Multiprocessor Execution of Functional Programs

Benjamin F. Goldberg
Research Report YALEU/DCS/RR-618
April 1988

A dissertation presented to the faculty of the Graduate School of
Yale University in candidacy for the degree of Doctor of Philosophy.

Multiprocessor Execution of Functional Programs

Benjamin F. Goldberg

YALEU/DCS/RR-618
April 1988

A dissertation presented to the faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy.

Abstract

Functional languages have recently gained attention as vehicles for programming in a concise and elegant manner. In addition, it has been suggested that functional programming provides a natural methodology for programming *multiprocessor* computers. This dissertation demonstrates that multiprocessor execution of functional programs is feasible, and results in a significant reduction in their execution times.

Two implementations of the functional language ALFL were built on commercially available multiprocessors. *Alfalfa* is an implementation on the Intel iPSC hypercube multiprocessor, and *Buckwheat* is an implementation on the Encore Multimax shared-memory multiprocessor. Each implementation includes a compiler that performs *automatic decomposition* of ALFL programs. The compiler is responsible for detecting the inherent parallelism in a program, and decomposing the program into a collection of tasks, called *serial combinators*, that can be executed in parallel. One of the primary goals of the compiler is to generate serial combinators exhibiting the coarsest granularity possible without sacrificing useful parallelism. This dissertation describes the algorithms used by the compiler to analyze, decompose, and optimize functional programs.

The abstract machine model supported by Alfalfa and Buckwheat is called *heterogeneous graph reduction*, which is a hybrid of graph reduction and conventional stack-oriented execution. This model supports parallelism, lazy evaluation, and higher order functions while at the same time making efficient use of the processors in the system. The Alfalfa and Buckwheat run-time systems support dynamic load balancing, interprocessor communication (if required), and storage management. A large number of experiments were performed on Alfalfa and Buckwheat for a variety of programs. The results of these experiments, as well as the conclusions drawn from them, are presented.

Acknowledgments

First and foremost, I would like to thank my advisor Paul Hudak who, through his insights and careful guidance, has made this dissertation possible. I would also like to thank my other committee members, Joe Fasel and Alan Perlis, for their suggestions and conscientious reading.

My work has benefited greatly from others at Yale, especially those in Hudak's "wrestling" group: Adrienne Bloss, David Kranz, Jonathan Young, Jim Philbin, Richard Kelsey, Lauren Smith, Rick Mohr, Juan Guzman, and Steve Anderson. Josh Cohen Benaloh proved invaluable for his advice on LaTeX as well as some of the contents of this work. I would like to thank Chris Hatchell for his flawless administrative support.

I benefited greatly from my collaboration with researchers at Los Alamos National Laboratory, especially Joe Fasel, Randy Michelsen, Bonnie Yantis, and Elizabeth Williams. I would also like to thank those in the functional programming community, especially Simon Peyton Jones, for their advice.

I would sincerely like to thank those people who made my life in New Haven interesting, stimulating, and most importantly, fun. Because of my friends, I will be able to look back on my student days with great fondness.

My wife Wendy has been far more supportive and understanding than I could ever have asked. I don't know how I would have survived without her. Baby... you're the greatest!

Finally, I would like to thank my parents. They have inspired me through their achievements, their commitment to learning, and above all, their faith in me. I dedicate this dissertation to them.

This research was supported in part by National Science Foundation grants DCR-8302018 and DCR-8521451, by a DARPA subcontract with SDC/Unisys, and by gifts from Burroughs Austin Research Center and the Intel Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Functional languages have recently gained attention as vehicles for programming in a concise and elegant manner. The merits of functional languages have been well argued [4,25,75,16,60]. The development of functional languages has benefited greatly from a large body of research on the foundations of programming languages and models of computation. In particular, functional languages have benefited from research on programming language semantics [68,69,77], the lambda calculus [9,5] and type theory [57,15,72].

It has been suggested that functional programming provides a natural methodology for programming *multiprocessor* computers. Several prototype machines have been built specifically for the purpose of executing functional programs in parallel. Until now, however, no functional language implementations have been built on the multiprocessors that are commercially available. This dissertation describes the first working implementation of a functional language on two commercially available multiprocessors.

1.1 Objectives

This dissertation seeks to answer the following question:

> Is it feasible to execute conventional functional programs on currently available multiprocessors, such that a significant reduction in the execution time is achieved?

Some of the terms used in this question need to be defined:

- *Conventional functional programs*: We seek to create an implementation for a functional language that contains *no special constructs* for specifying the parallel behavior of a program. Our implementation must be able to *automatically* decompose functional programs to run on a multiprocessor.

  If an algorithm is specified in a purely sequential manner, the implementation cannot hope to execute it efficiently on a multiprocessor. We do not seek to transform sequential algorithms into parallel algorithms, but rather to detect and exploit the parallelism that is implicit in a given functional program.

- *Currently available multiprocessors*: The multiprocessors available today are generally comprised of processors designed to execute programs written in sequential imperative languages. No special hardware support is provided for executing functional programs.

- *Reduction in execution time*: We are investigating whether a functional program can run significantly faster on a multiprocessor than on a sequential (uniprocessor) computer. We we would ultimately like to show that functional programming is the *most* appropriate method for programming parallel computers. However, in this dissertation we restrict ourselves to the investigation of the advantages of using parallel machines instead of sequential machines to execute functional programs.

In order to build an efficient multiprocessor implementation, a number of new evaluation techniques were developed for executing functional programs. These techniques include:

1. A compile-time analysis of the sharing that occurs during execution. This analysis uses abstract interpretation, a program analysis technique based on denotational semantics.

2. A heterogeneous evaluation model based on graph reduction and conventional sequential execution. Graph reduction is the evaluation model most commonly used by functional language implementations. While it is a powerful method for exploiting the generality of functional programs, it incurs unnecessary computational overhead in cases where its power is not needed. The implementation described here uses the power of graph

reduction when required, but reverts to a sequential stack-based execution of function calls whenever possible.

3. An automatic program partitioning technique that strives to exploit as much of the parallelism in a program as possible while maintaining a sufficiently coarse program granularity for efficient execution on current multiprocessors.

The first two techniques described above can also be used to make *sequential* evaluation of functional programs more efficient than it currently is.

Two multiprocessor implementations were built to test a variety of run-time support mechanisms for making effective use of the machines. A large number of empirical results on the effectiveness of various processor scheduling algorithms as well as a discussion of the strengths and weaknesses of the compiler are presented in this dissertation.

We assume that the reader has some familiarity with functional languages, graph reduction, and combinators. In the following section, we provide a very short introduction to these subjects. Those readers who have a strong background in functional languages and their evaluation can skip to section 1.3. An excellent discussion of these topics can be found in [61].

1.2 Functional Languages, Graph Reduction, and Combinators

1.2.1 Functional Languages

Functional languages are programming languages exhibiting the following characteristics:

- *Mathematical Notation*: The programs are written in a high-level notation resembling that of mathematics.
- *Referential Transparency*: There is no side-effect operator (such as assignment). Thus the programs exhibit referential transparency, the property in which identical expressions have identical values (within the same lexical scope).

- *Applicative structure*: A program consists of a collection of function and constant definitions, and an expression whose value constitutes the result of the program. Each expression in the program consists only of constants, identifiers, function applications, and perhaps nested definitions.

There are many functional languages. Some of the better known ones are FP [4], Miranda[1] [72], LML [2], SASL [74], ALFL [28], and FEL [47].

Most modern functional languages exhibit some additional properties:

- *Higher-Order Functions*: Functions are treated as *first class objects* in these languages. They can be passed as arguments to other functions and may be returned as the result of a function application. Functions that take functions as arguments or return functions as values are called higher-order functions. Functional languages that provide the ability to define higher-order functions are called *higher-order functional languages.*

  Higher-order functional languages generally allow function applications to be *curried*: If a function is defined to take several arguments, it can be thought of as a function that takes one argument and returns a function that takes another argument and so on.

- *Non-strict Semantics*: In functional programs written in languages with non-strict semantics, no argument in a function application is evaluated unless its value is required. The execution of a program written in a non-strict functional language is more likely to terminate than the same program written in a strict functional language. For example, non-strict languages provide the ability to write programs that use infinite data structures and still terminate.

For the rest of this dissertation, the term functional language will be used to refer only to non-strict, higher-order functional languages. Of the languages mentioned above, Miranda, LML, SASL, ALFL, and FEL are higher-order and non-strict.[2]

---

[1] 'Miranda' is a trademark of Research Software Ltd.

[2] For the remainder of this dissertation, we will use the term *lazy language* informally to mean a non-strict language.

1.2.2 ALFL: A non-strict, higher-order functional language

This dissertation describes an implementation of **ALFL**, a non-strict higher-order functional language [28]. ALFL is similar in many ways to the other lazy higher-order functional languages mentioned above. It provides implicit typing and requires that any implementation perform run-time type checking. ALFL functions are fully curried although many common infix operators, such as arithmetic operators, are provided for convenience. Like many other functional languages, ALFL provides pattern matching as an elegant way to define functions based on the structure of their arguments.

An ALFL program consists of an *equation group* which is a set of equations and a result expression delimited by braces. Each equation defines either a function or a constant (i.e. a function with no arguments):

$$
\begin{array}{l}
\{\ f_1\ x_{11} \ldots x_{1m_1} \texttt{==}\ e_1\texttt{;} \\
\quad \ldots \\
\quad f_n\ x_{n1} \ldots x_{nm_n} \texttt{==}\ e_n\texttt{;} \\
\quad \texttt{result}\ e\texttt{;} \\
\quad \}
\end{array}
$$

Each expression may consist of applications of functions, applications of primitive operators (such as + and -), and nested equation groups. ALFL uses block structure and static scoping to resolve identifiers. Functions defined within the same equation group may be mutually recursive. Here is a sample ALFL program that defines and uses the higher-order `map` function to form a list whose elements are the square of the elements of a given list:

```
{ map f l == l=[] -> [], f (hd l) ^ map f (tl l);
  square l == map { sq n == n*n;
                    result sq;
                    }
                  l;
  result square [1,2,3,4,5];
  }
```

The conditional operator in ALFL has the form `p ->c,a` where `p`, `c`, and `a` are the predicate, consequent, and alternate respectively. The infix operator `^`

denotes the list construction operator (similar to cons in Lisp) and [] denotes the null list.

The function map can be rewritten using ALFL's pattern matching syntax:

```
{ map f [] == []
  ' f (x^l) == f x ^ map f l;
  square l == map { sq n == n*n;
                    result sq;
                    }
                  l;
  result square [1,2,3,4,5];
  }
```

A function may be defined by a number of equations. The left hand side of each equation is a pattern that specifies the forms the arguments must take for the equation to apply. In the above program, if the second argument is [], then map returns []; otherwise the second argument is non-null and the identifiers x and l are bound to the head and the tail of the list, respectively.

It is straightforward (via the insertion of conditionals) to transform a program that uses pattern matching into one that does not. For simplicity, we will not discuss any methods for compiling and executing pattern matching, but rather we will assume that the pattern matching has been eliminated by a previous phase of the compiler.

Figure 1.2.2 describes the abstract syntax of the simplified version of ALFL that we have implemented. Arithmetic operators follow the usual precedence rules and function applications associate to the left. For a more complete description of ALFL, see [28].

1.2.3 The Lambda Calculus

The untyped lambda calculus is a formal model for specifying computation. It it is an extremely simple language that consists of a few kinds of syntactic objects and a few syntactic conversion rules. It is also very powerful,being equivalent to a Turing machine in computational power. Its simple structure provides a useful basis for reasoning about programs. In this section, we provide an informal introduction to the lambda calculus; a formal, in-depth discussion can be found in [5].

```
program        ::=  equation_group
equation_group ::=  { (equation ;)*
                      result exp ; }
equation       ::=  id ( id)* == exp
exp            ::=  id | constant | exp bin_op exp | -exp |
                    exp -> exp, exp | ẽxp
bin_op         ::=  + | - | * | / |  ^ | ^^ | < | > | ...
constant       ::=  integer | float | [] | predefined identifiers
```

Figure 1.1: The syntax of a simplified version of ALFL

Every expression in the lambda-calculus is a *lambda expression*, defined as follows (using the notation used in [61]):[3]

```
exp ::=  constant                    (constants)
       | id                          (identifiers)
       | exp exp                     (applications)
       | λ id . exp                  (lambda abstractions)
```

In a lambda abstraction A such as $\lambda\ x.E$, where E is some lambda expression, the variable x (following the λ symbol) is said to be *bound* in A and is called a *bound variable*. Any variable y in E (other than x) that is not bound in a lambda abstraction inside E, or that occurs outside of the lambda abstraction in which it is bound, is said to occur *free* in A and is called a *free variable*.

The lambda calculus provides several conversion rules for converting one lambda expression into an equivalent one. Evaluation of a lambda expression proceeds by repeatedly applying conversion rules until no more conversions can be applied. The conversion rule most often used is *β-reduction*:

$$(\lambda\ x\ .\ E)\ M \implies E[M/x]$$

This rule states that an application of an abstraction $(\lambda x.\ E)$ to an expression M can be converted into the expression obtained by replacing all free occurrences of $\mathbf{x}$ in E by copies of the expression M (and renaming bound variables in E as necessary to avoid name conflict with free variables in M). This new

[3]Actually, the *pure* lambda calculus has no constants (or primitive operators). For our purposes, we extend the lambda calculus these items.

expression is denoted by $E[M/x]$. The built-in constants of the lambda calculus considered here include operators such as + and - with their own conversion rules. For example, (* 3 2) $\Longrightarrow$ 6.

A given lambda expression may contain a number of expressions to which β-reduction can be applied. These expressions are called reducible expressions or *redexes*. If β-reduction is always applied to the leftmost outermost redex first, then the resulting evaluation order is *normal order*. Otherwise, if β-reduction is applied to the innermost redexes before any others then the resulting evaluation order is *applicative order*. The reduction process terminates when the lambda-expression has been reduced to a *normal form*, in which there are no redexes. Not all expression have normal forms, in which case the reduction process would not terminate.

Consider the lambda expression (λ x. + x x) (* 3 2). Applicative order evaluation would proceed as follows:

(λ x. + x x) (* 3 2) $\Longrightarrow$ (λ x. + x x) 6
$\Longrightarrow$ + 6 6
$\Longrightarrow$ + 12

Notice that (* 3 2) was evaluated first. Normal order evaluation would proceed in the following way:

(λ x. + x x) (* 3 2) $\Longrightarrow$ + (* 3 2) (* 3 2)
$\Longrightarrow$ + 6 6
$\Longrightarrow$ + 12

Unfortunately, since every occurrence of x is replaced by a copy of the argument, the expression (* 3 2) is evaluated twice.

If an argument is not used in the body of a function, however, normal order evaluation may be more efficient since the argument would not be evaluated at all. In fact, if the argument has no normal form, the normal order evaluation of the application might still terminate while applicative order evaluation would not. Normal order evaluation is *normalizing*, that is it terminates for any expression that has a normal form.

In the lambda calculus, recursion is implemented via the Y combinator, whose behavior is $Y\ f = f\ (Y\ f)$ and can be defined as a lambda-abstraction:

$$Y = (\lambda h.\ (\lambda x.\ (x\ x))\ (\lambda x.\ h\ (x\ x)))$$

One can express a recursive function, say factorial, as a lambda expression containing Y:

$$Y\ (\lambda fac.\ \lambda x.\ IF\ (=\ x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1))))$$

For convenience, the lambda calculus is often enriched to allow one to provide names for lambda expressions. For example, given the definitions

$$\begin{aligned} y &= 3 \\ f &= \lambda x.\ +\ x\ 1 \end{aligned}$$

the application $(f\ y)$ would be evaluated by substituting 3 for y, substituting $\lambda x.\ +\ x\ 1$ for f and performing β-reduction in the usual way. In addition, this *enriched lambda calculus* allows explicitly recursive (and mutually recursive) function definitions For example,

$$fac\ =\ \lambda x.\ IF\ (=\ x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1))))$$

would be a valid definition of factorial and would exhibit the same behavior as our prior definition.

In ALFL and most functional languages, this definition can be written with the bound variable x occurring on the left hand side:

```
fac x == x=0 -> 1, x * fac (x-1);
```

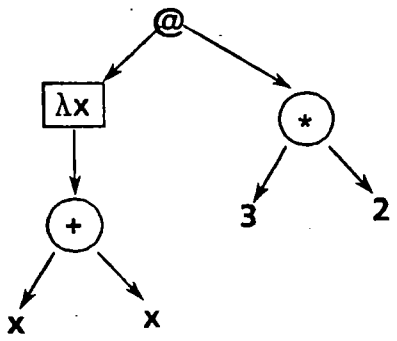
The translation of expressions from ALFL into the enriched lambda calculus is straightforward. Where convenient, we will use ALFL notation for expressions in the enriched lambda calculus.

1.2.4 Graph Reduction

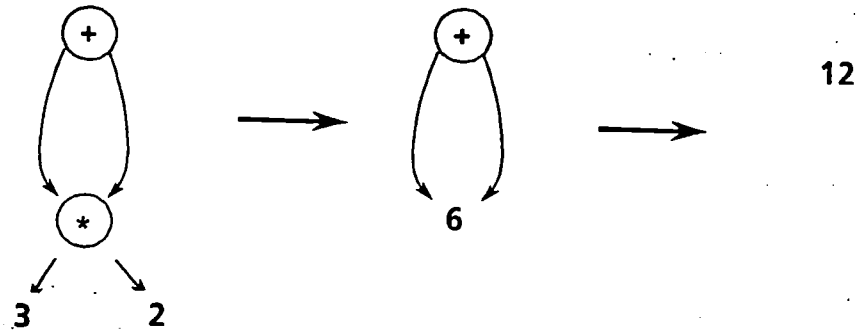
Graph reduction [78] is the evaluation method most often used to execute functional programs. It can be thought of as the graphical equivalent of reduction in the lambda calculus, and supports higher-order functions and normal order evaluation in a very natural manner. In graph reduction, the expression to be evaluated is represented as a graph. During execution, reductions are applied to the graph until it has been reduced to a normal form.

