October 1990 Report No. STAN-CS-90- 1340

Programming in QLisp

by

I. A. Mason, J. D. Pehoushek, C. Talcott, J. Weening

Department of Computer Science

Stanford University
Stanford, California 94305

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Programming in QLisp	5. FUNDING NUMBERS
6. AUTHOR(S) Ian A. Mason, Joseph D. Pehoushek, Carolyn L. Talcott Joseph S. Weening	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Stanford University	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA/ISTO	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Qlisp is an extension of Common Lisp, to support parallel programming. It was initially designed by John McCarthy and Richard Cabriel in 1984. Since then it has been under development both at Stanford University and Lucid, Inc. and has been implemented on several commercial shared-memory parallel computers. Qlisp is a queue-based, shared-memory, multi-processing language. This report is a tutorial introduction to the Stanford dialect of Qlisp.

14. SUBJECT TERMS parallel Lisp, queue based multi-processing, shared memory, dynamic task spawning	15. NUMBER OF PAGES 56
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRAC

Programming in Qlisp

Ian A. Mason
Joseph D. Pehoushek
Carolyn L. Talcott
Joseph S. Weening

October 24, 1990

Chapter 1

Introduction

Qlisp is an extension of Common Lisp, to support parallel programming. It was initially designed by John McCarthy and Richard Gabriel [5] while they were affiliated with the Lawrence Livermore National Laboratory's S1 project in 1984. Since then it has been under development both at Stanford University and Lucid, Inc. and has been implemented on several commercial shared-memory parallel computers. Qlisp is a queue-based, shared-memory, multi-processing language. A program must explicitly indicate where parallelism is possible by special language constructs. When such constructs are executed a collection of tasks are added to a queue for subsequent evaluation. For more details concerning the rationale behind the languages design, the reader is referred to [5]. The complete definition of Qlisp is in its reference manual [7]; this report is a tutorial introduction to the Stanford dialect of Qlisp.

The reader of this report is expected to have a good working knowledge of Common Lisp, as described in [10] or in textbooks on programming in Common Lisp. Most of the new constructs of Qlisp are described here, but some will not be presented in their most general form; see the Qlisp reference manual for full details.

This primer is organized as follows. Chapters 1 through 3 provide an introduction to the basic Qlisp primitives and gives simple examples of their use. Chapters 4 through 7 contain more substantial applications. Chapter 8 summarizes the main points.

1.1 Reasons for parallel programming

It is worthwhile to keep in mind one's goals and reasons for using a parallel programming language. Programming a parallel computer will probably never be as easy as programming a sequential computer, just because parallel computers are more complicated. The main reason for using a parallel computer is that it offers a potential speedup over sequential computers.

If your goal is to speed up a particular program (or to write a new program for a parallel machine with speed as a primary goal), it is worthwhile to consider other methods of speedup in addition to applying parallelism. Usually these can be found at both the high level (improving the algorithms used by a program) and the low level. In preparing examples for this report, we invariably found low-level speedups, such as reducing the allocation of memory, or replacing linear access to list structures (via functions such as member and assoc) by the use of data structures such as hash tables or trees.

When a parallel program runs on a machine with p processors, one usually expects a speedup of at most p over a machine with one of the same kind of processor. A speedup close to p means you are making efficient use of the machine, so further work on parallelizing the program is not necessary. Qlisp tries to make it easy to achieve such speedup in as many cases as possible. A speedup much less than *p* suggests that better performance is possible, but will require some restructuring of your program, so you must decide whether it is worthwhile to do this. This is obviously a tradeoff between your time and the machine's time!

Occasionally you may be lucky enough to see speedup greater than *p*. At first thought this appears impossible, but it can happen when a parallel program does less work than a corresponding sequential program-this often happens in search algorithms-or if the parallel program makes good use of hardware other than the additional processors. A parallel machine generally has more memory, caches, I/O bandwidth, etc., than a sequential machine and in some cases this additional hardware reduces a bottleneck in the program.

1.2 Overview of how to use Qlisp

In the following chapters we will introduce Qlisp by means of several example programs. Some of these were originally written as ordinary Common Lisp programs, while others were developed completely in Qlisp. A summary of the topics we will cover is as follows.

1. *Correctness.* The most important aspect of any program is that it be correct. This is just as true for parallel programs, but the process of making a program correct, debugging, is often harder in a parallel environment. So, you should do as much debugging as possible using a single processor, and only then try to run the program in parallel. Some bugs will not show up when a program is run sequentially; we will give examples of these and show how to avoid them in writing parallel programs.

2. *Things to avoid.* Certain programming techniques make parallelism difficult to achieve. Also, some algorithms are inherently more sequential than others and need to be avoided. In our examples we will show what makes a program difficult to parallelize and how to fix such programs.

3. *Identifying parallelism.* Even after the obstacles to achieving parallelism have been removed, the computer may need some help in deciding what to do. Qlisp provides several ways for the programmer to indicate which parts of a program should be run in parallel.

4. *Limiting parallelism.* Most programs have either too little parallelism, or more parallelism than necessary, for the particular environment in which they are run. When there is too little parallelism, you must either look for more, or use what parallelism there exists and accept less than perfect efficiency. When there is too much, however, it often pays to limit the amount of parallelism actually used, to avoid unnecessary overhead. Qlisp's language constructs are designed to make the expression of these tradeoffs straightforward, and to make the necessary decisions at runtime, when the largest amount of information is available about the program, its data, and the state of the runtime environment.

5. *Dynamic Scheduling.* Qlisp comes with a scheduling environment that automatically decides when to limit parallelism, based on the runtime state of the machine. Qlisp's default scheduling heuristics work well on most programs, so we always try them first before trying to come up with other runtime decisions that let a specific program control its parallelism.

6. *Profiling tools.* Qlisp comes with several tools for measuring the performance of a program, and indicating what parts of the program may be causing poor performance. Using these tools is an effective way to "tune" a program until a desired level of performance is reached. The major parameters estimated by the timing tool are idle time and scheduling overhead. An effective way to use this information when developing a program will be demonstrated in the following chapters.

1.3 Introduction to Qlisp

1.3.1 Creating parallel processes

The creation of parallel processes is expressed by `qlet`. Here is an example:

```
(qlet (spawnp) ((a (search u v1))
                (b (search u v2)))
  (append a b))
```

The effect of this expression is similar to the Common Lisp code:

```
(let ((a (search u v1))
      (b (search u v2)))
  (append a b))
```

except that in the qlet form, the expressions (search u vl) and (search u v2) will be executed in parallel if the expression (spawnp) returns a non-null value. The body of the `qlet` form, (append a b) , is executed after the processes computing the calls to search have finished and returned their values.

More generally, `qlet` forms have the following syntax.

(`qlet` *control* ((var_1 *expr,)* . . . *(var,* $expr_n$)) *body)*

This is like let except for the additional form *control,* which is called a *control expression.* A control expression should be either a constant or an expression that returns nil, t or the keyword : eager.

In most of our examples, *control* will be the form (spawnp). The initial definition of spawnp is

```
(defmacro spawnp () '(dynamic-spawn-p))
```

dynamic-spawn-p implements a control algorithm, to be described later, that works well on a variety of programs. Using this extra level of indirection, instead of coding (dynamic-spawn-p) directly into your program, will allow you to try other control algorithms without changing your code.

When *control* evaluates to nil, `qlet` behaves just like ordinary let, i.e., there is no parallelism. When *control* evaluates to t (or any other non-null value except : eager), Qlisp creates processes to evaluate the *expr,* forms of the `qlet` in parallel, waits for these to finish, binds the resulting values to the approapriate var_i and then evaluates the body.

Evaluation of the body can begin before the $expr_i$ forms have finished, if *control* has the value : eager. You might ask, what values are the variables var_i bound to during the evaluation of the body, since the expressions whose values they are supposed to contain have not yet been evaluated? A new data type, called a *future*, is used for this purpose. It represents a. value being computed by another process, and can be passed around like other Lisp data objects. (In the implementation, it is a pointer with a special tag.) Certain Lisp functions, such as car, cdr, atom, +, need the actual values of their arguments; when passed a future in Qlisp, they will automatically wait for the process computing this value to finish, and then resume when there is an available processor.

1.3.2 Synchronization

When parallel processes share data that can be modified, it is often necessary to add synchronization in order to make the program correct. Qlisp's basic form for this is qlambda. It has the following syntax.

(qlambda *control* (var_1 . . . var_n) *body)*

This is just like lambda except for the *control* argument, which controls parallelism. Also like lambda, you must use function around a qlambda expression in the places where Common Lisp requires it, or use the `#'` syntax of the Lisp reader, which is equivalent. The object returned by `#'(qlambda ...)` is called a *process closure* and may contain free variables like an ordinary closure.

In all cases (even if *control* is nil), Qlisp will allow only one process at a time to call a process closure. If a process tries to call a process closure while another call is in progress, it will be suspended; later it will be resumed and proceed to execute the forms in the qlambda body.

If *control* is nil, this is the only difference between qlambda and lambda. If *control* is t, the process closure returns immediately to its caller, with a future as the returned value. Thus the callers of a (qlambda t . . .) process closure never wait, although the calls themselves are performed sequentially by another process. The returned futures might cause suspension of a process later on.

Note that the *control* argument of a qlambda form is evaluated when the process closure is created, not each time it is called. An elaborate example of the use of qlambda may be found in parallelizing the OPS5 Matching Algorithm [8]

1.3.3 Speculative computation

Sometimes a program may create a process and discover, before the process finishes, that its result is not needed. In this case the program can make better use of resources by terminating the process rather than allowing it to continue execution. This is called *speculative computation* because the program could have avoided starting such a process until it was certain to be needed.

Qlisp supports speculative computation by estending the meaning of Common Lisp's catch and throw forms. When a throw returns control to a corresponding catch, in addition to performing any unwind-protect forms, Qlisp will kill processes, started within the catch that are still executing. The exact definition of which processes are killed is in the Qlisp reference manual.

The most common way of doing speculative computation is

```
(catch 'found
  ((code to try multiple solutions in parallel)))
```

and when a. solution is found,

```
    (throw 'found (solution))
```

The throw returns control to the catch form, and any processes started by the code to try multiple solutions in parallel that are still running will be killed. Thus, the solution returned will be that of the first process that finds one.

Chapter 2

Writing parallel programs

Common Lisp is designed to allow a variety of programming styles. It allows both pure functional programming and programming with effects; recursion and iteration; global and local variables; global and local data; lexical and special binding, etc. In a sequential program, the choice of which constructs to use is usually a matter of personal style or finding ones appropriate to the problem at hand. There are some efficiency considerations, but they are not too significant, and vary from one implementation of Common Lisp to another.

In Qlisp, because of the need to use multiple processors effectively, some styles of programming can lead to significant performance loss. We will discuss the most common cases of this phenomenon, and explain how to write programs that have a good chance of running well in parallel without significant additional effort.

2.1 High-level programming forms

The use of high-level programming styles such as iteration can make a program appear to have constraints on the sequence of operations, when in actuality those constraints are unnecessary. For example, a program may loop over the elements of a list or array, but the operations done on each element are independent and can be done in any order, or in parallel.

For parallel programming, it is useful to have the program explicitly indicate that such operations are safe to perform in parallel. Qlisp's "parallel iteration forms," along with other ways of indicating parallelism, can then be used to build higher-level functions that operate on aggregate data objects.

Qlisp has functions and macros that parallelize the work done by the Common Lisp forms `dolist`, `dotimes`, and the mapping functions `mapc`, `mapcar` etc. These are called `qdolist`, `qdotimes`, `qmapc`, `qmapcar` etc. The syntax of these Qlisp forms is exactly like the corresponding Common Lisp forms. For example, you can write

```
(qmapcar #'paint widgets colors)
```

and Qlisp will generate code equivalent to that for

```
(mapcar #'paint widgets colors)
```

except that it uses parallelism. The method that Qlisp uses for parallelizing iterations is based on the (dynamic-spawn-p) control form; it is described in more detail in chapter 3 of this report.

To use the parallel iteration functions, your program must work correctly no matter what order the iterations are performed, or if they are performed in parallel. If the body of an iteration form (or an argument function passed to a mapping function) performs side effects, you should be careful before trying to execute it in parallel. The mapping functions that return lists do guarantee that the result lists are in the right order, but they may be computed in a different order.

2.2 Variables

The two types of variables in Common Lisp are lexical and *special.* In addition, we will sometimes talk about *global* variables; these are symbols that have been assigned values (by means of setq, defvar and similar forms) but have not been bound as function parameters or by let or lambda. Common Lisp treats global variables as special variables.

We also distinguish between two ways of using variables, which we will call *shared* and *private.* Shared variables are those which may be used (their values read or changed) by more than one process at a time. Private variables can only be used by a single process. Sometimes we can't tell whether a variable is shared or private, because we don't have the whole program in hand to make a determination.

Here are the ways in which we determine if a variable is shared or private. When in doubt, we always take the conservative approach of saying that a variable *might* be shared if we cannot prove that it is private.

1. Global variables are shared, since references to global variables can appear in any function. You must therefore be cautious when using global variables in a parallel program. Legitimate uses of global variables are:

    - Variables that are modified only while the program is not using parallelism, but that may be read by concurrent processes. For example, you might have some "program parameters" that are easier to handle this way than by passing them as arguments to all the functions that need them.
    - Variables that are used in conjunction with synchronization, to ensure that only "safe" parallel references are possible. We will show how to do this later. Generally, this requires making sure that only one process is using the variable at a time, and so may reduce the amount of parallelism in your program. Thus it should be avoided when possible.

2. Lexical variables can be shared or private. The lexical scope rules of Common Lisp make it possible to determine, by inspection of the program, whether a variable is private. This is the case if, inside the scope of the form where the variable is defined, there are no forms that can possibly execute in parallel, and no forms that "capture" the binding of the variable in a closure. (Because such a closure'.could then be called in more than one process simultaneously.)

3. Special variables can also be shared or private. When a special variable is bound by a process, the binding is only seen in that process and its descendants; there is no effect on other processes that were previously sharing the variable's binding. Thus, rebinding a special variable can make it private during the extent of the binding, even though it is otherwise shared.

However, there are several reasons to avoid the use of special variables in Qlisp programs. One is that, as with global variables, it is hard to tell whether a reference to a variable is safe when it is not inside a binding form. Programs using special variables are therefore often harder to maintain and modify, because you need to understand more of the program than just the part you are modifying.

Another reason to avoid special variables is that they are somewhat slower, both to read and write, than lexical variables. The difference is not very great, so this should not be the primary consideration. But with other things being equal the use of lexical variables should be preferred.

