `qlambda` process closure), the simulator stores a process continuation representing

the work to be done after that event happens, in a data structure associated with the waiting process. Calling the process continuation resumes the suspended process.

3.4.1 Processors

The variable `*number-of-processors*` is used at the beginning of each top-level evaluation to determine how many processors to simulate. Each processor is always running a process, possibly an “idle” process. A processor is represented by a data structure containing its current process and its current simulated time.

The simulated times are the key to deciding when to switch the simulation from one processor to another. As long as CSIM performs operations that can have no effect on processors other than the current one, it continues to simulate the same processor, incrementing that processor’s time.⁴ The only operations by which one processor can affect others are those that read or write data in shared memory. To make sure that these operations are done in the correct order, CSIM enforces the following rule:

Any operation that can affect other processors must be done when the current processor’s time is the lowest of any in the system.

To see why this works, consider two processors, P_1 and P_2 , that perform shared-memory operations at times t_1 and t_2 , with $t_1 \leq t_2$. Without following the rule above, we might run the simulation of P_2 beyond time t_2 before we have simulated P_1 at time t_1 . This would be wrong: for instance, if P_1 ’s operation is a write and P_2 ’s is a read of the same memory location, then we would not read the correct value. (We call this a write/read conflict. Read/write or write/write conflicts cause similar problems.) However, because of the above rule this cannot happen. When we see that P_2 is about to perform a memory operation at time t_2 , we stop its simulation. We do not restart it until it has the lowest simulation time of any processor (or is equal to others with the same time). At that point, P_1 must have been simulated past time t_1 , because if it hasn’t been, then its time is less than t_1 and $t_1 \leq t_2$, so P_2 ’s time isn’t the lowest.

What this does is *serialize* all of the shared-memory operations that can cause one processor to affect another. We do this for unsynchronized memory operations (i.e., ordinary reads and writes) as well as synchronous operations such as acquiring locks. This ensures that our simulation corresponds to the order of operations that would occur in a real multiprocessor. However, we do not place any restrictions on shared-memory operations performed at the exact same time by two processors. The results of these are unpredictable.

⁴Actually, it increments the global variable `*time*`, and will store its value back into the processor’s structure before switching to a new processor.

Serialization is implemented by means of a priority queue (called the “run queue”) that holds the structures representing processors, sorted in increasing order of simulation time. When the interpreter is about to perform a shared memory operation (for instance, at the call to `switch-processors` above), it updates the data structures for the current process and processor and inserts the processor into the run queue. Then, the processor with the lowest simulation time is removed from the run queue and its simulation is resumed.

CSIM’s serialization method was chosen because it is easily to implement and prove correct. Since CSIM does not itself attempt to do work in parallel, this is a reasonable choice. Serialization would become a bottleneck if we were to try to speed up CSIM by having it simulate several processors at the same time, and we would probably need to use a more sophisticated mechanism, such as the “time warp” system described in [19].

3.4.2 The scheduler

As described at the beginning of Section 3.3, we assume there is a queue or some other data structure to hold processes that are ready to run. We call the code that maintains this data structure the *scheduler*, since it decides in what order the processes will run.

Scheduling algorithms are one of our objects of study, and we do not want to build one into the design of our simulator. Instead, we want to make it possible for a user of the simulator to write a scheduler in ordinary Lisp code (not in continuation passing style). CSIM models the execution of the scheduler by simulating it in the same way as other Lisp code.

The scheduler consists of two functions:

- `add-process` is called whenever a new process is created. It is given a process as its argument, and inserts it into whatever queue or other data structure is being used to schedule processes.
- `get-process` is called whenever a processor is idle. It finds a process to run and returns it.

When a processor becomes idle, the simulator creates a temporary “idle” process in which the call to `get-process` takes place. (Since this call is interpreted, there must be a process for it to run in.) Upon return from `get-process`, the new process replaces the idle process.

Currently, CSIM gives the user a choice of four schedulers: FIFO, LIFO, FIFO* and LIFO*. FIFO and LIFO each use a single global queue, while FIFO* and LIFO*

implement a separate queue per processor. LIFO scheduling also allows some optimizations in process management.

1. When a process is about to create a child process and immediately wait for its result, as in the `(qlet t ...)` construct of Qlisp, it can perform an ordinary function call instead, since there is no reason to put a process on the queue, make the processor become idle, and have it then remove the same process right away.
2. When a process finishes and has a list of waiting processes to wake up, its processor can put all but one of them on the process queue and run the last one itself, since it otherwise would become idle and immediately choose the last process that it added.

CSIM has a flag that is turned on by the LIFO scheduler, and turned off by the FIFO scheduler, which enables these optimizations. This interaction between the scheduler and the simulator is needed because creation and termination of processes are simulated directly, not interpreted as the scheduler is.

3.4.3 Processes

A process is represented by a structure containing its current process continuation, a flag to indicate whether it has terminated, and a list of other processes that are waiting for it (if it has not yet terminated). When interpreting a form that creates a process, such as `qlet` or `future`, the simulator calls a function `create-process` defined as follows:

```
(defun create-process (form lex-env dyn-env new-cont cont)
  (let ((new-proc (make-proc
                      :pcont #'(lambda ()
                                (eval form lex-env dyn-env
                                      new-cont))))))
    (call-user-function 'add-process (list new-proc)
      #'(lambda (v)
          (funcall cont new-proc)))))
```

The `form` argument is what the new process will evaluate, using `lex-env` and `dyn-env` as its initial environment. `New-cont` is the continuation that the new process will call with the value of `form`. `Cont` is the continuation for the parent process. The call to `call-user-function` tells CSIM to interpret the definition of `add-process` with the new process as an argument, and pass the result to the specified continuation. This

continuation returns the new process to the parent, which may have a need to refer to it. (For instance, `qlet` may wait for the new process to finish.)

The continuation `new-cont` called by the new process is always written to end with a call to the function `finish-process`, which wakes up any processes that have decided to wait for the given process to terminate. It does this by calling `add-process` on each of these. After this, the process is done. Its processor becomes idle and will try to find a new process. If we are using the LIFO scheduler described in the previous section, then if there were waiting processes we switch directly to one of them, avoiding a call to both `add-process` and `get-process`.

3.4.4 Process closures

Qlisp defines a new type of object called a *process closure*, which provides both concurrency and synchronization. A call to a process closure may proceed without the caller waiting for the result (but only when the call is in a position where the result value is ignored). Calls to each process closure are serialized; if one happens while a previous call is still in progress, it is put on a queue.

At present, CSIM implements only the synchronization features of process closures. To do this, we represent a process closure by a structure containing a (first-in, first-out) queue of waiting processes and a closure. When a process closure is called, the calling process is added to the queue. If it is the only one there, it proceeds by calling the closure. Otherwise, its processor becomes idle and calls `get-process` as described above.

When the call to the closure returns, the simulator removes the current process from the process closure's queue. If there are now other processes waiting on the queue, it calls `add-process` to resume the first one.

3.5 Miscellaneous details

In the previous sections, we omitted certain details in order to simplify the presentation. This section explains several of them.

3.5.1 Use of symbols

We find it convenient to have the interpreter share symbols with its underlying Common Lisp environment. As mentioned in Section 3.2.2, the values of unbound special variables are shared between the simulator and the program being simulated. Other

information about symbols is kept on their property lists, using the following property names:

- **cexpr** is the **lambda** expression for an interpreted function definition. CSIM's version of **defun** sets the value of this property.
- **csubr** is a function to handle a special form. It is called by CSIM's **eval** to handle such a form, with the form, the current lexical and dynamic environments, and the current continuation as arguments. The **defcsubr** macro defines such a function. This makes the code more modular, since we do not need to enumerate all of the special forms inside **eval**.
- **esubr** is a function to handle a primitive Lisp function that cannot be called directly, such as **apply**, because its operation needs to be simulated. The arguments to an **esubr** are evaluated normally before the function is called.
- **cinfo** is a list whose **car** is the number of time units that the simulator should charge to interpret the function or special form named by this symbol, and whose **cdr** indicates which arguments to the function may be passed without being touched. These are both meant mainly for functions that are interpreted by calling the Common Lisp functions directly. Functions that are simulated (using either **cexpr** or **csubr** definitions) do not make use of the **cdr** part of this property. For interpreted **cexpr** definitions, if this property is present it overrides the normal time charged for a function call.

3.5.2 Preprocessing of definitions

CSIM has its own version of **defun**, which stores the function definition of a symbol as its **cexpr** property. Before doing so, it preprocesses the function definition to perform the following transformations.

1. Macros are expanded wherever they are recognized. This avoids having to expand them in the interpreter.
2. Parallel Lisp constructs (**qlet**, **future**, etc.) are converted to a form that assigns a unique tag to each process creation point. For example, if a function **foo** contains several calls to **future** and the second one is **(future expr)** then it is converted to **(future-tagged foo2 expr)**. When **future-tagged** is interpreted, it is treated just like **future** except that the tag is assigned to the new process that is created. CSIM keeps track of how much time is spent in each call to a process with a given tag. With this information the user can decide

whether the processes created at each point in the program are of a reasonable size.

It is possible for a macro to be encountered in interpreted code even after preprocessing; if this occurs, CSIM expands it and then destructively replaces the original form by the expansion. This avoids the overhead of expanding the same expression each time it is encountered.

Many of the basic Common Lisp forms described in [24] are macros. Unfortunately, different implementations of Common Lisp expand these forms in different ways, causing noticeable changes in the times charged by the simulator. Even more of a problem is that the expansions may use implementation-specific functions. Because of this, CSIM must include timing information for functions that are not part of standard Common Lisp, particularly those resulting from expansions of `setf`.

CSIM also has a special version of `defstruct`, so that it can perform preprocessing to define timing information for the accessor, constructor, copier and predicate functions of the structure being defined.

3.5.3 Interpreted primitives

Many Common Lisp functions may be simulated by calling them directly with the values of their argument expressions. CSIM must be careful not to pass futures to these functions, because they are not part of Common Lisp. Therefore in most cases it “touches” arguments before calling a Common Lisp function. This would have to be done anyway in a Lisp system that uses futures, except for functions such as `cons` that do not depend on the values of their arguments; for those cases we have included a mechanism (the `cinfo` property described above) to avoid unnecessary touches.

Some functions cannot be called directly, however, because they reference objects other than their direct arguments, and these may be futures. Consider, for instance, the `cddr` function (`cdr` applied twice). Even if we ensure that the argument to `cddr` is not a future, it may be a cons cell whose `cdr` is a future, so calling Common Lisp’s `cddr` would result in an error. CSIM uses an interpreted definition of `cddr` and most other such functions for this reason.

Another class of functions that cannot be called directly is those whose running time depends on the size (or some other properties) of their input. The `equal` and `length` functions are examples of this. CSIM uses interpreted definitions of these functions also.

3.5.4 Top level

To interact with the user, CSIM provides a read-eval-print loop, but the top-level evaluator does a number of special things. It begins by initializing the processor data structures and clearing all of the statistics counters. Then it creates an initial process whose process continuation is set to call CSIM's `eval` with the form to be evaluated. This process is passed to initialization code for the scheduler, which sets it up as ready to run, with no other processes in the system. One of the processors is then chosen to begin the simulation.

While running, CSIM keeps track of the number of running processes. A process is said to be running between the time it is created and the time it terminates, except when it is suspended to wait for an event (such as a future being determined). If the number of running processes drops to zero, we halt the simulation and return to top level. Usually this happens when the top-level process returns a value (which is then printed) and terminates. But there may still be other running processes at this point, because of futures that have not yet been determined, or for other reasons. In this case, we continue simulation until the number of running processes is zero.

3.5.5 Memory allocation and garbage collection

CSIM calls the underlying Common Lisp functions (`cons` etc.) to simulate memory allocation by the program we are interpreting, and assumes that each such call takes a constant amount of time in a parallel machine. A real parallel Lisp can achieve this by giving each processor a private pool of free cells to allocate from, so this is realistic.

CSIM does not model garbage collection at all, except to estimate its eventual cost and include this in the simulated times for `cons` and other allocation functions. It assumes that the garbage collector will achieve the same amount of parallelism as the rest of the program.