The graph structure provides a fundamental improvement in β-reduction over the method described in the previous section. Performing β-reduction on

the application $(\lambda x.E)\ M$ proceeds via the construction of an instance (copy) of E with all free occurrences of x replaced by *pointers* to the argument M. For example, the expression $(\lambda x.\ +\ x\ x)\ (*\ 3\ 2)$ would be represented by



where each node labeled "@" represents an application.[4] When reduction is performed, the graph is reduced to



The operator + requires the values of both arguments; when the first argument is evaluated, the subgraph representing $(*\ 3\ 2)$ is reduced to 6. Since the second argument to + is represented by a pointer to this subgraph, the value 6 is available without recomputing $(*\ 3\ 2)$. Thus, graph reduction supports normal order evaluation *without* duplicating expressions. Normal order evaluation without duplication (and recomputation is called *lazy evaluation.*[5]

[4]For convenience, we place the primitive operators at the nodes in the graph.

[5]In most implementations of normal order and lazy evaluation, reduction terminates when a *weak head normal form*, in which there is no top-level redex, is reached.

It should be pointed out that:

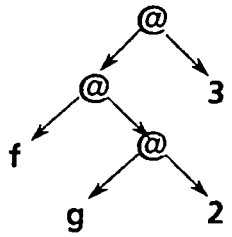
1. A redex may be shared (pointed to by several objects in the graph). When the redex is reduced, the root of the redex must be overwritten with the result in order to avoid having to reduce it again later on.

2. A lambda abstraction may be shared, thus new a instance of its body must be constructed during β-reduction, instead of substituting pointers into its body directly.

Graph reduction can also be used to evaluate expressions written in the enriched lambda calculus (or an equivalent functional language). For example, the initial graph representing the ALFL program

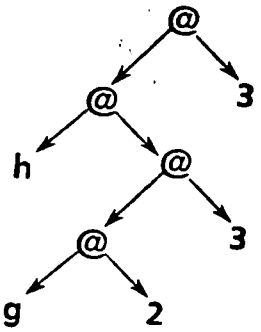
```
{  f x y == h (x y) y;
   g a b == a + b;
   h c d == c * d;
   result f (g 2) 3;
   }
```

would be

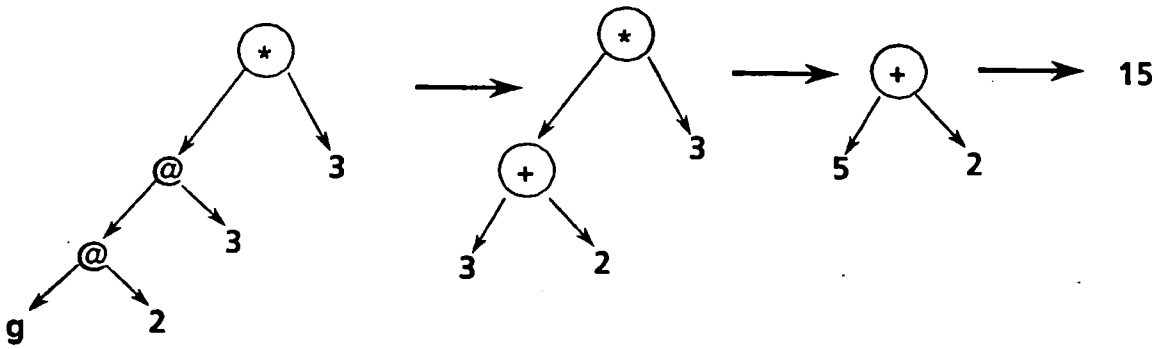


Notice that the application of `f` to two arguments is curried and is represented by two application nodes. Reduction proceeds via the construction of an instance of `f`'s body with its formal parameters replaced by pointers to the cor-

responding arguments:[6]



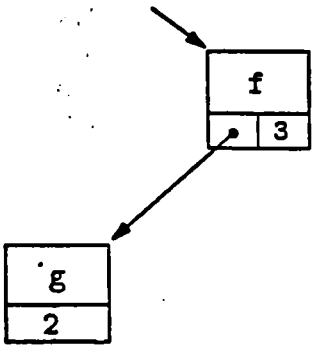
According to the definitions of g and h, the reduction of the graph proceeds as follows:



In the above example, the function identifier in an application always resided at a leaf in the graph to support currying. Each interior node represented the application of its left child to its right child. If a function is supplied with all the arguments it needs (as in the above application of `f`), an *uncurried* application could be represented by a node containing the function and arguments. For example, the uncurried version of `f (g 2) 3` in the above program would be

[6]Actually, this takes two reduction steps: For the first application, an instance of `f`'s body is constructed with the first argument substituted for `x`. For the second application, a new instance of the subgraph resulting from the first application is created with the second argument substituted for `y`.

represented by:



In this case, the node serves as an *activation record* for the function call.[7]

This brief description of graph reduction will be elaborated upon throughout this dissertation. For those readers who are completely unfamiliar with graph reduction, an excellent discussion of the subject can be found in [61].

1.2.5 Combinators

A costly feature of graph reduction is that each time a lambda abstraction (or a function identifier) is applied, an instance of its body must be constructed and have its bound variables replaced by the arguments. We would prefer to be able to *compile* the body of each lambda abstraction into a sequence of instructions that compute a result (and overwrite the root of the redex). The arguments to a lambda abstraction would comprise the *context* in which its sequence of instructions is executed.

Unfortunately, free variables within the body of a lambda abstraction provide an obstacle to its compilation. For example, the result of evaluating the expression

$$(\lambda x.\ \lambda y.\ +\ x\ y)\ 5$$

is $(\lambda\ y.\ +\ 5\ y)$, a new lambda abstraction. Every application of $(\lambda x.\ \lambda y.\ +\ x\ y)$ to a different argument will create a new and different lambda abstraction. We must be provide the sequence of instructions representing $(\lambda y.\ +\ x\ y)$ with a way to access the value of the free variable x. There are basically two ways to

[7] In applications that cannot be uncurried, an explicit apply node is required.

provide access to free variables:

1. A hierarchical environment structure could be maintained that provides a path between the use of a free variable and the context (activation record) of the lambda abstraction in which it was bound. In conventional languages, this is usually implemented by either a static chain or a display.
2. The lambda abstractions can be transformed into *combinators*. Combinators are simply functions that contain no free variables and no nested lambda abstractions. In a combinator body, all variables references are to variables that occur in the formal parameter list. No hierarchical environment structure is required to support the evaluation of combinators since the values of all variables in a combinator body have been passed as arguments.

There is an advantage to using combinators instead of a hierarchical environment structure that is particular to parallel implementations of functional programs. In a parallel implementation using environments, the use of a free variable in a function could occur on a processor other than the one on which the variable was bound. Thus, interprocessor communication would be required to resolve the variable reference. This would not be the case with combinators, since all variables are bound and are contained in the local context.

Any lambda expression (and thus any functional program) can be translated into an expression containing only references to a *fixed* set of combinators [13,67]. In fact, the two combinators S and K,

$$\begin{aligned} Sfgx &= fx(gx) \\ Kxy &= x \end{aligned}$$

are sufficient.

Combinator reduction is a special case of graph reduction in which all functions are combinators. Although the S and K combinators are sufficient to execute any program, several other combinators have been added for efficiency [73]. Because the behavior of the fixed combinators is so simple, they can be viewed as the instructions of an abstract machine (or even a real machine).[8] However,

[8]Several machines designed to execute fixed combinators have been built and are described

this kind of combinator reduction tends to be very fine-grained and to incur a high overhead in space (because of large combinator expressions) and time (because of the large number of reductions required to perform rather simple operations).

To improve combinator reduction, Hughes observed that instead of relying on a *fixed* set of combinators, one could derive a *different* set of combinators for each program [40]. He called these derived combinators *supercombinators*. His algorithm for deriving supercombinators is discussed in detail in chapter 2.

The translation of expressions in the enriched lambda calculus (or any functional language) into supercombinators is called *lambda lifting* [45]. In its simplest form, lambda lifting essentially adds all free variables in a function definition to its formal parameter list. Any application of the function is also modified to include the free variables as arguments. For example, the ALFL program

```
{ f x == { g y == x + y;
           result g 1;
           };
  result f 7;
  }
```

would be transformed by lambda lifting to

```
{ f x ==  g x 1;
  g x y == x + y;
  result f 7;
  }
```

Lambda lifting may also be applied to expressions in the ordinary lambda calculus. Lambda lifting proceeds by giving each lambda abstraction a name and creating a formal parameter list that includes all variables referenced in the body. The occurrence of the lambda abstraction is replaced an application of its given name to the variables that were free in its body. For example, lambda lifting

```
λ x. (+ x (λ y. (* x y)) 1) 2
```

---

in chapter 9.

would result in the following supercombinator definitions:

$$f1\ x\ y = (*\ x\ y)$$
$$f2\ x = +\ x\ (f1\ x\ 1)$$

The supercombinator expression corresponding to the above lambda expression would be $(f2\ 2)$.

This form of lambda lifting creates supercombinators that may be less efficient than those generated by Hughes's method of lambda lifting. Hughes's supercombinators can be considered more lazy than the ones generated via the above method (see chapter 2).

1.3 Dissertation Outline

The chapters in this dissertation roughly follow the order in which we compile ALFL programs into target code for the various multiprocessor implementations. The last few chapters describe the implementations, and present execution timings for a number of different programs.

The compilation process described here does not include lexing and parsing. but proceeds from an abstract syntax tree. In addition, several transformations are assumed to have already been applied. These include:

- *Common subexpression elimination*: Common subexpression elimination may be performed via the abstraction of multiple occurrences of a subexpression from the expression containing all such occurrences. For example, the program

```
{ f x y == (x+y) + (x * (x+y));
  result f 3 4;
  }
```

would be transformed into:

```
{ f x y == { g z == z + x * z;
             result g;
             } (x+y);
  result f 3 4;
  }
```

- *Partial Evaluation*: Optimizations such as constant folding, integration of non-recursive functions, and evaluation of simple expressions may have already been applied using techniques described in [34].

This dissertation is organized as follows:

- **Chapter 1. Detecting Sharing of Partial Applications in Functional Programs.** This chapter describes an analysis technique for detecting when partial applications of functions are shared. The analysis is used to transform a lambda-lifted ALFL program into a new set of combinators called *refined supercombinators* in the first phase of the compilation process. Refined supercombinators are more efficient versions of Hughes' supercombinators. This chapter has appeared previously in a slightly different form [20].

- **Chapter 2. A Characterization of Parallelism and Granularity in Functional Programs.** This chapter digresses from the compilation process in order to discuss formally the issues involved in decomposing functional programs for multiprocessors. It characterizes the types of parallelism that can be exploited as well as the costs involved with each type. The transformations presented in chapter 3 are based upon the insights provided in chapter 2.

- **Chapter 3. Automatic Partitioning of Functional Programs.** This chapter describes the decomposition of a program (represented as a set of refined supercombinators) into a set of *serial combinators*. Each Serial combinator is a function that specifies the behavior of an individual task. Parallelism is exploited by executing a large number of these tasks simultaneously. Unlike ALFL functions and refined supercombinators, serial combinators contain constructs for creating other tasks and synchronizing their execution. A serial combinator call can only be executed on a single processor and thus determines the *granularity* of the parallel computation. The second phase of the compilation process strives to define serial combinators with the appropriate granularity for the target multiprocessor. A preliminary definition of serial combinators appeared in [33].

- **Chapter 4. A Heterogeneous Graph Reduction Model.** This chapter describes a new evaluation model for graph reduction that combines graph allocation of activation records (nodes) with stack allocation of activation records. Graph allocation is generally required when a serial combinator is higher order or non-strict (requiring the creation of a node to represent the delayed evaluation of an argument) or if it spawns parallel invocations of other serial combinators. If a serial combinator exhibits none of these properties, then its activation records may be allocated on a stack for greater efficiency. The compiler performs this analysis and modifies the serial combinators to explicitly specify whether an application should be graph or stack allocated.

- **Chapter 5. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor.** This chapter describes the implementation of a heterogeneous graph reducer on the Intel iPSC hypercube multiprocessor. The implementation is called *Alfalfa* and provides run-time support for serial combinator execution. This support includes dynamic scheduling of tasks among the processors, message handling, and storage management. The last phase of the compiler, code generation for Alfalfa, is discussed in detail here. A brief description of Alfalfa appeared in [21].

- **Chapter 6. Dynamic Scheduling in Alfalfa.** This chapter explores a number of methods for scheduling tasks onto the processors in Alfalfa at run time. These methods fall into the class of algorithms known as *diffusion scheduling*. Alfalfa was tested on five different programs using a variety of diffusion heuristics. The results of these experiments are presented along with conclusions about Alfalfa performance.

- **Chapter 7. Buckwheat: Graph Reduction on a Shared Memory Multiprocessor.** This chapter describes an implementation, called *Buckwheat*, on the Encore Multimax shared memory multiprocessor. The description is brief since Buckwheat is essentially a simplified version of Alfalfa. In Buckwheat, a shared queue is used for task scheduling. In order to reduce contention for the queue, a hierarchical queue structure is used. Buckwheat was tested on the same programs as Alfalfa; the re-

sults of these experiments are presented, along with a comparison of the effectiveness of Buckwheat and Alfalfa.

- **Chapter 9. Related Work, Future Work and Conclusions**. This chapter provides a brief description of research projects that have had an effect on the work described here. It also presents some ideas for future exploration and summarizes the conclusions reached as a result of this research.

For readers mainly interested in the mechanics of parallel graph reduction, chapters 4, 5, and 6 provide a largely self-contained coverage of this topic. Chapter 2 should be skipped by any reader not interested in (nor already familiar with) the details of abstract interpretation. Those readers already familiar with parallel graph reduction can read chapters 6, 7, and 8 for a description of the manner in which graph reduction was implemented on the Intel iPSC and Encore Multimax multiprocessors.

Chapter 2

Detecting Sharing of Partial Applications in Functional Programs

In this chapter, we describe the first phase of our compilation process. It contains two components:

1. An analysis, based on denotational semantics, of the sharing of partial function applications.
2. A source-to-source transformation of the functions defined in the source program into a new set of functions that execute more efficiently on both uniprocessors and multiprocessors.

2.1 Sharing in functional programs

A key aspect of graph reduction is its ability to share expressions (subgraphs) during beta reduction. For example, given the program

```
{  f x y == x + y;
   g a == a 1 + a 2;
   result g (f 3);
   }
```

the expression `(g (f 3))` is evaluated by substituting a pointer to `(f 3)` for each occurrence of `a` in the body of `g`. This substitution is pictured in Figure 2.1.

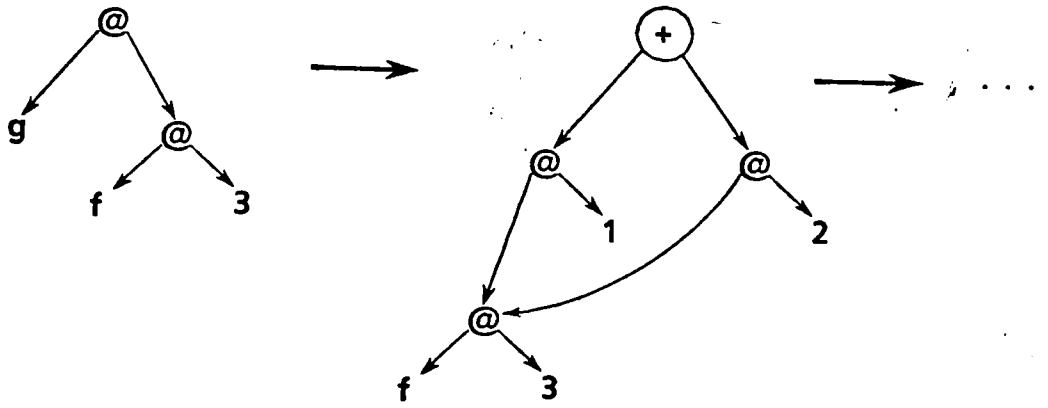


Figure 2.1: Sharing of expressions in graph reduction

If there are multiple references to an expression e we say that e is *shared.* For reasons described in section 2.4.2 we are interested in determining which *partial applications* are shared. Given a function definition of the form

$$f\ x_1 \ldots x_n = \ \ldots$$

any application of f to k arguments, $k < n$, is called a *partial application of* f. An application of f to n arguments is called a *complete application.*We would like our analysis to answer the following question:

For each function f defined by

$$f\ x_1 \ldots x_n = \ \ldots$$

and for each value of k, $0 \leq k < n$, what is the maximum number of times a partial application of f to k arguments could be shared?