Here are some examples of the use of variables in Qlisp programs.

```
(let ((item (car objects)))
  (qlet (spawnp) ((x (search item list1))
                  (y (search item list2)))
    (cons x y)))
```

Here the variable `item` is shared by the forms (search item listl) and (search item `list2`). Since both of them simply pass the value of item as an argument to the function search, the usage of this shared variable is safe.

```
(let ((index 0))
  (flet ((new-index ()
           (incf index)))
    (qlet (spawnp) ((x1 (make-object (new-index) y1))
                    (x2 (make-object (new-index) y2))
                    (x3 (make-object (new-index) y3)))
      (list x1 x2 x3))))
```

This code is **incorrect** since the variable index is modified in an unsafe way, by processes running in parallel. The Common Lisp form (`incf` index) is equivalent to (setq index (+ index 1)), and when this is executed in parallel by multiple processes, it does not necessarily return a unique value in each process. To fix it we must synchronize using `qlambda`, or we can use the equivalent form qflet as follows:

```
(let ((index 0))
  (qflet nil ((new-index ()
                (incf index)))
    (qlet (spawnp) ((x1 (make-object (new-index) y1))
                    (x2 (make-object (new-index) y2))
                    (x3 (make-object (new-index) y3)))
      (list x1 x2 x3))))
```

Note that we have specified nil as the control form of the process closure defined by qflet. This is because we only need the synchronization features of the process closure; having it execute in parallel with the processes that call it would create more overhead (in the use of futures) than it would gain in extra parallelism.

```
(defvar *color-list* '(yellow))

(defun test-color (color)
  (let ((*color-list* (cons color *color-list*)))
    (check-color)))
```

```
(qlet (spawnp) ((x (test-color 'blue))
                (y (test-color 'green))
                (z (test-color 'red)))
  (append x y z))
```

This code is safe in its use of the special variable *color-list*. The function test-color, called in three parallel processes, rebinds *color-list* in each process. These bindings have no effect on each other, so the value of *color-list* seen by check-color is always (blue yellow) in the first process, (green yellow) in the second process, and (red yellow) in the third process, no matter how they execute.

Here are some other ways of writing test-color in the above program.

```
(defun test-color (color)
  (push color *color-list*)
  (check-color)
  (pop *color-list*))
```

This is **incorrect** as a parallel program, even though it is correct as sequential code.[1] The reason it doesn't work is that (push color *color-list*) expands to

```
(setq *color-list* (cons color *color-list*))
```

and this code contains a critical region. In this version of test-color, the variable *color-list* is shared by all of the processes.

We might try to create "atomic push" and "atomic pop" operations that add synchronization around the critical region. However, this would still not fix the function as written above. The reason is that with several processes doing push and pop to the same variable, *color-list* may have a value such as (green blue yellow) when check-color is called. This does not happen when the program is run sequentially.

```
(qdefun test-color (color)
  (push color *color-list*)
  (check-color)
  (pop *color-list*))
```

This adds synchronization at a high enough level to execute correctly. Calling a function defined by qdefun, which performs the same synchronization as qlambda, ensures that only one process at a time enters the code that does the push and pop. However, in doing this we have removed all the parallelism from the program! This is why rebinding of special variables, as done in the first version of `test-color`, is the best method to use.

2.3 Shared data

Even when private variables are used, it is possible for data to be shared between processes in a program. If shared data is modified by one process, it may affect other processes in unintended ways.

There is no problem with shared data as long as the program does not use destructive operations, since then the only time that words in memory are written is when they are allocated. (For example, cons writes its two argument values into the slots of a newly-allocated cons cell.)

[1] If there is a possibility of a `throw` from inside `check-color`, then `(pop *color-list*)` should be contained in an `unwind-protect` form.

All other references just read the data, and there is no problem with doing this concurrently in several processes.

Destructive operations include rplaca, rplacd, setf and anything else that modifies storage after it has been allocated. One approach to writing correct parallel code is to avoid entirely the use of these forms. Often this is undesirable because it would sacrifice efficiency, or because you are modifying a sequential program that already contains destructive code, and want to make as few changes as possible.

If destructive operations are used, then, they must be used safely. To do this, you must either prove that the program is correct when run in parallel in spite of the destructive operations, or add synchronization code to the program where it uses shared data. In the latter case, it is usually necessary to modify code that reads the shared data, as well as the code that writes it.

Chapter 3

Using Qlisp

In this chapter, we illustrate the tools and techniques of Qlisp program development, debugging and performance analysis that have been described. Our experiments were done with 'an implementation of Qlisp, based on Lucid Common Lisp, running on an Alliant FX/8, a shared-memory multiprocessor with eight processors.

3.1 Expressing parallelism

If the parallelism in your program is not easy to express with the high-level forms that Qlisp provides, you will need to describe it more directly. In most cases this is done with `qlet`.

In order to write a `qlet` expression, you must decide which Lisp forms should be evaluated in separate processes. Then create a local variable to receive the returned value of each of the parallel forms, and use these variables in an expression. For example:

```
(qlet (spawnp) ((x (crunch a1 b1))
                (y (crunch a2 b2)))
  (list x y))
```

performs two function calls to crunch in parallel, and then uses the returned values in an ordinary sequential computation. The use of (spawnp) as a control expression was described in section 1.3.1. If you decide that a particular `qlet` expression needs a different form of control than others, you should give it a different control expression, which you can then redefine as necessary.

When writing new code in Qlisp, the style just described is fairly natural and easy to write. However, you may be converting an existing sequential program to Qlisp. It probably does not contain a form like

```
(let ((x (crunch a1 b1))
      (y (crunch a2 b2)))
  (list x y))
```

since this is very verbose. Instead, normal Common Lisp style would be to write

```
(list (crunch a1 b1) (crunch a2 b2))
```

In order to convert this to Qlisp, you need to take the parallel forms (the calls to crunch in this case) out of the expression they are contained in, replacing them by variables, and then write a `qlet` form to create the parallel processes.

In this way, you can selectively decide what is to be done in parallel. For example, suppose the original form was

```
(list (list a1 b1) (list a2 b2) (crunch a1 b1) (crunch a2 b2))
```

The expressions (list a1 b1) and (list a2 b2) are too small to benefit from running in parallel with the calls to crunch. It would take longer to create processes for them, than to simply execute them sequentially. But perhaps crunch performs a long computation and there are no side effects that prevent running the two calls to crunch in parallel. In this case, an appropriate `qlet` form would be

```
(qlet (spawnp) ((x (crunch a1 b1))
                (y (crunch a2 b2)))
  (list (list a1 b1) (list a2 b2) x y))
```

3.2 Abbreviated syntax

A pattern that comes up often is the evaluation of all of the arguments of a function call in parallel, using the default control expression (dynamic-spawn-p). Converting such a form

(*fun* $arg_1 \ldots arg_n$)

into the corresponding `qlet`, although straightforward, requires a lot of editing and forces you to come up with new variable names. Instead of this, you can write

#? (*fun* $arg_1 \ldots arg_n$)

The **#?** syntax is very convenient and we will use it in many of the examples below. You can also write

#! (*fun* $arg_1 \ldots arg_n$)

to represent the corresponding (`qlet` t . . .) form, i.e., a form that creates processes unconditionally. The **#?** and **#!** constructs can also be placed before a progn form, i.e.,

#? (progn $form_1 \ldots form_n$)

In this case, the values returned by all but the last $form_i$ are discarded, and the value of $form_n$ is returned.

3.3 Dynamic Scheduling

A major feature of Qlisp is the ability to control, at runtime, the way in which processes are created and distributed among the processors. There are two ways in which you can tell the system how to manage processes.

- The control expressions of the forms that create processes determine whether or not a process is actually created at each evaluation. This is the main way in which programs interact with the Qlisp runtime environment.
- The *scheduler* determines which processes are run on which processors at any given time. A default scheduler is supplied with Qlisp, but you can replace it with your own scheduler. Before you do this, though, you should determine whether it is really necessary. Most programs can easily be made to perform as well with the default scheduler as with any other, and a specialized scheduler that you write may not work well if your code is combined with other Qlisp code.

The default Qlisp scheduler is based around a set of process queues, one per processor. These are actually double-ended queues (sometimes called "deques"), so that processes can be added or removed at each end. When processes are created, they are placed in the "local" queue (the queue of the processor that created the process), at a particular end. Let us call this end of the queue the "head," and the other end the "tail." When there are several processes in a queue, then, the one most recently created is at the head and the one least recently created is at the tail.

When a processor becomes idle (because the process it was running has either finished or suspended), it first tries to remove a process from its own queue by taking the most recently created process, from the head of the queue. However, when the local queue is empty, it looks in the queues of other processors. When it finds a non-empty queue, it takes the process at the tail of the queue. These choices (head from one's own queue, tail from another processor's) are not arbitrary; they have been shown both theoretically and experimentally to improve performance for many programs. A phrase that we associate with this scheduling strategy is "locally LIFO, globally FIFO." Effectively, the local queue behaves like a stack of processes; tasks get pushed onto and popped from the local queue (last in first out). When accessing other processors' queues, the behavior is closer to standard, first in first out, queue-like behavior.

The (dynamic-spawn-p) control expression uses the state of the scheduler to control the creation of processess. In the normal case, it tests whether the queue of the processor that it is running on is empty. If so, it returns t and causes a process to be created, thus making the queue non-empty for the next test (until the process is removed to be run on some processor). If the queue is non-empty, (dynamic-spawn-p) returns nil and no process is created.

This form of control may appear somewhat strange, but, like the default scheduler, it has been shown to perform well for many programs. In some cases, however, (dynamic-spawn-p) may not create enough processes, and there is unnecessary idle time. An extension of the basic idea is to look at the size of the local processor's queue, and keep generating processes when it is below some threshold. This is expressed by writing

```
(dynamic-spawn-p n)
```

as the control expression, where n is a small integer that gives the maximum process queue size. Using $n = 1$ is equivalent to writing just (dynamic-spawn-p). You should try this form if you discover your program has too much idle time. Try $n = 2, 3, \ldots$. If there is no improvement, you must look elsewhere for the cause of the idle time. Usually there will be an improvement up to a certain value of n, but for higher values of n the performance will start to degrade as the program starts to create many unnecessary processes. To make such fine tuning as simple as possible, the abbreviated syntax has been extended to include

#n? *(fun* $arg_1 \ldots arg_n$)

3.4 Using Qlisp

Let's assume that you have written some Qlisp code and would like to test it. Running Qlisp is very much like running ordinary Lisp. To start, type "qlisp,' to the Unix shell prompt.

```
% qlisp
```

After some initial comments, Qlisp will print its prompt, and you are now in a Lisp read-eval-print loop.

```
;;; Lucid Common Lisp, Qlisp version 1.1
>
```

At this point you can interact with Qlisp just as with an ordinary Lisp interpreter. Unless you use the function qeval, however, it will not create any parallel processes[1] Therefore, you must type

(qeval *form*>

in order to evaluate a *form* that uses parallelism.

Interpreted Lisp forms can use parallelism, but the speed gained by parallel processes is offset by the relative slowness of interpreted code over compiled code. Therefore, you will usually want to compile your Qlisp code as soon as it seems to be bug-free, before doing any performance measurements. The functions compile and compile-file behave just as in Common Lisp. The compiler itself is not (yet) a parallel program, so you should not try to compile in parallel.

The usual steps in developing a Qlisp program are therefore:

1. Make sure your program works as a sequential program. For debugging purposes, it is often better to use the interpreter at this stage.

2. Test the parallel version of the program, either as interpreted or compiled code.

3. Run the parallel program with the default runtime environment (i.e., with (spawnp) forms set to (dynamic-spawn-p). If this produces acceptable performance, you are done.

For performance measurement, Qlisp has a qtime form that works like Common Lisp's time, but uses qeval automatically. Note that time and qt ime are macros that don't evaluate their argument forms, so you should not quote the form to be evaluated.

If the program's performance after the steps above is not acceptable, you will need to decide whether it has too little or too much parallelism. The statistics typed out by qtime indicate whether there was a lot of idle time (indicating not enough parallelism to keep the processors busy) or a lot of overhead caused by creating and scheduling processes.

Excessive idle time can happen when not enough parallelism has been identified, or when the control algorithm is not creating enough processes. To distinguish between these, you can temporarily substitute a control algorithm that creates all possible processes. This is easy to do if you have used control expressions such as (spawnp) in our examples, which can be redefined as functions or macros that always return t. If you have used the abbreviated form **#?**, replace it with **#!**. In either case, you will need to recompile your code, since these control espressions are partly processed at compile time by macro expansion.

Note that the parallel iteration functions qdotimes, qmapcar, etc. cannot be converted to "spawn-always" forms. They cont rol their process creation in a way that will not improve by spawning more processes.

If, after converting your expressions to create all possible processes, the program still has a lot of idle time, then it does not have sufficient parallelism to run efficiently on the given number of processors. You should look for more parallelism in the program, or a new algorithm with more parallelism, in order to speed it up.

On the other hand, if the program now has very little idle time, then it is just a matter of finding the right control expressions to avoid excessive overhead. Try redefining your control

[1]This may change in future versions of Qlisp.

expressions to (dynamic-spawn-p n), with n a small integer. Recompile and test again to see if this improves the performance.

In the early stages of testing, you should allow for the. fact that your program may run a lot slower than you would like. This is to be expected when you are testing it sequentially, and may still be the case if there are performance problems. Therefore, it will be very helpful to prepare test data that represents a simpler version of the problem that your program is expected to solve, and that should run much faster. -Otherwise, the time you spend waiting during debugging of you code may exceed the savings you achieve by parallelizing it!