Parallel garbage collection is an important problem and there are many approaches currently under investigation. However, we view this area of research as orthogonal to our main interests, which are modelling the execution of processes and investigating partitioning and scheduling algorithms.

3.6 Accuracy and performance

To produce meaningful results, CSIM's timings must approximate those of an actual machine. And to be usable, the simulator must be fairly fast. This section describes the results of some experiments done to see how well it meets these goals.

3.6.1 Accuracy of simulated times

To derive timings for basic Lisp operations, we compiled and ran a set of small test programs. Each consisted of a loop performing a primitive Lisp operation; one of these was a "no-op" to measure the overhead of the loop code. Subtracting the time for the "no-op" test from the time of each other test, and dividing by the number of iterations of the loop, indicated how much time was spent in each function being tested. These tests were performed on a single processor of an Alliant FX/8 running Lucid Common Lisp⁵ and scaled to a set of small integer values. Here are some of these values:

Lexical var. ref.	1
CDR	1
+	2
EQ	3
Function call	4
Special var. ref.	5
CONS	15
*	17

We then ran several of the Gabriel benchmarks [7], first as ordinary compiled programs and then using CSIM with the timings derived from the test programs. The table below shows, for each program, the compiled time in seconds, the simulated time in units of 10^6 steps, and the ratio of simulated time to compiled time. The compiled times are the average of five runs of each program.

The accuracy of our simulator is reflected by how close the ratios are to each other. They are not as close as we might like, but they are all of the same general order of magnitude. To account for the differences, we can provide several explanations:

- CSIM's interpreter sometimes performs different operations than the compiled code. For example, CSIM does not optimize the evaluation of common subexpressions, and charges for each reference to a variable, whereas in compiled code some of these might be eliminated. The most extreme ratios each have an explanation of this sort:
 - *destructive* contains *do* loops that the compiler can optimize, while CSIM treats them as ordinary loops performing index computation and conditional branches.

⁵Actually, we used a version of the Qlisp system in development on the Alliant, running on a single processor. In this Lisp, memory allocation and special variable references are somewhat slower than a Lisp designed for only one processor would be.

- *stak* uses special variables, which are quite slow on the version of Lisp that we used. The compiled code uses deep binding, which takes a varying amount of time per reference, while CSIM charges a constant amount of time.
- *takl* does a lot of tail-recursive function calling, which is optimized by the compiler.

	Compiled Time	Simulated Time	Ratio
<i>boyer</i>	22.06	27.74	1.3
<i>browse</i>	19.63	41.02	2.1
<i>ctak</i>	1.56	3.26	2.1
<i>dderiv</i>	6.99	7.72	1.1
<i>deriv</i>	5.96	7.15	1.2
<i>destructive</i>	2.18	7.89	3.6
<i>div test-1</i>	2.63	4.22	1.6
<i>div test-2</i>	3.44	3.62	1.1
<i>stak</i>	6.09	2.39	0.4
<i>tak</i>	0.53	1.11	2.1
<i>takl</i>	2.04	8.39	4.1
<i>takr</i>	0.72	1.11	1.5

- CSIM pretends that garbage collection time is a constant multiple of the time spent in allocation functions, by including it in the cost of these functions. This is not accurate; a copying garbage collector takes time proportional to the amount of memory in use when it is called, which may be large or small depending on the program being tested.

3.6.2 Speed of the simulator

Next we will compare the speed of CSIM itself with the speed of compiled code that it is simulating. The computation of function timing statistics (see Section 3.2.6) was disabled during these tests; turning it on slows CSIM by an extra factor of 2 or more. We also ran the programs through the Lucid Common Lisp interpreter for comparison.

The times in the table below are all in seconds. The CSIM runtimes are the average of three runs, except for *boyer* and *browse* which were only run once. The runtimes

for interpreted and compiled code are the average of five runs.

	CSIM	Interpreted		Compiled	
	Runtime	Time	Ratio	Time	Ratio
<i>boyer</i>	9441.32	1313.54	7.2	22.06	428
<i>browse</i>	12125.03	1142.68	10.6	19.63	618
<i>ctak</i>	488.61	94.65	5.2	1.56	313
<i>dderiv</i>	1105.40	103.66	10.7	6.99	158
<i>deriv</i>	1192.60	116.65	10.2	5.96	200
<i>destructive</i>	2315.31	244.35	9.5	2.18	1062
<i>div test-1</i>	1406.86	207.71	6.8	2.63	535
<i>div test-2</i>	999.59	118.05	8.5	3.44	291
<i>stak</i>	475.61	107.76	4.4	6.09	78
<i>tak</i>	433.87	65.67	6.6	0.53	818
<i>takl</i>	3230.16	582.13	5.5	2.04	1583
<i>takr</i>	456.09	66.67	6.8	0.72	633

During these tests there was some variation in running conditions. Running time on the Alliant generally increases when several programs are executing simultaneously. This is probably due to contention for the cache, which is shared between its processors. This factor makes as much as a 10% difference, so the figures above should be taken as rough approximations.

In some of the tests there was a significant amount of garbage collection. Enough memory was allocated to limit the garbage collection to once every few seconds, but not so much as to cause paging of the Lisp process.

CSIM generally took 300 to 1000 times as long as the compiled version of the code it was interpreting, i.e., 5 to 15 minutes to simulate a second of compiled code.

The comparison with the Lucid interpreter shows much less variation in the speed ratio, reflecting the fact that CSIM and an ordinary interpreter do similar things with a program. In general, CSIM is about 5 to 10 times slower than the interpreter, which is a reasonable price to pay for the extra work that CSIM does to handle parallel programs.

Both CSIM and the Lucid interpreter spend a lot of time doing storage allocation and garbage collection. The Lucid interpreter dynamically allocates lexical environments just as CSIM does [31], but it uses a stack for dynamic binding and function calls. CSIM spends much of its time creating lexical closures for use as continuations. It runs best when given a lot of free storage, since this decreases the frequency of garbage collection. But the physical memory of the machine provides a limit to the amount of useful storage we can allocate; once this is exceeded and we start paging, performance drops tremendously.

Although these tests simulated only one processor, they are indicative of the times that we get simulating parallel programs, since none of the code to manage concurrency has been removed. The time to simulate a parallel program is roughly proportional to the product of the number of processors we are simulating and the parallel runtime, with the same ratio as above, as long as most of the processors are doing useful work. Simulating idle processors turns out to be more expensive than simulating processors running ordinary code, because they are generally in a loop referencing memory (checking a queue for work to do), and each such reference must be serialized as described in Section 3.4.1. We could probably modify CSIM to avoid this source of inefficiency, but the difference does not seem worth the effort it would require.

3.7 A Parallel example

As an example of how CSIM is used, we will try to apply parallelism to the *boyer* benchmark. Boyer [7, pp. 116–135] is a simple theorem prover that works by rewriting a formula into a canonical form (a structure of nested *if*-expressions), and then applying a tautology checker to the result.

Converting *boyer* to a parallel program is mainly an exercise; it is unlikely that anyone will want to use the result. This is because there are better algorithms to do what *boyer* does, so it would pay to start from scratch and write a good parallel theorem prover. Still, the case of parallelizing an existing sequential program is an important one, and we expect to see it come up fairly often.

We begin with little knowledge of where the program spends its time. The first step, therefore, is to simulate it running as a sequential program on one processor and look at the function timing statistics (Section 3.2.6). Unfortunately, the benchmark as given takes too much time and memory for easy experimentation; a single run through CSIM with statistics gathering turned on takes about 20 hours and causes a large amount of paging. Therefore we will modify it to create a faster test.

The top level of the program is a function called *test*, which first constructs a term by calling

```
(apply-subst
  (quote ((x f (plus (plus a b)
                     (plus c (zero))))
          (y f (times (times a b)
                     (plus c d)))
          (z f (reverse (append (append a b)
                                (nil))))))
```

```

(u equal (plus a b)
  (difference x y))
(w lessp (remainder a b)
  (member a (length b))))
(quote (implies (and (implies x y)
  (and (implies y z)
    (and (implies z u)
      (implies u w))))
  (implies x w))))

```

and then calls `tautp`, the main function of the theorem prover, with this term. Our simplified test case uses instead the term

```

(apply-subst
  (quote ((x f (plus (plus a b)
    (plus c (zero))))
    (y f (times (times a b)
      (plus c d)))
    (z f (reverse (append (append a b)
      (nil))))))
  (quote (implies (and (implies x y)
    (implies y z))
    (implies x z))))

```

Running this test through CSIM, we get three sets of function timing statistics. First, for each function we have the amount of time spent just in that function:

ONE-WAY-UNIFY1	358379	36.5%
REWRITE-WITH-LEMMAS	145404	14.8%
ONE-WAY-UNIFY1-LST	127357	13.0%
REWRITE	115362	11.7%
REWRITE-ARGS	91899	9.3%
ONE-WAY-UNIFY	66324	6.7%
ASSQ	49480	5.0%
APPLY-SUBST-LST	12504	1.3%
APPLY-SUBST	10442	1.1%
...		

Next, we have the time spent in each function including other functions that it calls:

TEST	983065	100.0%
TAUTP	982079	99.9%
REWRITE	976180	99.3%
REWRITE-WITH-LEMMAS	973337	99.0%
REWRITE-ARGS	972961	99.0%
ONE-WAY-UNIFY	635750	64.7%
ONE-WAY-UNIFY1	608115	61.9%
ONE-WAY-UNIFY1-LST	420197	42.7%
ASSQ	179506	18.3%
APPLY-SUBST	38393	3.9%
APPLY-SUBST-LST	37718	3.8%
TAUTOLOGYP	5899	0.6%
...		

Finally, we have the average time per call to each function.

TEST	983065.0	(1 call)
TAUTP	982079.0	(1 call)
REWRITE	4083.7	(2595 calls)
TAUTOLOGYP	3038.5	(13 calls)
REWRITE-ARGS	2486.4	(4626 calls)
REWRITE-WITH-LEMMAS	907.2	(7559 calls)
APPLY-SUBST-LST	280.3	(494 calls)
APPLY-SUBST	231.2	(394 calls)
TRUEP	158.4	(24 calls)
ONE-WAY-UNIFY	115.0	(5527 calls)
FALSEP	110.4	(19 calls)
ASSQ	83.5	(2798 calls)
ONE-WAY-UNIFY1	73.0	(12951 calls)
ONE-WAY-UNIFY1-LST	72.0	(7513 calls)
...		

These statistics show that most of the time is spent in `rewrite` and `one-way-unify` and their subsidiary functions. But the calls to `one-way-unify` are, on the average, much smaller than calls to `rewrite`. This suggests that we should try to parallelize calls to `rewrite`, since this will create processes of larger size and thus reduce the process creation overhead. If this does not achieve enough speedup, we will look at calls to `one-way-unify`.

Notice that the first set of statistics, which is what ordinary "profiling" of the program would produce, tells us that `one-way-unify1` accounts for a large portion of the execution time, but it does not tell us that calls to this function are parts of

higher-level tasks. Thus, it does not tell us as much about the places to look for effective use of parallelism as the second set of statistics does.

Before we investigate `rewrite`, we must notice that *boyer* contains some uses of global (special) variables that would cause improper sharing of data in parallel processes. One of these is easy to fix: the variable `temp-temp`, declared special with a `defvar` at the beginning of the program, is used only as a local temporary variable in the functions `apply-subst` and `one-way-unify1`. By removing the `defvar` and adding `&aux temp-temp` to the parameter lists of these functions, we avoid the use of the global variable.

The other global variable, `unify-subst`, is a bit more difficult to deal with. It is used in the following way:

```
(defun rewrite-with-lemmas (term lst)
  (cond ((null lst)
        term)
        ((one-way-unify term (cadr (car lst)))
         (rewrite (apply-subst unify-subst (caddr (car lst)))))
        (t (rewrite-with-lemmas term (cdr lst)))))
```

Each call to `one-way-unify` sets `unify-subst` to `NIL`, and then incrementally modifies it (in `one-way-unify1`). When `one-way-unify` returns, `unify-subst` contains a list which is referenced by the code shown above, and then there are no further references. Since we are going to parallelize calls to `rewrite`, several processes may be running `rewrite-with-lemmas` at the same time and they should not share the same global variable.