2.2 A Naive Approach

A naive approach to sharing detection would be to examine how each partial application is used. If a partial application occurs once and is not passed as an argument to another function then the partial application is not shared. If the partial application is passed as an argument to a function, the number of

occurrences of the corresponding formal parameter in the body of the function would seem to determine if the partial application was shared. If so, a purely syntactic analysis in which the bound variables in function definitions are counted would suffice. However, such a syntactic analysis is insufficient for two reasons:

1. Our source language, ALFL, is a lazy functional language. Although a bound variable occurs multiple times in the body of a function, it may never be used. For example in the program

   ```
   {  f x y == x + y;
      h a b == 1;
      g c == h c c;
      result g (f 1);
      }
   ```

   the bound variable c corresponding to `(f 1)` in the body of g occurs twice but is never used.

2. Because ALFL is higher order, it may not be obvious which function a partial application is passed to. For example, in the program

   ```
   {   f x y == x+y;
       h c  == c 1 + c 2;
       g a b == a (f b);
       result g h 1;
       }
   ```

   `(f 1)` is passed to a function that the variable a is bound to. A more sophisticated analysis is required to determine the behavior of this function.

2.3 Semantics-based sharing analysis

The method we use to detect sharing is one that has given promising results in recent work on other aspects of functional languages [7,36]. We first define *non-standard* denotational semantics for ALFL in which the meaning of a program is an exact description of the sharing properties of the program. Unfortunately, obtaining such exact information is undecidable since it amounts to running

the program. Thus this method is not a tool that can be used at compile time.

We define an *abstraction* of the nonstandard semantics that will provide us with useful, although less complete, sharing information. The compile-time "interpretation" of a program using an abstraction of an exact semantics is called *abstract interpretation.* Abstract interpretation was invented by Cousot and Cousot [12] as an analysis tool for conventional languages. Alan Mycroft first used it for strictness analysis of programs written in first order functional languages [58]. Recently it has been extended to analyze the strictness properties of programs written in higher order languages and programs that contain lists [8,10,39,76].

In graph reduction, sharing can occur only when a variable that has been bound to a partial application is used multiple times. Our analysis does not perform common subexpression elimination (cse) and assumes that it has already been performed by the abstraction of common subexpressions from the expressions in which they occur (section 1.3).

2.3.1 Representing Sharing Information

The first step in describing a nonstandard sharing semantics for ALFL is to define a semantic domain. In this section we describe a domain S called the *sharing domain.* The value of each expression in a program is an element of S and must contain a description of the sharing properties of the expression.

Each value in S must contain information indicating whether or not that value represents a partial application. If a variable is bound to to a value that represents a partial application and the variable is used more than once, then the partial application must have been shared. If a variable is not bound to a partial application, no sharing will occur no matter how many times the variable is used.

Different partial applications must be represented by different values, even if the partial applications are lexically identical and have the same value in the standard semantics. For example, during the evaluation of the program

```
{  f x y == x + y;
   g a b == a 1 + b 1;
   result g (f 1) (f 1);
   }
```

the variables a and b in the body of g are bound to *different partial applications* and no sharing of (f 1) occurs.

When an expression is evaluated, its value must also contain a list of the partial applications that were shared during the evaluation of the expression. For example, the result of executing the program

```
{  f x y == x + y;
   j z w == z + w;
   g a == h a (j 1);
   h c d == (c 1 + c 2) * (d 3 + d 4);
   result g (f 1);
   }
```

should be a value in S that indicates partial applications of both f and j to a single argument were shared.

Since the value of an expression in the standard semantics may be a function, a value in S must also be able to capture the behavior of a function over S. Therefore, the value $s \in S$ resulting from the evaluation of an expression e is a triple of the form $\langle p,\ l,\ f \rangle$ where:

- p indicates whether e represents a partial application and, if so, provides enough information to differentiate it from other partial applications. This information is called the *p-value* of e.
- l is a list of partial applications that were used during the evaluation of e and is called the *l-value* of e.
- f is a function over S that captures the higher order behavior of e and is called the *f-value* of e.

The p-value

The p-value of an expression e is a pair of the form

$$[id\ n]$$

where *id* is an identifier and *n* is a natural number (which serves as a count). This tuple can be interpreted as follows: "Expression *e* represents the application of the bound variable *id* to *n* arguments. The result of this application, and thus the value of *e*, is a partial application." If the *p*-value of an expression *e* is [] then *e* does not represent a partial application.

For example, in the program

```
{  f x y z == x+y+z;
   g b == b 2;
   result g (f 2);
   }
```

the *p*-value of `(b 2)` inside the body of `g` is `[b 1]` since the variable `b` is bound to a partial application and was applied to one argument. Likewise, the *p*-value of `(f 2)` in the expression `g (f 2)` would be `[f 1]` since `(f 2)` represents an occurrence of a partial application.

Suppose an expression *e* is a partial application with a *p*-value of $[id\ j]$ for some identifier *id* and some value *j*. The *p*-value of an application of *e* to another argument would be $[id\ (j+1)]$ as long as the result is still a partial application.

Since the *p*-value of `(b 1)` does not indicate which function is partially applied, how can it be determined that the partial application bound to `b` is `(f 1)`? When `g` was called, the *p*-value of its argument was `[f 1]`. However, the *p*-value of the corresponding formal parameter, `b`, was bound to `[b 0]`. After the body of *g* has been evaluated, the identifier `b` is replaced by the identifier `f` in the resulting *p*-value and the number of arguments that `b` was applied to in `g` (namely 1) is added to the number of arguments that `f` had been applied to when `g` was called. After this substitution, the correct *p*-value for the program, namely `[f 2]`, is returned.

In the standard semantics, when a function is applied its body is evaluated with the values of the arguments substituted for the formal parameters. In the sharing semantics described in section 2.3.2, *two* substitutions occur during the evaluation of a function application. On entry to a function the *p*-value of each formal parameter is bound to a "placeholder" (such as `[b 0]` above). After the body of the function has been evaluated, any placeholder in the result is replaced by the original value of the corresponding argument.

Why not simply bind the formal parameter to the value of the corresponding argument in a function application. A problem arises in the following program:

```
{  f x y == x+y;
   g a b == a 1 + b 2;
   result g (f 1) (f 1);
   }
```

Although both arguments to `g` have identical values in the standard semantics, they represent *different* partial applications.[1] Therefore, the corresponding formal parameters, `a` and `b`, must be recognized as being bound to different partial applications. If `a` and `b` were bound to the same value then it would appear as though they represented the same partial application and that `(f 1)` was shared in the body of `g`. One solution to this would be to create a unique identifier for every partial application. However, as discussed in section 2.3.4 creating new identifiers creates a termination problem for the analysis.

It may seem that another solution would be to label each *syntactic* occurrence of a function application in order to be able to differentiate between different partial applications. In the above example, the two occurrences of `(f 1)` would be given different labels and could be recognized as being different partial applications. Consider the following program:

```
{ f x y z == x+y+z;
   h a == a 4;
   g b == h b 2 + h b 3;
   result g (f 1);
   }
```

Notice that no partial application of `f` to two arguments is shared. Two *different* partial applications (each representing `(f 1 4)`) are created by executing the body of `h` twice. However, both of these partial applications would incorrectly be labeled identically since they were created by the same sections of the program.

Instead we use the names of the formal parameters of `g` as placeholders to distinguish between the two partial applications of `f`. The process of binding the p-values of formal parameters to placeholders on entry to a function and

[1] Although common subexpression elimination could have been applied in this case, in general it cannot determine that two expressions are semantically identical.

back-substituting real values into the result is described in section 2.3.1.

The *l*-value

The *l*-value is the second element of a value in S. It is a *set* of tuples, $\{t_1, \ldots, t_n\}$, where each tuple, t_i has the form

$$[id\ v_0 \ldots v_n]$$

The value of each v_i represents the maximum number of times that an application of the variable id to i arguments has occurred. Thus, if $v_i \geq 2$ then the application of id to i arguments is shared. For example, the tuple [b 1 2 1] indicates that there is

- one occurrence of `b` applied to no arguments,
- two occurrences of an application of `b` to one argument, and
- one occurrence of an application of `b` to two arguments.

Given the program

```
{  f x y == x+y;
   j z w == z+w;
   g a == h a (j 1);
   h c d == (c 1 + c 2) * (d 3 + d 4);
   result g (f 1);
   }
```

the *l*-value of the result expression, `g (f 1)`, would be:

$$\{[\texttt{g 1 1}], [\texttt{h 1 1 1}], [\texttt{f 1 2 1}], [\texttt{j 1 2 1}]\}$$

This *l*-value indicates that only the partial applications of `f` and `j` to single arguments were shared.

In the body of `h`, the *l*-value of the expression `(d 3 + d 4)` would be {[d 2 1]}, since `d` occurs twice. There is no sharing of an application of *d*.

Formal parameter names (such as `d` above) occur in elements of an *l*-value as placeholders in the same way they occur in a *p*-value. Before the value of the function application is returned, all formal parameter names occurring in the resulting *l*-value are replaced by the *p*-values of the corresponding actual parameters (after converting the *p*-values to a form that is appropriate for an element of an *l*-value)

Merging *l*-values

During the evaluation of an expression e several applications of a variable id may have occurred. Thus the l-value of e would contain several tuples representing the application of id. These tuples must be merged to indicate that id was shared.

The function that merges two tuples representing applications of the same variable is called *merge_tuple* and is defined as follows:

$$\begin{aligned} &merge\_tuple([id\ v_0 \ldots v_n], [id\ v'_0 \ldots v'_m]) = \\ &\qquad [id\ (v_0 + v'_0)\ \max(v_1,\ v'_1)\ \ldots\ \max(v_n,\ v'_n)\ v'_{n+1} \ldots v'_m] \end{aligned}$$

where both tuples have the same id (and it is assumed that $m \geq n$). Only the count of the occurrences of id applied to no arguments is increased, to $v_0 + v'_0$, when the tuples are merged.The number of occurrences of id applied to i arguments, $i > 0$, is the maximum such number in the two tuples, namely $\max(v_i, v'_i)$. If two tuples do not represent application of the same id, then *merge_tuple* cannot be applied to them.

The function *merge* takes two l-values and merges them as follows:

$$\begin{aligned} &merge(l_1, l_2) = \\ &\quad \{merge\_tuple(t, t') \mid t \in l_1, t' \in l_2 \text{ and } \mathrm{id}(t) = \mathrm{id}(t')\} \\ &\quad \cup \\ &\quad \{t \mid\ (t \in l_1 \text{ and } \forall t' \in l_2,\ \mathrm{id}(t) \neq \mathrm{id}(t')) \text{ or} \\ &\qquad\ (t \in l_2 \text{ and } \forall t' \in l_1,\ \mathrm{id}(t) \neq \mathrm{id}(t'))\} \end{aligned}$$

where each t and t' is a tuple and $id(t)$ is the bound variable associated with t.

Given the program

```
{ f x == x 1 2 + x 2 3;
  g b c d == b + c + d;
  result f (g 1);
  }
```

the l-values of both `(x 1 2)` and `(x 2 3)` in the body of `f` will be {[x 1 1 1]}. Since both these expressions occur in `(x 1 2 + x 2 3)`, the resulting l-value is:

$$merge(\{[\mathtt{x}\ 1\ 1\ 1]\}, \{[\mathtt{x}\ 1\ 1\ 1]\}) \Longrightarrow \{[\mathtt{x}\ 2\ 1\ 1]\}$$

The f-value

The f-value is the third element of a value in S and reflects the value's higher-order behavior. If, in the standard semantics, an expression e denotes a function, the f-value for e is a function over S. The precise definition of an f-value is presented in section 2.3.2. The next section describes how the f-value is used.

Function Applications in the Sharing Domain

Suppose the p-value of an expression e is $[id\ j]$. If the application of e to an argument x represents a partial application, then the p-value of the result of applying the f-value of e to the value of x must be $[id\ j+1]$. For example, if the value of e is $\langle [b\ 0], \{\}, f\rangle$ then the value in S of $(e\ x)$ would be $\langle [b1], \{\}, f'\rangle$ where f' is a function capturing the higher order behavior of $(e\ x)$. The function *add_p* is used to for addition on the count in a p-value.

$$\begin{aligned} add\_p(p,k) = \ & \text{if } (p = []) \text{ then } [] \\ & \text{else let } [id\ j] = p \\ & \qquad \text{in } [id\ j+k] \end{aligned}$$

In many cases, the count in a p-value is simply incremented and we define the function *incr* as follows:

$$incr(p) = add\_p(p,1)$$

Suppose e represents a partial application of a function g and needs only one argument to become a complete application. When e is applied, the body of g is evaluated. The environment in which the body of g is evaluated binds the formal parameters of g to values in S that have placeholder p-values but whose f-values are the same as the corresponding actual parameters. For example, in the program

```
{  h x y == x+y;
   f a == a 1 + b 2;
   g b == b (h 1);
   result g f;
   }
```

the variable `b` is bound to the function `f`. When `b` is applied in the expression `(b (h 1))`, the body of `f` is evaluated in an environment in which the variable `a` is bound to

$$\langle [\texttt{a } 0], \{\}, f' \rangle$$

where f' captures the higher order behavior of `(h 1)`.

When the body of f in the above program has been evaluated, any occurrence of the identifier `a` in the p-value and l-value of the result has to be replaced by the p-value of the argument to `f`. The function *backsub_p* takes the p-value of the result of executing the body of a function and replaces the identifier (bound variable name) with the identifier in the p-value of the corresponding actual parameter. It also adds the count in the result p-value to the count of the p-value of the actual parameter. The second argument to *backsub_p* is an environment, *arg_env*, mapping formal parameters to the values of the corresponding arguments.

$$\begin{aligned} backsub\_p(p, arg\_env) = \ & \text{let } [id\ n] = p \\ & \text{in } \text{if } arg\_env[\![id]\!] = \{\} \text{ then } p \\ & \quad \text{else } \text{let } \langle p', l', f' \rangle = arg\_env[\![id]\!] \\ & \quad\quad \text{in } add\_p(p', n) \end{aligned}$$

The function *backsub_l* takes the l-value of the result of executing the body of a function and replaces any occurrence of a formal parameter name in a tuple by the p-value of the actual argument. It does so by expanding the p-value of the argument into a form that is appropriate for an element of the l-value. This expansion is performed by the function *expand* as follows:

$$\begin{aligned} expand(p) = \ & \text{let } [id\ j] = p \\ & \text{in } [id\ \underbrace{1\ 1 \ldots 1}_{j+1}] \end{aligned}$$

where the last line of *expand* denotes a tuple consisting of id and $j + 1$ occurrences of the number 1. This expansion is appropriate because the p-value $[id\ j]$ representing a partial application of id to j arguments is equivalent to the l-value element $[id\ 1 \ldots 1]$ (containing $(j + 1)$ ones) indicating that id was applied to zero arguments once, to one argument once, and so on.

To replace an identifier in an l-value element, we use the function $repl$ which is defined as follows:

$$
\begin{aligned}
repl(l\_elt, p) = \ & \text{let} \ \ [id\ v_0 \ldots v_n] = l\_elt \\
& \qquad [id'\ v'_0 \ldots v'_j] = expand(p) \\
& \text{in} \begin{cases} l\_elt & \text{if } id \neq id' \\ [id\ v'_0 \ldots v'_{j-1}\ (v'_j \times v_0)\ v_1 \ldots v_m] & \text{otherwise} \end{cases}
\end{aligned}
$$

The function $backsub\_l$ replaces all bound variables in an l-value with the expanded versions of the p-values of the corresponding arguments as described above.

$$
\begin{aligned}
& backsub\_l(l, arg\_env) = \\
& \quad \text{let} \ \ \{t_1 \ldots t_m\} = l \\
& \qquad [id_i\ v_{i1} \ldots v_{in}] = t_i, \ \ \text{for each } t_i \in l \\
& \qquad s_i = \begin{cases} \{t_i\} & \text{if } arg\_env[\![id_i]\!] = \{\} \\ \{repl(t_i, p'_i)\} \cup l'_i & \text{if } arg\_env[\![id_i]\!] = \langle p'_i,\ l'_i, f'_i \rangle \end{cases} \\
& \quad \text{in} \ \bigcup s_i
\end{aligned}
$$

If a bound variable name occurs in the l-value of the result, then during the back-substitution

1. the bound variable name is replaced by the p-value of the corresponding argument, and
2. if the bound variable name occurs in the l-value of the result then value of corresponding argument was used. Therefore, all the sharing that occurred during the evaluation of the argument, as indicated by its l-value, should be included in the l-value of the result.

After back-substitution, the l-value of the result may include several tuples that contain the same identifier. This occurs when several formal parameters are bound to the same partial application. Thus, after back-substitution all tuples that contain the same identifier are merged. The function $combine$ accomplishes this:

$$
\begin{aligned}
combine(l) = \ & \text{if } (l = \{\}) \text{ then } \{\} \\
& \text{else} \ \ \text{let } \{t_1,\ t_2, \ldots t_n\} = l \\
& \qquad \text{in } merge(\{t_1\}, combine(t_2, \ldots, t_n))
\end{aligned}
$$

2.3.2 An Exact Sharing Semantics

The semantics which we are about to give specifies the exact sharing that occurs during an ALFL program's execution. For simplicity, though, we assume that all nested functions in an ALFL program have been lambda-lifted to the top level. These top-level functions are the ones whose sharing properties will be determined. The semantic domains and functions are:

$$\begin{array}{rcll} P & = & (V \times \mathcal{N}) + \{[]\} & \text{the domain of tuples} \\ L & = & \mathcal{P}(V \times \mathcal{N}^+) & \\ F & = & (S \times P) \rightarrow S & \\ S & = & (P \times L \times F) + \{error\} & \text{the sharing domain} \\ Env & = & V \rightarrow S & \\ E & : & Exp \rightarrow Env \rightarrow S & \text{the semantic function for expressions} \\ E_p & : & Prog \rightarrow S & \text{the semantic function for programs} \end{array}$$

where $\mathcal{N}$ is the set of natural numbers and and $\mathcal{P}$ is the power set notation. The semantic functions E and E_p are defined below.