3.5 A simple test program

Our first example program is very simple, so that we can concentrate on illustrating the usage of Qlisp. It computes numbers in the *Fibonacci sequence:* 0, 1, $1, 2, 3, 5, 8, 13, 21, \ldots$; each Fibonacci number is the sum of the previous two. Here is a sequential version of the program:

```
(defun fib (n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This program uses an inefficient algorithm, since it computes many values of (fib n) more than once. So in a real-life situation, we would use a new algorithm before attempting speedup by using parallelism. Nevertheless, let us proceed to convert the above code directly into Qlisp.

Each call to (fib n) with n $\geq$ 2 results in the two recursive calls (fib (- n 1)) and (fib (- n 2)). These can run in parallel since they are completely independent computations. Since these forms make up the arguments to a function call (the call to +), we can use the abbreviated syntax described in section 3.2 and rewrite the program as follows:

```
(defun fib (n)
  (if (< n 2)
      n
      #?(+ (fib (- n 1)) (fib (- n 2)))))
```

There is no other obvious parallelism, so we are ready to test the program. The sequential version of fib takes about 2.3 seconds to compute (fib 25), but only 0.2 seconds to compute (fib 20). Let us use (fib 20) as our debugging test, and (fib 25) when we are done as the "real data" test.

After compiling the code above, we test it and get the following output:

```
> (qtime (fib 20) )
Parallel Time: 43 msecs on 8 processors
Processes:       1924
Overhead:       74.8 msecs, 21.7%
Idle :          31.7 msecs,  9.2%
6765
```

The value 6765, typed at the end, is the result of (fib 20). The interesting figures are the percentage of time in overhead (process creation and scheduling) and idle time. Both of these are rather high. In order to see if idle time can be reduced, we change the use of #? to #!, recompile, and run it again:

```
> (qtime (fib 20))
Parallel Time: 63 msecs on 8 processors
Processes:      10946
Overhead:      277.2 msecs, 55.0%
Idle:           20.6 msecs, 4.1%
6765
```

The idle time has decreased somewhat, but not as much as the increase in overhead, so the total running time is worse. We therefore make one more test, using `#2?`, to see if it will make an improvement:

```
> (qtime (fib 20))
Parallel Time: 39 msecs on 8 processors
Processes:       1874
Overhead:       54.8 msecs, 17.5%
Idle:           22.3 msecs, 7.1%
6765
```

This is slightly better than the original test with `#?`.

However, when the total running time is as short as in these tests (43, 63 and 39 milliseconds), you should beware of the significance of the timings. In our "real data" test of `(fib 25)`, the time becomes large enough to put more faith in the numbers.

With `#?`, we get:

```
> (qtime (fib 25))
Parallel Time: 367 msecs on 8 processors
Processes:       3673
Overhead:      112.2 msecs, 3.8%
Idle:           29.2 msecs,  1.0%
75025
```

Using # ! , we get:

```
> (qtime (fib 25))
Parallel Time: 718 msecs on 8 processors
Processes:      121393
Overhead:      3023.1 msecs, 52.6%
Idle:           22.4 msecs, 0.4%
75025
```

Finally, with `#2?`, the running time is:

```
> (qtime (fib 25))
Parallel Time: 367 msecs on 8 processors
Processes:       5268
Overhead:      141.9 msecs, 4.8%
Idle:           151.2 msecs, 5.2%
75025
```

These and other results are summarized in the following table.

(fib 25) (sequential 2.540)						
para-macro	para-time	processes	overhead	%	idle- time	%
-						
#?	2.564 0.367	3673 1	0.1122 0.0	0.0 3.8	0.0292 17.702	86.3 1.0
#2?	0.367	5268	0.1419	4.8	0.1512	5.2
#3?	0.402	15952	0.4092	12.7	0.0248	0.8
#4?	0.471	36790	0.9251	24.6	0.0203	0.5
#!	0.718	121393	3.0231	52.6	0.0224	0.4
Best Speed-up (/ 2.540 0.367) = 6.921						

Creating all processes with **#!** still gives the worst time, because of too much overhead. but now the best times are achieved either by the default control algorithm **#?** or by **#2?**. This trend continues as n increases, as the following table of `fib 30` illustrates.

(fib 30) (sequential 28.249)						
para-macro	para-time	processes	overhead	%	idle-time	%
-						
#?	28.259 3.850	10733 1	0.3074 0.0	0.0 1.0	196.022 0.0198	86.7 0.1
#2?	3.888	14515	0.3743	1.2	0.0178	0.1
#3?	3.952	40329	1.0145	3.2	0.0130	0.0
#!	7.596	1346269	33.4049	55.0	0.0147	0.0
Best Speed-up (/ 28.249 3.850) = 7.34						

Chapter 4

Theorem proving

This program is a Common Lisp version of the well known Boyer benchmark. Here is what Bob Boyer said about the original program ([4]. p116).

> J Moore and I wrote the rewrite program as a quick means of guessing how fast our theorem-proving program would run if we translated it into some other Lisp system. Roughly speaking, it is a rewrite-rule-based simplifier combined with a very dumb tautology-checker, which has a three-place IF as the basic logical connective.

In the first section we describe the general structure of the program and its underlying algorithms. Then we describe new-boyer, our implementation using abstract syntax and other high-level programming constructs. In the second section we discuss how to parallelize new-boyer and present a spectrum of experimental results. In the third section we give two levels of optimization of new-boyer and corresponding experimental results. The final version is equivalent to the original benchmark code, Gabriel [4], p.116, with the exception that global variables are made local. Two observations about this set of examples are the following. Firstly, parallelization of the optimized versions exactly corresponds to that of new-boyer-we haven't changed the algorithm, just the representation of structures. Secondly, certain optimizations not only speed up the sequential version but also give improved speed-up in the parallel case. This is due to the reduction of time spent locking and unlocking shared data structures. This phenomenon is discussed further in the final chapter.

4.0.1 The Program

The benchmark consists of running the program tautp on a particular symbolic term. A symbolic term is either an atomic term or a composite term consisting of an operator and a list of arguments. tautp rewrites the term according to a long list,

REALLY-BIG-LIST-OF-LEMMAS,

of rewriting rules or lemmas as they are called by Boyer and Moore. The dumb tautology-checker `tautologyp` is then applied to the rewritten term. In new-boyer the structure of terms is expressed abstractly using the Common Lisp def `struct` mechanism. An atomic term simply consists of a name (represented as a Lisp symbol), while a composite term is consists of an operation together with a list of arguments, themselves terms.

```
(defstruct (term (:print-function term-print)))
(defstruct (atomic-term (:include term)) name)
(defstruct (composite-term (:include term)) op args)
```

An operation consists of a name (represented as a Lisp symbol) together with a list of lemmas associated with it. The lemmas component of an operation is an annotation that allows us to speed up the search for a lemma that applies to a give term. Given the name, **<name>**, of an operation the corresponding operation is the value of (the-op <name>). This is to insure that there is unique operation associated with each name, and hence a unique list of lemmas for each operation.

```
(defstruct (op (:print-function op-print)) name lemmas)

(defun the-op (sym &optional symbol-table)
  (if symbol-table
      (get-op sym symbol-table)
      (let ((op (get sym (quote op))))
         (if op
             op
             (let ((op (make-op :name sym)))
                (setf (get sym (quote op)) op) op)))))
```

A lemma has three components: an operator, a list of arguments, and a right-hand term. A lemma corresponds to an equation whose left-hand term is the term whose operation and arguments are those of the lemma and whose right-hand term is the right-hand term of the lemma.

```
(defstruct lemma op args rhs)
```

Since terms include annotations such as the list of lemmas associated with an operator we also define a term equality test term-eq which only looks at the abstract term structure and not at the annotations.

```
(defun atomic-term-eq (term1 term2)
  (eq (atomic-term-name term1) (atomic-term-name term2)))

(defun op-eq (op1 op2) (eq (op-name op1) (op-name op2)))

(defun term-eq (term1 term2)
  (cond ((and (atomic-term-p term1) (atomic-term-p term2))
         (atomic-term-eq term1 term2))
        ((and (composite-term-p term1) (composite-term-p term2))
         (and (op-eq (composite-term-op term1) (composite-term-op term2))
              (term-list-eq (composite-term-args term1)
                            (composite-term-args term2))))
        (t nil)))

(defun term-list-eq (terms1 terms2)
  (if (and (null terms1) (null terms2))
      t
      (and (term-eq (car terms1) (car terms2))
           (term-list-eq (cdr terms1) (cdr terms2)))))
```

In the original boyer benchmark symbolic terms are represented using Lisp symbols and lists in the usual way, with no abstract syntax used in writting the program. Lemmas are terms with operator EQUAL and left-hand side a composite term. In order to speed up the search for a lemma matching a given term, the list of lemmas is partitioned according to the operator of the left-hand term and the sublist for a given operator is stored on the property list of that operator. The functions term-2-sexp, sexp-2-term, and sexp-2-lemma give the formal correspondence between our representation of terms and lemmas and the representation used in the original program. These allow us to translate the original lemma list into our representation and to compare intermediate results of the two versions.

```
(defun term-2-sexp (term)
  (cond ((atomic-term-p term) (atomic-term-name term))
        ((composite-term-p term)
         (cons (op-name (composite-term-op term))
               (mapcar #'term-2-sexp (composite-term-args term))))
        (t (error "~%term-2-sexp did not understand the term: ~a" term))))

(defun sexp-2-term (sexp)
  (if (atom sexp)
      (make-atomic-term :name  sexp)
      (make-composite-term :op (the-op (car sexp))
                           :args (mapcar #'sexp-2-term (cdr sexp)))))

(defun sexp-2-lemma (sexp)
  (make-lemma :op (the-op (caadr sexp))
              :args (mapcar #'sexp-2-term (cdadr sexp))
              :rhs (sexp-2-term (caddr sexp))))
```

We provide printing functions for terms in order to be able to examine intermediate results more easily.

```
(defun term-print (term stream pl) (pprint (term-2-sexp term) stream))
(defun op-print (op stream pl) (print (op-name op) stream))
```

Finally we provide special abstract syntax for conditional expressions: make-if, ifp, if-test, if-then, if-else. Note that ifp assumes it is given a composite-term and the selectors assume the argument is an if-term.

```
(defun make-if (test-term then-term else-term)
  (make-composite-term :op (the-op (quote if))
                       :args (list test-term then-term else-term)))
(defun if-test (term)  (car (composite-term-args term)))
(defun if-then (term)  (cadr (composite-term-args term)))
(defun if-else (term)  (caddr (composite-term-args term)))
(defun ifp (term)  (op-eq (the-op (quote if)) (composite-term-op term)))
```

The rewrite program takes as input a term. If the term is atomic the program exits with that term as its value. Otherwise the term is a composite term consisting of an operation, op, and an list of argument terms. The argument subterms are first rewritten and then the resulting whole term is rewritten via the first lemma of op that it matches. This is repeated until no more rewriting can be done. The auxiliary function rewrite-with-lemmas does the actual lemma matching and rewriting using the programs match-args and apply-subst, respectively.

```
(defun rewrite (term)
 (labels ((rewrite-with-lemmas (op args lemmas)
            (if (null lemmas)
                (make-composite-term :op op :args args)
                (multiple-value-bind (success? sublist)
                    (match-args args (lemma-args (car lemmas)) nil)
                    (if success?
                        (rewrite (apply-subst sublist (lemma-rhs (car lemmas))))
                        (rewrite-with-lemmas op args (cdr lemmas)))))))
        (cond ((atomic-term-p term) term)
              ((composite-term-p term)
               (let ((op (composite-term-op term)))
                 (rewrite-with-lemmas op
                                      (mapcar #'rewrite
                                              (composite-term-args term))
                                      (op-lemmas op))))
              (t (error "~%rewrite did not understand the term: ~a" term)))))
```

apply-subst applies a substitution list to a term in the usual manner.

```
(defun apply-subst (sublist term)
  (cond ((atomic-term-p term)
         (let ((bind (assoc term sublist :test #'atomic-term-eq)))
           (if bind (cdr bind) term)))
        ((composite-term-p term)
         (make-composite-term  :op (composite-term-op  term)
                               :args (mapcar #'(lambda (t1)
                                                 (apply-subst sublist t1))
                                             (composite-term-args term))))
        (t (error "~%apply-subst did not understand the term: ~a" term))))
```

The main work in rewriting is matching the arguments of a term to the arguments of a lemma (called match-args in our version and one-way-unify-1st in the original). The task that match-args performs is to determine whether or not it its first argument is a substitution instance of its second argument via a substitution (a map from variables to terms) that extends its third argument. In other words whether there is a substitution sublist such that (term-eq (mapcar #'(lambda (x) (apply-subst sublist x)) args2) args1) is t and sublist extends its third argument. So match-args must return two pieces of information. Firstly whether or not such a match is possible, and secondly, when a match is possible, the substitution that achieves this match. We use the Common Lisp multiple values feature to accomplish this.

```
(defun match-args (args1 args2 sublist)
  (labels ((match (term1 term2 sublist)
             (cond ((atomic-term-p term2)
                    (let ((bind (assoc term2 sublist :test #'atomic-term-eq)))
                      (if bind
                          (if (term-eq term1 (cdr bind))
                              (values t sublist)
                              (values nil nil))
                          (values t (cons (cons term2 term1) sublist)))))
                   ((composite-term-p term2)
                    (cond ((atomic-term-p term1) (values nil nil))
                              ((composite-term-p term1)
                               (if (op-eq (composite-term-op term1)
                                          (composite-term-op term2))
                                   (match-args (composite-term-args term1)
                                               (composite-term-args term2)
                                               sublist)
                                   (values nil nil)))
                              (t (error "~%match did not understand the term:
                                        ~a" term1))))
                       (t (error "~%match did not understand the term: ~a" term2)))))
  (if (null args1)
     (values t sublist)
     (multiple-value-bind (success? sublist)
        (match (car args1) (car args2) sublist)
        (if success?
           (match-args (cdr args1) (cdr args2) sublist)
           (values nil nil))))))
```

The dumb tautology checker is the program **tautp**. It is dumb simply because it only works for terms correctly for terms which are in if-normal form. Where the set of if-normal forms are defined to be the smallest set of terms containing the atomic propositions and closed under the formation rule: if then and else are if-normal forms and test is an atomic proposition then (make-if test then else) is an if-normal form. In the simplest case atomic propositions are (boolean) variables. In practice atomic propositions also include terms whose operation symbol is treated as an uninterpreted predicate symbol for the purposes of tautology testing. tautologyp takes two additional arguments true-list and false-list. true-list (false-list) is a list of atomic propositions assumed true (false). The invariant assumption is that the propositional term currently being considered is in the true branch of conditionals with tests in true-list and the false branch of conditionals with tests in false-list. (See [1] Chapter 4.) Except for the use of abstract syntax, our version of the tautology checker is the same as the original.

```
(defun tautp (term) (tautologyp (rewrite term) nil nil))

(defun tautologyp (term true-lst false-lst)
  (cond ((truep term true-lst) t)
        ((falsep term false-lst) nil)
        ((atomic-term-p term) nil)
        ((composite-term-p term)
         (when (ifp term)
               (cond ((truep (if-test term) true-lst)
                      (tautologyp (if-then term) true-lst false-lst))
                     ((falsep (if-test term) false-lst)
                      (tautologyp (if-else term) true-lst false-lst))
                     (t (and (tautologyp (if-then term)
                                         (cons (if-test term) true-lst)
                                         false-lst)
                             (tautologyp (if-else term)
                                         true-lst
                                         (cons (if-test term) false-lst)))))))
        (t (error "~%tautologyp did not understand the term: ~a" term))))
```

```
(defun truep (term true-lst)
  (cond ((atomic-term-p term) (member term true-lst :test #'term-eq))
        ((composite-term-p term)
         (or (op-eq (composite-term-op term) (the-op (quote t)))
             (member term true-lst :test #'term-eq)))
        (t (error "~%truep did not understand the term: ~a" term))))

(defun falsep (term false-lst)
  (cond ((atomic-term-p term) (member term false-lst :test #'term-eq))
        ((composite-term-p term)
         (or (op-eq (composite-term-op term) (the-op (quote f)))
             (member term false-lst :test #'term-eq)))
        (t (error "~%falsep did not understand the term: ~a" term))))
```

The remaining code is for initializing the system and carrying out a standard test. The setup procedure carries out the task of partitioning the list of lemmas and annotating each operator with the appropriate sublist. The constant REALLY-BIG-LIST-OF-LEMMAS is defined in [4] pp.118–126. The test term is constructed by applying a substitution to boolean expression.