CSIM's definition of dynamic binding (see Section 3.3.1) makes it possible to establish a separate instance of `unify-subst` for each call to `one-way-unify`, as follows:

```
(defun rewrite-with-lemmas (term lst)
  (let ((unify-subst nil))
    (cond ((null lst)
          term)
          ((one-way-unify term (cadr (car lst)))
           (rewrite (apply-subst unify-subst (caddr (car lst)))))
          (t (rewrite-with-lemmas term (cdr lst)))))
```

Recall that `unify-subst` is a special variable because of the `defvar` at the beginning of the program. If `rewrite-with-lemmas` is called concurrently in different processes, they will each perform a dynamic binding of `unify-subst`, which will be invisible to

other processes because each establishes a new dynamic environment. Thus, the references to `unify-subst` in `one-way-unify` will not interfere with each other.

Having made these changes, we now proceed to examine the functions `rewrite` and `rewrite-args`.

```
(defun rewrite (term)
  (cond ((atom term)
        term)
        (t (rewrite-with-lemmas (cons (car term)
                                       (rewrite-args (cdr term)))
                                (get (car term)
                                    (quote lemmas))))))

(defun rewrite-args (lst)
  (cond ((null lst)
        nil)
        (t (cons (rewrite (car lst))
                  (rewrite-args (cdr lst))))))
```

The main potential for parallelism here is in `rewrite-args`, which performs independent computations on each member of the list given as its argument. We can create a separate process for each one of these. We can also use futures to return a value from `rewrite-args` before these processes finish, which may add some more parallelism.

A single change to the function accomplishes this:

```
(defun rewrite-args (lst)
  (cond ((null lst)
        nil)
        (t (cons (future (rewrite (car lst)))
                  (rewrite-args (cdr lst))))))
```

When we run the resulting program through CSIM, we get the results shown below.⁶ CSIM provides the “running time” and “idle overhead” data, and we have computed the other numbers in the table from these. Speedup versus one processor is the time for the parallel program on one processor divided by the time on n processors. Speedup versus the serial program is a more meaningful measure, since it accounts

⁶CSIM’s default LIFO scheduler was used for these tests. Different results would be obtained using the FIFO scheduler, or if we changed `FUTURE` to `DFUTURE`. This is mainly because a reference to an undetermined future causes extra overhead, and the order in which processes are scheduled decides whether the values of futures are computed by the time they are referenced.

for overhead in the parallel program that we must try to avoid. The serial program's time is 983065 steps, as computed in the earlier simulator run.

# of Proc.	Running Time	Speedup vs. 1 proc.	Speedup vs. serial	Useful Work	Idle Overhead	Other Overhead
1	1367478	1.00	0.72	0.72	0.13	0.15
2	704366	1.94	1.40	0.70	0.15	0.15
3	491812	2.78	2.00	0.67	0.19	0.15
4	396999	3.44	2.48	0.62	0.22	0.16
5	346768	3.94	2.84	0.57	0.27	0.17
10	310896	4.40	3.16	0.32	0.50	0.18
15	315602	4.33	3.12	0.21	0.63	0.17
20	299398	4.57	3.28	0.16	0.68	0.15

The last three columns show the fractions of processor time spent doing useful work and overhead of various sorts. Useful work is the speedup vs. the serial time, divided by the number of processors. This number stays well below 1.00 because of overhead in the parallel program. For each future, the parallel program does extra work to create the future, to add a process to the queue, to remove it when a processor becomes idle, and to reference data indirectly through the future. The costs of future creation and adding processes are part of the "other overhead" above. The costs of finding processes in the queue and removing them are counted in "idle overhead." Idle overhead also counts time spent waiting for the lock on the queue, and time when there is no work for a processor to do.

Beyond about 10 processors, there is simply not enough work to keep all the processors busy, and the idle overhead begins to climb rapidly, while the "other overhead" fraction drops because the idle processors are not doing the operations that are charged to that category.

Chapter 4

Experimental results and analysis

In this chapter we present the results of experiments that compare the partitioning and scheduling methods described in Chapter 2, and analyze these results to explain what is happening and draw some general conclusions about the methods.

It is difficult to draw conclusions about the partitioning and scheduling methods without looking at specific examples of their use. Therefore, our approach is to begin with experimental results on specific programs, explain them, and try to generalize them to a larger class of programs.

4.1 The Fast Fourier Transform

The Fast Fourier Transform (FFT) is a computation with a very regular structure, and should be easy to parallelize in any shared-memory programming model, even though its running time is only $O(n \log n)$ for input of size n . In addition to its applications in signal processing, the FFT is used as a subroutine in the fastest known algorithms for multiplying polynomials and large integers. This use of the FFT is the symbolic processing application we have in mind when choosing FFT as a test program.

Figure 4.1 shows a Common Lisp program for the Fast Fourier Transform.¹ This program differs in several ways from the more usual implementations.

- The basic structure of the program is recursive, not iterative. Each FFT of size n performs two FFT's of size $n/2$ and merges the results. Changing from an iterative to a recursive program has a negligible effect on performance, and will allow us to add parallelism to the program more easily.

¹The function `cis` called by the program computes $\cos \theta + i \sin \theta$, returning a complex number. The symbol `pi` is the mathematical constant π . The function `ash` is an arithmetic shift, and is used in several places to multiply and divide integers by 2. Using shifts instead of division turns out to make a noticeable improvement in the program's performance.

```

;;; ARRAY is the input array, of size ARRAY-SIZE. RESULT is the
;;; output array, of size RESULT-SIZE, which is the size of the FFT
;;; that is performed. RESULT-SIZE must be a power of 2; if
;;; ARRAY-SIZE < RESULT-SIZE then ARRAY is considered to be padded
;;; with 0's.

(defun fft (array array-size result result-size)
  (labels ((fft-inner (n r j m)
    ;; Store FFT of size N into RESULT starting at position
    ;; R, using elements starting at position J in ARRAY with
    ;; step M.
    (if (= n 1)
      (setf (aref result r)
        (if (< j array-size) (aref array j) 0))
      (let ((n2 (ash n -1)))
        (fft-inner n2 r j (ash m 1))
        (fft-inner n2 (+ r n2) (+ j m) (ash m 1))
        (dotimes (k n2)
          (let ((x (aref result (+ r k)))
                (y (* (aref result (+ r k n2))
                     (cis (/ (* -1 pi k) n2))))))
            (setf (aref result (+ r k)) (+ x y))
            (setf (aref result (+ r k n2)) (- x y)))))))
    (fft-inner result-size 0 0 1))
  result)

```

Figure 4.1: The Fast Fourier Transform in Common Lisp

- The source and destination values are in different arrays. This arrangement of the data has several advantages over performing the FFT in place on the input array: no shuffling (using bit-reversal) is necessary, and the input argument is not modified, allowing its value to be reused. This version of the FFT algorithm can therefore be used by a program written in a functional style. On the other hand, we have decided to pass the output array as an argument to the `fft` function instead of having `fft` create and return a new array each time; the caller of the function can then reuse an array when it is obviously safe to do so. An example of such reuse is when multiplying polynomials using the equation

$$a * b = FFT^{-1}(FFT(a) \cdot FFT(b)).$$

where $*$ denotes polynomial multiplication and $\cdot$ denotes term-by-term multiplication. If the coefficients of the polynomials a and b are stored in the arrays A and B , then we can allocate two new arrays T_1 and T_2 , compute $T_1 \leftarrow FFT(A)$; $T_2 \leftarrow FFT(B)$; $T_1 \leftarrow T_1 \cdot T_2$; and finally $T_2 \leftarrow FFT^{-1}(T_1)$. (The inverse FFT is a slight variation of the FFT program. See [3] for a more detailed explanation.)

There are two sources of parallelism in the FFT program shown. The first is that the recursive calls to `fft-inner` may be performed in parallel, since they reference different parts of the input and output arrays. Secondly, we can parallelize the `dotimes` loop in which pairs of input values are merged in a "butterfly" pattern. All of the iterations of the loop are independent and can be executed concurrently.

The recursive calls to `fft-inner` form a binary tree, and fit well into the height cutoff and depth cutoff partitioning models that we have proposed. The computation depth is easy to keep track of, and the height can be determined by looking at the size of the subproblem being solved in the current call to `fft-inner`.

The height of a subcomputation is directly related to the value of the argument `n` passed to `fft-inner`. To implement a height cutoff, we replace the two calls to `fft-inner` by the form

```
(qlet (> n *height*)
  ((x1 (fft-inner n2 r j (ash m 1)))
   (x2 (fft-inner n2 (+ r n2) (+ j m) (ash m 1))))))
```

where `*height*` is a global variable that will contain the height cutoff value that we decide to use. For a depth cutoff, we write

```
(let ((*depth* (1- *depth*)))
  (qlet (>= *depth* 0)
    ((x1 (fft-inner n2 r j (ash m 1)))
     (x2 (fft-inner n2 (+ r n2) (+ j m) (ash m 1))))))
```

Here we are using Lisp's dynamic binding mechanism to give each process the appropriate value of `*depth*`. Before the top-level call to `fft`, we set `*depth*` globally to the number of levels in the tree that should cause processes to be created. When `*depth*` is rebound by the `let` expression, the effect is local to the process performing the binding. This value is inherited by any subprocesses that are created, but when they reach the same `let` expression they will create new local bindings of `*depth*`, and so on. Thus each process uses the appropriate value in its partitioning test.

Note that in both of these versions of the parallel program, we use the `qlet` form to bind variables `x1` and `x2`, but never use their values. This is because the basic operations of the program are side effects on an array.

The `dotimes` loop in the original program does not immediately lend itself to these methods. We could convert it to a loop that creates a process for each iteration, but then all of the potential processes would be the same size, and process creation overhead might be too high to make any of them worthwhile. However, it is not hard to transform the loop into code that recursively splits the range of indices in half at each call, resulting in a binary tree of potential processes. These processes will vary in size just as the recursive calls to `fft-inner` do, and we will be able to apply height or depth cutoff methods to this computation tree.

In our first experiment, we used height cutoff as the partitioning method and examined four different scheduling methods. These are FIFO with a global process queue, LIFO with a global queue, FIFO with a separate queue per processor, and LIFO with a separate queue per processor. We abbreviate these to FIFO, LIFO, FIFO* and LIFO*. Rather than try to guess a good height cutoff, we tested a wide range of values to see how the choice of cutoff value affects performance.

Figure 4.2 shows a selection of the results of this experiment. The most obvious thing to notice is the different performance of the various scheduling algorithms. In all cases, LIFO* was the best-performing scheduler and FIFO the worst. The order of the other two varied depending on the number of processors and height cutoff. The greatest differences in performance occurred at low values of the height cutoff, especially for the larger problem instances. Since there is such a clear winner among the schedulers, LIFO*, we will use it for the remainder of the FFT experiments.

In Section 2.3.1, we conjectured that the optimal height cutoff value would depend on both the number of processors and the problem size. The optimal depth cutoff, on the other hand, might be independent of the problem size. Figure 4.3 confirms the first prediction; in each graph, as the problem size increases the height cutoff giving the best performance also increases. Also, the optimal cutoff varies for a given problem size n as we increase the number of processors p .

Figure 4.4 shows the corresponding curves when a depth cutoff is used. For some values of p , there is a single best cutoff value for all of the problem sizes n . In other cases, the optimal depth cutoff increases as n grows, though not as fast as the optimal height cutoff did. It is much easier to choose a depth cutoff value that gives reasonably good performance over the range of problem sizes tested. In particular, the optimal cutoff value for the largest n tested results in reasonably good performance over the entire range of problem sizes.