A constant is not a partial application and does not contribute to the sharing of any other partial application. Thus

$$E[\![c]\!]env = \langle [], \{\}, err \rangle$$

where c is a constant and err is a function that returns an error if applied (for simplicity, we are not concerning ourselves with constant primitive functions).

The meaning of a variable is whatever it is bound to in the current environment.

$$E[\![x]\!]env = env[\![x]\!]$$

The result of a binary operation is never a partial application, although sharing of partial applications may have occurred during the evaluation of the operands. To distinguish between syntactic objects and semantic objects in the following equations, all variables representing elements of S (or sub-elements of a element of S) are written with a "^".

$$\begin{array}{rl} E[\![e_1 + e_2]\!]env = & \text{let } \langle \hat{p}_1, \hat{l}_1, \hat{f}_1 \rangle = E[\![e_1]\!]env \\ & \quad \langle \hat{p}_2, \hat{l}_2, \hat{f}_2 \rangle = E[\![e_2]\!]env \\ & \text{in } \langle [],\ merge(\hat{l}_1, \hat{l}_2),\ err \rangle \end{array}$$

In a well-typed program, no partial application can serve as an operand in a binary operation. Therefore, $\hat{p}_1$ and $\hat{p}_2$ above will be [].

The conditional is handled as follows:

$$\begin{aligned} E[\![e_1 \rightarrow e_2, e_3]\!]env = \ & \text{let } \langle \hat{p}_1, \hat{l}_1, \hat{f}_1 \rangle = E[\![e_1]\!]env \\ & \text{in if } (\text{Oracle}[\![e_1]\!] = True) \text{ then} \\ & \quad \text{let } \langle \hat{p}_2, \hat{l}_2, \hat{f}_2 \rangle = E[\![e_2]\!]env \\ & \quad \text{in } \langle \hat{p}_2,\ merge(\hat{l}_1, \hat{l}_2),\ \hat{f}_2 \rangle \\ & \text{else let } \langle \hat{p}_3, \hat{l}_3, \hat{f}_3 \rangle = E[\![e_3]\!]env \\ & \quad \text{in } \langle \hat{p}_3, merge(\hat{l}_1, \hat{l}_3),\ \hat{f}_3 \rangle \end{aligned}$$

To provide an exact semantics, conditionals must be resolved correctly. To do so, we defer to an oracle to determine the correct meaning of each conditional.[2] In the next section we provide an abstract sharing semantics that does not rely upon such an oracle, but provides less precise sharing information.

Function application is defined as follows:

$$\begin{aligned} E[\![e_1\ e_2]\!]env = \ & \text{let } \langle \hat{p}_1, \hat{l}_1, \hat{f}_1 \rangle = E[\![e_1]\!]env \\ & \quad \langle \hat{p}_3, \hat{l}_3, \hat{f}_3 \rangle = \hat{f}_1(E[\![e_2]\!]env, \hat{p}_1) \\ & \text{in } \langle \hat{p}_3, merge(\hat{l}_3, \hat{l}_1), \hat{f}_3 \rangle \end{aligned}$$

Since $\hat{f}_1$ is the function that captures the higher order behavior of e_1, $\hat{f}_1$ is applied to the value of e_2. The sharing information gained from evaluating e_1 is then merged with the result of the application.

The extra argument, $\hat{p}_1$, to $\hat{f}_1$ indicates which partial application $\hat{f}_1$ represents. This is required in order for the p-value of a partial application of a function to contain the name of the variable which the function happens to be bound to.

The meaning of a program is the value of the result expression in an envi-

[2]This oracle is actually a shorthand for carrying the standard semantics around and consulting it for the conditionals.

ronment in which all function names are bound to values in S.

$$
\begin{aligned}
&E_p[\![\{\ F_1\ x_{11} \ldots x_{1k_1} = e_1; \\
&\quad \ldots \\
&\quad F_n\ x_{n1} \ldots x_{nk_n} = e_n; \\
&\quad \texttt{result}\ e; \\
&\quad \}]\!] = E[\![e]\!]env'
\end{aligned}
$$

whererec

$$env' = [\hat{s}_1/F_1, \ldots, \hat{s}_n/F_n]$$

and for each i, $1 \leq i \leq n$,

$$
\begin{aligned}
\hat{s}_i = \ & \langle [F_i\ 0], \{\}, \hat{f}_i^1 \rangle \\
& \text{where } \hat{f}_i^1 = \lambda \hat{y}_1\ \hat{p}_1.\langle incr(\hat{p}_1),\ \{\}, \hat{f}_i^2 \rangle \\
& \qquad \text{where } \hat{f}_i^2 = \lambda \hat{y}_2\ \hat{p}_2.\langle incr(\hat{p}_2),\ \{\}, \hat{f}_i^3 \rangle \\
& \qquad\quad \vdots \\
& \qquad\quad \text{where } \hat{f}_i^{k_i} = \ \lambda \hat{y}_{k_i}\ \hat{p}_{k_i}. \\
& \qquad\qquad \text{let } \ arg\_env = [\hat{y}_1/x_{i1}, \ldots, \hat{y}_{k_i}/x_{ik_i}] \\
& \qquad\qquad\quad \text{For each } j,\ 1 \leq j \leq k_i, \\
& \qquad\qquad\quad\ \langle \hat{p}_j, \hat{l}_j, \hat{f}_j \rangle = \hat{y}_j \\
& \qquad\qquad\quad\ \hat{p}'_j = \begin{cases} [] & \text{if } \hat{p}_j = [] \\ [x_{ij}\ 0] & \text{otherwise} \end{cases} \\
& \qquad\qquad\quad\ \hat{y}'_j = \langle \hat{p}'_j, \{\}, \hat{f}_j \rangle \\
& \qquad\qquad\quad \langle \hat{p}', \hat{l}', \hat{f}' \rangle = \\
& \qquad\qquad\qquad\ E[\![e_i]\!]env'[\ \hat{y}'_1/x_{i1}, \ldots, \hat{y}'_{k_i}/x_{ik_i}] \\
& \qquad\qquad\quad \hat{p}'' = backsub\_p(\hat{p}', arg\_env) \\
& \qquad\qquad\quad \hat{l}'' = \ \{expand(\hat{p}_{k_i})\} \cup \\
& \qquad\qquad\qquad\qquad backsub\_l(\hat{l}', arg\_env) \\
& \qquad\qquad \text{in } \langle \hat{p}'', \text{combine}(\hat{l}''), \hat{f}' \rangle
\end{aligned}
$$

The utility functions *backsub_p*, *backsub_l*, *combine*, *expand* and *incr* were defined in section 2.3.1.

2.3.3 Abstract Interpretation of the Sharing Semantics

Since we are unable to resolve conditionals at compile time, we define an abstract sharing semantics such that the meaning of a program is a description of the maximum sharing that *could* occur.

We define abstract sharing domain S' whose elements are sets representing alternate possibilities for the sharing occurring in an expression. The abstract semantic domains and functions are:

P	$=$	$(V \times \mathcal{N}) + \{[]\}$	the domain of tuples
L	$=$	$\mathcal{P}(V \times \mathcal{N}^+)$	
F'	$=$	$(S' \times P) \rightarrow S'$	
S'	$=$	$\mathcal{P}(P \times L \times F') + \{error\}$	the abstract sharing domain
Env'	$=$	$V \rightarrow S'$	
E'	:	$Exp \rightarrow Env \rightarrow S'$	the abstract semantic function for expressions
E'_p	:	$Prog \rightarrow S'$	the abstract semantic function for programs

The value of a constant is a singleton set:

$$E[\![c]\!]env = \{\langle [], \{\}, err\rangle\}$$

As in the exact sharing semantics, the value of a variable is determined by the current environment.

$$E[\![x]\!]env = env[\![x]\!]$$

In a binary operation, each operand may represent many possible sharing situations. The result of the operation has to account for each possible combination by forming a "cartesian product" of the elements of the operations.

$$\begin{aligned} E'[\![e_1 + e_2]\!]env = \ & \text{let} \quad \{\langle \hat{p}_0, \hat{l}_0, \hat{f}_0\rangle, \ldots, \langle \hat{p}_n, \hat{l}_n, \hat{f}_n\rangle\} = E'[\![e_1]\!]env \\ & \qquad \{\langle \hat{p}'_0, \hat{l}'_0, \hat{f}'_0\rangle, \ldots, \langle \hat{p}'_m, \hat{l}'_m, \hat{f}'_m\rangle\} = E'[\![e_2]\!]env \\ & \text{in } \{\langle [], merge(\hat{l}_i, \hat{l}'_j), err >\rangle \mid 0 \leq i \leq n,\ 0 \leq j \leq m\} \end{aligned}$$

The conditional is handled in a similar manner:

$$
\begin{aligned}
&E'[\![e_1 \rightarrow e_2, e_3]\!]env = \\
&\quad \text{let} \quad \{\langle \hat{p}_0, \hat{l}_0, \hat{f}_0\rangle, \ldots, \langle \hat{p}_n, \hat{l}_n, \hat{f}_n\rangle\} = E'[\![e_1]\!]env \\
&\qquad \{\langle \hat{p}'_0, \hat{l}'_0, \hat{f}'_0\rangle, \ldots, \langle \hat{p}'_m, \hat{l}'_m, \hat{f}'_m\rangle\} = E'[\![e_2]\!]env \\
&\qquad \{\langle \hat{p}''_0, \hat{l}''_0, \hat{f}''_0\rangle, \ldots, \langle \hat{p}''_q, \hat{l}''_q, \hat{f}''_q\rangle\} = E'[\![e_3]\!]env \\
&\quad \text{in} \quad \{\langle \hat{p}'_j, merge(\hat{l}_i, \hat{l}'_j), \hat{f}'_j\rangle \mid 0 \leq i \leq n, 0 \leq j \leq m\} \cup \\
&\qquad \{\langle \hat{p}''_j, merge(\hat{l}_i, \hat{l}''_j), \hat{f}''_j\rangle \mid 0 \leq i \leq n, 0 \leq j \leq q\}
\end{aligned}
$$

An interesting aspect of the abstract semantic definition of function application is that applications cannot be implemented lazily. A function (each $\hat{f}_i$ below) must evaluate its argument.

$$
\begin{aligned}
E'[\![e_1\ e_2]\!]env = \ &\text{let } \{\langle \hat{p}_0, \hat{l}_0, \hat{f}_0\rangle, \ldots, \langle \hat{p}_n, \hat{l}_n, \hat{f}_n\rangle\} = E'[\![e_1]\!]env \\
&\qquad \{\langle \hat{p}'_0, \hat{l}'_0, \hat{f}'_0\rangle, \ldots, \langle \hat{p}'_m, \hat{l}'_m, \hat{f}'_m\rangle\} = E'[\![e_2]\!]env \\
&\qquad \{\langle \hat{p}''_{ij0}, \hat{l}''_{ij0}, \hat{f}''_{ij0}\rangle, \ldots, \langle \hat{p}''_{ijq}, \hat{l}''_{ijq}, \hat{f}''_{ijq}\rangle\} = \\
&\qquad\quad \hat{f}_i(\{\langle \hat{p}'_j, \hat{l}'_j, \hat{f}'_{ij}\rangle\},\ \hat{p}_i), \text{ for each } i \leq n,\ j \leq m \\
&\quad \text{in } \{\langle \hat{p}''_{ijk}, merge(\hat{l}''_{ijk}, \hat{l}_i), \hat{f}''_{ijk}\rangle \mid i \leq n,\ j \leq m,\ k \leq q\}
\end{aligned}
$$

The definition of E'_p simply extends the definition of E_p to handle values that are sets. Notice that in the definition of E' for function application, a function is passed a *singleton* set as its first argument. This is reflected in the definition

of each $\hat{f}_i^j$ below.

$$
\begin{array}{l}
E_p[\![\{\ F_1\ x_{11} \ldots x_{1k_1} == e_1; \\
\quad \ldots \\
\quad F_n\ x_{n1} \ldots x_{nk_n} == e_n; \\
\quad \texttt{result}\ e; \\
\quad \}]\!] = E[\![e]\!]env'
\end{array}
$$

whererec

$$env' = [\hat{s}_1/F_1, \ldots, \hat{s}_n/F_n]$$

For each i, $1 \leq i \leq n$,

$$
\begin{array}{l}
\hat{s}_i = \{\langle [F_i\ 0], \{\}, \hat{f}_i^1 \rangle\} \\
\quad \text{where } \hat{f}_i^1 = \lambda \hat{y}_1\ \hat{p}_1.\{\langle incr(\hat{p}_1),\ \{\}, \hat{f}_i^2 \rangle\} \\
\quad\quad \text{where } \hat{f}_i^2 = \lambda \hat{y}_2\ \hat{p}_2.\{\langle incr(\hat{p}_2),\ \{\}, \hat{f}_i^3 \rangle\} \\
\quad\quad\quad \vdots \\
\quad\quad\quad \text{where } \hat{f}_i^{k_i} = \lambda \hat{y}_{k_i}\ \hat{p}_{k_i}. \\
\quad\quad\quad\quad \text{let } arg\_env = [\hat{y}_1/x_{i1}, \ldots, \hat{y}_{k_i}/x_{ik_i}] \\
\quad\quad\quad\quad\quad \text{For each } j,\ 1 \leq j \leq k_i, \\
\quad\quad\quad\quad\quad\quad \{\langle \hat{p}_j, \hat{l}_j, \hat{f}_j \rangle\} = \hat{y}_j \\
\quad\quad\quad\quad\quad\quad \hat{p}'_j = \begin{cases} [] & \text{if } \hat{p}_j = [] \\ [x_{ij}\ 0] & \text{otherwise} \end{cases} \\
\quad\quad\quad\quad\quad\quad \hat{y}'_j = \{\langle \hat{p}'_j, \{\}, \hat{f}_j \rangle\} \\
\quad\quad\quad\quad\quad \{\langle \hat{p}'_0, \hat{l}'_0, \hat{f}'_0 \rangle, \ldots, \langle \hat{p}'_m, \hat{l}'_m, \hat{f}'_m \rangle\} = \\
\quad\quad\quad\quad\quad\quad E'[\![e_i]\!]env'[\ \hat{y}'_1/x_{i1}, \ldots, \hat{y}'_{k_i}/x_{ik_i}] \\
\quad\quad\quad\quad\quad \text{For each } j \leq m, \\
\quad\quad\quad\quad\quad\quad \hat{p}''_j = backsub\_p(\hat{p}_j, arg\_env) \\
\quad\quad\quad\quad\quad\quad \hat{l}''_j = \{expand(\hat{p}_{k_i})\} \cup \\
\quad\quad\quad\quad\quad\quad\quad\quad backsub\_l(\hat{l}'_j, arg\_env) \\
\quad\quad\quad\quad \text{in } \{\langle \hat{p}''_j, \text{combine}(\hat{l}''_j), \hat{f}'_j \rangle, \mid 0 \leq j \leq m\}
\end{array}
$$

2.3.4 Termination

In order for our analysis to be useful, we need to guarantee that the value in S' for any program can be determined in a finite amount of time. In particular, each value in S' is set of tuples and each tuple contains a function in the subdomain

$$F' = (S' \times P) \rightarrow S'$$

Each such function may be recursive and we must show that the fixed point (of its corresponding functional) can be found in a finite amount of time. This is the case if the following conditions are satisfied:

1. The subdomain P of p-values is finite: This is clearly the case since a p-value is a pair containing a variable name (from the finite set V) and a natural number less than some finite value. Since a p-value pair represents a partial application, the size of the natural number is limited by the largest number of formal parameters that can occur in a function definition.

   In section 2.3.1 we mentioned that it is undesirable to create a new identifier for each partial application created during execution of the program. If new identifiers were created in such a fashion it would be very difficult (if not impossible) to insure that there were a finite number of elements in the sharing domain.

2. The subdomain L of l-values is finite: Each l-value is a set of tuples that describes the number of occurrences of partial applications. We can make the set of possible tuples finite by setting a limit on the maximum number of occurrences of a partial application that the analysis can detect. In most cases, we simply want to know if a partial application of some function occurred more than once. If the number of occurrences of a partial application reaches the maximum value then no further occurrences of that partial application will be counted.

3. The domain $F' = (S' \times P) \rightarrow S'$ contains a finite number of functions: The number of functions of a given arity (the number of arguments that a function can be applied to before the result is no longer a function) over

a finite domain is finite. Thus, we can ensure that F' is finite by requiring that the arity of each function in F' be finite. This is a reasonable restriction and is often enforced by a type inferencing system.

In addition, we can define an ordering of the elements of S' and prove monotonicity properties about the functions in F'. The ordering of S', based on *powerdomain* ordering, is a subject of ongoing research.

2.4 An Application: Efficient Full Laziness

Unlike supercombinators generated by lambda-lifting, the supercombinators defined by Hughes exhibit the property of being *fully lazy* [40]. Loosely speaking, we say that a function is fully lazy if shared uses of any of its partial applications do not result in the evaluation of the same subexpression more than once. (Examples of this will be given shortly.)

Unfortunately, the algorithm for generating supercombinators turns out to be excessively conservative in preserving the property of full laziness. As a result, the supercombinators are very often much more fine-grained than they need to be, resulting (as with a fixed set of combinators) in more reductions and greater consumption of space.