```
(defun add-lemma (sexp)
  (let* ((lemma (sexp-2-lemma sexp))
         (op (lemma-op lemma)))
    (setf (op-lemmas op) (cons lemma (op-lemmas op)))))
(defun sbind-2-tbind (sbind)
  (cons (sexp-2-term (car sbind)) (sexp-2-term (cdr sbind))))
(defun setup ()   (mapc #'add-lemma REALLY-BIG-LIST-OF-LEMMAS))
(defun test () (tautp (test-term)))
(defun test-term ()
 (apply-subst
  (mapcar #'sbind-2-tbind
          (quote ((x foo (plus (plus a b)
                               (plus c (zero))))
                  (y foo (times (times a b)
                                (plus c d)))
                  (z foo (reverse (append (append a b)
                                          (nil))))
                  (u equal (plus a b)
                     (difference x y))
                  (w lessp (remainder a b)
                     (member a (length b))))))
   (sexp-2-term (quote (implies (and (implies x y)
                                     (and (implies y z)
                                          (and (implies z u)
                                               (implies u w))))
                                (implies x w))))))
```

4.0.2 Parallelizing the Program

In the benchmark test, the rewritten term is large (order of fifty thousand nodes) and hence most of the work in evaluating (tautp term) is spent in rewriting the term and its subterms. It seems an ideal place to start parallelizing. Fortunately this is a very simple task. There are two obvious places where there is inherent parallelism in the rewrite program and its auxiliary programs. These points are where the functional **mapcar** is used, in rewrite and apply-subst. Here we can replace **mapcar** by one of several parallel versions qmapcar! , qmapcarl, **qmapcar2**, qmapcar3 and **qmapcar4**.

These parallel versions of **mapcar** are most appropriate for short lists such as argument lists for terms. They are all constructed from the same template, differing only in the dynamic parallelism macro used. For example

```
(defun qmapcar! (f l)
  (if l #!(cons (funcall f (car l)) (qmapcar f (cdr l)))) nil)
(defun qmapcar<n> (f l)
  (if l #<n>(cons (funcall f (car l)) (qmapcar f (cdr l)))) nil)
```

Another possible point for introducing parallelism is where tautologyp calls itself recursively on both branches of an if-term. Here we can prefix the conjunction with one of the dynamic parallelism macros.

To summarize we have three sites for the addition of parallelism which we will call (A), (R), and (T). (A) consists in replacing the call to **mapcar** in apply-subst by one of the parallel versions. (R) consists in replacing the call to **mapcar** in rewrite by one of the parallel versions. (T) consists in prefixing the and expression in tautologyp by one of the dynamic parallelism macros. The first tests simply used unconstrained parallelism to see what the potential parallelism is at each site, as always we are running on a machine with eight processors.

New-Bover (sequential 18.538)						
R	T	A	para-time	processes	overhead	idle-time
-	-	-	20.210	1	0.0	137.9889
#!	#!	#!	3.683	98626	2.5769	0.9364
-	-	#!	19.896	8550	0.6791	135.7723
#!	-	#!	3.701	98612	2.5811	2.9396
#!	-	-	3.655	90062	2.3456	1.7393
#!	#!	-	3.444	90076	2.3513	0.9313

From these results we see that putting parallelism only in apply-subst has almost no effect, while putting parallelism only in rewrite gives substantial **speedup**. This is due to the fact that most of work in rewriting is finding a match. There are many more failures than successes and hence many more matching tasks than applications of substitutions. Adding tautologyp parallelism to rewrite parallelism decreases the idle time by a factor of two and produces a small increase in parallelism. On the other-hand adding apply-subst parallelism to either (R) or (R,T) increases the overhead and idle time and decrease the **speedup**. This is presumably due to the fact that the tasks created by apply-subst parallelism are small. The next experiments are testing the effects of various fine tunings in the parallelism.

New-Boyer (sequential 18.538)						
R	T	A	para-time	processes	overhead	idle-time
#?	#?	-	13.080	54788	4.4300	75.6937
#3?	#3?	-	3.363	14822	0.4833	1.0278
#?	#!	-	12.913	54783	4.3974	74.2607
#2?	#!	-	3.448	10599	0.4182	1.1196
#3?	#!	-	3.291	12075	0.4217	1.0587
#4?	#!	-	3.344	15649	0.5022	1.0511
#?	-	-	13.706	54474	4.3859	78.9108
#2?	-	-	3.533	9492	0.3784	1.8355
#3?	-	-	3.438	11569	0.3973	1.7265
#4?	-	-	3.430	14766	0.4842	1.6686
#?	-	#?	13.562	60073	4.9065	78.4283
#2?	-	#2?	3.545	15204	0.6166	1.8917
#3?	-	#3?	3.450	14770	0.5151	1.6650
#4?	-	#4?	3.454	17678	0.5671	1.5985
Best Speed-up (/ 18.538 3.291) = 5.633						

4.0.3 Optimizing

As already remarked, the original Boyer. benchmark program used lists and list operations to represent composite terms and lemmas. Atomic terms and operations were represented as Lisp symbols and lemmas relevant to a given operation were stored on the property list of the corresponding symbol. We represented atomic terms, composite terms, operations and lemmas as distinct structures and stored lemmas in the operation structure. The original code for rewrite was different only in its failure to use either abstract syntax or the functional `mapcar`. These are also the only differences between the original apply-subst and our version. match-args must return two pieces of information. We use the Common Lisp multiple values feature to express this. The original version returned t or nil depending on success, and if successful set the value of a global variable to be the resulting substitution.

Although use of structure definitions and other high-level constructs results in elegant and easy to understand code, these constructs are part of a rather complex machinery and may well not produce the most efficient implementation of the underlying algorithm. The original program can (essentially) be recovered by carrying out the simple set of transformations described below.

- Step zero consists in replacing structures by lists. Common Lisp provides a mechanism that allows the programmer to maintain the elegant look of the code with more efficient implementation by using the `:type` option for structure definitions. By simply changing the **`defstructs`** as shown below we force a representation close to that of the usual list structure representation.

```
(defstruct (atomic-term   (:type list) :named) name)
(defstruct (composite-term (:type list) :named) op args)
(defstruct (lemma (:type list) :named) op args rhs)
(defstruct (op (:type list) :named) name lemmas)
```

- Step one is to eliminate the atomic-term structure definition. This is achieved by omitting the atomic-term defstruct and making the following definitions.

```
(defun make-atomic-term (x) x)
(defun atomic-term-name (x) x)
(defun atomic-term-p (x) (atom x))
(defun atomic-term-eq (x y) (eq x y))
```

- Step two consists in making the composite-term structure unnamed. This is achieved by replacing the composite-term defstruct by

```
(defstruct (composite-term  (:type list)) op args)
```

and redefining the (no longer defined) test function composite-term-p.

```
(defun composite-term-p (x) (consp x))
(defun term-eq (x y) (equal x y))
```

- Step three involves making the lemma structure unnamed. This consists of replacing the original lemma defstruct by

```
(defstruct (lemma (:type list)) op args rhs)
```

- Step four consists in eliminating the operation structure. Here again we omit the op defstruct and adding the following definitions.

```
(defun make-op (x) x)
(defun op-name (x) x)
(defun op-lemmas (x) (get x (quote lemmas>>>
(defun op-p (x) (atom x))
(defun the-op (x) (atom x))
(defun op-eq (x y) (eq x y))
```

- Step five involves eliminating the use of multiple values. For this we replace occurrences of (values nil nil) by **'fail**, (values t **<exp>**) by **<exp>** and __

```
(multiple-value-bind (?success sublist) <bndexp> <body>)
```

by

```
   (let ((sublist <bndexp>))
      (let ((?success (if (eq sublist 'fail) nil t))) <body>))
```

- Step six consists in unfolding **mapcar** in apply-subst. For this we replace the call to **mapcar** by (apply-subst-list (composite-term-args term)) and add the definition

```
(defun apply-subst-lst (alist lst)
  (cond ((null lst)
         nil)
        (t (cons (apply-subst alist (car lst))
                 (apply-subst-lst alist (cdr lst))))))
```

- Step seven step consists in unfolding **mapcar** in rewrite. For this we replace the call to **mapcar** by (rewrite-args (composite-term-args term>> and add the definition

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

The program nlboyer is obtained by carrying out step zero of the transformations. The major differences between nlboyer and that of the original bench-mark are the following. Atomic terms are not atoms, operators are not atoms and lemmas are part of operator structure rather than being on the property list of the corresponding symbol, the use of multiple values rather than global variables for passing the substitution list, and the use of mapcar rather than auxiliary functions for applying a function to a list of items.

nlboyer is roughly fifty percent faster than new-boyer and one gets roughly ten percent better speed up. Clearly a win if speed is the goal. The data for nlboyer is summarized in the following table.

nlboyer Structures as Lists (sequential time 14.823)					
R	T	para-time	processes	overhead	idle-time
- #3?	#!	15.242 2.294	13590 0	0.4783 0.0	119.052 0.855
Best Speed-up (/ 14.823 2.294) = 6.462					

The program oboyer is obtained by carrying out all of the transformations. This differs from the original boyer benchmark code, assuming a suitably smart compiler, only in the elimination of global variables. Note that sexp-to-term and term-to-sexp are now identity functions – hopefully we don't need to redefine them explicitly. The timings oboyer are given in the following table.

Original Boyer (sequential time 13.153)					
R	T	para-time	processes	overhead	idle-time
- #!	-	13.355 2.262	90062 0	2.3472 0.0	93.1092 1.1649
#3?	-	1.997	13676	0.4773	1.2800
#2?	#!	1.954	10117	0.4207	0.6772
#3?	#!	1.976	11739	0.4101	0.9977
Best Speed-up (/ 13.153 1.954) = 6.731					

Chapter 5

The parallel iteration forms

In section 2.1 we presented parallel versions of the Common Lisp iteration forms:

- `dotimes`,
- `dolist`,
- `mapcar`, `mapc`, etc.

Now we will describe how the parallel forms are implemented. We would like qdotimes, qmapcar, etc. to satisfy the following goals.

- Any number of iterations should be handled as efficiently as possible. When the number of iterations is large, this means generating fewer processes than the number of iterations.
- The efficiency should depend as little as possible on the size of the computation that is done in each iteration. I.e., the parallel forms should be able to handle fine-grained iteration almost as well as coarse-grained iteration.
- The parallel forms should work well if they are called at the "top level" of a parallel program, or when they are used inside other code that is already parallel. In the latter case, it may not be necessary to create any processes, and doing so would be inefficient.

The last goal suggests using the dynamic-spawn-p control form, which will interact well with other code in the program that uses dynamic-spawn-p. However, there is a problem in doing this. Dynamic spawning works best when the processes are arranged in a fairly balanced tree. The straightforward way to create processes from an iteration, however, results in an unbalanced tree, and does not execute efficiently under dynamic-spawn-p, or under any other form of parallelism control that we know about.

Our solution to this is to restructure the computation as a balanced tree whenever possible. For `dotimes`, we can do this because the number of iterations is computed right at the beginning. Let n be the number of iterations of a `dotimes` form. If we divide the whole problem into two subproblems: (1) do iterations 0 to $\lfloor n/2 \rfloor$; and (2) do iterations $\lfloor n/2 \rfloor$ to $n - 1$; then these computations can be done in parallel, and they can in turn be subdivided. We repeat this process until we create processes that perform a single iteration.

Here is some code that illustrates the above-described method. We convert a qdotimes form such as:

```
(qdotimes (i n) (body i))
```

to:

```
(labels ((do-range (low high)
           (if (= low high)
               (body low)
               #?(progn
                   (do-range low (ash (+ low high) -1)))
                   (do-range (+ 1 (ash (+ low high) -1)) high))))
  (do-range 0 (1- n)))
```

The internal function do-range evaluates (body `i`) for the range of values between low and high (inclusive). If the range contains more than one value, it is split into two smaller ranges. The control form (spawnp) (called implicitly because of the **#?** syntax) determines whether this is done by means of parallel processes or ordinary function calls.

The above solution meets some of our criteria, but has significant overhead for small-grained iteration. The reason is that we have added a function call (to do-range) for each iteration. We would like to do an extra function call only when necessary, i.e., when a new process is actually created by the (spawnp) control expression.

In order to do this, we modify our strategy slightly. The code above takes a range [low, high], splits it in half, and then decides whether to do the two halves in parallel. Instead of this, we will first call (spawnp) to see if a new process is needed. If it is not, we will perform one iteration sequentially, and then call (spawnp) again. The range to be split is now [**low**+1,**high**]. As long as **(spawnp)** returns nil, we will avoid creating a process *and* we will avoid a function call for the current iteration. The only overhead is therefore the calls to (spawnp). Each of these takes less than a function call, if the default form (dynamic-spawn-p) is used, because it is a macro that expands into a very small number of machine instructions.

Using this new strategy, our code becomes:

```
(labels ((do-range (low high)
           (loop
             (when (and (spawnp) (< low high))
               #!(progn
                   (do-range low (ash (+ low high) -1))
                   (do-range (+ 1 (ash (+ low high) -1)) high))
               (return-from do-range nil))
             (when (> low high) (return nil))
             (body low)
             (incf low))))
  (do-range 0 (1- n)))
```

Notice that if (spawnp) always returns nil, then all that is executed is a loop that computes the **dotimes** body for each value in the desired range.

The method we have just outlined does not extend directly to the parallel iteration functions on lists: qdolist, qmapcar, etc. The reason is that we do not know the length of the list, so we cannot easily split the range of iterations in half. Also, even if we knew the length of the list to be n, it would take $O(n)$ time to reach the beginning of the second half, while in the code above it takes just constant time to start the two subprocesses. This extra time is spent in a sequential computation and thus adds idle time, if there is nothing else for our processors to do.

Given a list mapping operation whose iterations are independent computations, our goal is to execute this computation as efficiently as possible on a shared-memory multiprocessor. The following variables characterize our parallel machine. Those that represent time are all multiples of some basic time unit, whose exact value is not important.

p is the number of processors.

s is the amount of time needed to create ("spawn") a process.

d is the amount of time needed to evaluate the cdr function.

We use the following to describe a specific instance of the use of a mapping function.

n is the length of the list being mapped over.

c is the time needed to apply the function to each list element. (We assume c is constant .)

The above descriptions combine some of the primitive operations needed to evaluate a mapping function. All of the work done in stepping from each iteration to the next (testing for the end of the list, calling cdr, and whatever else is needed) is subsumed in the parameter d, and all of the work needed to create and schedule a process is contained in s.

Our potential speedup is limited by Amdahl's Law, which predicts a maximum speedup on any parallel program based on its inherently sequential component, In our case, the list data structure requires n cdr operations to be performed in order, since each cons cell contains the pointer to the next one. So the minimum time for any parallel mapping function is nd, the time needed to perform the n cdr operations.

A straightforward sequential version of the mapping function takes time $n(c + d)$, since we perform one function application on each element of the list, and step from each element to the next. Therefore Amdahl's Law limits the speedup to

$$\frac{T_{seq}}{\min T_{par}} = \frac{n(c \ t \ d)}{nd} = \frac{c+d}{d}.$$

Unless we change the list data structure to something else, there is no way to overcome this limitation.