Our final set of graphs, in Figure 4.5, compares the “best” depth cutoff for each value of p with the results of the dynamic partitioning method using a separate queue per processor, and creating a process only when this queue is empty. The dynamic method does not perform as well as the depth cutoff method for this range of problem

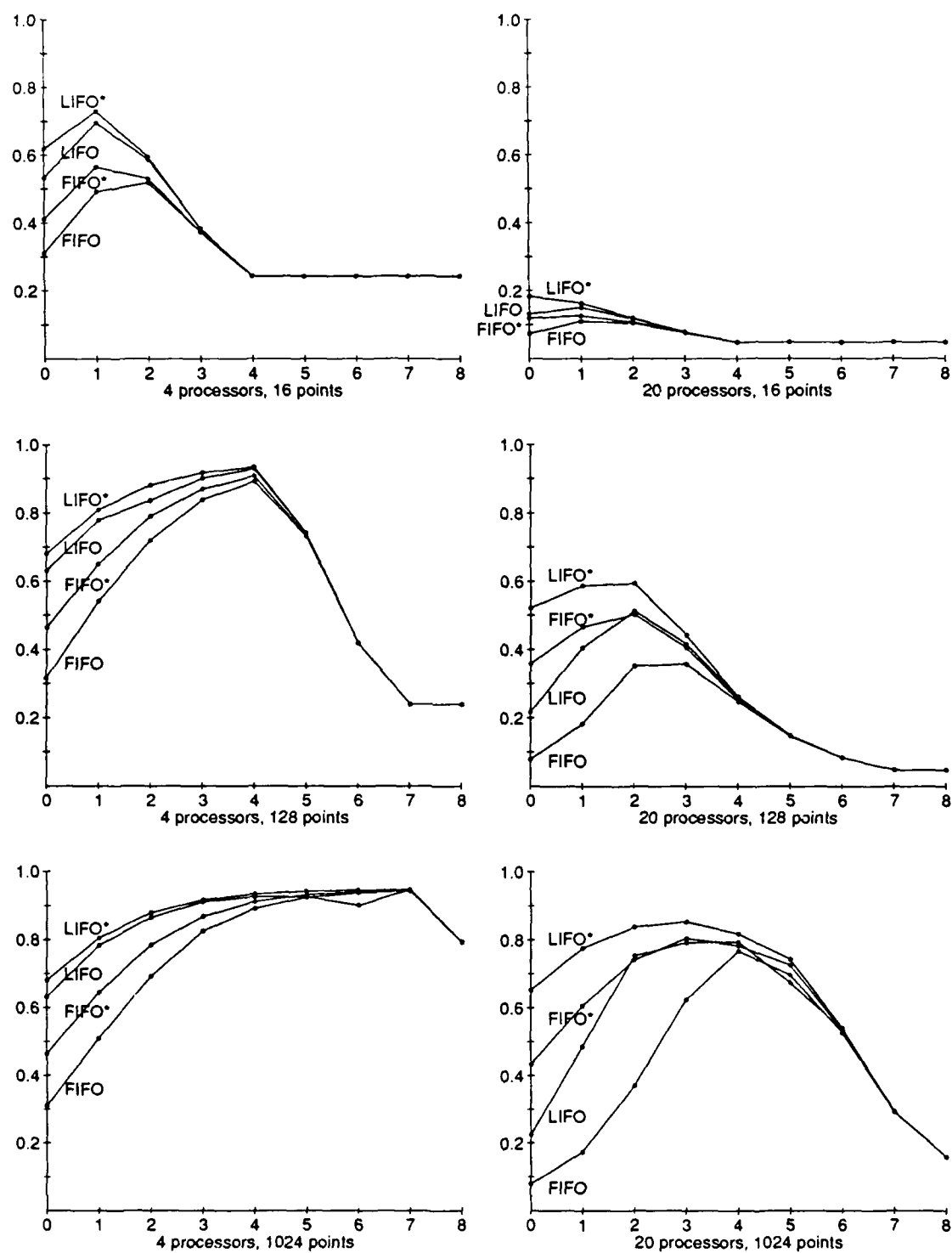


Figure 4.2: Efficiency vs. height cutoff value, four scheduling methods

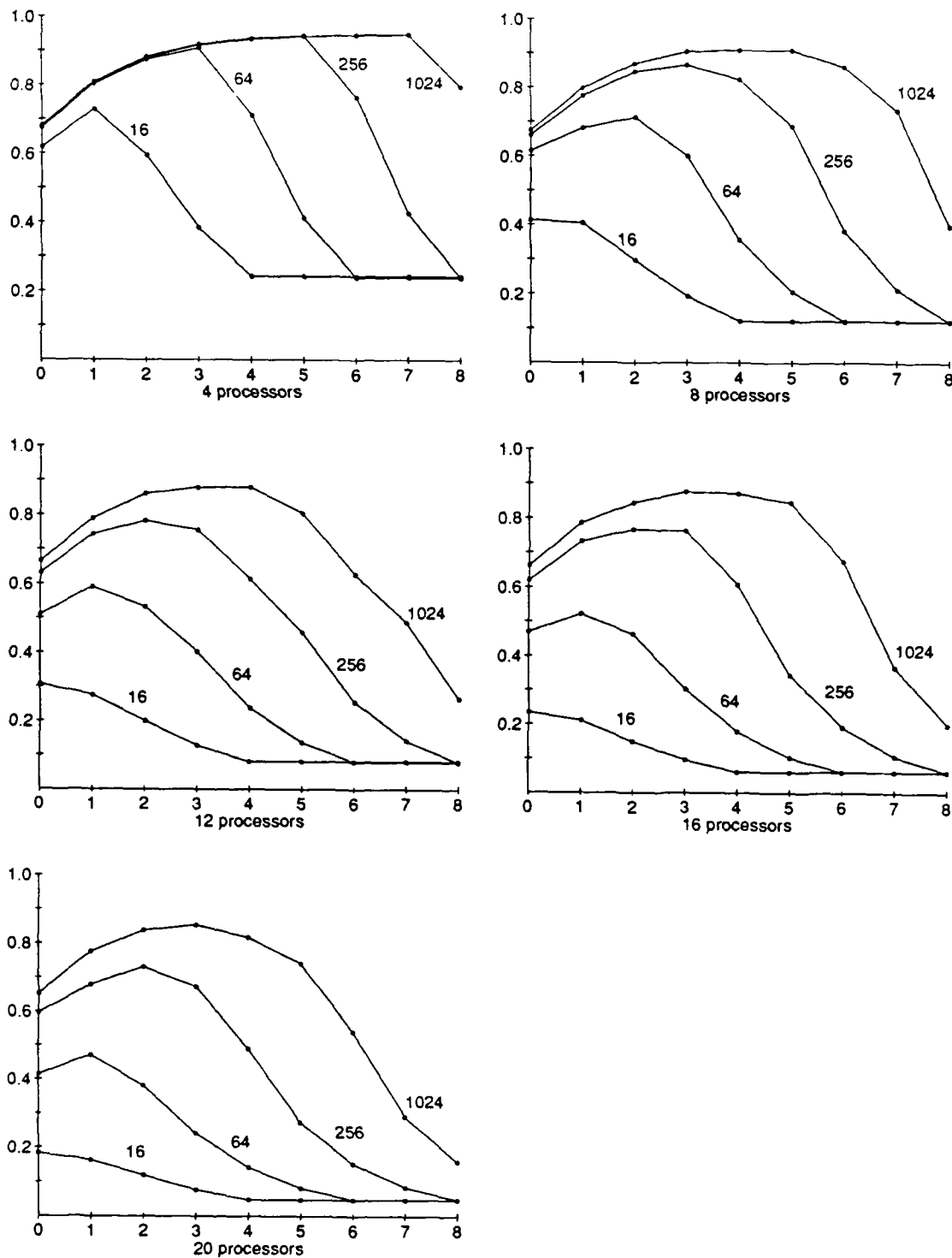


Figure 4.3: Efficiency vs. height cutoff value, various problem sizes

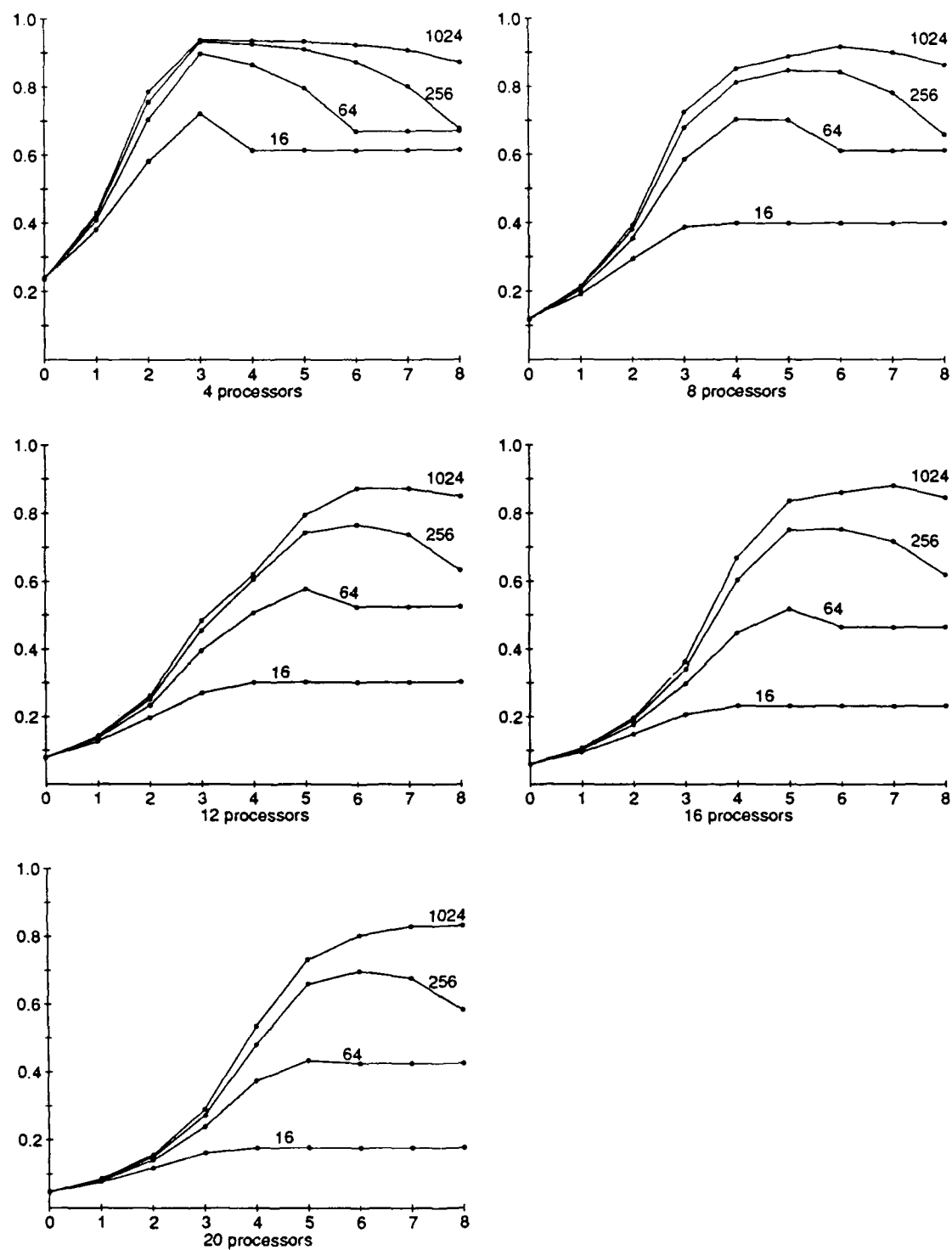


Figure 4.4: Efficiency vs. depth cutoff value, various problem sizes

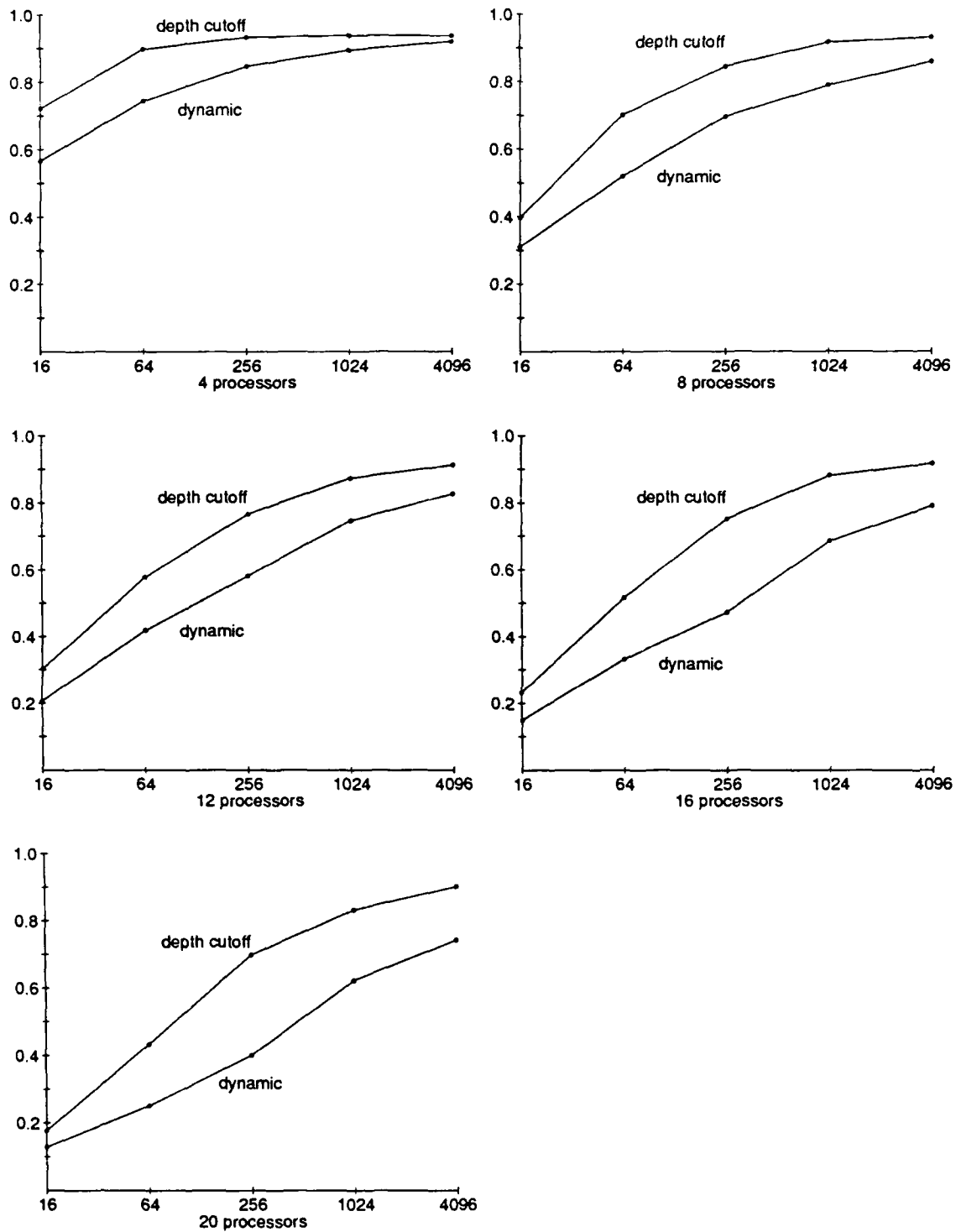


Figure 4.5: Efficiency vs. problem size, for "optimal" depth cutoff and dynamic partitioning

sizes: on the other hand, it is easier to apply in practice because there is no need to experiment to determine the appropriate cutoff values. Also, as the problem size increases, both methods approach optimal performance and therefore the dynamic method is worth considering.

These results shown here are for programs that continue making partitioning tests even after the height or depth cutoff has decided to run a potential process sequentially. This additional work slows the program by a constant factor, no matter how many processors are used. For the examples tried, the slowdown was about 5%. Changing the program to avoid partitioning tests when running a process sequentially recovered practically all of this 5% in the best cases. This shows that an appropriately chosen cutoff value can result in essentially optimal performance.

4.2 The CYK parsing algorithm

Our next example is a program with a less regular structure than the FFT. The Cocke-Younger-Kasami (CYK) parsing algorithm is a simple polynomial time method to decide membership of strings in a context-free language, given a Chomsky normal form grammar for the language. Figure 4.6 shows a Common Lisp program for this algorithm.

A grammar in Chomsky normal form consists of productions of the form $N \rightarrow AB$ and $N \rightarrow t$, where N , A and B are nonterminal symbols and t is a terminal symbol. Our program represents each nonterminal N by a structure (defined with `defstruct`) that contains a list of the terminals t for which $N \rightarrow t$, and a list of the productions $N \rightarrow AB$. Each production is represented by the S-expression $(A . B)$, where A and B are themselves the structures for the nonterminal symbols A and B . The resulting data structure may be "circular," but the program expects this. The entire grammar is represented as a list of its nonterminal symbols, with the start symbol as the first member of the list.

The `parse` function constructs an array V such that V_{ij} is the set of nonterminals that can produce the substring of the input starting at position i and ending at position $i + j$. (At the beginning of the string, $i = 0$. Our program is equivalent to the one in [17, p. 140] except that our variables i , j and k begin their range at 0 rather than 1.) After completing the computation of this array, `parse` accepts the input string if the start symbol of the grammar is a member of $V_{0,n-1}$ where n is the length of the input string.

In order to parallelize the `parse` function, we examine each of the loops to see if their iterations can be executed in parallel instead of sequentially. For the outermost loop, on the variable j , this is not possible because the computation of each V_{ij}

```

;;; Grammar      = list of nonterminals (beginning with start symbol)
;;;
;;; Nonterminal = structure
;;;              { list of terminals t such that N -> t;
;;;                list of productions (A . B) such that N -> A B }

(defstruct (nonterminal (:conc-name nil))
  (terminals nil)
  (productions nil))

(defun parse (str grammar)
  (let* ((n (length str))
        (v (make-array (list n n) :initial-element nil)))
    ;; Initialization
    (dotimes (i n)
      (let ((c (char str i)))
        (dolist (a grammar)
          (when (member c (terminals a))
            (push a (aref v i 0))))))
    ;; Main loop
    (do ((j 1 (1+ j)))
      ((>= j n))
      (dotimes (i (- n j))
        (dotimes (k j)
          ;; The body of this loop takes time bounded by a constant,
          ;; for any given grammar.
          (let ((left (aref v i k))
                (right (aref v (+ i k 1) (- j k 1))))
            (when (and left right)
              (dolist (a grammar)
                (dolist (p (productions a))
                  (when (and (member (car p) left)
                              (member (cdr p) right))
                    (unless (member a (aref v i j))
                      (push a (aref v i j))))))))))
      (let ((left (aref v i k))
            (right (aref v (+ i k 1) (- j k 1))))
        (when (and left right)
          (dolist (a grammar)
            (dolist (p (productions a))
              (when (and (member (car p) left)
                          (member (cdr p) right))
                (unless (member a (aref v i j))
                  (push a (aref v i j))))))))))