In this section we discuss a refinement of supercombinators that overcomes this conservatism, resulting in larger and more efficient combinators called *refined supercombinators*. We present an effective algorithm for translating a set of lambda expression definitions into refined supercombinators. The algorithm uses the sharing analysis of the previous section.

2.4.1 Supercombinators

Consider the function f, expressed in the syntax of the lambda calculus, defined by:

$$f = \lambda a.\lambda b.\lambda c. * (+ a^2\, b)\, c$$

From f we can define the combinator α:

$$\alpha\, a\, b\, c = * (+ a^2\, b)\, c.$$

and α can be used in place of f. Now suppose the following expression is evaluated:

$$(\lambda g. (* (g\, 5)\, (g\, 6)))\, (f\, 3\, 4)$$

Since g occurs twice, $(f\, 3\, 4)$ is shared. Yet because of our choice of the combinator α, both $(\alpha\, 3\, 4\, 5)$ and $(\alpha\, 3\, 4\, 6)$ will be evaluated independently. As a result, $(+ 3^2\, 4)$ will be computed twice. Hence the combinator α does not have the property of being fully lazy, and results in more computation than necessary.[3]

[3]This combinator definition is essentially what would result from *lambda lifting* [44].

To improve this situation, one might generate *supercombinators* from f, in which full laziness is (conservatively) guaranteed by generating one combinator for every bound variable. To see how this works, we first define a *free expression* with respect to a particular bound variable v as an expression in which there are no free occurrences of v. A *maximally free expression* (mfe) with respect to v is a free expression which is not contained within any larger free expression (with respect to v). When the context is clear, we omit naming the bound variable with respect to which an expression is maximally free.

The algorithm for generating supercombinators begins with the innermost lambda expression and works out, abstracting at each level all mfe's with respect to the bound variable at that level. For example, for the definition of f above, we see that the mfe of the innermost lambda expression is $(+\, a^2\, b)$. This expression is abstracted to form the supercombinator:

$$\alpha\, x\, c = *\, x\, c$$

and thus $f = \lambda a.\lambda b.\, \alpha\, (+\, a^2\, b)$. Next we note that a^2 is the mfe of the new innermost lambda expression, so it is abstracted, forming the combinator:

$$\beta\, y\, b = \alpha\, (y + b)$$

and thus $f = \lambda a.\, \beta\, (a^2)$. Since there is no (non-trivial) expression in f that is free with respect to a, the next (and final) supercombinator is:

$$\gamma\, a = \beta\, (a^2)$$

and all occurrences of f in the program are replaced by γ.

The shared expression mentioned earlier, $(f\ 3\ 4)$, will reduce as follows:

$$\begin{aligned} (f\ 3\ 4) &\Rightarrow (\gamma\ 3\ 4) \\ &\Rightarrow (\beta\ 9\ 4) \\ &\Rightarrow (\alpha\ 13) \end{aligned}$$

and therefore $(+\ 3^2\ 4)$ is only computed once — thus achieving fully lazy evaluation.

Even if a function definition contains no explicit nesting of lambda expressions it can still be transformed into a set of supercombinators. This is possible

because a definition of the form,

$$f\ x_1\ x_2 \ldots\ x_n = e$$

can be transformed into

$$f = \lambda x_1.\ \lambda x_2.\ \ldots x_n.e$$

and the supercombinator algorithm can then be applied.

A formal algorithm for generating supercombinators from a program P is:

1. Find the leftmost, innermost lambda expression L, of the form $\lambda v.exp$.
2. Find the maximally free expressions, $e_1 \ldots e_n$, of exp with respect to v.
3. Create a new combinator (say α) defined by:

$$\alpha\ i_1 \ldots i_n\ v = exp[i_1/e_1, \ldots, i_2/e_2]$$

   where formal parameters $i_1 \ldots i_n$ do not occur free in exp.

4. Substitute $(\alpha\ e_1 \ldots e_n)$ for L in P.
5. Repeat steps 1-4 until step 1 fails.

There are a few obvious optimizations to this algorithm, such as eliminating redundant combinators, as in: $\alpha\ a\ b = \beta\ a\ b$.

2.4.2 Refined supercombinators

Although preserving full laziness is a worthy goal, the supercombinator approach is too conservative. To see this, note in the previous example that the original single-combinator definition for f would be perfectly satisfactory if no partial application of f were ever shared, for then there would be no partial result that might be computed more than once. And because one combinator is used instead of three, the single-combinator solution would be more efficient in time and space. Using the sharing analysis from section 2.3 we can determine whether or not a partial application of f is shared, and can choose either a one-, two-, or three-combinator implementation for it, as appropriate.

Refined supercombinators are obtained by adapting Hughes's algorithm to use sharing information. These combinators are more efficient than supercombinators for most user-defined functions. Despite the elegance of higher-order

functions created through partial application, they are seldom used and more seldom shared.

For purposes of our presentation, we modify the notation of a function definition to incorporate sharing information. We write "$g\ \underline{w\ x}\ \underline{y\ z}\ = \ldots$" to indicate, for example, that g applied to two arguments is shared, but not to one or three arguments. In other words, the breaks in the underline indicate which partial applications are shared. We then generalize, in the obvious way, the notion of a maximally free expression with respect to a single variable, to that of a maximally free expression with respect to a set of variables.[4] We define $MFE(exp, S)$ be the set of mfe's of exp with respect to the set of identifiers S.

When generating refined supercombinators for a function definition such as

$$g\ \underline{w\ x}\ \underline{y\ z} = \ldots$$

we treat each "group" of formal parameters as a single unit by abstracting mfe's with respect to each group, working innermost out as before. Given the previous discussion, the rationale for doing this should be obvious – we cannot make the groups any larger, for that might violate full laziness, nor is there any reason to make them any smaller, since no finer partial application is shared.

Here is the algorithm for generating refined supercombinators:

1. Let $f_1 \ldots f_n$ be the names of the defined functions in the program.
2. Partition the formal parameters of each definition f_i so as to reflect the sharing of partial applications f_i.
3. For each definition f_i, repeat this step until there is just one partition of bound variables remaining. Let $\underline{x_{il} \ldots x_{in}}$ be the right-most partition; i.e.,

$$f_i\ \underline{x_{i1} \ldots \ldots \ldots x_{i(k-1)}}\ \underline{x_{ik} \ldots x_{in}}\ =\ e_i$$

Define a new combinator (say α) by:

$$\alpha\ v_1 \ldots v_p\ x_{ik} \ldots x_{in} = exp_i[v_1/e_1, \ldots, v_p/e_p]$$

where $v_1 \ldots v_p$ are new variable names not occurring free in exp_i, and:

$$\{e_1, \ldots, e_p\} = MFE(exp_i, \{x_{ik} \ldots x_{in}\})$$

[4] In this case, the variables underlined together are in the same set.

Then replace the previous definition of f_i by:

$$f_i \underbrace{x_{i1} \ldots \ldots x_{i(k-1)}} = \alpha \; exp_1 \ldots exp_p$$

2.5 An Example

Consider the following (somewhat contrived) ALFL program:

```
{ f a b c d == ((a*a)+b) * (a-b) + (2 * c) + d;
  g x == h (x 1) 2 + x 2 3;
  h y z == y z;
  result g (f 1 2);
  }
```

Applying sharing analysis, we obtain

$$\{\langle(),\ \{[\mathrm{g}\ 1\ 1]\ [\mathrm{h}\ 1\ 1\ 1]\ [\mathrm{f}\ 1\ 1\ 2\ 1\ 1]\},\ err\rangle\}$$

which indicates that the only shared partial application in the program is that of `f` to two arguments.

Converting the program into lambda-expression form and applying Hughes's supercombinator algorithm to the definition of `f` proceeds as follows:

1. Initially,

   $$f = \lambda a\ b\ c\ d.\ +\ (+\ (*\ (+\ (*\ a\ a)\ b)\ (-\ a\ b))\ (*\ 2\ c))\ d$$

   and the first combinator generated is:

   $$\alpha\ i\ d = (+\ i\ d)$$

2. Thus f can be redefined,

   $$f = \lambda a\ b\ c.\ \alpha\ (+\ (*\ (+\ (*\ a\ a)\ b)\ (-\ a\ b))\ (*\ 2\ c))$$

   and the next combinator is:

   $$\beta\ j\ c = \alpha\ (+\ j\ (*\ 2\ c))$$

3. Now,

   $$f = \lambda a\ b\ .\ \beta\ (*\ (+\ (*\ a\ a)\ b)\ (-\ a\ b))$$

   and

   $$\gamma\ k\ a\ b = \beta\ (*\ (+\ k\ b)(-\ a\ b))$$

4. Finally,

$$f = \lambda a.\ \gamma\ (*\ a\ a)\ b$$

and

$$\delta\ a = \gamma\ (*\ a\ a)$$

which means that $f = \delta$ and can be replaced by δ in the above program. The same transformation is applied to **g** and **h**, resulting in three more combinators.

However, using the above sharing information, the generation of refined supercombinators for f proceeds as follows:

1. Initially, the arguments are grouped according to how f is shared:

$$f\ \underline{a\ b}\ \underline{c\ d} = +\ (+\ (*\ (+\ (*\ a\ a)\ b)\ (-\ a\ b))\ (*\ 2\ c))\ d$$

and thus the first refined supercombinator generated is:

$$\eta\ m\ c\ d = +\ (+\ m\ (*\ 2\ c))\ d$$

2. Now,

$$f\ \underline{a\ b} = \eta\ (*\ (+\ (*\ a\ a)\ b)(-\ a\ b))$$

and since there is no more sharing the second (and last) combinator generated is:

$$\mu\ a\ b = \eta\ (*\ (+\ (*\ a\ a)\ b)(-\ a\ b))$$

Therefore $f = \mu$ and can be replaced by μ in the program.

Only one refined supercombinator will be generated for each of **g** and **h** in the program.

The compiler that we have implemented performs sharing analysis on a lambda-lifted program and generates a set of refined supercombinators using the algorithm described above. Naturally, this phase of the compiler can be used to make sequential implementations of functional languages more efficient. The rest of the chapters discuss issues of parallel execution of functional programs.

Chapter 3

A Characterization of Parallelism and Granularity in Functional Programs

In this chapter we develop a theoretical basis for the algorithms used by the compiler to partition a functional program for multiprocessor execution. We present a characterization of the potential costs and benefits of parallel execution of the expressions in a program.

3.1 Parallelism in Functional Languages

Automatic partitioning of functional programs is tractable because functional languages exhibit the Church-Rosser property. Operationally speaking, the Church-Rosser property (actually a corollary to the Church-Rosser Theorem) states all orders of evaluation of an expression that terminate will give the same result. The arguments in a function application, for example, can be evaluated in any order we desire, including in parallel.

Since we are executing programs written in non-strict functional languages, we need to preserve the termination properties of normal order evaluation. In order to do so and still exploit parallelism, we use *strictness analysis* to determine which arguments in a function application will always be needed.

Parallelism is exhibited in two ways in functional programs. The first is

what we call *horizontal parallelism*, which results from evaluating the strict arguments in a function application in parallel. If only horizontal parallelism is exploited then the values of all strict arguments must be available before the body of the function is executed. Horizontal parallelism is exhibited in strict languages (such as Lisp) that utilize a *call-by-value* evaluation order. In lazy languages, however, non-strict arguments are not evaluated before the function call. In the program,

```
{ f x y z == x=y -> z, 1;
  result f (6+7) (8*9) (10+11);
  }
```

the result expression could be executed by evaluating `(6+7)` and `(8*9)` in parallel (but not `(10+11)`). The body of `f` would not be executed until the two strict arguments had been evaluated.

A function may be able to do a significant amount of work before needing the value of a particular argument. It may therefore be beneficial to start executing the body of a function before the values of its strict arguments have returned. If an argument that is still being evaluated is referenced, the function must block until the needed value arrives. This type of parallelism, which we call *vertical parallelism*, results from executing the body of a function and its arguments in parallel. For example, in the program,

```
{ f x y == g x -> y+1, y-1;
   g x == ...
   h y == ...
   result f (6+7) (h 5);
  }
```

where `g` and `h` are very complex functions, there is no reason for `f` to wait for the value of `(h 5)` before executing `(g x)`. Vertical parallelism results from executing `(g x)` and `(h 5)` simultaneously.

In the next section, we analyze the various costs and benefits of utilizing horizontal and vertical parallelism.

3.2 Communication Costs and Granularity

In any multiprocessor system, there is an overhead cost involved in creating parallel computations, or *tasks*. In many architectures this cost is significant and, depending on the particular tasks being executed, may outweigh any benefit gained from exploiting parallelism.

In chapters 6 and 8 we discuss the overhead costs associated with two particular multiprocessor architectures. All parallel architectures incur overhead for the following actions:

1. *Creation of parallel tasks.* This involves creating a new process state which contains the information needed to execute a task. This action may be as inexpensive as creating a simple activation record or as expensive as invoking an operating system call to create a heavyweight process.

   When a task is created, some communication must occur between the processor that initiated the creation and the processor that will execute the new task. This communication may be as inexpensive as accessing a shared queue of tasks or as expensive as sending a message over a network.

   In the multiprocessor implementation described in chapter 6 the communication cost involved in task creation is significantly greater than the cost of creating a structure to contain the process state information. In the following discussion the term *communication cost* includes all of the costs involved in creating a task.

2. *Communication between parallel tasks.* During execution, interprocessor communication may occur between tasks in order to provide synchronization or share data. This may either involve accessing a shared variable or sending a message. Whether the communication involves enforcing mutual exclusion on areas of shared memory or invoking message passing routines, a non-trivial cost is incurred.

3. *Cleaning up completed tasks.* After a task has finished executing, the process state of the task must be removed and, if necessary, a value returned to the task that invoked it.

The costs listed above lead to the notion of the *granularity* of a parallel com-

putation. Granularity is a measure of how much computation occurs on each processor between periods of communication and is an indication of how often the execution of a task will incur communication overhead. If the grain size is large, resulting in a *coarse-grained* computation, a large amount of computation is done between periods of communication. Thus communication costs are incurred relatively infrequently. If the grain size is small, resulting in a *fine grained* computation, then the communication overhead will be incurred often during the computation.

We will be partitioning functional programs into expressions that will be evaluated by individual tasks. Because functional programs contain no side-effect operators, two tasks executing in parallel cannot have an effect upon each other. Therefore, they will not have to communicate with each other while they are both executing; communication is needed only when a task is to be created or has finished executing and needs to return a value. The total overhead due to communication is therefore directly proportional to the number of tasks created during execution and the granularity of the computation is directly related to the execution time of each task.

Although it is clearly desirable to keep communication overhead low, there is a tradeoff between coarse- and fine-grained parallelism. If a large grain is desired, then a program will be partitioned into fewer tasks than if a small grain size is used. Each of these large grained tasks will execute longer, but the potential for exploiting parallelism will be reduced because each task may perform work serially that could be performed in parallel. On the other hand, if the granularity of a computation is too fine, any benefit gained from creating a large number of tasks may be outweighed by the communication costs involved. We use the term *useful parallelism* to describe the parallelism in a program that, if exploited, results in a reduction of the overall execution time. If every partitioning of an expression increases its execution time, then the expression contains no useful parallelism.

On a multiprocessor with relatively low communication costs, fine grained tasks may exploit the parallelism in a computation without being heavily penalized for the large amount of communication needed. The higher the communications cost of a multiprocessor, the more likely it is that a coarse-grained

computation will perform better. Thus, the optimal grain size varies with the communication costs of the host machine.

In the following discussion, we use the term *task* loosely to represent either one of the expressions into which a program is partitioned or the *evaluation* of such an expression. Although technically a task is the latter, where the meaning is unambiguous we will use it in either way.

3.3 Program granularity in dynamic load balancing systems

There are two methods for scheduling the execution of tasks in a multiprocessor. The first, *static scheduling*, is performed before the computation begins. Either the programmer or the compiler partitions a program into tasks and specifies exactly which processor will execute each task. The number of processors generally determines the number of tasks into which the program is partitioned.

The second scheduling method is *dynamic scheduling*, also known as *dynamic load balancing*, in which the allocation of tasks to specific processors is performed during execution. Generally, the decision of where to allocate a newly created task is based on the current state of the system. If the system is heavily loaded, then several tasks that could be executed in parallel may be executed on the same processor. Therefore, the creation and completion of tasks may or may not incur the overhead of interprocessor communication.

In chapters 7 and 8 we describe a number of dynamic scheduling algorithms. In this chapter, we discuss the compile-time partitioning of programs into tasks that will be allocated to specific processors using dynamic scheduling. The goal of compile-time partitioning is to ensure that the granularity of the computation is never fine enough to cause a degradation of performance if each task were allocated to a different processor. That is, the partitioning of the program should be optimized for the case where an idle processor is available to execute each task. The dynamic scheduler may choose to allocate several tasks onto a single processor. However, if it chooses to execute several tasks in parallel,

there must be a possibility that performance is improved over executing them sequentially.[1]

3.4 Finding the Appropriate Granularity

As mentioned above, our goal is to find the appropriate granularity into which a program should be partitioned assuming that each task will be executed on a different processor. In this section we outline an exact (but infeasible) analysis that will guide us in choosing the heuristics that will be used by the compiler.