```
(defun qmapa (fn list)
  (if (null list)
      nil
      #!(progn (funcall fn (car list))
               (qmapa fn (cdr list)))))
```

A simple way to parallelize the computation is shown in the function qmapa. The main loop of this function creates a new process for each iteration of the loop; this process will perform the c units of work required to apply the function to one element of the list. Even if enough processors are available to handle the processes that are created, the minimum time for qmapa is $n(s + d)$, and by the argument above, its maximum speedup is now $(c + d)/(s + d)$ instead of $(c + d)/d$. If the spawning time s is large, this is a significant loss.

The function qmapa has other problems. If there are not enough processors to handle all of the processes as they are created, then proper scheduling of the processes becomes important.

Also, the amount of memory needed to hold data structures describing the waiting processes can become a serious obstacle.

Our experience in Qlisp programming has shown that programs that work by top-down recursive splitting (such as the Quicksort algorithm for sorting) are easy to parallelize. Such computations can be viewed as a tree of processes, where the root represents the entire computation, and each process's children are subcomputations that may be executed in parallel. We have studied in some depth the particular case where each node in the tree has two children, the work performed at each node is roughly constant, and a "dynamic partioning" method is used to avoid creating many more processes than are necessary to keep the parallel machine busy [11].

Dynamic partitioning, in its simplest form, uses a separate queue of processes for each of the p processors. When the program allows a new process to be created, a processor does so only if its own queue is empty, as indicated by the function dynamic-spawn-p. Processes are inserted only into a processor's own queue. When it is idle, a processor first tries to take work from its own queue; if the queue is empty, it cycles among the other processors' queues, removing a process from the first non-empty one that it finds. If there are p processors and the computation tree has height h, this results in $O(p^2h^4)$ processes being created.

```
(defun qmapb (fn list)
  (labels
      ((map-loop (k list)
         (cond ((or (null list) (= k 0))
                nil)
               ((not (dynamic-spawn-p))
                (funcall fn (car list))
                (map-loop (1- k) (cdr list)))
               ((= k 1)
                (funcall fn (car list)))
               (T (let ((k2 (halve k)))
                    #!(progn (map-loop k2 list)
                             (map-loop (- k k2)
                                       (nthcdr k2 list))))))))
   (map-loop (length list) list))
  list)

(defun halve (k) (ash k -1))
(defun double (k) (ash k 1))
```

Function qmapb uses a modified divide-and-conquer method, dividing only when it spawns a process. Initially, qmapb computes the length of the list n. It is the job of the inner function map-loop to perform the actual calls to the function being mapped, as well as to check to see if it is reasonable to split the task into two equal sub-tasks. The answer to the latter question is provided by a call to dynamic-spawn-p. This predicate returns `T` if the local task queue (the current processor's queue of things to do) is empty, and `NIL` otherwise.

When the predicate causes a partition, the algorithm divides the list into two parts of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, creates subprocesses to perform the mapping function on these sublists, and combines the results of these processes. There is also a test prior to spawning, insuring non- trivial processes.

Dynamic partitioning applied to qmapb yields a significant reduction in the overhead due to process spawning, compared to qmapa, which spawned n processes. The height h of the computation tree is $O(\log n)$, so for a fixed number of processors p, the number of processes spawned is at most $O(\log^4 n)$, in the worst case, using the analytical result previously mentioned. In practice, the average number of spawns is $O(\log^2 n)$, but in either case, this function grows much more slowly than n.

However, there is still a problem-idle time. We divide idle time into three components.

- At the beginning of the computation, only one processor is busy. Other processors remain idle until enough processes have been created to make them busy.
- Once all of the processors become busy, the machine reaches a "steady state" where there is very little idle time. (This is true for the algorithms we are describing, but it is not true in general for all programs.)
- The steady state ends when the computation has passed the point when any new processes can be created, and all of the queues used by the dynamic scheduler are empty. Then, once a processor becomes idle it remains idle for the rest of the computation. This is because no new process can be created for it, and whenever another processor finishes a process, allowing its parent to resume, that processor is available to run the parent.

Of the three components of idle time in qmapb, the first is the most significant. To compute the length of the list requires n cdr operations, which takes time nd. All of this is done on one processor, while the others wait, since this cannot be parallelized. Additionally, the time until all p processors are busy is $O(nd \log p)$, due to the large number of calls to cdr near the beginning of the computation. Even if the rest of the computation is done in the fastest possible time, which we observed above to also be nd, the minimum time for the parallel algorithm is at least $n(2d + d\log p)$, and hence the potential speedup is less than half of the limit imposed by Amdahl's Law.

The idle time at the end is not as large. During the "steady state" period, all p processors remain busy. (Here we assume that p is not more than $(c + d)/d$, the speedup limit imposed by Amdahl's Law.) As long as some of the processes are performing the mapping operation on lists of length greater than 1, the steady state continues, since such processes can be partitioned whenever needed to provide work for a processor that has become idle. After the steady state period, therefore, all processors are either idle, are applying the function to lists of length 1, or are combining the results of subcomputations.

Only c time units (a constant number) can be spent in finishing the work on lists of length 1. The combination of subcomputations takes time proportional to the height of the computation tree, which is $O(\log n)$. Therefore the idle time at the end of the computation is $O(\log n)$. As n increases, this becomes insignificant compared to both the idle time at the beginning (which is at least nd) and the overall runtime (at least n/p).

The function qmapb eliminated one obstacle to achieving the optimal speedup given by Amdahl's Law, namely the overhead of process creation, but the excessive idle time at the beginning of the computation still stands in the way. We now describe an improved function qmapc that reduces this idle time.

Rather than precompute the length, n, of the list, we use a parameter k as an initial estimate, and divide the work into two tasks. The first task applies the function to the first k elements of the list, while the second task is a recursive call with the length estimate k doubled. The repeated doubling of k insures that the end of the list is reached after $\log n$ tasks have been spawned. This virtually eliminates the idle time at the beginning of the computation (assuming the initial value of *k* is small). However it does not insure that the machine reaches a steady state, in particular the last task spawned is as large as all the others combined. By using the dynamic partioning method within each of these logn tasks we can insure that a steady state is reached, and maintained as long as possible. Each of these tasks divides into equal sized subtasks whenever the dynamic partitioning predicate is true. While the predicate is false each task simply performs the desired mapping operations.

```
(defun qmapc (fn list)
  (macrolet
      ((*map-apply* (fn list) '(funcall ,fn (car ,list))))
    (labels
        ((map-loop (k list)
           (cond ((or (null list) (= k 0))
                  nil)
                 ((not (dynamic-spawn-p))
                  (*map-apply* fn list)
                  (map-loop (1- k) (cdr list)))
                 ((= k 1)
                  (*map-apply* fn list))
                 (T (let ((k2 (halve k)))
                      #!(progn (map-loop k2 list)
                               (map-loop (- k k2)
                                         (nthcdr k2 list)))))))
         (map-rest (k list)
           (when list
             #!(progn (map-loop k list)
                      (map-rest (double k)
                                (nthcdr k list))))))
      (map-rest 1 list)))
  list)
```

We begin by describing the simpler non-value accumulating mapping functionals qmapc and qmapl, concentrating on the former for ease of exposition. The qmapc program has two local functions map-loop and map-rest. The function map-rest spawns the first logn tasks. Each of these tasks consists of a call to the second local function map-loop, which is identical to map-loop in qmapb. In this version of the program we take 1 to be our initial estimate of the length of the list to be processed.

The qmapc function is written using `macrolet` to capture the uniformities between this function and the related function qmapl. The definition of qmapl is obtained by modifying the macro *map-apply* so that it expands to (`funcall` fn list).

To extend this technique to the value returning mapping functionals, `mapcar`, `mapcan`, `mapcon` and `maplist`, we need to accumulate and pass along the values of the respective calls to the function. To do this efficiently we use cyclic lists in the following way. Rather than have map-loop return the list of accumulated values that would then have to be cdr-ed down to be attached to the remaining result. The program map-loop is written so as to return the last cell in this list, modified so that the cdr points to the first cell of the list. We shall call such a cyclic representation (or modification) of a list a cycle. The transformations from lists to cycles, `list-2-cycle`, and from cycles to lists, `cycle-2-list`, explicitly explains this representation.

```
(defun list-2-cycle (list)
  (when list
    (let ((cycle (last list))) (setf (cdr cycle) list) cycle)))

(defun cycle-2-list (cycle)
  (when cycle
    (let ((first-cell (cdr cycle))) (setf (cdr cycle) nil) first-cell)))
```

The functions map-loop and map-rest are modified so as to return cycles, which in the case of map-loop entails adding a new argument, cycle, representing the cycle up to the current point in the loop. This also entails that the cycles returned by spawned tasks must be remembered and linked together. This linking is performed by the function link-cycles. It takes two cycles as arguments and links them toget her to form a third cycle. The resulting cycle encodes the list obtained by nconc-ing the list encoded by the first cycle onto the list encoded by the second cycle. In other words a call to (link-cycles cycle-l cycle-2) is equivalent to a call to (list-2-cycle (nconc (cycle-2-list cycle-1) (cycle-2-list cycle-2))).

Similarly when map-loop applies the function to the appropriate argument it must splice the resulting list into the cycle accumulated so far, i.e. the value of cycle. This is accomplished by the program splice-cycle which takes a cycle, and a list and returns the same cycle that would result from a call to (link-cycles cycle-l (list-2-cycle list)).

```
(defun link-cycles (cycle-1 cycle-2)
  (cond ((and cycle-1 cycle-2)
         (let ((temp (cdr cycle-1)))
           (rplacd cycle-1 (cdr cycle-2))
           (rplacd cycle-2 temp)
           cycle-2))
         (cycle-1 cycle-1)
         (cycle-2 cycle-2)))

(defun splice-cycle (cycle list)
  (cond ((and cycle list)
         (let ((new-cycle (last list))
               (temp (cdr cycle)))
           (rplacd cycle list)
           (rplacd new-cycle temp)
           new-cycle))
        (cycle cycle)
        (list (let ((new-cycle (last list)))
                (rplacd new-cycle list)
                new-cycle))))

(defun qmapcar (fn list)
  (macrolet
      ((*map-apply* (fn list cycle)
         '(splice-cycle ,cycle
           (cons (funcall ,fn (car ,list)) nil))))
    (labels
        ((map-loop (k list cycle)
           (cond ((or (null list) (= k 0)) cycle)
                 ((not (dynamic-spawn-p))
                  (map-loop (1- k)
                      (cdr list)
                      (*map-apply* fn list cycle)))
           ((= k 1) (*map-apply* fn list cycle))
           (T
            (let ((k2 (halve k)))
              (multiple-value-bind (second third)
                  (qvalues (map-loop k2 list nil)
                            (map-loop (- k k2)
                                      (nthcdr k2 list)
                                      nil))
                (link-cycles cycle
                             (link-cycles second third)))))))
   (map-rest (k list)
     (when list
       (multiple-value-bind (first second)
           (qvalues (map-loop k list nil)
                    (map-rest (double k) (nthcdr k list)))
         (link-cycles first second)))))
(cycle-2-list (map-rest 1 list)))))
```

These modifications result in the function **qmapcar**. The parallelism is expressed by using the Qlisp form **qvalues**, which creates processes for each of its argument forms, waits for them to finish, and returns their values. Again the actual program is written using **macrolet** so as to capture the uniformities between this program, **mapcar**, and its sister programs **mapcan**, **maplist** and **mapcon** whose definitions are obtained by modifying the macro *map-apply* suitably. In particular for **mapcan** the macro definition expands to (splice-cycle cycle (**funcall fn** (car **list**))), for **maplist** it expands to (splice-cycle cycle (cons (**funcall** fn list))), and for **mapcan** it expands to (splice-cycle cycle (**funcall** fn list)).

5.1 Analysis of qmapc

The function qmapc outperforms qmapb in several respects. Here we will show that the idle time at the beginning of the computation, which was the main source of overhead in qmapb, becomes negligible as n increases.

The key idea is to show that enough work to keep p processors busy is found in $O(p \log p)$ time, instead of the O(n logp) that we needed for qmapb. If the lowest-level processes are large enough, the first p iterations of the function provide this work, and our method of doubling the process size at the beginning of the computation ensures that these processes are created in $O(p \log p)$ time.

It may happen that some of the initial processes finish before the steady state is reached, and in that case the initial idle time is longer. Eventually, though, the doubling of the segment size produces a process large enough so that all p processors remain busy while the beginning of the next segment is found. The size of this segment is some constant multiple of p, so the time needed to reach it is $O(p)$, and the time to partition it into p processes is $O(p \log p)$. This is the initial idle time of the computation.

For small values of n, the input list may be exhausted before the situation described above holds. The result is therefore true asymptotically as n increases. In the next section. oui experimental results show how large *n* needs to be as a function of the work performed in each iteration of the mapping function.

5.2 Experimental results

Each experiment consisted of mapping the function