    ;; Test for acceptance.
    (not (null (member (car grammar) (aref v 0 (1- n))))))

```

Figure 4.6: The CYK parsing algorithm in Common Lisp

depends on some of the values of $V_{i'j'}$, with $j' < j$. The loops on i and k can be parallelized, however, since their iterations are completely independent.

Because `parse` performs assignments to shared memory, synchronization must be considered. The innermost loop performs the operation `(push a (aref v i j))`, creating a critical region of code. (The Common Lisp macro `(push x y)` is an abbreviation for `(setf y (cons x y))`, which involves a read followed by a write to y .) Several processes doing this `push` concurrently with the same values of i and j would create a race condition, with the possibility of losing one of the pushed values if there is no synchronization. If we only parallelize the i loop, there is no problem—any two processes executing `(push a (aref v i j))` will have different values of i . But if we parallelize the k loop, two processes with the same values of i and j may perform conflicting `push` operations, so we must modify the program to synchronize these. As will be explained shortly, parallelizing both the i and k loops is necessary in order to get reasonable speedup.

Of the synchronization constructs discussed in Chapter 1, either locks, as provided by Multilisp, or process closures, as in Qlisp, are appropriate for this purpose. In our experiment we chose to use locks because the synchronization is at a very low level, and the overhead of process closures would slow down the program significantly. Given the choice between locks that loop while waiting for access to the critical region (spin locks), or those that suspend the waiting process and resume it later, we chose spin locks both to avoid overhead, and because we expect the delay in entering the critical region to be short most of the time.

The following new version of the inner loops in the program includes the necessary locking constructs.

```
(dotimes (i (- n j))                ;for i := 0 to n-j-1
  (let ((l (make-lock)))
    (dotimes (k j)                  ;for k := 0 to j-1
      (let ((left (aref v i k))
            (right (aref v (+ i k 1) (- j k 1))))
        (when (and left right)
          (dolist (a grammar)
            (dolist (p (productions a))
              (when (and (member (car p) left)
                          (member (cdr p) right))
                (get-lock l)
                (unless (member a (aref v i j))
                  (push a (aref v i j)))
                (release-lock l))))))))))
```

Any processes that are created for the same value of i but different values of k will share access to the lexical variable l , and thus manipulate the same lock. Processes working on different values of i will have different locks bound to the variable l , so there will be no unnecessary synchronization.

Modifying the program in this way to ensure correct behavior will necessarily increase its sequential runtime as well as its parallel runtime, because of the cost of manipulating the locks even if they never cause any processes to wait. Once we have done this, we can not hope to achieve perfect speedup over the sequential running time. We will measure the sequential time of both the original program and the program with locks. The former should be used as the basis for reporting speedup, to provide an honest comparison of the sequential and parallel programs. The latter should be used to decide when we have achieved a reasonable parallel running time, because it measures the work that is actually performed by the parallel program.

In the FFT example, the focus was on partitioning and scheduling strategies, because identification of parallelism was fairly straightforward. In this program, finding sufficient parallelism is more difficult, and we concentrate on that. The experiments about to be described all used a single scheduling strategy—the LIFO* method that showed the best performance on FFT—and use variants of the dynamic partitioning strategy.

The loops in `parse` have the same problem as those in FFT—the only concurrency that is immediately available is to perform all of the iterations in parallel. A difference in this program is that the number of iterations of each loop varies, depending on the value of j in the outer loop. For small values of j , the loop on i has a large number of iterations, each of which does a relatively small amount of work; while for large j , the i loop has a small number of iterations and each does a large amount of work. This is why parallelizing just the i loop is not sufficient; during the last few iterations of the j loop, there will not be enough work to keep all of the processors busy.

Dynamic partitioning provides a useful means of resolving this problem. We allow all iterations of the i and k loops to define potential processes, and create an actual process only when the queue of work on the current processor is empty. As a point of comparison, we show what happens when just the i loop is parallelized in this way, to justify the claim in the previous paragraph.