We have seen that any application of a function to one or more strict arguments has the potential for exhibiting parallelism. If communication overhead were zero, we would exploit the full parallelism in a program by executing all strict arguments, along with the body of the function, in parallel. Thus even for a simple expression like

```
(4*5)+(6*7)
```

two tasks would be created to evaluate `(4*5)` and `(6*7)` in parallel. Obviously, tasks that accomplish the parallel evaluation of such a simple expression constitute a very fine grained computation.

In the previous example, we assumed that communication overhead was zero. Since there is no multiprocessor architecture that provides zero-cost communication, it is unlikely that such a fine granularity would prove worthwhile. How then do we partition a functional program into tasks of the appropriate granularity? If we are given an expression e, how can we determine whether or not the evaluation of e should be decomposed into a number of tasks?

Since every expression in a functional program is an application of a function (or primitive operator) to zero or more arguments, the expression e that we are interested in is of the form

$$(f\ e_1 \ldots e_n)$$

where f is a function (or primitive operator) that is applied to arguments e_1

[1] There is an alternative method called *run-time partitioning*, in which a program is decomposed into tasks at run-time. Because there is a high overhead associated with this method and because our goal is to develop compile-time methods for program decomposition, we will not discuss run-time partitioning further in this dissertation.

through e_n. In the above example, the expressions corresponding to f, e_1, and e_2 would be `+`, `(4*5)`, and `(6*7)`, respectively.

In the analysis, the communication overhead is broken down into two parts:

- The time spent by the local processor spawning a task onto a remote processor: We consider this cost, denoted C_{loc}, to be a constant whose value depends only upon the implementation. Even though the cost of creating a process state may vary from task to task, the cost incurred locally of sending it to another processor is generally far greater (at least on loosely-coupled multiprocessors) and is generally constant. For simplicity, we assume that C_{loc} is incurred before the task is actually sent to the remote processor.[2]

- The sum of the communication delays (latencies), denoted C_{lat}, that occur when a task is spawned and when its value is returned: C_{lat} is twice the delay that occurs between the time a message is sent and is actually received. Again, the analysis considers this to be a constant.[3]

Suppose we are given an expression e of the form $(f\ e_1 \ldots e_n)$, where f is strict in all of its arguments. If the lowest possible cost of evaluating each e_i (whether sequentially or in parallel) is known, our analysis should indicate whether or not it is worth decomposing e into a number of tasks. If the answer is negative then the evaluation of e should proceed as the evaluation of $e_1 \ldots e_n$ in sequence followed by the instantiation of the function application. The evaluation of each e_i proceeds in the manner providing the lowest cost for each e_i. To be more precise, we define the following terms:

- $T(e)$ is the minimum cost of executing an expression e either sequentially or in parallel (either $T_p(e)$ or $T_s(e)$, below).

- $T_p(e)$ is the cost of evaluating e in parallel. This cost is, of course, dependent on the way e is partitioned.

- $T_s(e)$ is the cost of executing the components of e, namely $e_1 \ldots e_n$, in

[2]For simplicity, we have ignored the cost of receiving a value returned by a completed task. This cost can be incorporated into C_{loc} without significantly altering the accuracy of our model.

[3]This is a reasonable assumption of tasks are always spawned on neighboring processors. The task scheduling methods described in chapter 7 exhibit this property

sequence plus the cost of evaluating the body of f. The cost of executing each e_i is $T(e_i)$. Therefore,

$$T_s(f\ e_1 \ldots e_n) = (\sum_{i=1}^{n} T(e_i)) + T(f[e_1, \ldots, e_n])$$

where $f[e_1, \ldots, e_n]$ represents the application of f to the arguments after all e_i have been evaluated.

In the preceding discussion f is assumed to be strict in all of its arguments. This restriction will be relaxed.

The analysis proceeds bottom-up: Given an expression e, the analysis is recursively applied to each component e_i of e in order to determine the best way to evaluate e_i. Then, based on the value of $T(e_i)$ for each e_i, the analysis determines if it is worthwhile to partition e into parallel tasks.

3.4.1 Analysis of Horizontal Parallelism

In this section, we concentrate on the costs and benefits of exploiting *horizontal parallelism.* Eventually, we extend the analysis to allow for vertical parallelism.

Horizontal parallelism in binary function calls

Suppose we are given an expression of the form $(f\ e_1\ e_2)$, where f is strict in both arguments.[4] If $(f\ e_1\ e_2)$ is evaluated sequentially then

$$T_s(f\ e_1\ e_2) = T(e_1) + T(e_2) + T(f[e_1, e_2]) \tag{3.1}$$

The simplest way to exploit the horizontal parallelism in $(f\ e_1\ e_2)$ would be for the local processor to create tasks on two remote processors to evaluate e_1 and e_2. The execution time would be

$$\begin{aligned} T_p(f\ e_1\ e_2) \;=\; & \max(C_{loc} + T(e_1) + C_{lat},\ 2C_{loc} + T(e_2) + C_{lat}) + \\ & T(f[e_1, e_2]) \end{aligned} \tag{3.2}$$

[4]Note that this subsumes the case of an expression of the form $(e_1\ e_2)$ in which case f is simply the strict version of the `apply` function.

if the task evaluating e_1 were created first, or

$$T_p(f\ e_1\ e_2) = \max(C_{loc} + T(e_2) + C_{lat},\ 2C_{loc} + T(e_1) + C_{lat}) + T(f[e_1, e_2]) \tag{3.3}$$

if the task evaluating e_2 were created first. Suppose that $T(e_1) \geq T(e_2)$. Thus

$$\begin{aligned}\max(C_{loc} + T(e_1) + C_{lat},\ 2C_{loc} + T(e_2) + C_{lat}) \\ &\leq 2C_{loc} + T(e_1) + C_{lat} \\ &= \max(2C_{loc} + T(e_1) + C_{lat},\ C_{loc} + T(e_2) + C_{lat})\end{aligned}$$

and equation 3.2 results in a lower value for $T_p(f\ e_1\ e_2)$ than equation 3.3. This indicates that it is better to start the remote evaluation of the *more complex* argument first.

Partitioning of $(f\ e_1\ e_2)$ would be worthwhile only if

$$T_p(f\ e_1\ e_2) \leq T_s(f\ e_1\ e_2) \tag{3.4}$$

Again assume that $T(e_1) \geq T(e_2)$. Substituting the definitions of T_s and T_p (from equations 3.1 and 3.2, respectively) into the inequality 3.4 we see that there are two possible cases:

1. If $T(e_1) \geq C_{loc} + T(e_2)$ then the value of e_2 arrives before the value of e_1 and inequality 3.4 is satisfied iff $C_{loc} + C_{lat} \leq T(e_2)$.
2. Otherwise, if $C_{loc} + T(e_2) \geq T(e_1)$ then the value of e_1 arrives first and inequality 3.4 is satisfied iff $2C_{loc} + C_{lat} \leq T(e_1)$.

This analysis is based on the assumption that the tasks for both e_1 and e_2 are sent to remote processors to be executed. A better way to evaluate e would be to send only one argument to a remote processor and execute the other locally. If e_1 is evaluated locally then the parallel execution time, denoted T_p^1, is described by

$$T_p^1(f\ e_1\ e_2) = \max(C_{loc} + T(e_1),\ C_{loc} + T(e_2) + C_{lat}) + T(f[e_1, e_2]) \tag{3.5}$$

If e_2 is evaluated locally then

$$T_p^2(f\ e_1\ e_2) = \max(C_{loc} + T(e_1) + C_{lat},\ C_{loc} + T(e_2)) + T(f[e_1, e_2]) \tag{3.6}$$

The communication costs, C_{loc} and C_{lat} are incurred only once because only one argument is evaluated on a remote processor.

To minimize the parallel execution time of $(f\ e_1\ e_2)$ the argument with the *shorter* execution time must be evaluated remotely. If $T(e_1) \leq T(e_2)$, then equation 3.6 minimizes the parallel execution time because

$$\max(T(e_1) + C_{lat},\ T(e_2)) \leq \max(T(e_1),\ T(e_2) + C_{lat})$$

Parallel evaluation is worthwhile only if

$$\max(C_{loc} + T(e_1) + C_{lat},\ C_{loc} + T(e_2)) \leq T(e_1) + T(e_2)$$

which holds if either

$$T(e_2) \leq T(e_1) + C_{lat} \text{ and } C_{loc} + C_{lat} \leq T(e_2) \tag{3.7}$$

or

$$T(e_1) + C_{lat} \leq T(e_2) \text{ and } C_{loc} \leq T(e_1) \tag{3.8}$$

Although strict binary function calls exhibiting only horizontal parallelism may seem too simple to warrant examining, they occur very frequently. The primitive binary operators such as + and * behave in precisely this fashion.

Function calls with multiple arguments

Suppose we are given an expression of the form $(f\ e_1 \ldots e_n)$ where f is only strict with respect to some of its arguments. For simplicity, assume that f is strict in $e_1 \ldots e_k$ and not in $e_{k+1} \ldots e_n$. In this case,

$$T_s(f\ e_1 \ldots e_n) = (\sum_{i=1}^{k} T(e_i)) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \tag{3.9}$$

where $f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]$ is the computation needed to evaluate $(f e_1 \ldots e_n)$ after $e_1 \ldots e_k$ have been evaluated (but not $e_{k+1} \ldots e_n$).

Since we provide no mechanism for supporting *eager evaluation*—the premature evaluation of non-strict arguments—only the strict arguments can be executed immediately. If all of the strict arguments were sent (in left to right order) to remote processors to be evaluated then

$$T_p(f\ e_1 \ldots e_n) = \max(t_1,\ t_2, \ldots,\ t_k) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \tag{3.10}$$

where, for $1 \leq i \leq k$,

$$t_i = iC_{loc} + T(e_i) + C_{lat} \tag{3.11}$$

Equations 3.10 and 3.11 take into account that the local processor incurs C_{loc} for each task sent to a remote processor. The task is spawned at time C_{loc}, the second expression at time $2C_{loc}$, and so on.

Instead of spawning the arguments in left to right order, the parallel execution time is minimized by spawning the arguments in *decreasing order* of their complexity. That is, for strict arguments e_i and e_j, if $T(e_i) \geq T(e_j)$ then e_i should be spawned before e_j. Suppose e_i is spawned after e_j. If e_j is the pth expression spawned and e_i is the $(p+q)$th expression spawned then the total execution time is described by

$$T_p(f\ e_1 \ldots e_n) = \max(\ldots, t_p, \ldots, t_{p+q}, \ldots) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \tag{3.12}$$

where

$$t_p = pC_{loc} + T(e_j) + C_{lat}$$
$$t_{p+q} = (p+q)C_{loc} + T(e_i) + C_{lat}$$

If e_i and e_j exchange their positions in the order, then

$$T'_p(f\ e_1\ e_n) = \max(\ldots, t'_p, \ldots, t'_{p+q}, \ldots) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \tag{3.13}$$

where

$$t'_p = pC_{loc} + T(e_i) + C_{lat}$$
$$t'_{p+q} = (p+q)C_{loc} + T(e_j) + C_{lat}$$

and all other terms in equation 3.13 are the same as in equation 3.12. We can see that $t'_{p+q} \leq t_{p+q}$ and $t'_p \leq t'_p$, and therefore

$$T'_p(f\ e_1 \ldots e_n) \leq T_p(f\ e_1 \ldots e_n)$$

Thus execution time is minimized when the arguments are spawned in decreasing order of complexity.

Assume the arguments are spawned off in decreasing order of their complexities and, for simplicity, that $T(e_i) \geq T(e_{i+1})$ for all $i < k$. What must be true in order for

$$T_p(f\ e_1 \ldots e_n) \leq T_s(f\ e_1 \ldots e_n) \tag{3.14}$$

to be satisfied? Suppose that t_m is the largest term described in equation 3.11. That is, $t_m \geq t_i$ for all $i \leq k$. Thus

$$mC_{loc} + T(e_m) + C_{lat} \geq iC_{loc} + T(e_i) + C_{lat}$$

for all $i \leq k$. In this case,

$$\begin{aligned} T_p(f\ e_1 \ldots e_n) &= mC_{loc} + T(e_m) + C_{lat} + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \\ &\leq T(e_1) + \ldots + T(e_k) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \\ &= T_s(f\ e_1 \ldots e_n) \end{aligned}$$

iff

$$mC_{loc} + C_{lat} \leq \sum_{\substack{i=1 \\ i \neq m}}^{k} T(e_i)$$

When exploiting horizontal parallelism, one of the strict arguments should be evaluated locally. The other strict arguments should still be spawned off in decreasing order of complexity. The problem remains to determine which argument to evaluate locally. The answer is not obvious:

- The expression that is evaluated locally does not incur the communication latency C_{lat}. This favors the choice of a complex argument to evaluate locally.
- The local argument cannot start evaluating until all of the remote expressions have been spawned, namely at time $(k-1)C_{loc}$. This favors the choice of an expression of low complexity to evaluate locally since the more complex expressions should start being evaluated as early as possible.

If e_j is evaluated locally then the parallel execution time, denoted T_p^j, is described by

$$T_p^j(f\ e_1 \ldots e_n) = \max(t'_1, \ldots, t'_k) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n])$$

where

$$t'_i = \begin{cases} t_i & \text{if } i < j \\ t_i - C_{loc} & \text{if } i > j \\ t_i + (k-i-1)C_{loc} - C_{lat} & \text{if } i = j \end{cases} \tag{3.15}$$

Because e_j is evaluated after all other tasks have been spawned, each e_i, $i > j$, is spawned C_{loc} sooner than if all of the arguments were evaluated remotely. However, the time at which e_j finishes executing is changed by adding $(k-1-j)C_{loc} - C_{lat}$. This amount may be negative. Since $(k-1)C_{loc} - C_{lat}$ is a constant, we refer to it as K. The evaluation of e_j completes $K - jC_{loc}$ later than if all expressions were evaluated remotely.

We can minimize $T_p(f\ e_1 \ldots e_n)$ by comparing the values of $T_p^j(f\ e_1 \ldots e_n)$ for each value of j, $1 \leq j \leq k$. The solution can be found more directly, however.

We define M to be the time required to evaluate all of the strict arguments remotely.

$$M = \max(t_1, \ldots, t_n)$$

We define M_j to be the time required to evaluate all of the strict arguments remotely, except for e_j which is evaluated locally.

$$M_j = \max(t'_1, \ldots, t'_k)$$

where each t'_i is defined by equation 3.15. In addition:

1. Since the arguments are sorted in decreasing order of complexity and each successive argument is spawned off C_{loc} later than the previous one, for any i,j such that $i < j$,

$$t_i \geq t_j - (j-i)C_{loc} \tag{3.16}$$

   This indicates that each argument returns no sooner than C_{loc} before the next argument.

2. By equation 3.16, for any i, j, such that $i < j$,

$$\begin{aligned} t_i + K - iC_{loc} &= t_i + K - jC_{loc} + (j-i)C_{loc} \\ &\geq t_j - (j-i)C_{loc} + K - jC_{loc} + (j-i)C_{loc} \\ &= t_j + K - jC_{loc} \end{aligned} \tag{3.17}$$

Again suppose that $t_m \geq t_i$ for all $i \leq k$. Either e_m should be executed locally or some e_p, $p \neq m$, should be. It is clear that p must be no greater than m: Otherwise $t'_m = t_m$ and $M_p \geq t_m = M$. There are several cases to be considered:

- Suppose $K - mC_{loc} \geq 0$. This means that $T_p(f\ e_1 \ldots e_n)$ cannot decrease by executing e_m locally. Is there any e_p, $p < m$, that should be evaluated locally? If so, then t_m is reduced by C_{loc} but t_p is increased by $K - pC_{loc}$. For all $p < m$,

$$\begin{aligned} M_p &\geq t_p + K - pC_{loc} && \text{(by definition of } M_p\text{)} \\ &\geq t_m + K - mC_{loc} && \text{(by equation 3.17)} \\ &\geq t_m && \text{(since } K - mC_{loc} \geq 0\text{)} \\ &= M \end{aligned}$$

  If evaluating e_m locally causes the total execution time to increase then benefit can be derived from executing any argument locally.

- Suppose $K - mC_{loc} \leq 0$. There are two cases to consider:

  1. Assume that for all $i < m$, $t_i \leq t_m - C_{loc}$ (that is, all expressions spawned before e_m return at least C_{loc} before e_m does). If e_m is executed locally, then

$$\begin{aligned} M_m = \max(\ & t_1, \ldots,\ t_{m-1},\ t_m + K - mC_{loc}, \\ & t_{m+1} - C_{loc}, \ldots,\ t_k - C_{loc}) \end{aligned}$$

  If e_p, $p < m$, is executed locally, then

$$M_p = \max(t_1, \ldots,\ t_{p-1},\ t_p + K - pC_{loc},\ t_{p+1} - C_{loc}, \ldots,\ t_k - C_{loc})$$

  By our hypothesis, for all $i < m$, $t_i \leq t_m - C_{loc}$. By definition of t_m, $t_m - C_{loc} \geq t_i - C_{loc}$ for all $i > m$. Therefore,

$$M_p = \max(t_p + K - pC_{loc},\ t_m - C_{loc})$$

  By equation 3.17

$$t_p + K - pC_{loc} \geq t_m + K - mC_{loc}$$

  and thus $M_p \geq M_m$. This shows that in this case it is best to evaluate e_m locally.