```
(defun work (m) (if (<= m 0) 0 (work (1- m))))
```

over a list containing *n* copies of a number *m*. Thus *m* represents the granularity and *n* the problem size for a more general list mapping operation.

The function work runs in roughly *6m* microseconds on input *m*. We examined the behavior of qmapa, qmapb and qmapc on a variety of lists of various lengths, L, ranging from 10 to 100000, the problem size, N, ranging from (work 0), which takes a few microseconds per element, up to (work 160), which takes nearly 1 millisecond per element.

(QMAP WORK 0)									
L	X	pure serial	serial time	parallel time	number processes	overhead time	%	idle time	%
10A	0	0.000	0.000	0.001	11	0.000	8.1	0.004	47.3
10B	0	0.000	0.000	0.001	8	0.000	10.0	0.004	50.4
10C	0	0.000	0.000	0.001	11	0.001	13.5	0.004	56.5
100A	0	0.001	0.004	0.005	101	0.007	18.3	0.028	70.9
100B	0	0.001	0.002	0.001	32	0.001	14.3	0.006	54.1
100C	0	0.001	0.002	0.001	45	0.003	26.4	0.005	40.1
1000A	0	0.016	0.046	0.048	1001	0.073	18.9	0.271	70.4
1000B	0	0.016	0.020	0.008	110	0.006	9.7	0.028	45.2
1000C	0	0.016	0.017	0.005	154	0.010	23.3	0.007	16.5
10000A	0	0.161	0.484	0.651	10001	0.728	14.0	3.867	74.2
10000B	0	0.161	0.202	0.061	318	0.014	2.9	0.230	47.6
10000C	0	0.161	0.176	0.031	433	0.021	8.5	0.009	3.4
100000B	0	1.619	2.045	0.586	609	0.025	0.5	2.278	48.6
100000C	0	1.619	1.758	0.275	991	0.042	1.9	0.010	0.5

(QMAP WORK 10)									
L	N	pure serial	serial time	parallel time	number processes	overhead time	%	idle time	%
10A	10	0.000	0.001	0.001	11	0.000	8.9	0.004	52.9
10B	10	0.000	0.001	0.001	8	0.000	9.6	0.004	47.7
10C	10	0.000	0.001	0.001	10	0.001	13.3	0.004	59.0
100A	10	0.007	0.010	0.005	101	0.007	18.2	0.023	56.2
100B	10	0.007	0.008	0.002	33	0.001	9.9	0.006	36.0
100C	10	0.007	0.008	0.002	48	0.004	23.8	0.005	31.6
1000A	10	0.076	0.107	0.047	1001	0.073	19.5	0.205	54.9
1000B	10	0.076	0.080	0.015	125	0.007	5.7	0.029	23.6
1000C	10	0.076	0.077	0.013	169	0.010	10.4	0.006	6.2
10000A	10	0.758	1.117	0.547	10001	0.728	16.6	3.014	68.9
10000B	10	0.758	0.798	0.134	291	0.014	1.3	0.234	21.8
10000C	10	0.758	0.772	0.106	462	0.023	2.7	0.009	1.0
100000B	10	7.569	7.998	1.323	551	0.023	0.2	2.274	21.5
100000C	10	7.569	7.710	1.011	890	0.038	0.5	0.010	0.1

(QMAP WORK 40)									
L	N	pure serial	serial time	parallel time	number processes	overhead time	%	idle time	%
10A	40	0.002	0.003	0.001	11	0.000	9.2	0.005	61.1
10B	40	0.002	0.003	0.001	8	0.000	8.7	0.005	64.8
10C	40	0.002	0.003	0.001	10	0.001	12.3	0.005	60.4
100A	40	0.025	0.028	0.007	101	0.007	12.3	0.022	38.5
100B	40	0.025	0.026	0.005	34	0.002	5.4	0.007	19.0
100C	40	0.025	0.026	0.005	49	0.004	10.1	0.006	15.1
1000A	40	0.254	0.284	0.066	1001	0.069	13.0	0.189	35.7
1000B	40	0.254	0.259	0.038	129	0.007	2.4	0.029	9.5
1000C	40	0.254	0.256	0.035	181	0.011	4.0	0.007	2.5
10000A	40	2.544	2.901	0.709	10001	0.700	12.3	2.236	39.4
10000B	40	2.544	2.585	0.358	332	0.016	0.6	0.232	8.1
10000C	40	2.544	2.558	0.329	472	0.024	0.9	0.010	0.4
100000B	40	25.446	25.856	3.552	561	0.024	0.1	2.276	8.0
100000C	40	25.446	25.572	3.238	870	0.038	0.1	0.011	0.0

(QMAP WORK 160)									
		pure	serial	parallel	number	overhead		idle	
L	N	serial	time	time	processes	time	%	time	%
10A	160	0.009	0.010	0.002	11	0.000	3.9	0.009	47.8
10B	160	0.009	0.010	0.002	8	0.000	3.6	0.010	53.9
10C	160	0.009	0.010	0.002	10	0.001	4.7	0.009	46.7
100A	160	0.097	0.100	0.016	101	0.007	5.3	0.025	19.8
100B	160	0.097	0.097	0.014	35	0.002	2.0	0.012	10.2
100C	160	0.097	0.097	0.005	51	0.004	3.4	0.010	8.4
1000A	160	0.968	0.999	0.155	1001	0.067	5.4	0.189	15.3
1000B	160	0.968	0.973	0.128	136	0.008	0.7	0.033	3.2
1000C	160	0.968	0.970	0.125	182	0.012	1.2	0.011	1.1
10000A	160	9.684	10.015	1.639	10001	0.678	5.2	2.078	15.9
10000B	160	9.684	9.726	1.250	315	0.015	0.2	0.236	2.4
10000C	160	9.684	9.701	1.224	435	0.022	0.2	0.014	0.1
100000B	160	25.446	25.856	3.552	561	0.024	0.1	2.276	8.0
100000C	160	25.446	25.572	3.238	870	0.038	0.1	0.011	0.0

Chapter 6

Backtracking search

The N-Queens problem entails counting the number of distinct placements of N queens on a chessboard such that no queen attacks any other queen. Recall that queens attack all squares in the same row, column, and the two diagonals. The most common algorithms for solving the problem use backtracking; queens are placed in feasible locations until either a solution is obtained or no more queens can be placed. In either case, the program "backs up" to the last queen placement and tries the next possibility. There are many possible heuristics which can reduce the search space, but we simply use the straightforward column numbering scheme, and build the solutions from left to right.

The following function serial-column solves the serial N-Queens problem. serial-column tries to place a queen in each row of the specified column. It succeeds in placing the queen when the specified row, and both diagonals are free of attack. It then recursively tries to place a queen in the next column, until reaching the last column. The function returns the total number of solutions that were found.

```
(defun serial-column (column row-state left-diag right-diag N)
  (let ((count-solutions 0))
    (dotimes (row N)
      (when (free-p row row-state)
        (when (free-p (left-diagonal column row)  left-diag)
          (when (free-p (right-diagonal column row n) right-diag)
            (if (= column (- n 1))
                (incf count-solutions)
                (incf count-solutions
                      (serial-column (next-column column) (add-attack row row-state)
                       (add-attack (left-diagonal column row) left-diag)
                       (add-attack (right-diagonal column row n) right-diag)
                       n)))))))
    count-solutions))
```

These macros make the code more legible. free-p uses an index to test a bit in a bit-vector. add-attack turns on a bit in a bit-vector. left and right-diagonal compute the appropriate diagonal index from the row, column, and N. To find the number of placements on an empty NxN board, we use (serial-column 0 0 0 0 N).

```
(defmacro free-p (index state) '(not (logbitp ,index ,state)))
(defmacro add-attack (index state) '(+ ,state (ash 1 ,index)))
```

```
(defmacro left-diagonal (column row) `(+ ,column ,row))
(defmacro right-diagonal (column row n) `(- (+ ,column ,n -1) ,row))
(defmacro next-column (column) `(+ 1 ,column))
```

This backtracking algorithm has an abundance of potential parallelism. The individual iterations of the dotimes are almost completely independent; if they were completely independent, we could parallelize the program by changing the dotimes to qdotimes. the only difficulty is that the count-solutions counter could be updated in any single iteration; since Qlisp does not assume atomic read-modify-write operations, modifying the count-solutions variable is a critical region, in a parallel version of the program.

6.0.1 Two Solutions

We will present a synchronous solution, using locks, and an asynchronous solution, using a distributed variable. Both solutions use qdotimes to express the parallelism.

Synchronous Solution

To implement the synchronous solution, we use a lock structure and the with-lock macro. Each incf of the count-solutions variable is wrapped in a with-lock, to protect it as a critical region. Note that the (incf count-solutions) form is critical, because all of the processes share the variable. We use the temp variable to avoid locking the recursive call to synch-column.

```
(defun synch-column (column row-state left-diag right-diag N)
  (let ((count-solutions 0)
        (lock (make-lock :type :spin)))
    (qdotimes (row N)
      (when (free-p row row-state)
        (when (free-p (left-diagonal column row)  left-diag)
          (when (free-p (right-diagonal column row n) right-diag)
            (if (= column (- n 1))
                (with-lock lock (incf count-solutions))
                (let ((temp
                       (synch-column (next-column column) (add-attack row row-state)
                         (add-attack (left-diagonal column row) left-diag)
                         (add-attack (right-diagonal column row n) right-diag)
                          n)))
                (with-lock lock (incf count-solutions temp))))))))
    count-solutions))
```

There are some disadvantages to this synchronous solution,' however. When the number of solutions (and number of processors) is large, then the lock may be a significant bottleneck, even though the increment operation is itself quite tritial. The make-lock operation is also non-trivial. Although this code could be modified to pass a lock, it would be better to avoid using locks at all, via the asynchronous method.

An Asynchronous Solution

The asynchronous solution uses a distributed counter. During the computation, each processor updates its "own" number of solutions independently of a.11 other processors. At the end, the total number of solutions is the sum of all processors number of solutions. It has a small disadvantage, in that the pre-processing and post-processing require time proportional to the number of processors. However, since the backtracking algorithm is exponential, this overhead is not significant.

Solve first creates a vector with length *number-of -processors*. Then it calls asynch-column, passing the vector as an argument. When asynch-column finds a solution, it increments the appropriate vector element as indexed be (get-processor-number), instead of incrementing a local variable. This asynchronous version works primarily by side-effectine the vector, and does not return any useful value.

```
(defun solve (n)
  (let ((distributed-counter (make-array *number-of-processors*
                                         :initial-element 0)))
    (asynch-column 0 0 0 0 n distributed-counter)
    (let ((solutions 0))
      (dotimes (i *number-of-processors*)
        (incf solutions (svref distributed-counter i)))
      solutions)))

(defun asynch-column (column row-state left-diag right-diag N
              distributed-counter)
  (qdotimes (row N)
    (when (free-p row row-state)
      (when (free-p (left-diagonal column row)  left-diag)
          (when (free-p (right-diagonal column row n) right-diag)
            (if (= column (- n 1))
                (incf (svref distributed-counter (get-processor-number)))
                (asynch-column (next-column column) (add-attack row row-state)
                  (add-attack (left-diagonal column row) left-diag)
                  (add-attack (right-diagonal column row n) right-diag)
                  n distributed-counter))))))
```

The only disadvantages of this met hod are the overhead of qdot imes, and the passing of an extra argument, the distributed-counter. In the current implementation, qdotimes expands into a labels form, which causes a closure to be created upon each qdotimes entry. Once entered, qdotimes is quite inexpensive and highly parallel, but the initial entry causes a closure to be consed up. In the case of N-Queens, N is usually small enough for this particular source of overhead to matter. (As a technical aside, if downward funargs could be declared and compiled effectively in common lisp, qdotimes would benefit.) This leads us to the depth cutoff solution; generally, we like to avoid cutoff types of solutions, but in this case the closure creation overhead is just too high, and it is fairly easy to do a depth cutoff.

6.1 The Solution

The most effective solution combines the asynchronous parallel code for the first few columns and the serial version for the rest of the board. This method is typical of depth-cutoff approaches.

Solve calls depth-column with a depth of 2, which means the first 2 columns will be done in parallel, and the rest serially.

```
(defun solve-depth (n)
  (let ((distributed-counter (make-array *number-of-processors*
                                         :initial-element 0)))
    (depth-column 2 0 0 0 0 n distributed-counter)
    (let ((solutions 0))
      (dotimes (i *number-of-processors*)
        (incf solutions (svref distributed-counter i)))
      solutions)))

(defun depth-column (d column row-state left-diag right-diag N
               distributed-counter)
  (if (> column d)
      (incf (svref distributed-counter (get-processor-number))
            (serial-column column row-state left-diag right-diag N))
  (qdotimes (row N)
    (when (free-p row row-state)
      (when (free-p (left-diagonal column row)  left-diag)
        (when (free-p (right-diagonal column row n) right-diag)
          (depth-column d (next-column column) (add-attack row row-state)
            (add-attack (left-diagonal column row) left-diag)
            (add-attack (right-diagonal column row n) right-diag)
            n distributed-counter)))))))
```

6.1.1 Results

The results show that we have succeeded in parallelizing a fast N-Queens program. In all three versions of the program, there is very little idle time or scheduling overhead as N increases. There is, however, a noticeable difference in the amount of serial work done in each of the three versions. In the final version, there is virtually no time difference between the pure serial code, and the serial final version; when coupled with virtually zero idle overhead and zero scheduling overhead, the implication is that the final version gets nearly perfect speed-up.

N-Queens Synchronous Solution								
	pure	serial	parallel	number	overhead		idle	
N	serial	time	time	processes	time	%	time	%
5	0.013	0.019	0.006	96	0.008	15.4	0.018	37.0
6	0.043	0.063	0.013	224	0.012	11.6	0.018	16.7
7	0.159	0.226	0.040	403	0.018	5.8	0.027	8.5
8	0.656	0.914	0.139	644	0.027	2.4	0.032	2.9
9	2.904	3.975	0.554	1173	0.045	1.0	0.069	1.6
10	13.512	18.117	2.498	2060	0.069	0.3	0.045	0.2

N-Queens Asynchronous Solution

N	pure serial	serial time	parallel time	number processes	overhead time	%	idle time	%
5	0.013	0.020	0.007	95	0.007	12.6	0.019	32.2
6	0.043	0.071	0.015	238	0.014	11.6	0.024	20.2
7	0.159	0.265	0.043	384	0.018	5.1	0.021	6.0
8	0.656	1.093	0.159	781	0.033	2.6	0.047	3.7
9	2.904	4.787	0.687	1244	0.049	0.9	0.062	1.1
10	13.512	22.100	3.036	1217	0.045	0.2	0.053	0.2

N-Queens Depth Cutoff Solution

N	pure serial	serial time	parallel time	number processes	overhead time	%	idle time	%
5	0.013	0.014	0.006	51	0.004	9.1	0.019	44.2
6	0.043	0.045	0.009	80	0.005	7.1	0.018	24.6
7	0.159	0.162	0.025	103	0.006	2.8	0.023	11.5
8	0.656	0.662	0.092	131	0.007	1.0	0.041	5.6
9	2.904	2.910	0.385	173	0.009	0.3	0.074	2.4
10	13.512	13.510	1.763	224	0.012	0.1	0.239	1.7

Chapter 7

Polynomial GCD

In this section we develop a parallel version of an algorithm for computing the greatest common divisor (gcd) of two polynomials. It is simple to compute polynomial gcds using the euclidian algorithm. The problem is that in the case of polynomials, intermediate coefficients often become quite large. A standard solution to this problem is to compute gcds modulo a suitable set of primes and combine the results using an algorithm based on the Chinese remainder theorem. Our algorithm is a variant on a standard solution that combines the results in a tree-like fashion rather than sequentially. For details on the underlying mathematics the reader is referred to Collins [3], or Lauer [6].

In this example we restrict our attention to polynomials in one variable over the integers. The main phenomena observed here are the effects of very coarse grained parallelism and a trade-off between opportunities for parallelism and amount of work required. The main source of parallelism is the independent gcd computation for each prime. In the case of 8 processors, if there are 7 primes, we get very good speed up. While if there 9 primes, then 8 primes get processed in parallel, leaving the last one to be shared (not very efficiently) by the 8 processors. There is finer grain parallelism within the processing of each of the primes, but it is fairly flat.

7.1 The Program

A polynomial in single variable x over the integers can be thought of as a formal sum $P = \sum_{0\leq i\leq d} c_i x^i$ where each c_i is an integer and $c_d \neq 0$. d is the degree of P and c_d is the leading coefficient. The poly-gcd program uses a simple representation of univariate polynomials as lists of monomials, where a monomial consists of an coefficient (an integer in our case) and an exponent (a natural number). The list with monomials (c_k, e_k) for $k < n$ represents the polynomial whose degree d is the largest e_k such that c_k is non-zero. The degree is 0 if there are no non-zero coefficients. The coefficient of x^i is the sum of the c_k such that $e_k = i$. For example, the empty list represents the 0 polynomial and the list

```
(#mono(:exp 2 :coef 1) #mono(:exp 1 :coef -2) #mono(:exp 0 :coef 1))
```

represents the polynomial commonly written $x^2 - 2x + 1$. For efficiency, we require that the exponents of a monomial list form a strictly decreasing sequence and that first element of a non-empty monomial list has non-zero coefficent. In particular, the zero polynomial is uniquely represented by the empty list, and an allowed non-empty list represents a non-zero polynomial whose degree is the exponent of the first monomial and whose leading coefficient is the coefficient of the first monomial. The basic polynomial structure is implemented by the following code. The operation `poly-extend` adds a monomial to a polynomial. It is used only in the case that exp is greater than degree of poly .