Several versions of the CYK program were run. Input strings of sizes from 1 to 20 were used, with three strings of each size chosen randomly and then used for all of the experiments. From 1 to 8 processors were simulated. The results shown are speedups in relation to the serial code including locking, as mentioned above, so the best possible value is not a true speedup of 8.0 but is the best that we can hope to achieve. The overhead for this locking was less than 5%, however, so the true speedup

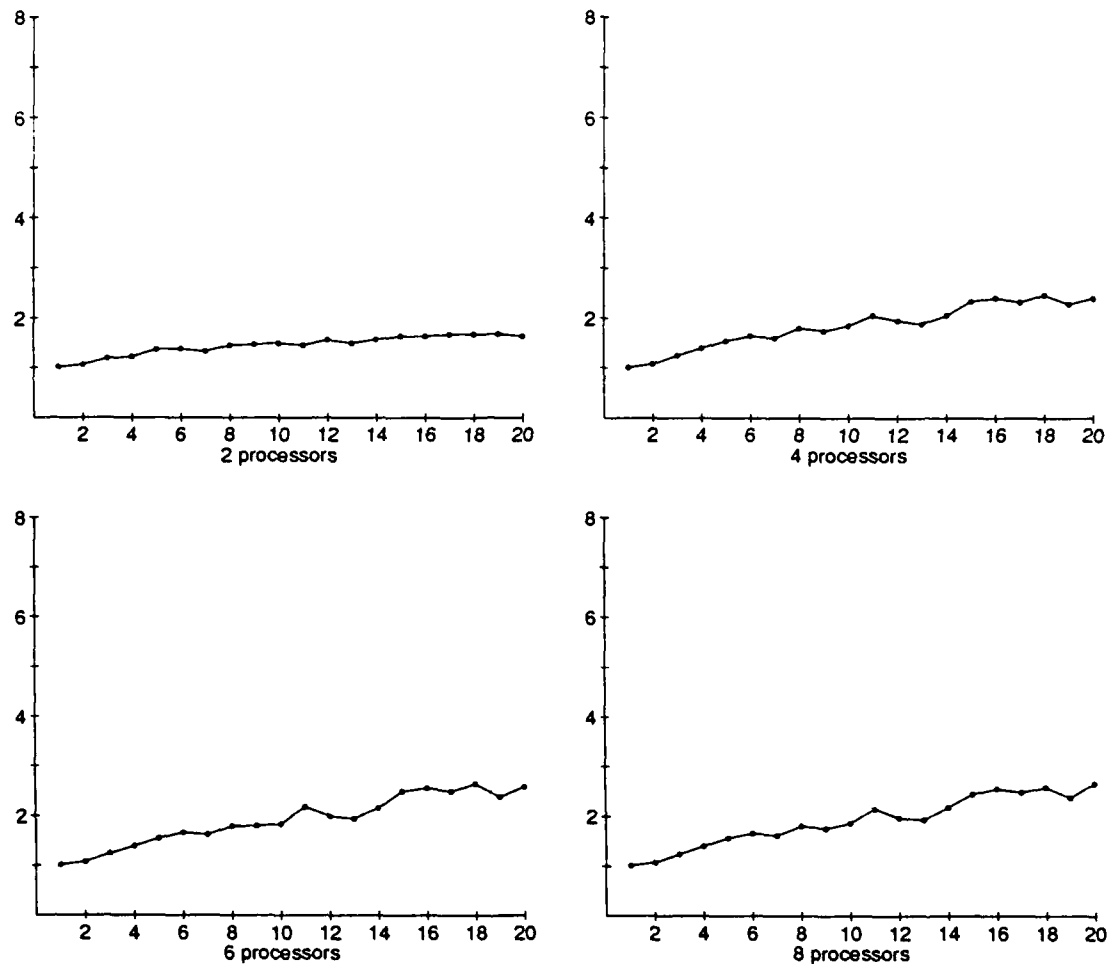


Figure 4.7: Speedup vs. problem size, parallelizing just the i loop

results do not differ much from what is shown.

Figure 4.7 shows what happened when just the loop on i was parallelized. As mentioned above, this does not create enough processes near the end of the computation, so the poor performance that we see is understandable. Next, in Figure 4.8, we see the result of also parallelizing the loop on k . There is a slight improvement, but we are still far from approaching optimal speedup as we attained in the FFT example.

We may ask whether the dynamic partitioning method is creating enough processes. To answer this question, we ran the program with *all* potential processes created. Figure 4.9 shows the result of this version. On 2 and 4 processors, the performance was slightly worse, due to the extra overhead, but on 6 and 8 processors the performance improved. This shows that there is potential for improvement over the set of processes created by the dynamic partitioning method.

We next modified the dynamic partitioning method to create processes whenever

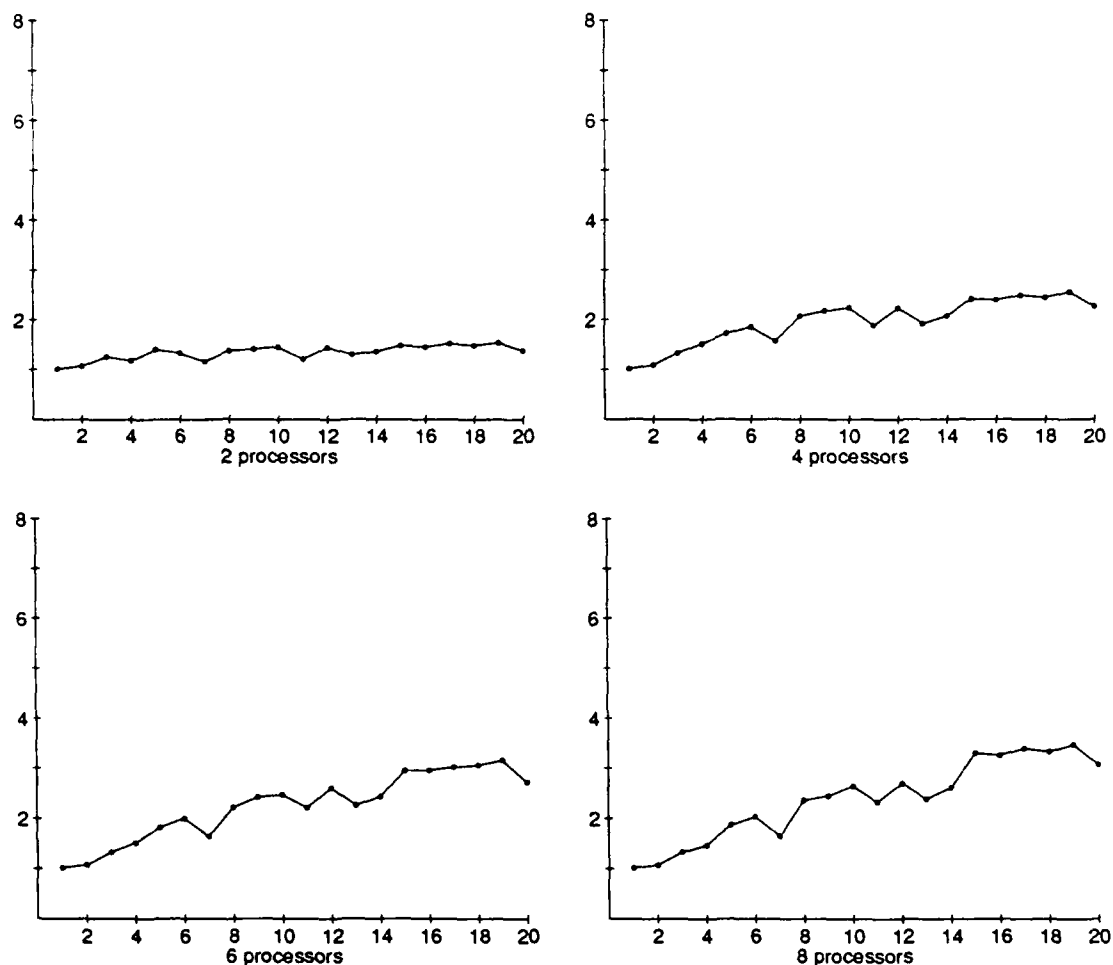


Figure 4.8: Speedup vs. problem size, parallelizing both the i and k loops

the size of a processor's queue was less than 4, instead of less than 1 as we have done so far. This will give more processes an opportunity to be created, while still providing a means to limit the amount of process creation. Figure 4.10 shows the result of this experiment. There is some improvement over Figure 4.8, but we are still far from optimal speedup.

With additional work, it may be possible to achieve better speedups for this program than those presented here, but we believe that the main obstacle to achieving good speedup easily is the lack of parallelism at the highest level of the program (the j loop). The most promising approach to achieving further speedup is, therefore, eliminating this bottleneck. Making such a change to the program qualifies as redesigning the algorithm. As we outlined at the beginning of Chapter 2, proper algorithm design is a vital part of achieving good parallel performance.