  2. If there is an $i < m$ such that

$$t_i > t_m - C_{loc} \tag{3.18}$$

then it can be shown that e_j should be executed locally, where

$$-C_{loc} \leq K - jC_{loc} \leq 0$$

In other words, $j = \lceil K/C_{loc} \rceil$. This says that we should chose the most complex argument to evaluate locally such that the local evaluation of that argument completes before the remote evaluation of that argument would have. In this case,

$$\begin{aligned} M_j &= \max(\ t_1, \ldots,\ t_{j-1},\ t_j + (K-j)C_{loc}, \\ &\qquad t_{j+1} - C_{loc},\ \ldots,\ t_k - C_{loc}) \\ &= \max(t_n,\ t_j + (K-j)C_{loc},\ t_m - C_{loc}) \end{aligned}$$

where $t_n = \max(t_1, \ldots, t_{j-1})$ and therefore $K - nC_{loc} > 0$. In order to prove that M_j is less than M_p for any $p \neq j$, we consider the different possible values for M_j.

- Suppose $M_j = t_n$ and we choose to execute some e_p locally. If $p = n$ then

  $$M_p \geq t_n + (K-n)C_{loc} \geq T_n = M_j$$

  If $p > n$ then $M_p \geq t_n = M_j$. Otherwise, if $p < n$ then

  $$t_p \geq t_n - (n-p)C_{loc}$$

  and

  $$\begin{aligned} t_p + K - pC_{loc} &= t_p + K - nC_{loc} + (n-p)C_{loc} \\ &\geq t_n - (n-p)C_{loc} + K - nC_{loc} + (n-p)C_{loc} \\ &= t_n + K - nC_{loc} \\ &\geq t_n \end{aligned}$$

  and therefore $M_p \geq M_j$.

- Suppose $M_j = t_m - C_{loc}$. For all $p < m$,

  $$M_p \geq t_m - C_{loc} = M_j$$

since $t_m - C_{loc}$ is a term in each M_p. By inequality 3.18, if $p = m$, then

$$M_p > t_m - C_{loc} = M_j$$

since there is an $i < m$ such that $t_i > t_m - C_{loc}$.

- Finally, suppose $M_j = t_j + (K - j)C_{loc}$. If we choose to evaluate e_p locally and $p > j$ then

$$M_p \geq t_j \geq t_j + K - jC_{loc} = M_j$$

since t_j is a term in M_p. If $p < j$ then by equation 3.17

$$\begin{aligned} M_p &\geq t_p + K - pC_{loc} \\ &\geq t_j + (K - j)C_{loc} \\ &= M_j \end{aligned}$$

Now that we have specified which argument, if any, should be evaluated locally, it remains to be seen if parallel evaluation is worthwhile. Suppose, for some $j \leq k$, e_j is to be computed locally. In this case,

$$\begin{aligned} T_p(f\ e_1 \ldots e_n) &= M_j + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \\ &= t'_m + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n]) \end{aligned}$$

where $t'_m \geq t'_i$ for all $i \leq k$. As in equation 3.9,

$$T_s(f\ e_1 \ldots e_n) = (\sum_{i=1}^{k} T(e_i)) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n])$$

If $t'_m = t_m = mC_{loc} + T(e_m)$ then inequality 3.14 is satisfied iff

$$mC_{loc} + T(e_m) + C_{lat} \leq (\sum_{i=1}^{k} T(e_i))$$

or, equivalently,

$$mC_{loc} + C_{lat} \leq \sum_{\substack{i=1 \\ i \neq m}}^{k} T(e_i)$$

If $t'_m = t_m - C_{loc} = (m-1)C_{loc} + T(e_m)$ then inequality 3.14 is satisfied iff

$$(m-1)C_{loc} + T(e_m) + C_{lat} \leq (\sum_{i=1}^{k} T(e_i))$$

that is, iff

$$(m-1)C_{loc} + C_{lat} \leq \sum_{\substack{i=1 \\ i \neq m}}^{k} T(e_i)$$

Otherwise, if e_m was evaluated locally then

$$t'_m = (k-1)C_{loc} + T(e_m)$$

in which case inequality 3.14 is satisfied iff

$$(k-1)C_{loc} \leq (\sum_{i=1}^{k} T(e_i))$$

which is equivalent to

$$C_{loc} \leq (\sum_{\substack{i=1 \\ i \neq m}}^{k} T(e_i))/(k-1) \tag{3.19}$$

Inequality 3.19 demonstrates that if the local expression is the last one to finish being evaluated, then C_{loc} must be no greater than the average time of local execution of all of the other expressions. Otherwise the parallelism in $(f\ e_1 \ldots e_n)$ cannot be exploited.

Evaluating a set of arguments locally

We may wish to evaluate several arguments locally, choosing a set S of such arguments in order to minimize the parallel execution time T_p^S:

$$T_p^S(f\ e_1 \ldots e_n) = \max(m_0, \ldots,\ m_p,\ m) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n])$$

where

$$\begin{aligned}
&S = \{e_{j_0}, \ldots, e_{j_p}\} \subset \{e_1, \ldots, e_k\} \text{ and } j_i < j_{i+1} \\
&m_i = \max(\ (j_i + 1 - i)C_{loc} + T(e_{j_i+1}) + C_{lat}, \\
&\qquad\qquad (j_i + 2 - i)C_{loc} + T(e_{j_i+2}) + C_{lat}, \\
&\qquad\qquad \vdots \\
&\qquad\qquad (j_{i+1} - 1 - i)C_{loc} + T(e_{j_{i+1}-1}) + C_{lat})) \\
&m = (\textstyle\sum_{i=0}^{p} T(e_{j_i})) + (k-p)C_{loc}
\end{aligned}$$

In order to find the set S that minimizes $T_p^S(f\ e_1 \ldots e_n)$, one could perform a search (which is exponential with respect to k) over the possible alternatives for

S. Any set S that minimizes $T_p^S(f\ e_1 \ldots e_n)$ will exhibit a property that provides the basis for our heuristic partitioning algorithm described in section 4.3: For every e_i such that $T(e_i) \leq C_{loc}$, $e_i \in S$. This is easily seen because the cost (to the local processor) of creating a remote task for each such e_i would be greater than the cost of evaluating e_i locally. If a set S does not contain an expression e_i such that $T(e_i) \leq C_{loc}$, then the set $S' = S \cup \{e_i\}$ of expressions to be evaluated locally will result in a lower total execution time. Those arguments e_j, $j > i$, that are evaluated remotely will be spawned off C_{loc} sooner, and while the local evaluation of the expressions in S' will take $T(e_1)$ longer, it will start C_{loc} sooner.

Evaluating sets of arguments on each processor

The most general method for exploiting horizontal parallelism is to evaluate sets of arguments on each processor. Let

$$S = \{S_1, \ldots, S_p\}$$

where

$$S_i = \{e_{i1}, \ldots, e_{in_i}\}$$

and

$$\bigcup S_i = \{e_1, \ldots, e_k\}$$

and each S_i is a set of arguments that gets evaluated on a single processor (assume S_p gets evaluated locally). In this case,

$$T_p^S(f\ e_1\ e_n) = \max(t1, \ldots, t_p) + T(f[e_1, \ldots, e_k][e_{k+1}, \ldots e_n])$$

where

$$t_i = \begin{cases} iC_{loc} + (\sum_{j=1}^{n_i} T(e_{ij})) + C_{lat} & \text{if } i < p \\ pC_{loc} + \sum_{j=1}^{n_p} T(e_{pj}) & \text{if } i = p \end{cases}$$

Again, a straightforward (although expensive) way to find the set S that minimizes $T_p^S(f\ e_1 \ldots e_n)$ would be to perform a search over the possible alternatives for S. The systems described in chapters 6 and 8 do not, however, have the ability to combine a set of expressions into a single task to be executed on a remote processor.

3.4.2 An Analysis of Vertical Parallelism

At this point, we incorporate the possibility of *vertical parallelism*, which occurs when the body of a function is entered while some of the strict arguments are still being evaluated.

Strict binary function calls

We again consider an expression of the form $(f\ e_1\ e_2)$. The simplest way to achieve vertical parallelism would be to evaluate both arguments on remote processors while the body of f is evaluated as much as possible locally. Let f^- represent the evaluation of that part of the body of f that can be performed without the values of any arguments. Let $f^+[e_1, e_2]$ represent the evaluation of the body of f after its arguments e_1 and e_2 have been evaluated (exclusive of that already computed by f^-). We can rewrite $T_s(f\ e_1\ e_2)$ as follows:

$$T_s(f\ e_1\ e_2) = T(e_1) + T(e_2) + T(f^-) + T(f^+[e_1, e_2])$$

Assume that $T(e_1) \geq T(e_2)$. By the same reasoning used in section 3.4.1 it is best to start the remote evaluation of e_1 before e_2. Therefore,

$$\begin{aligned} T_p(f\ e_1\ e_2) = \max(\ & C_{loc} + T(e_1) + C_{lat},\ 2C_{loc} + T(e_2) + C_{lat}, \\ & 2C_{loc} + T(f^-)) + T(f^+[e_1, e_2]) \end{aligned} \tag{3.20}$$

In order to determine if $T_p \leq T_s$, the possible values of T_p are considered separately.

- If $2C_{loc} + T(f^-)$ is greater than both $C_{loc} + T(e_1) + C_{lat}$ and $2C_{loc} + T(e_2) + C_{lat}$ then

  $$T_p(f\ e_1\ e_2) = 2C_{loc} + T(f^-) + T(f^+[e_1, e_2])$$

  Therefore $T_p(f\ e_1\ e_2) \leq T_s(f\ e_1\ e_2)$ iff $2C_{loc} \leq T(e_1) + T(e_2)$. That is, C_{loc} must be less than the average cost of the expressions being evaluated remotely.

- If $C_{loc} + T(e_1) \leq 2C_{loc} + T(e_2)$ and $T(f^-) \leq T(e_2) + C_{lat}$ then

  $$T_p(f\ e_1\ e_2) = 2C_{loc} + T(e_2) + C_{lat} + T(f^+[e_1, e_2])$$

  and $T_p(f\ e_1\ e_2) \leq T_s(f\ e_1\ e_2)$ iff

  $$2C_{loc} + C_{lat} \leq T(e_1) + T(f^-)$$

- Otherwise, if $T(e_1) \geq C_{loc} + T(e_2)$ and $T(e_1) + C_{lat} \geq 2C_{loc} + T(f^-)$ then

$$T_p(f\ e_1\ e_2) = C_{loc} + T(e_1) + C_{lat} + T(f^+[e_1, e_2])$$

and $T_p(f\ e_1\ e_2) \leq T_s(f\ e_1\ e_2)$ only if

$$C_{loc} + C_{lat} \leq T(e_2) + T(f^-)$$

The only vertical parallelism exploited was the concurrency between the evaluation of the arguments and the work that f could do without the value of any of the arguments. However, f may be able to perform a significant amount of work with the value of only one of its arguments. It should resume processing as soon as the value of either argument returns.

If we restrict the function f to be a combinator (such as a refined supercombinator from chapter 2) then the values of all identifiers referenced in the body of f (with the exception of global combinator names) are passed as arguments. Therefore, if we assume that constant folding has occurred during compilation, there is no work that f can perform without having the values of any arguments. If this is the case, then it makes sense to evaluate one of the arguments locally - but which one? Should it be the one that f could perform more work with, or the one that takes longer to execute (as is the case when we were only allowing horizontal parallelism)? As it turns out, the answer is not so simple.

Let $f^1[e_1]$ denote the work that f can perform after only e_1 has been evaluated. When the value of e_2 becomes available, the rest of the body of f can be computed and is denoted $f^{1,2}[e_1, e_2]$. Likewise, $f^2[e_2]$ and $f^{2,1}[e_2, e_1]$ denote the work that f performs after e_2 returns and then after e_1 returns, respectively. An important assumption is that

$$T(f^1[e_1]) + T(f^{1,2}[e_1, e_2]) = T(f^2[e_2]) + T(f^{2,1}[e_2, e_1]) \tag{3.21}$$

That is, if both arguments have already been evaluated by the time they are needed, then whether one performs f^1 and then $f^{1,2}$ or f^2 and then $f^{2,1}$ the total execution time will be identical. This is a reasonable assumption if one considers the body of f to be a number of independent computation steps.

The cost of executing $(f\ e_1\ e_2)$ in parallel is either

$$\begin{aligned} T_p^1(f\ e_1\ e_2) &= \max(\ C_{loc} + T(e_2) + C_{lat}, \\ &\qquad C_{loc} + T(e_1) + T(f^-) + T(f^1[e_1])) + \\ &\quad T(f^{1,2}[e_1, e_2]) \end{aligned} \tag{3.22}$$

when e_1 is computed locally, or

$$\begin{aligned} T_p^2(f\ e_1\ e_2) &= \max(\ C_{loc} + T(e_1) + C_{lat}, \\ &\qquad C_{loc} + T(e_2) + T(f^-) + T(f^2[e_2]) + \\ &\quad T(f^{2,1}[e_2, e_1]) \end{aligned} \tag{3.23}$$

when e_2 is computed locally.

We need to determine under what conditions $T_p^1(f\ e_1\ e_2) \leq T_p^2(f\ e_1\ e_2)$. There are three separate cases that must be considered.

1. Assume

$$T(e_1) + T(f^-) + T(f^1[e_1]) \geq T(e_2) + C_{lat}$$

and

$$T(e_2) + T(f^-) + T(f^2[e_2]) \geq T(e_1) + C_{lat}$$

If so,

$$\begin{aligned} T_p^1(f\ e_1\ e_2) &= C_{loc} + T(e_1) + T(f^-) + T(f^1[e_1]) + T(f^{1,2}[e_1, e_2]) \\ &\leq C_{loc} + T(e_2) + T(f^-) + T(f^2[e_2]) + T(f^{2,1}[e_2, e_1]) \\ &= T_p^2(f\ e_1\ e_2) \end{aligned}$$

iff $T(e_1) \leq T(e_2)$.

Therefore, if the remote argument (whether it is e_1 or e_2) returns before it is needed then it is best to choose the argument with the *shorter* execution time to evaluate locally.

2. Assume

$$T(e_1) + T(f^-) + T(f^1[e_1]) \geq T(e_2) + C_{lat} \tag{3.24}$$

and

$$T(e_1) + C_{lat} \geq T(e_2) + T(f^-) + T(f^2[e_2]) \tag{3.25}$$

If so,

$$\begin{aligned} T_p^1(f\ e_1\ e_2) &= C_{loc} + T(e_1) + T(f^-) + T(f^1[e_1]) + T(f^{1,2}[e_1, e_2]) \\ &\leq C_{loc} + T(e_1) + C_{lat} + T(f^{2,1}[e_2, e_1]) \\ &= T_p^2(f\ e_1\ e_2) \end{aligned}$$

iff

$$T(f^-) + T(f^1[e_1]) + T(f^{1,2}[e_1, e_2]) \leq T(e_1) + C_{lat} + T(f^{2,1}[e_2, e_1])$$

which, by equation 3.21, is equivalent to

$$T(f^-) + T(f^2[e_2]) \leq C_{lat} \tag{3.26}$$

The assumption indicates that:

- If e_1 is computed locally then the value for e_2 will return before $f^1[e_1]$ has completed. Therefore, the local processor will not have to wait for the value of e_2 at any point during the evaluation of $(f\ e_1\ e_2)$.
- Otherwise, if e_2 is evaluated locally, then $f^2[e_2]$ will finish before the value of e_1 arrives, and the local processor will have to wait for the value of e_1 before staring the execution of $f^{2,1}[e_2, e_1]$.

It would seem that the parallel execution time would be less in the case in which the local processor does not have to wait. However, if the other choice were made, more of the computation might occur in parallel. Inequality 3.26 shows, surprisingly, that the decision should be based only on the communication overhead and certain properties of f and not on the relative execution times of e_1 and e_2 (once inequalities 3.24 and 3.25 are satisfied).

3. Assume

$$T(e_2) + C_{lat} \geq T(e_1) + T(f^-) + T(f^1[e_1])$$

and

$$T(e_1) + C_{lat} \geq T(e_2) + T(f^-) + T(f^2[e_2])$$

If so,

$$\begin{aligned} T_p^1(f\ e_1\ e_2) &= C_{loc} + T(e_2) + C_{lat} + T(f^{1,2}[e_1, e_2]) \\ &\leq C_{loc} + T(e_1) + C_{lat} + T(f^{2,1}[e_2, e_1]) \\ &= T_p^2(f\ e_1\ e_2) \end{aligned}$$

iff

$$T(e_2) + T(f^{1,2}[e_1, e_2]) \leq T(e_1) + T(f^{2,1}[e_2, e_1])$$

which, (again using equation 3.21) is equivalent to

$$T(e_2) + T(f^2[e_2]) \leq T(e_1) + T(f^1[e_1]) \tag{3.27}$$

The assumption states that the local processor will have to wait for the value of the remote argument. Inequality 3.27 indicates that the local argument should be chosen such that the amount of work that the local processor can do before waiting is maximized.

Once we have determined which argument is better to compute locally, we must determine if the parallelism available is worth exploiting; that is if

$$T_p(f\ e_1\ e_2) \leq T_s(f\ e_1\ e_2) \tag{3.28}$$

Assume that $T_p(f\ e_1\ e_2)$ is minimized by evaluating e_1 locally, in which case $T_p(f\ e_1\ e_2)$ is determined by equation 3.22. There are two cases to be considered:

1. If

$$T(e_1) + T(f^-) + T(f^1[e_1]) \geq T(e_2) + C_{lat} \tag{3.29}$$

then

$$T_p(f\ e_1\ e_2) = C_{loc} + T(e_1) + T(f^-) + T(f^1[e_1]) + T(f^{1,2}[e_1, e_2])$$

Therefore $T_p(f\ e_1\ e_2)$ is less than $T_s(f\ e_1\ e_2)$, as defined in equation 3.1, iff $C_{loc} \leq T(e_2)$.