```
(defstruct (mono (:print-function mono-print)) exp coef)

(defun mono-print (mono stream pl)
  (pprint (list :c (mono-coef mono) :e (mono-exp mono)) stream))

(defun pair2mono (pair) (make-mono :exp (car pair) :coef (cdr pair)))

(defun deg (poly) (mono-exp (car poly)))

(defun lead-coef (poly) (mono-coef (car poly)))

(defun poly-extend (coef exp poly)
  (if (zerop coef) poly (cons (make-mono :exp exp :coef coef) poly))))
```

We need some arithmetic operations on polynomials. Multiplication of a polynomial by a monomial is defined by

$$cx^e * \sum_{0\leq i\leq d} c_i x^i = \sum_{e\leq i\leq d+e} (c * c_i)x^{i+e}$$

`(mono*poly coef` exp `poly)` multiplies `poly` by the monomial with coefficient `coef` and exponent exp.

```
(defun mono*poly (coef exp poly)
  (mapcar
   #'(lambda (mono)
       (make-mono :exp (+ (mono-exp mono) exp)
                  :coef (* (mono-coef mono) coef)))
   poly))
```

Division of a polynomial by a scalar is defined by

$$\sum_{0\leq i\leq d} c_i x^i \div c = \sum_{0\leq i\leq d} (c_i \div c)x^i$$

This operation is only applied if c is non-zero and divides each c_i. (poly-div-scalar `poly coef`) divides poly by `coef`.

```
(defun poly-div-scalar (poly coef)
  (mapcar
   #'(lambda (mono)
       (make-mono :exp (mono-exp mono) :coef (/ (mono-coef mono) coef)))
   poly))
```

Addition of two polynomials is defined by

$$\sum_{0\leq i\leq d_0} c_i^0 x^i + \sum_{0\leq i\leq d_1} c_i^1 x^i = \sum_{0\leq i\leq max[d_0,d_1]} (c_i^0 + c_i^1)x^i$$

`poly-add` adds two polynomials.

```
(defun poly-add (poly1 poly2)
  (cond ((null poly1) poly2)
        ((null poly2) poly1)
        ((< (deg poly1) (deg poly2))
         (cons (car poly2) (poly-add poly1 (cdr poly2))))
        ((< (deg poly2) (deg poly1))
         (cons (car poly1) (poly-add (cdr poly1) poly2)))
        (t (let ((coef (+ (lead-coef poly1) (lead-coef poly2)))
                 (rest (poly-add (cdr poly1) (cdr poly2))))
             (poly-extend coef (deg poly1) rest)))))
```

Polynomial multiplication is defined by

$$\sum_{0\leq i\leq d_0} c_i^0 x^i \ * \sum_{0\leq i\leq d_1} c_i^1 x^i \ = \sum_{0\leq i\leq d_0+d_1} (\sum_{j,k|j\leq d_0,k\leq d_1\ ,j+k=i} c_j^0 + c_k^1) x^i.$$

poly-mul multiplies two polynomials.

```
(defun poly-mul (poly1 poly2)
  (if (null poly1)
      nil
      (poly-add (poly-mul (cdr poly1) poly2)
                (mono*poly (lead-coef poly1) (deg poly1) poly2))))
```

gcd-poly-coef computes the gcd of the coefficients of a polynomial.

```
(defun gcd-poly-coef (poly)
  (apply #'gcd (mapcar #'mono-coef poly)))
```

The polynomial gcd algorithm also requires modular arithmetic for polynomials. Operations modulo a prime are only carried out on polynomials whose leading coefficient is not divisible by that prime, thus preserving the requirement for allowable monomial lists.

(mod-inv n p) computes the inverse of n modulo p for n and p relatively prime.

```
(defun mod-inv (n p)
  (labels
      ((m-inv (j k)
         (if (= j 1) 1 (/ (- 1 (* k (m-inv (mod k j) j))) j))
           ))
    (mod (m-inv n p) p)))
```

(mod-scalar*poly coef poly prime) multiplies each coefficient of poly by the scalar coef modulo prime. This operation is only applied when coef is non-zero modulo prime.

```
(defun mod-scalar*poly  (coef poly prime)
  (mapcar
   #'(lambda (mono)
       (make-mono :coef (mod (* coef (mono-coef mono)) prime)
                  :exp (mono-exp mono)))
   poly))
```

(mod-mono*poly coef exp poly prime) multiplies poly by the monomial with coefficient coef and exponent exp, reducing the resulting coefficients modulo prime. This operation is only applied when coef is non-zero modulo prime.

```
(defun mod-mono*poly (coef exp poly prime)
  (mapcar
   #'(lambda (mono)
       (make-mono :coef (mod (* coef (mono-coef mono)) prime)
                  :exp (+ exp (mono-exp mono))))
   poly))
```

(mod-add polyl poly2 prime) adds the polynomials polyl and poly2, reducing the resulting coefficients modulo prime.

```
(defun mod-add (poly1 poly2 prime)
  (cond ((null poly1) poly2)
        ((null poly2) poly1)
        ((< (deg poly1) (deg poly2))
         (cons (car poly2) (mod-add poly1 (cdr poly2) prime)))
        ((< (deg poly2) (deg poly1))
         (cons (car poly1) (mod-add poly2 (cdr poly1) prime)))
        (t (let ((coef (mod (+ (lead-coef poly1) (lead-coef poly2)) prime))
                 (rest (mod-add (cdr poly1) (cdr poly2) prime)))
             (poly-extend coef (deg poly1) rest))) ))
```

(mod-rmd polyl poly2 prime) computes the remainder of polyl divided by poly2 modulo prime. If poly 1, poly2 represent polynomials P_1, P_2 with P_2 non-zero and prime is a prime π then (mod-rmd polyl poly2 prime) represents a polynomial P_0 of degree less than P_2 such that for some polynomial Q $P_1 = Q * P_2 + P_0$ modulo π.

```
(defun mod-rmd (poly1 poly2 prime)
  (cond ((null poly1) nil)
        ((< (deg poly1) (deg poly2)) poly1)
        (t (let ((d1 (deg poly1))
                 (c1 (lead-coef poly1)))
             (mod-rmd (mod-add (mod-scalar*poly (lead-coef poly2) poly1 prime)
                               (mod-mono*poly (- c1)
                                              (- d1 (deg poly2))
                                              poly2
                                              prime)
                               prime)
                      poly2
                      prime)))))
```

(mod-poly-gcd polyl poly2 prime) is a polynomial of greatest degree dividing polyl and poly2 modulo prime.

```
(defun mod-poly-gcd (poly1 poly2 prime)
  (let ((r (mod-rmd poly1 poly2 prime)))
    (if (null r) poly2 (mod-poly-gcd poly2 r prime))))
```

The basic algorithm for computing polynomial gcds involves computing local polynomial gcds, each one modulo some prime number, and then joining these intermediate results using the Chinese remainder theorem. More precisely, given a pair of polynomials: (i) choose a set of primes and form a binary tree whose leaves are these primes; (ii) transform the tree of primes into a tree whose leaves are pairs consisting of the corresponding prime and the gcd of the given polynomials modulo that prime; (iii) for each pair of leaves combine the leaves according to the Chinese algorithm to obtain a new leaf consisting of a product of primes and a polynomial; (iv) repeat (iii) until only one leaf is left. The polynomial part will be the desired gcd, provided that the product of the primes we use is sufficiently large and that none of the primes are *unlucky* (see Collins [3] for details).

The Chinese algorithm combines two local gcds to produce a new local gcd. A local gcd is represented as a pair consisting of a produce of primes and a polynomial (the gcd modulo that product of primes). The primes of the two pairs must be disjoint. Let $lp_j = (\pi_j , \sum_{i \leq d_j} c_i^j x^i)$ for j $\in \{1, 2\}$ and let $d = max[d_1, d_2]$ then

$$chinese(lp_1, lp_2) = (\pi_1 * \pi_2, \sum_{i \leq d} ch(\pi_1, \pi_2, c_i^1, c_i^2)x_i$$

where $ch(\pi_1, \pi_2, c_1, c_2) = c_2 + (\pi_2 * c_3)$ modulo $\pi_1 * \pi_2$ and c_3 = mod-inv($\pi_2, \pi_1) * (c_1 - c_2)$ modulo π_1.

```
(defun chinese (p1 p2)
  (let* ((poly1 (cdr p1))
         (poly2 (cdr p2))
         (pp1 (car p1))
         (pp2 (car p2))
         (pp1-times-pp2 (* pp1 pp2))
         (inv (mod-inv pp2 pp1)))
    (labels
     ((chin (poly1 poly2)
            (cond ((null poly1)
                   (mapcar #'(lambda (mono)
                               (make-mono :coef (chin-eq 0 (mono-coef mono))
                                          :exp (mono-exp mono)))
                           poly2))
                  ((null poly2)
                   (mapcar #'(lambda (mono)
                               (make-mono :coef (chin-eq (mono-coef mono> 0)
                                          :exp (mono-exp mono)))
                           poly1))
                  ((= (deg poly1) (deg poly2))
                   (let ((coef (chin-eq (lead-coef poly1) (lead-coef poly2)))
                         (rest (chin (cdr poly1) (cdr poly2))))
                     (poly-extend coef (deg poly1) rest)))
                  ((< (deg poly1) (deg poly2))
                   (let ((coef (chin-eq 0 (lead-coef poly2)))
                         (rest (chin poly1 (cdr poly2))))
                     (poly-extend coef (deg poly2) rest)))
                  ((< (deg poly2) (deg poly1))
                   (let ((coef (chin-eq (lead-coef poly1) 0))
                         (rest (chin (cdr poly1) poly2)))
                     (poly-extend coef (deg poly1) rest)))))
         (chin-eq (c1 c2)
           (mod (+ c2 (* pp2 (mod (* inv (- c1 c2)) pp1)))
                pp1-times-pp2)))
      (cons pp1-times-pp2 (chin poly1 poly2)))))
```

Now for the toplevel code. `list2tree` takes a list and returns a balanced binary tree whose fringe is the input list. find-primes constructs a tree of primes suitable for the given pair of polynomials. It uses a global list of candidate primes *primes*. Bad primes are those dividing the leading coefficient of either polynomial. They are detected by bad-prime and rejected. The optional argument prime-floor determines where in the list of candidates to begin accumulating primes, and the accumulation stops when the product of the accumulated primes exceeds the estimated maximum coefficient.

```
(defun list2tree (l)
  (labels
   ((l2t (l n)
         (if (= n 1)
             (car l)
             (let ((nleft (ash n -1))) ;; n ge 2 size of left partition
               (cons (l2t l nleft)
                     (l2t (nthcdr nleft l) (- n nleft)))))))
   (if (null l) l (l2t l (length l)))))

(defun find-primes (poly1 poly2 &optional (prime-floor 1))
  "estimate maximal coef of gcd"
  (let ((primes *primes*))
    (do () ((>= (car primes) prime-floor)) (setf primes (cdr primes)))
    (let ((estimate (* 2 (max (max-coef poly1) (max-coef poly2)))))
      (do ((working-primes nil)
           (rest-of-primes primes (cdr rest-of-primes))
           (prod 1))
          ((> prod estimate) (list2tree working-primes))
        (unless (bad-prime (car rest-of-primes) poly1 poly2)
          (push (car rest-of-primes) working-primes)
          (setq prod (* prod (car rest-of-primes))))))))

(defun bad-prime (prime poly1 poly2)
  (or (eq 0 (mod (lead-coef poly1) prime))
      (eq 0 (mod (lead-coef poly2) prime))))

(defun max-coef (l)
  (if (null l) 0 (max (abs (lead-coef l)) (abs (max-coef (cdr l))))))
```

poly-gcd computes prime-tree using find-primes. Then for each leaf it computes the gcd modulo the prime at that leaf and normalizes the result to have leading coefficient equal to the gcd of the leading coefficients of the input polynomials (modulo the given prime). Finally Chinese is applied to each internal node and the resulting 'raw' gcd is normalized by dividing out common factors of coefficients.