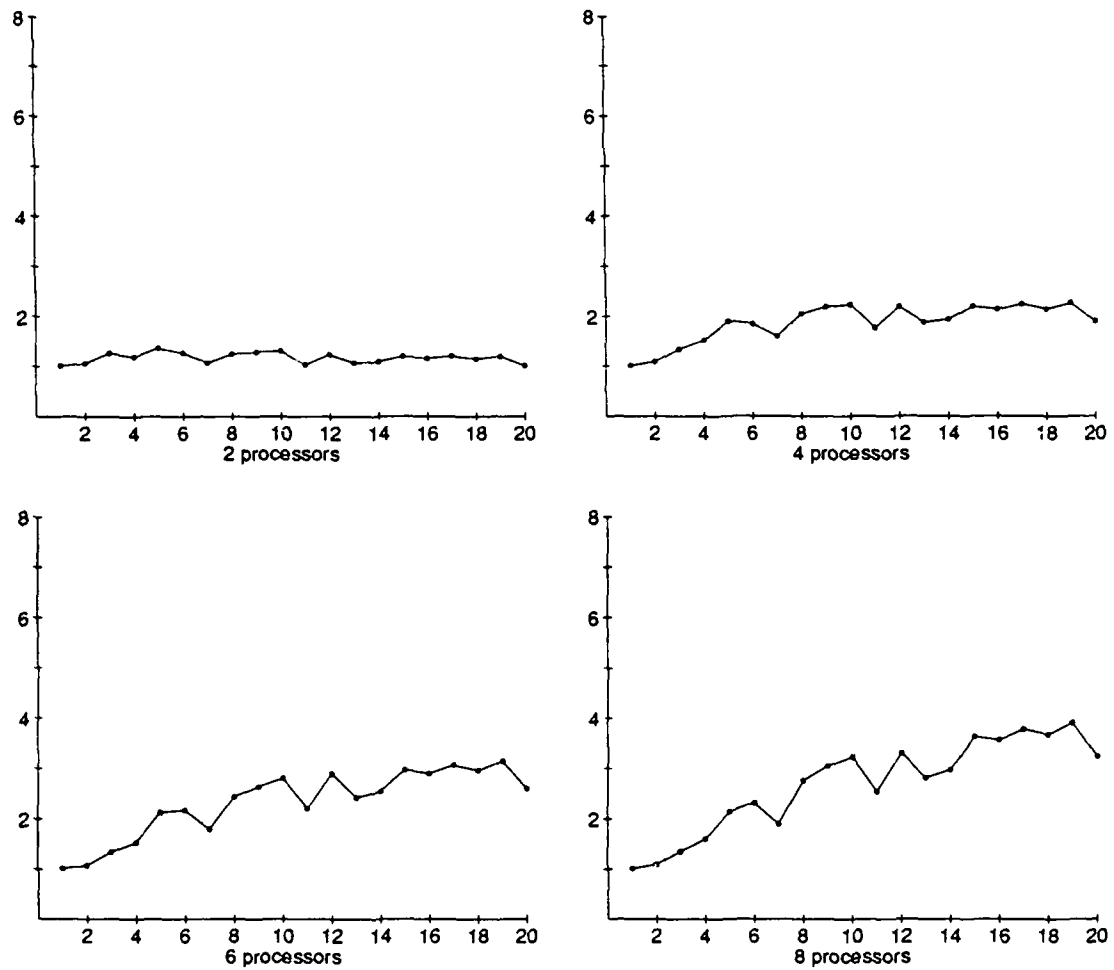


Figure 4.9: Speedup vs. problem size, creating all potential processes

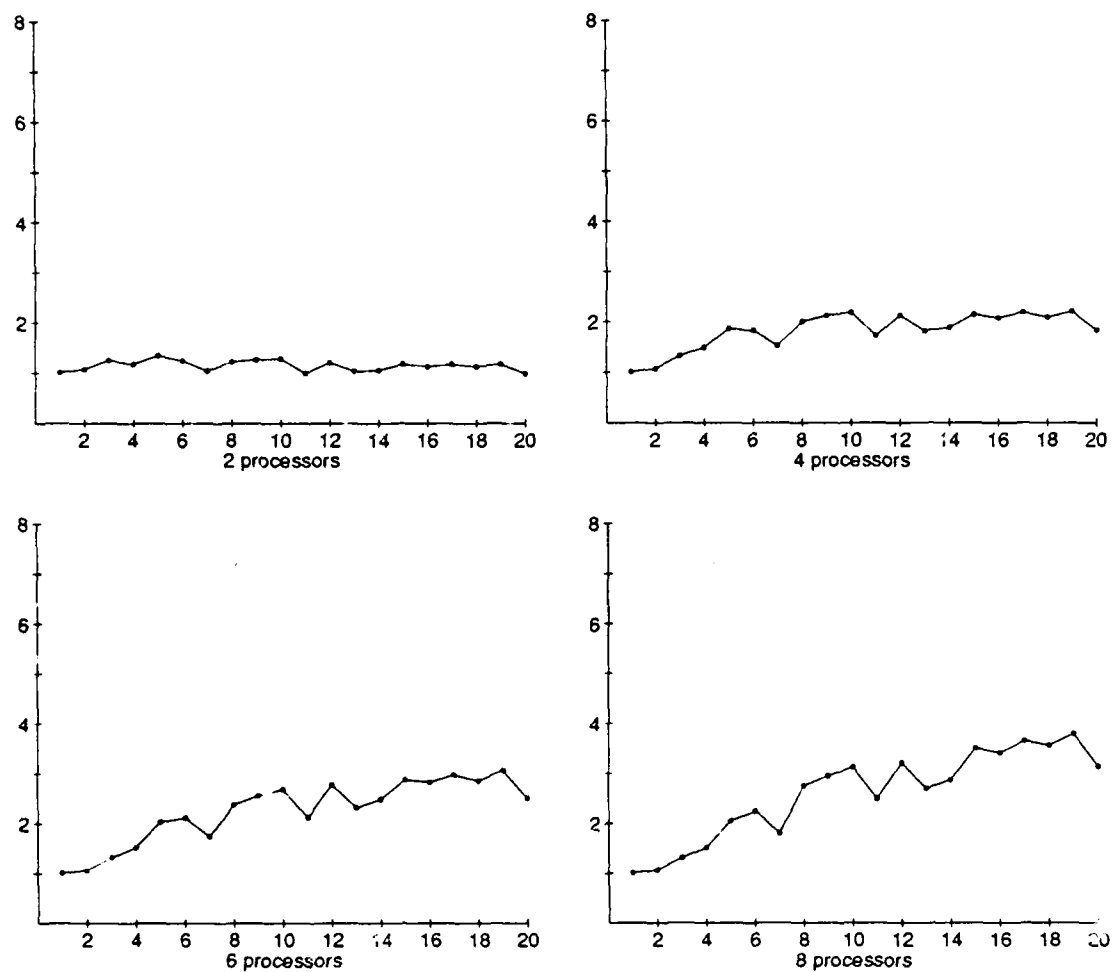


Figure 4.10: Speedup vs. problem size, creating processes when queue length $< \frac{1}{2}$

Chapter 5

Analysis of dynamic partitioning

In the experiments of the previous chapter, the performance of the dynamic partitioning method was often close to that of the height and depth cutoff methods. This is somewhat surprising, since the cutoff methods try to take advantage of knowledge about the program, such as the expected size of processes, while the dynamic method ignores such information.

This chapter examines the dynamic partitioning method in more detail. In doing so, we will see some of the reasons why it is successful in spite of its fairly simple approach to partitioning and scheduling.

5.1 Process creation behavior

We will analyze programs represented by computation trees, as described in Section 2.1. These trees correspond to non-eager programs in which each potential process has at most one partitioning decision. To simplify the discussion, we make the further assumption that at these partitioning points at most two subprocesses are created, i.e., the tree is binary. We will later extend these results to non-binary trees.

In Qlisp programs, these conditions restrict us to programs in which each potential process executes at most one `qlet` form, and this `qlet` is of the non-eager variety with (for the moment) two variable bindings. Two processes are created to evaluate these expressions. Since, after creating these processes, the parent process will wait for them to finish, the processor that was running the parent will immediately begin executing one of the two child processes. The other will remain in the processor's queue.

The scheduling method that we initially analyze is the one where each processor has a separate queue into which it adds processes that it creates, and from which it

removes processes when it is idle. If an idle processor's own queue is empty, it cycles among the queues of other processors and will remove a process from one of those.

At each partitioning decision, a processor checks to see if its own queue is empty. If so, it creates the two subprocesses, putting one in the queue and running one itself, as described above. If the queue is non-empty, however, it does not create the new processes. Our assumption of binary computation trees implies that the queue will never contain more than one process.

Suppose we are given a computation tree of height h satisfying the above assumptions, and execute it using our dynamic partitioning method on p processors. The computation starts with all queues empty and all processors idle, except for one processor that is executing the topmost node of the tree.

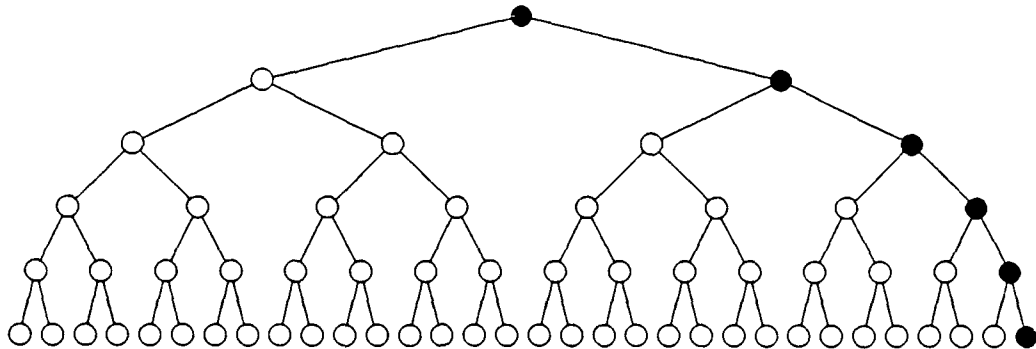
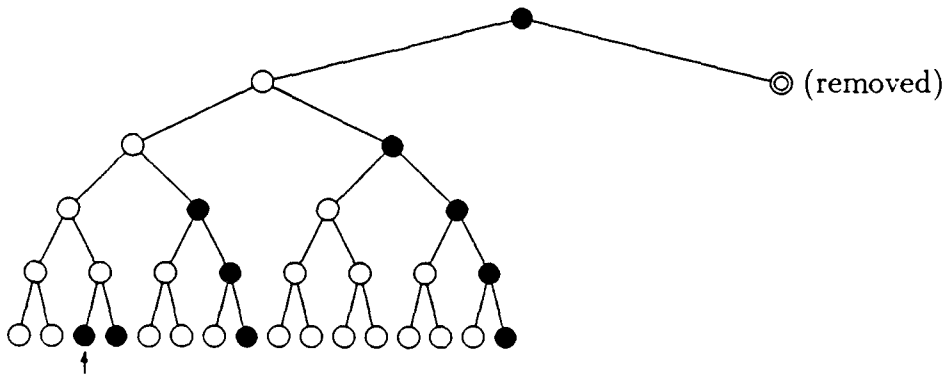
Consider a processor that is idle at some point in the computation. It removes a process from either its own queue or the queue of another processor. This process is either a subtree of the original computation tree, or it is the continuation of a process that was suspended waiting for one of its children to finish. For now, let us just consider the first case; later we will account for the resumption of suspended processes.

When the execution of this process begins, the processor's queue will be empty, because we assumed the processor was previously idle. (If its queue was not empty when the processor became idle, we will have made it empty by removing the process that was there—recall that there is never more than one process on a processor's queue.) Therefore, the partitioning test in the topmost node of the tree will be true, and that node will create subprocesses for its children. Let us assume without loss of generality that whenever a node creates subprocesses, it puts its right child tree onto its queue and the processor continues with the evaluation of its left child.

As long as all of the processors are busy, the processes that they create are all added to and removed from their own queues; none are taken from other processors' queues. This bounds the number of processes created as shown in the following lemma.

Lemma 5.1 *If a tree of height h is executed entirely on one processor, then $O(h)$ processes are created during its execution.*

Figure 5.1 shows the top part of a computation tree. Let h be the height of this tree. The processor that executes the root node of this tree will execute the entire tree, as long as no other processor removes processes from its queue during the computation. In this case, the dark circles indicate those processes that decide to create subprocesses for their children, and light circles mark those that do not. No processes are created while the left child of the root node is evaluated, because the right child of the root

Figure 5.1: $O(h)$ partitioning when there is no interferenceFigure 5.2: $O(h^2)$ partitioning when there is interference

remains on the queue. When the processor finishes with the left child of the root and becomes idle, it removes this process from its queue, and creates its right child process because the queue is now empty. Continuing in this way, it creates one process for each level in the tree, for $O(h)$ process creations altogether. $\square$

The rate of process creation can be higher when processors remove processes from each others' queues. For this case we show the following.

Lemma 5.2 *While a processor is executing a tree of height h , each time a process is removed from its queue $O(h^2)$ processes are created.*

Figure 5.2 shows what happens if another processor removes a process from the queue while a processor is evaluating a tree. In this example, the right child of the root

node has been removed before the evaluation of the left half of the tree is finished. The arrow points to the next node that will make a partitioning decision. Since the queue is now empty, a process is created, and (assuming this process is not removed by another processor) it will be executed on the original processor at its normal time. When it finishes, however, the queue is again empty and the next partitioning decision creates a process. The darkened nodes in the figure indicate the "cascade" of processes that is created as a result of the original right child's removal.

The cascade consists of a number of "branches," each of which starts one level higher in the tree than the previous branch, and extends down the right side of a subtree. When all of the work in such a subtree is finished, the program returns to the parent node of the subtree, which has now finished its left child and begins work on its right child. At this point the processor's queue is empty, so $O(h)$ processes are created during the execution of the right child. There may be as many as h branches, leading to the $O(h^2)$ bound. $\square$

The removal of a process from the queue by another processor will be called a "transfer," to distinguish it from a processor removing a process from its own queue.

We will now derive an asymptotic upper bound on the total number of processes created during the execution of a tree of height h . All subtrees of this tree have height less than h , so using h in place of the actual height of any subtree will still give an upper bound. Each tree that causes $O(h)$ process creations as described in Lemma 5.1 must at some time have been transferred from another processor's queue, and thus causes $O(h^2)$ additional process creations. The $O(h^2)$ term dominates the $O(h)$ term, so the total number of processes created is just $O(h^2)$ multiplied by the number of transfers. We now compute an upper bound for this quantity.

At any point during the computation, various subtrees of the original computation tree are unevaluated. These must all have height less than h , the height of the original tree. Let H be the maximum height of any potential process that may still be created in the remainder of the computation. H will decrease as the computation progresses and will be 0 at the end.

For each processor i , let H_i be the maximum height of any process that processor i can create before becoming idle. (By "becoming idle," we mean finishing its current process and any process on its queue.) Clearly $H_i < H$ for each i , and $H = \max H_i$, since all potential processes are part of a computation tree being executed or on the queue of some processor. In order to reduce H by 1, we must reduce H_i by 1 in each processor i for which $H_i = H$.

To illustrate using Figure 5.1, if the bottom level shown has height 1, then $H_i = 4$ after the process at the root node has created its children, because those children (whose height is 5) can create processes of height 4. During the evaluation of the

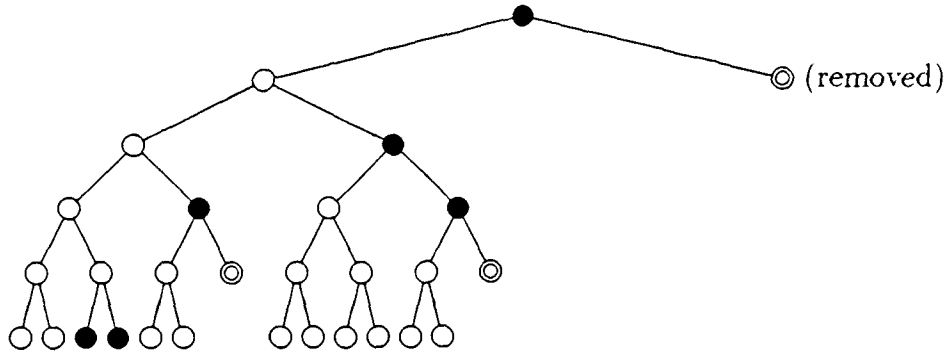


Figure 5.3: Multiple transfers

left half of the tree, H_i remains 4. If a process is transferred as in Figure 5.2, then H_i becomes 3 since this is now the largest height of any process that can be created. This is one way in which H_i can be reduced by 1 in a processor.

However, not every transfer causes H_i to be reduced by 1. A processor in the cascade situation shown in Figure 5.2 is creating many small processes, and if one of these is transferred, H_i will not change. We need to determine how often this can occur before H_i is guaranteed to be reduced.