```
(defun mod-poly-normalize (poly prime n)
         (mod-scalar*poly
          (* (mod-inv (lead-coef poly) prime) n) .
          poly
          prime)))

(defun normalized-mod-poly-gcd (prime poly1 poly2 lead)
  (mod-poly-normalize (mod-poly-gcd poly1 poly2 prime) prime lead))

(defun poly-gcd (poly1 poly2 &optional (prime-floor 2))
  (let ((lead (gcd (lead-coef poly1) (lead-coef poly2))))
  (labels
      ((gcd-tree (prime-tree)
         (cond ((atom prime-tree)
                (cons prime-tree
                      (normalized-mod-poly-gcd prime-tree poly1 poly2 lead)))
               (t (chinese (gcd-tree (car prime-tree))
                           (gcd-tree (cdr prime-tree)))))))
    (let ((raw-gcd (gcd-tree (find-primes poly1 poly2 prime-floor))))
      (poly-div-scalar (cdr raw-gcd)  (gcd-poly-coef (cdr raw-gcd)))))))
```

The algorithm as implemented above is a heavy consumer of space. The main work and space consumption is in the computation of the initial gcds at the leaves. Each gcd computation involves repeated application of `mod-rmd`, which in turn repeatedly subtracts a multiple of its second argument from its first. Each subtraction conses up a list of newly created monomials. It is easy to see that if the first argument to `mod-rmd` is not shared then the result can be computed by updating that polynomial thus saving space needed for repeated copying. To do this we define `inplace-mod-scalar*poly`, a destructive version 'of `mod-scalar*poly` that reuses the input polynomial. `nmod-rmd` is obtained from `mod-rmd` by replacing the call to `mod-scalar*poly` by a call to `nmod-scalar*poly`. `nmod-poly-gcd` is obtained from mod-poly-gcd by replacing `mod-rmd` by `nmod-rmd`. `nmod-poly-normalize` is obtained from `mod-poly-normalize` by replacing `mod-scalar*poly` by `inplace-mod-scalar*poly` and `mod-poly-gcd` by `nmod-poly-gcd`. Finally normalized-mod-poly-gcd is redefined, replacing mod-poly-normalize by nmod-poly-normaliz and mod-poly-gcd by nmod-poly-gcd applied to fresh copies of the input polynomials.

```
(defun inplace-mod-scalar*poly (coef poly prime)
  (mapc #'(lambda (mono)
            (setf (mono-coef mono) (mod (* coef (mono-coef mono)) prime)))
        poly)
  poly)

(defun nmod-rmd (poly1 poly2 prime)
  (cond ((null poly1) nil)
        ((< (deg poly1) (deg poly2)) poly1)
          (t (let ((d1 (deg poly1))
                   (c1 (lead-coef poly1)))
               (nmod-rmd
                (mod-add (inplace-mod-scalar*poly (lead-coef poly2) poly1 prime)
                         (mod-mono*poly (- c1)
                                        (- d1 (deg poly2))
                                        poly2
                                        prime)
                         prime)
                poly2
                prime)))))

(defun nmod-poly-gcd (poly1 poly2 prime)
  (let ((r (nmod-rmd poly1 poly2 prime)))
    (if (null r) poly2 (nmod-poly-gcd poly2 r prime))))

(defun nmod-poly-normalize (poly prime n)
  (inplace-mod-scalar*poly
    (* (mod-inv (lead-coef poly) prime) n)
    poly
    prime)))

(defun copy-poly (poly) (mapcar #'copy-mono poly))

(defun normalized-mod-poly-gcd (prime poly1 poly2 lead)
  (nmod-poly-normalize
    (nmod-poly-gcd  (copy-poly poly1) (copy-poly poly2) prime)
    prime
    lead))
```

By similar reasoning we can also replace the first conditional branch of Chinese

```
(mapcar #'(lambda (mono)
            (make-mono :coef (chin-eq 0 (mono-coef mono))
                       :exp (mono-exp mono)))
        poly2))
```

by

```
(mapc #'(lambda (mono)
          (setf (mono-coef mono)
                (chin-eq 0 (mono-coef mono))))
      poly2))
```

and similarly for the second branch, but this is less important as these branches are rarely taken.

7.2 The Parallelism

By organizing the primes as the leaves of a tree, we obtain a degree of large grained parallelism. The amount of parallelism depends on how many primes we use, which depends on the size of the largest coefficients of the polynomials and the list of candidate primes. To avoid bignum arithmetic, the largest prime should be not greater than the largest f ixnum. If we use too many small primes, then the tree may be unnecessarily large. To implement the large grained parallelism we surround the call to Chinese in poly-gcd by `#!` . In the tables below this source of parallelism is indicated by the presence of an "A" in the variant column.

A second source of parallelism is in the Chinese algorithm itself. In the case that neither input polynomial is null, the construction of the resulting polynomial can be parallelized by replacing the let by `qlet` t. In the tables below this source of parallelism is indicated by the presence of a "B" in the variant column.

Finally there is fine grained parallelism the initial gcd computations at the leaves of the prime tree. The arguments to mod-add in nmod-rmd can be computed in parallel by wrapping the call in `#!`. In addition, the mapping functions in mod-scalar*poly and `mod-mon*poly` can be replaced by their parallel versions. In the tables below this source of parallelism is indicated by the presence of a "C" in the variant column.

The test polynomials are constructed as follows.

```
(defvar p0
  (mapcar #'pair2mono
          '((8 . 8) (7 . 7) (6 . 6) (5 . 5) (3 . 3) (2 . 2) (1 . 1) (0 . 10))))
(defvar p1
  (mapcar #'pair2mono
        '((8 . 8) (7 . 7) (6 . 6) (5 . 5)(4 . 4)(3 . 3) (2 . 2) (1 . 1) (0 . 10))))
(defvar p2
  (mapcar #'pair2mono
        '((8 . 8) (7 . 7) (6 . 6) (5 . 5)(4 . 8) (3 . 3) (2 . 2) (1 . 1) (0 . 10))))
(defvar p00 (poly-mul p0 p0))
(defvar p11 (poly-mul p1 p1))
(defvar p22 (poly-mul p2 p2))
(defvar q1 (poly-mul p11 p00))
(defvar q2 (poly-mul p22 p00))
(defvar q3 (poly-mul q1 q1))
(defvar q4 (poly-mul q2 q2))
(defvar q5 (poly-mul q3 q1))
(defvar q6 (poly-mul q4 q2))
(defvar q7 (poly-mul q5 q1))
(defvar q8 (poly-mul q6 q2))
(defvar q9 (poly-mul q7 q1))
```

The global variable *primes* is an increasing list of primes. In the tables below, we summarize results for two values of prime-floor, 100 and 1000.

When prime-floor is 100, more primes are used to compute the gcd. This provides more opportunity for parallelism, but also increases the amount of work required. A smaller number of primes gives better overall performance.

Type "A" parallelism alone generates a process for each leaf in the tree of primes. Since this tree is relatively small there is no need to constrain this source of process generation. The opportunity for parallelism is not a smooth function of the number of primes when this number is small. For example, if there are 7 primes, we get very good speed up, while if there 9 primes, then 8 primes get processed in parallel, leaving the last one to be shared (not very efficiently) by the 8 processors. Adding type "B" parallelism has little or no effect in running or idle time, although the number processes created is an order of magnitude greater. Again there seems to be no need to constrain this source of parallelism. Adding type "C" parallelism increases the number of processes by three orders of magnitude over type "A" only, and produces at most a modest improvement. Replacing **#!** by **#2?** in the type "C" parallelism does not decrease the number of processes created. This is because the source of parallelism is "flat", i.e. the calling tree is essentially a list and each pair of parallel calls is completed before the nest pair is generated.

Polynomial GCD, prime-floor 100							
polys	variant	para- time	processes	overhead	%	idle- time	%
q1, q2	serial	.825	1	.ooo	0.0	5.828	88.3
	A	.298	3	.ooo	0.0	1.529	64.2
	AB	.295	37	.003	0.1	1.529	64.8
	ABC	.225	2212	.153	8.5	.607	33.6
q1, q4	serial	5.491	1	.ooo	0.0	39.084	89.0
	A	.904	7	.001	0.0	1.549	21.4
	AB	.884	109	.008	0.1	1.480	20.9
	ABC	.927	6178	.205	2.8	.679	9.2
q3, q4	serial	9.593	1	.ooo	0.0	68.724	89.5
	A	1.722	7	.001	0.0	3.386	24.6
	AB	1.590	205	.015	0.1	2.708	21.3
	ABC	1.413	6399	.207	1.8	.720	6.4
q1, q6	serial	13.115	1	.ooo	0.0	94.045	89.6
	A	2.858	10	.001	0.0	9.473	41.4
	AB	2.995	163	.012	0.0	10.492	43.8
	ABC	2.431	16966	.762	3.9	3.737	19.2
q5, q6	serial	26.962	1	.ooo	0.0	209.950	97.3
	A	6.382	10	.001	0.0	22.063	43.2
	AB	6.078	451	.030	0.1	20.173	41.5
	ABC	5.210	20544	.939	2.3	7.827	18.8
q2, q9	serial	36.793	1	.ooo	0.0	264.000	89.7
	A	5.390	15	.001	0.0	5.658	13.1
	AB	5.287	253	.016	0.0	5.402	12.8
	ABC	5.481	35216	1.010	2.3	1.789	4.1
q7, q8	serial	59.479	1	.ooo	0.0	433.553	91.1
	A	10.607	13	.001	0.0	22.549	26.6
	AB	10.045	793	.045	0.1	17.570	21.9
	ABC	9.849	36546	1.397	1.8	5.334	6.8
q3, q6	serial	71.088	1	.ooo	0.0	501.553	88.2
	A	14.368	10	.001	0.0	42.296	36.8
	AB	14.725	307	.020	0.0	42.574	36.1
	ABC	10.350	18805	.798	1.0	4.616	5.6
q4, q7	serial	167.906	1	.000	0.0	1250.776	93.1
	A	28.990	12	.001	0.0	61.431	26.5
	AB	30.330	375	.024	0.0	62.635	25.8
	ABC	23.461	29473	1.111	0.6	3.459	1.8

Polynomial GCD, prime-floor 1000							
polys	variant	para-time	processes	overhead	%	idle- time	%
ql, q2	serial	.706	1	.ooo	0.0	4.961	87.9
	A	.370	2	.ooo	0.0	2.793	94.3
	AB	.398	19	.001	0 . 0	2.469	77.6
	ABC	.221	1620	.123	7 . 0	.839	47.4
ql, q4	serial	4.083	1	.ooo	0.0	28.127	86.1
	A	.891	5	.000	0 . 0	2.990	42.0
	AB	.896	73	.005	0 . 1	3.008	42.0
	ABC	.792	6592	.321	5 . 1	1.035	16.3
q3, q4	serial	6.647	1	.ooo	0.0	50.181	94.4
	A	1.749	5	.ooo	0.0	6.492	46.4
	AB	1.552	137	.009	0 . 1	5.738	46.2
	ABC	1.189	7475	.361	3 . 8	1.176	12.4
ql, q6	serial	8.756	1	.ooo	0.0	63.993	91.4
	A	1.475	7	.ooo	0.0	2.303	19.5
	AB	1.495	109	.008	0 . 1	2.402	20.1
	ABC	1.550	10153	.341	2 . 7	1.356	10.9
q5, q6	serial	21.016	1	.ooo	0.0	147.279	87.6
	A	3.598	7	.ooo	0.0	8.628	30.0
	AB	3.258	301	.021	0 . 1	5.322	20.4
	ABC	3.177	11071	.358	1 . 4	2.152	8.5
q2, q9	serial	27.016	1	.ooo	0.0	199.059	92.1
	A	5.313	11	.001	0 . 0	14.331	33.7
	AB	5.604	181	.013	0 . 0	15.869	35.4
	ABC	4.767	37334	1.726	4.5	3.375	8.9
q7, q8	serial	42.623	1	.ooo	0.0	330.264	96.9
	A	10.718	9	.001	0 . 0	40.515	47.3
	AB	9.868	529	.356	0 . 0	34.120	43.2
	ABC	8.575	23898	.996	1 . 5	17.859	26.0
q3, q6	serial	55.478	1	.ooo	0.0	390.997	88.1
	A	8.744	7	.ooo	0.0	12.596	18.0
	AB	8.489	205	.014	0 . 0	11.182	17.3
	ABC	7.794	11522	.416	0 . 7	1.479	2.4
q4, q7	serial	138.732	1	.ooo	0.0	1079.731	97.3
	A	32.369	9	.001	0 . 0	116.027	44.8
	AB	31.451	273	.019	0 . 0	106.643	42.4
	ABC	20.322	22157	.930	0 . 6	11.445	7.0

Chapter 8

Caveats and Conclusions

It is relatively simple to realize inherent parallelism using the dynamic spawning primitives. One simple, but effective, approach is to start with a high-level abstract description of the underlying algorithm. Using this description, determine the likely points for introducing parallelism, and determine the degree of parallelism at each point that produces the best results. Likely points for introducing parallelism include non tail-recursive function calls and various forms of iteration and structure traversal. To determine the optimal degree of parallelism one starts with unconditional parallelism at each point, and introduces dynamic control at those points that generate excessive numbers of tasks. In this process the main guidelines are provided by the overhead, idle time and task creation data.

The abstract description may well not provide an efficient implementation of the underlying algorithm. One can easily refine the description into a more efficient program as for the sequential case. This refinement can be carried out without significantly altering the fine-tuning of the parallelism. In particular, the points of parallelism and their degrees will remain unchanged, and the speedup will be similar, possibly better if synchronization bottlenecks are removed in the process. Two examples of useful refinements are:

1. Rerepresentation of abstract data types — rerepresenting composite structures as lists and omitting structure definitions for variables and operations.

2. Elimination of overly general program abstractions — replacing typecase by conditional and multiple-values by cons.

The programmer should be aware that because Qlisp processes exist in a single Lisp environment they share the data bases that the Lisp implementation maintains. Various definitions and other events update these data bases and hence in the parallel version they may need to be locked when accessed. In particular this may cause certain primitive operations to impair speed up due to locking. For example rerepresentation of structures as lists generally yields an increase in performance due to a decrease in locking. A spectacular example of this is the typecase problem. Our first version of boyer used defstructs and typecase dispatch. The code was elegant but the program was ten times slower than the original. This is due to the fact that typecase expands into calls to typep, which is very expensive. Worse yet we observed a 60 second (fifty percent) slow down when the sequential program was run in parallel mode. After some detective work we discovered that this was due to the fact that typep locks the structure data base.

Bibliography

[1] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, 1979.

[2] B. Buchberger, G. E. Collins, and R. Loos (eds.). *Computer Algebra; Symbolic and Algebraic Computation.* Springer-Verlag, 1983.

[3] G. E. Collins. *Computer Algebra of Polynomials and Rational Functions.* Am. Math. Monthly 80, 725-755 (1973).

[4] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.

[5] Richard P. Gabriel, and John McCarthy. *Queue-based Multi-processing Lisp* Computer Science Technical Report STAN-CS-84-1007, Stanford University, Stanford, CA, June 1984.

[6] M. Laurer. *Computing by Homomorphic Images.* in [2] pp. 139-168.

[7] *Stanford Qlisp Reference Manual.* Available from the Stanford Qlisp Group.

[8] Daniel Scales. *Parallelizing the OPS5 Matching Algorithm in Qlisp* Computer Science Technical Report Stanford University, Stanford, CA, (in preparation).

[9] *Qlisp Reference Manual.* Lucid Inc., in preparation.

[10] *Guy* L. Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[11] Joseph S. Weening. *Parallel Execution of Lisp Programs.* Phd. Thesis, Stanford University. Computer Science Technical Report STAN-CS-89-1265, Stanford University, Stanford, CA. June 1989.