Lemma 5.3 *At most $p^2 h$ transfers are required to reduce the quantity $H = \max H_i$ by 1.*

Figure 5.3 shows what happens when some of the processes in the “cascade” are transferred. At most one process along each branch of the cascade can be transferred, because for each branch, the first process to be transferred carries with it all of the processes that remain to be created along that branch. Therefore, after at most h transfers, we can be sure that H_i has been reduced by 1 in the processor under consideration.

There are at most p processors for which $H_i = H$, and H_i must be reduced by 1 in each of them. Let us call these the “significant” processors. There are also at most p processors removing processes from them. Each of these cycles among the other processors whenever it becomes idle. Therefore, at least one out every p^2 transfers takes place from one of the significant processors. Furthermore, at least one out of every h of these reduces H_i in a significant processor, as explained above. $\square$

Suspension and resumption of processes can be handled in a way that does not affect the amount of process creation. Under the assumptions we have made, a process will only

suspend its execution when it has created processes for its children, finished execution of the left child, and cannot run the right child because it has been transferred to some other processor. The parent process is then suspended, and resumed when the child has finished execution. Meanwhile, the processor on which the parent was running becomes idle and tries to transfer a process from another processor's queue.

When the child finishes execution, the parent process is resumed. Rather than try to continue it on the processor that it originally ran on (which may now be running some other process), it can be resumed on the processor that was running the child, since this processor is now idle. This results in runtime behavior that is essentially the same as if the original processor had continued running the parent process. In each case, the processor's queue is in the same state (empty) after the child process has finished.

We can now tie the lemmas we have proved into the following result.

Theorem 5.4 *Under the assumptions made thus far, the total number of processes created, in executing a tree of height h , is $O(h^4 p^2)$.*

Since the initial value of H is h , and H is reduced by 1 after every $p^2 h$ transfers, the total number of transfers is bounded by $p^2 h^2$ since then H will be 0 and the computation will be done.

And since each transfer can cause $O(h^2)$ processes to be created, the total number of processes created is $O(h^4 p^2)$. $\square$

5.2 Extending the basic result

Theorem 5.4 is based on several assumptions, which we will now show can be removed with little or no change in the conclusion.

We first consider a generalization of the partitioning method. Instead of creating a process only when the current processor's queue is empty, which results in the queue always having either 0 or 1 processes in it, let us consider having a higher bound on the number of processes in the queue.

Lemma 5.5 *If a tree of height h is executed entirely on one processor, with processes created whenever the length of the queue is less than c , for some constant c , then the number of processes created is $O(h^c)$.*

The case $c = 1$ corresponds to Lemma 5.1. For larger values of c , after the root node has spawned its right child, the execution of the left half of the original computation tree behaves just as a tree of height at most $h - 1$ with a bound of $c - 1$ on the queue length. Then the right child is removed from the queue and executed, so the right

half of the tree, whose height is at most $h - 1$, is executed with a bound of c on the queue size. If $S(c, h)$ is the maximum number of processes created for any c and h , then

$$\begin{aligned} S(c, 0) &= 0 \\ S(0, h) &= 0 \\ S(c, h) &= 1 + S(c - 1, h - 1) + S(c, h - 1) \end{aligned}$$

The solution to this recurrence is the sum

$$S(c, h) = \binom{h}{c} + \binom{h}{c-1} + \cdots + \binom{h}{1},$$

which implies $S(c, h) = O(h^c)$. $\square$

Lemma 5.2 is also affected when up to c processes can be present in each processor's queue. If d of these processes are removed by other processors, then the processor in question can create a "cascade" of $O(h^{d+1})$ new processes. The worst case, therefore, is when $d = c$ and $O(h^{c+1})$ processes can be created.

Theorem 5.6 *If processes are created whenever the current size of the queue is less than c , then the total number of processes created in executing a tree of size h is $O(h^{c+3}p^2)$.*

As in the proof of Theorem 5.4, at most $p^2 h^2$ processes are transferred altogether. Each of these causes $O(h^{c+1})$ other processes to be created, and the product of these factors is the desired bound. As in Theorem 5.4, this term dominates the $O(h^c)$ processes that are created even if no processes are transferred. $\square$

We have also assumed that child processes are created just two at a time. Suppose that, instead, a program creates up to k processes at a time. The quantity k is constant for any program, because the language constructs we are considering, such as `qlet`, do not provide a way of creating a variable number of processes at one time. Such a program therefore corresponds to a computation tree in which each node has at most k children.

The analysis of this situation depends on whether a single partitioning decision is made once to create all k child processes or none; or if a separate decision is made for each child process. The first of these situations is difficult to deal with, because if we create the children whenever the queue size is less than a constant c , we may end up with $c + k - 1$ processes in the queue, whereas our analysis assumed that there are never more than c .

The second situation fits into the framework we have developed, however. If each a node in the computation tree with k children is transformed into a binary tree with

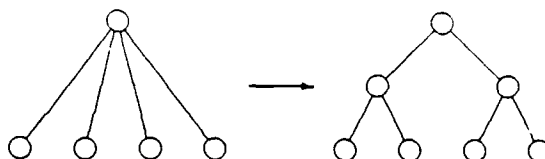


Figure 5.4: Transforming the creation of multiple processes

the k children as leaf nodes, as illustrated in Figure 5.4, then the new tree satisfies our assumptions. The height of this tree is bounded by a constant factor times the height h of the original tree. If we use balanced binary trees as in the figure, the new height is at most $h \lceil \log_2 k \rceil$.

5.3 Making use of dynamic partitioning

We have shown an upper bound of $O(h^{c+3}p^2)$ process creations when the dynamic partitioning method is used, in a certain restricted class of programs. (Binary computation trees, with at most one partitioning decision in each potential process.) For the method to be effective, the time T_{create} spent creating processes should grow more slowly than the sequential execution time T_{seq} .

Our examples satisfied these conditions by having balanced or nearly balanced computation trees, thus making h , the height of the tree, logarithmic in T_{seq} . The overhead of process creation, which is polynomial in h , therefore grows more slowly than T_{seq} as the problem size increases. If we can show that the idle time of a program is also asymptotically zero, then for any given value of p , the performance using the dynamic partitioning method will approach perfect speedup.

To make use of this result in implementing parallel language constructs, care should be taken to make computation trees balanced whenever possible. This strategy can be built into high-level language constructs that map over elements of a set or sequence, for instance.

Chapter 6

Conclusions

We have examined the issues of partitioning and scheduling in parallel Lisp programs by examining several example programs, experimenting with different methods, and performing an analysis of the dynamic partitioning method, because of its promising characteristics.

The simulator that we wrote as part of this research proved very useful, since it allowed fine-grain observation of the effects of the programs that were studied, gave reproducible results, and had the ability to simulate more processors than were present on available parallel machines.

From a small number of example programs we cannot draw fully general conclusions, but we can make some observations based on the experience that has been gained. The first is that the cutoff-based partitioning methods are harder to apply than one might expect. The proper choice of a cutoff parameter depends on many criteria, including the nature of the program, the size of the input data, and the number of processors. Although it was possible to get rid of some of these dependencies, the selection of good cutoff values still required experimentation, and this is not the kind of work that we expect programmers will be willing to spend much time on.

Considering the amount of work needed just to get acceptable performance on our small programs, we believe that partitioning methods such as height or depth cutoff will prove almost totally impractical for large programs. A program that performs more complicated functions will undoubtedly have potential parallelism at many points, and there may be complicated interactions when several sources of parallelism are creating processes simultaneously.

The dynamic partitioning method, on the other hand, is very appealing because it leaves the decision to create processes mostly to the system, but still bases it on information available at runtime, which we argued is necessary for satisfactory performance. Other projects in parallel Lisp have also come to this conclusion, but

without performing a detailed analysis of the reasons for this method's strengths and limitations. This analysis, we believe, is a useful contribution to the study of parallel program execution.

The analysis of dynamic partitioning can be extended in several ways.

1. We assumed that processes are always removed from the queue in the same way. In practice, better results have been observed when transferred processes are removed from the head of another processor's queue (as in FIFO scheduling), while processes removed from a processor's own queue are run in LIFO order. The reasons for this need to be explored.
2. When a process creates subprocesses at more than one point, our analysis does not apply. This kind of behavior occurs in many programs.
3. The average case behavior of dynamic partitioning, rather than the worst case, is of obvious interest. Preliminary experiments [22] show that, for some programs at least, the average-case behavior is proportional to the predicated worst-case performance, but the constant of proportionality is quite small.
4. Programs that use futures create a whole new realm of investigation. Other parallel constructs such as the process closures and `catch/throw` extensions of Qlisp also need to be analyzed.

At the end of the previous chapter we concluded that dynamic partitioning works well with balanced computation trees, specifically those whose total size grows faster than a polynomial function of their height. This bears a resemblance to programs studied in the theory of parallel computation. In that theory, the class NC of problems that can be done in polylogarithmic time on a polynomial number of processors is thought to be easy to parallelize. It might be possible to prove a relation between problems in NC and programs that speed up well using the dynamic partitioning method.

Finally, our analysis of partitioning and scheduling methods helps define goals for improved algorithms. The factor of p^2 in our bounds on process creation means that the overhead on a given problem rises as we increase the size of the machine. We would be happier if the factor was p , meaning that the average rate of process creation per processor is independent of the machine size. Perhaps some refinements to the test used in dynamic partitioning will achieve this goal.

Bibliography

- [1] Hal Abelson et al. The revised revised report on Scheme or an unCommon Lisp. AI Memo 848, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1985.
- [2] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.
- [3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
- [4] Thomas L. Anderson. The design of a multiprocessor development system. Technical Report TR-279, MIT Laboratory for Computer Science, Cambridge, Massachusetts, September 1982.
- [5] Isaac Dimitrovsky. *ZLISP—A Portable Parallel LISP Environment*. PhD thesis, New York University, New York, New York, March 1988.
- [6] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.
- [7] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.
- [8] Richard P. Gabriel and John McCarthy. Queue-based multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 25–44, Austin, Texas, August 1984.
- [9] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. In *Proceedings of HICSS-22, Hawaii International Conference on System Sciences*, January 1989.

- [10] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.
- [11] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164-189, April 1983.
- [12] Sharon L. Gray. Using futures to exploit parallelism in Lisp. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1986.
- [13] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, October 1985.
- [14] W. Ludwell Harrison. Compiling Lisp for evaluation on a tightly coupled multiprocessor. Technical Report 565, Center for Supercomputing Research and Development, Urbana, Illinois, March 1986.
- [15] Peter Henderson and James Morris, Jr. A lazy evaluator. In *Conference Proceedings of the Third ACM Symposium on Principles of Programming Languages*, pages 95-103, Atlanta, Georgia, January 1976.
- [16] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549-557, October 1974.
- [17] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
- [18] Takayasu Ito and Manabu Matsui. A parallel language PaiLisp and its kernel specification. In *Proceedings of the US/Japan Workshop on Parallel Lisp*, Sendai, Japan, June 1989.
- [19] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.
- [20] John McCarthy et al. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1963.
- [21] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1987.

- [22] Dan Pehoushek and Joseph S. Weening. 1989. Report in preparation.
- [23] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical Report CSL-TR-87-328, Stanford University Computer Systems Laboratory, Stanford, California, April 1987.
- [24] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Massachusetts, 1984.
- [25] Guy L. Steele Jr. and Gerald J. Sussman. LAMBDA: The ultimate imperative. AI Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976.
- [26] Guy L. Steele Jr. and Gerald J. Sussman. The revised report on SCHEME: A dialect of LISP. AI Memo 452, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1978.
- [27] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massachusetts, 1977.
- [28] Pete Tinker and Morry Katz. Parallel execution of sequential scheme with paratran. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 28-39, Snowbird, Utah, July 1988.
- [29] Mitchell Wand. Continuation based multiprocessing. In *Conference Record of the 1980 LISP Conference*, Stanford, California, August 1980.
- [30] Alexander Wang. Exploiting parallelism in Lisp programs with side effects. Bachelor's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1986.
- [31] Jon L. White. Personal communication.