2. If

$$T(e_2) + C_{lat} \geq T(e_1) + T(f^-) + T(f^1[e_1]) \tag{3.30}$$

then f must wait for the value of e_2 to return. Inequality 3.28 is satisfied iff

$$C_{loc} + C_{lat} \leq T(e_1) + T(f^-) + T(f^1[e_1])$$

The communication cost (local overhead plus latency) must be less than the work that can be done on the local processor before requiring the value of e_2.

Function calls with multiple arguments

We now extend the treatment of vertical parallelism to determine the best way to evaluate an expression of the form $(f\ e_1 \ldots e_n)$ where f is strict in only some of its arguments.

Again we assume that f is strict in only its first k arguments. The first question is: In what order should the arguments be spawned?

The answer will be based upon two pieces of information:

1. The minimum execution time for each e_i, namely $T(e_i)$.

2. The order in which the values of the arguments are needed in the function f and the amount of work that can be done with each argument before the next one is needed. In the previous section we assumed that f could resume executing when the value of either of its arguments was available. However, if the number k of strict arguments is large then the complexity of the code for f would have to be on the order of $k!$ to handle the values of the arguments in any order. Therefore, we specify an order in which f can use the values of its arguments.

Assume that f requires the values of its strict arguments in the order $e_1 \ldots e_k$ and that f^i is the work that f can perform after the value of e_i has returned and before it needs the value of e_{i+1}. The work that f^i performs may include creating new tasks to evaluate the non-strict arguments. After f^k finishes executing, the evaluation of the function application (including the appropriate non-strict arguments) is complete.

Suppose that $p_1 \ldots p_k$ is a sequence of integers that determines the order in which the strict arguments are spawned. That is, each e_i is the p_ith argument spawned. For example if $p_3 = 5$, then e_3 was the fifth argument spawned. If the arguments were spawned in the order in which they were needed by f, then $p_i = i$, $1 \leq i \leq k$.

We would like to determine the value of each p_i such that the total execution

time is minimized. The total parallel execution time is given by the following equation:

$$T_p(f\ e_1 \ldots e_n) = M_k + T(f^k)$$

where

$$\begin{aligned} M_1 &= \max(kC_{loc},\ p_1C_{loc} + T(e_1) + C_{lat}) \\ M_2 &= \max(M_1 + f^1,\ p_2C_{loc} + T(e_2) + C_{lat}) \\ &\vdots \\ M_k &= \max(M_{k-1} + f^{k-1},\ p_kC_{loc} + T(e_k) + C_{lat}) \end{aligned}$$

Each M_i represents the time at which:

1. the value of e_i is available, and
2. f is ready to use the value of e_i

Again a straightforward method for minimizing the parallel execution time would be an exponential search over the possible orders in which the arguments are spawned.

For the purpose of developing heuristics to be used by the compiler, there are several pieces of information that can be extracted from the above analysis:

1. If e_i and e_{i+1} are needed at the same time, that is if $T(f^i) = 0$, then the expression with the longer execution time should be spawned first. If $f^i = 0$ then

$$\begin{aligned} M_i &= \max(M_{i-1} + f^{i-1}, p_iC_{loc} + T(e_i)) \\ M_{i+1} &= \max(M_i, p_{i+1}C_{loc} + T(e_{i+1})) \\ &= \max(M_{i-1} + f^{i-1}, p_iC_{loc} + T(e_i), p_{i+1}C_{loc} + T(e_{i+1})) \end{aligned}$$

   Assume $T(e_{i+1}) \geq T(e_i)$. We show that if $p_{i+1} \geq p_i$ then the total execution time can be reduced by interchanging the order in which e_i and e_{i+1} are spawned. Suppose $p_{i+1} = p_i + q$, for some positive integer q. Therefore,

$$M_{i+1} = \max(M_{i-1} + f^{i-1}, p_iC_{loc} + T(e_i), p_i + qC_{loc} + T(e_{i+1}))$$

   If we switch the order in which e_i and e_{i+1} are spawned then

$$M'_{i+1} = \max(M_{i-1} + f^{j-1}, (p_i + q)C_{loc} + T(e_i), p_iC_{loc} + T(e_{i+1})$$

Since $T(e_{i+1}) \geq T(e_i)$,

$$p_i + qC_{loc} + T(e_{i+1}) \geq (p_i + q)C_{loc} + T(e_i)$$

and

$$p_i + qC_{loc} + T(e_{i+1}) \geq p_i + qC_{loc} + T(e_i)$$

Therefore $M'_{i+1} \geq M_{i+1}$. We can show that if, for all $j > i + 1$,

$$M'_j = \max(M'_{j-1} + f^{j-1}, p_j C_{loc} + T(e_j))$$

then $M'_j \leq M_j$ and therefore $M'_k \leq M_k$. The proof is trivial and proceeds by an induction on the value of j (and is left to the reader).

2. If two arguments have the same execution time then they should be spawned in the order in which they are needed.[5] In other words, if $T(e_i) = T(e_j)$ and $M_i < M_j$ for any values of p_i and p_j, then execution time is lower when $p_i < p_j$ than when $p_i > p_j$.

   Assume otherwise, namely that $p_i = p_j + q$ for some positive integer q. Since $T(e_i) = T(e_j)$,

   $$M_i = \max(M_{i-1} + f^{i-1}, (p_j + q)C_{loc} + T(e_i))$$

   and

   $$M_j = \max(M_{j-1} + f^{j-1}, p_j C_{loc} + T(e_i))$$

   Since $M_j \geq M_i$,

   $$M_j = \max(M_{j-1} + f^{j-1}, (p_j + q)C_{loc} + T(e_i))$$

   If we interchange the order in which e_i and e_j are spawned, then

   $$M'_i = \max(M_{i-1} + f^{i-1}, p_j C_{loc} + T(e_i)) \leq M_i$$

   For any l such that $i < l < j$,

   $$M'_l = \max(M'_{l-1} + f^{l-1}, p_l C_{loc} + T(e_l))$$

[5] A sophisticated analysis called *Path Analysis* has been developed by Bloss and Hudak [6] to determine this order

and it can be shown by induction on the value of l that $M'_l \leq M_l$ (because $M'_i \leq M_i$). Therefore,

$$\begin{aligned} M'_j &= \max(M'_{j-1} + f^{j-1}, (p_j + q)C_{loc} + T(e_i)) \\ &\leq \max(M_{j-1} + f^{j-1}, (p_j + q)C_{loc} + T(e_i)) \\ &= M_j \end{aligned}$$

Again by induction, if $M'_j \leq M_j$ then $M'_k \leq M_k$, and the total execution time is reduced if the arguments with the same execution time are spawned in the order that they are needed.

3. By the same reasoning used in section 3.4.1, any arguments whose execution time is less than C_{loc} should be executed locally.

3.5 Task Lifting

So far we have only considered the case in which each subexpression e_i in the expression $(f\ e_1 \ldots e_n)$ is turned into a task. This task may generate more parallel tasks. For example, if each e_i is of the form $(e_{i1}\ e_{12} \ldots e_{in_i})$ then after the task for e_i starts executing, tasks for each expression e_{ij} may be generated. This situation is pictured in figure 3.1 and shows that the task for e_i is responsible for creating the task for each e_{ij}.

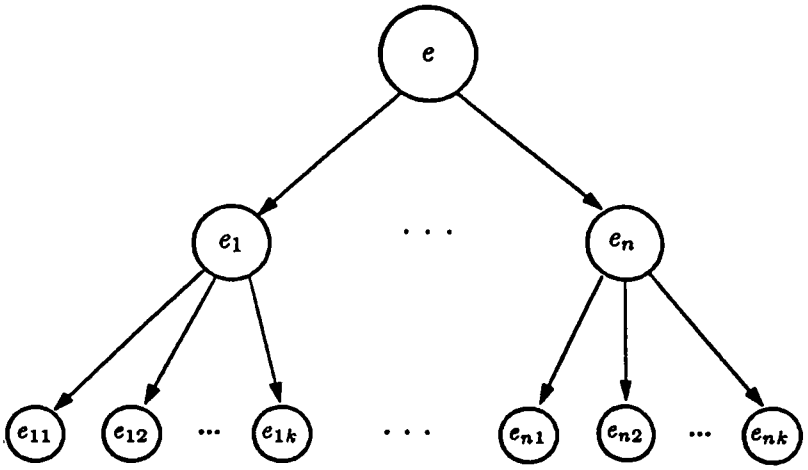


Figure 3.1: Task creation during execution

It may be possible for the creation of the task for each expression e_{ij} to be "lifted" out of the task for e_i and to occur at the same time that the task for e_i is created. This situation is pictured in figure 3.2 and would require that the task for e_i be responsible only for waiting for the value of each e_{ij} when needed.

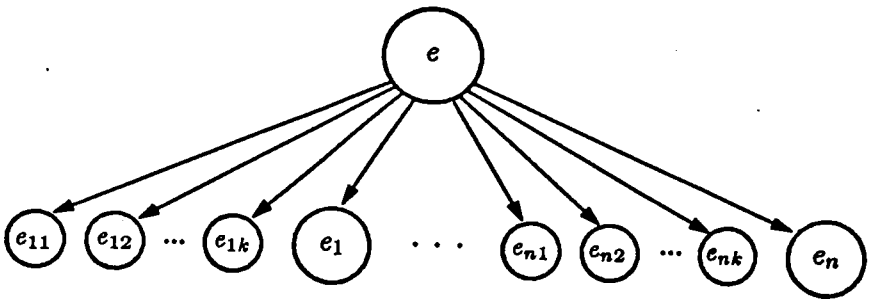


Figure 3.2: Task creation for "lifted" expressions

If this lifting were not performed, the creation of the tasks for each e_{ij} would not occur until after the task for e_i started executing on a remote processor, which would be at least $C_{lat}/2$ (the latency for a one-way message) after the task for e_i was created. The lifting allows the creation of tasks for each e_{ij} to occur earlier.

Lifting has a drawback, however. Without lifting, the task for each e_i incurs C_{loc} creating a task e_{ij} for each value of j. With lifting the *parent* task—evaluating $(f\ e_1 \ldots e_n)$—incurs C_{loc} for e_{ij} for all values of i and j. Without lifting, the task evaluating each e_i incurs the cost of sequentially creating sub-tasks for $e_{i1} \ldots e_{in_i}$, but runs in parallel with the tasks for e_m, for all $m \neq i$. Therefore, the total overhead execution time is affected by the *maximum* number of sub-tasks created (one for each e_{ij}) in the task for each e_i. If lifting has occurred, then the creation of all tasks for the subexpressions e_{ij} occurs sequentially and the total execution time is affected by the *total* number of such subexpressions. In short, without lifting the overhead cost of task creation is parallelized while with lifting it is sequentialized. This situation is pictured in figure 3.3.

Under what circumstances would be it be worthwhile for lifting to occur? For simplicity we assume that only horizontal parallelism is being exploited and

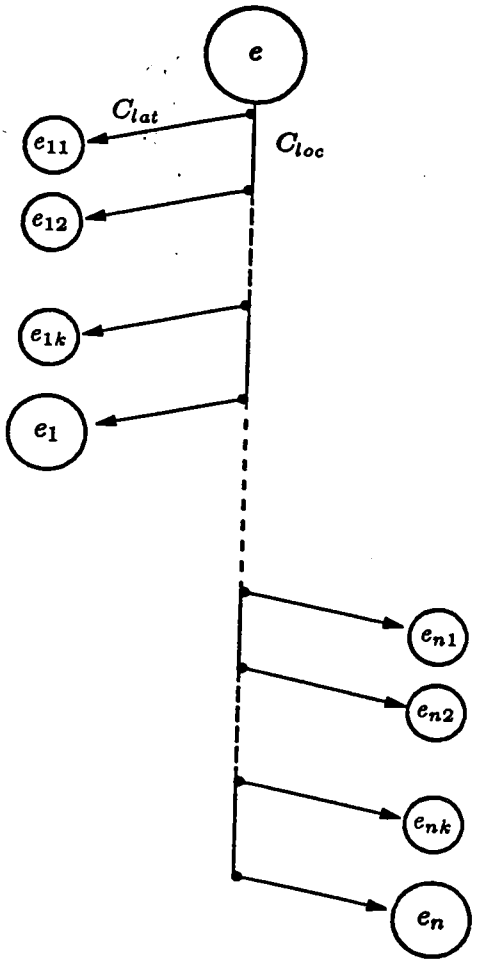


Figure 3.3: Sequential task creation due to "lifting"

in the expression $(f\ e_1 \ldots e_k)$, f is strict with respect to all of the arguments.

Without lifting,

$$T_p(f\ (e_{11} \ldots e_{1n_1}) \ldots (e_{k1} \ldots e_{kn_k})) = \max(t_1, \ldots, t_k)$$

where for each t_i,

$$t_i = iC_{loc} + \max(t_{i1}, \ldots, t_{in_i}) + T(e_{i1}[e_{i2}, \ldots, e_{in_i}]) + C_{lat}$$

and for all i,j,

$$t_{ij} = jC_{loc} + T(e_{ij}) + C_{lat}$$

For some q and r, t_q is the largest value over all t_i, and t_{qr} is the largest value over t_{qj} for all j and

$$T_p(f\ (e_{11}\ldots e_{1n_1})\ldots(e_{k1}\ldots e_{kn_k})) = \\ (q+r)C_{loc} + T(e_{qr}) + T(e_{q1}[e_{q2},\ldots,e_{qn}]) + 2C_{lat}$$

With lifting,

$$T_p(f\ (e_{11}\ldots e_{1n})\ldots(e_{k1}\ldots e_{kn})) = \max(t'_1,\ldots,t'_k)$$

where

$$t'_i = (\textstyle\sum_{m=1}^{i-1} n_m)C_{loc} + \max(t'_{i1},\ldots,t'_{in_i}) + T(e_{i1}[e_{i2},\ldots,e_{in}]) + C_{lat}/2$$
$$t'_{ij} = jC_{loc} + T(e_{ij}) + C_{lat}$$

For some q' and a r' such that $t'_{q'}$ is the largest value over all t'_i, and $t'_{q'r'}$ is the largest value over $t'_{q'j}$ for all j,

$$T^l_p(f\ (e_{11}\ldots e_{1n})\ldots(e_{k1}\ldots e_{kn})) = \\ (\textstyle\sum_{m=1}^{q'-1} n_m) + r')C_{loc} + T(e_{q'r'}) + \\ T(e_{q'1}[e_{q'2},\ldots,e_{q'n}]) + 3C_{lat}/2$$

In order to see if lifting is worthwhile, the values of T_p and T^l_p have to be computed. Obviously, if C_{loc} is quite large and the evaluation of each e_i can be broken down into a large number of tasks, lifting will probably not be worthwhile. On the other hand, if C_{lat} is much larger than C_{loc} and the number of subexpressions, e_{ij}, in each expression, e_i, is small, then lifting may be worthwhile.

3.6 Sequentiality in Functional Programs

In a program the evaluation of two expressions may not be able to occur in parallel with each other, even if their execution times are sufficiently large. If so, they said to be *sequentially ordered* with respect to one another.

There are two fundamental reasons for the evaluation of two expressions to be sequentially ordered. The first reason is that there may be *data dependencies* between them. Before a value can be used in one expression, it may have to be computed by the other expression. In the previous section we saw that it is beneficial to evaluate one of the arguments to a function call locally, since there is little the body of a function can do without the value of any of its arguments. There is sequential ordering between the evaluation of an argument and the evaluation of the part of the body of the function that requires the value of that argument. These two expressions can therefore be evaluated on the same processor without the loss of parallelism.

The second source of sequentiality in a program arises from the desire to preserve the termination properties of normal order evaluation. An expression e cannot be evaluated unless its value will be needed. Therefore, the expression that determines if e will be needed has to be evaluated before e is. All sequentiality of this kind is due to the conditional expression,

$$(e_1 \rightarrow e_2,\ e_3)$$

in which the predicate e_1 must be evaluated before e_2 or e_3.

It is principally because of the existence of the conditional that function applications may contain non-strict arguments.[6] Although strict arguments in a function application can always be evaluated in parallel, there is often little parallelism available between the evaluation of strict arguments and non-strict arguments. Although the execution of some non-strict arguments may overlap with the execution of some strict arguments, at least one strict argument must be *completely evaluated* before any non-strict argument can start executing.

Note that the term *sequentially ordered* refers to the order of evaluation among two or more expressions and does not imply that each such expression must itself be evaluated sequentially. In the above conditional, each of e_1, e_2, and e_3 may contain substantial parallelism. However, the conditional operator supplies no additional parallelism beyond that supplied by its arguments and can thus be considered a *sequential operator*. Other examples include the

[6]The other source of non-strict arguments lies in functions, such as the K combinator, that discard some of their arguments

standard boolean operators such as **and** and **or**.

3.6.1 Practical Sequentiality

Even if an expression contains some parallelism, unless that parallelism is useful (as defined in section 3.2) the expression should be evaluated by a single task on one processor. Therefore, we define an expression to be *practically sequential* if it contains no useful parallelism. Notice that whether an expression is practically sequential depends upon the communication costs of the particular multiprocessor being utilized.

Definition: An expression e is *practically sequential* if for any method of evaluating e in parallel,

$$T_s(e) \leq T_p(e)$$

where $T_p(e)$ is dependent on C_{loc} and C_{lat}.

Intuitively, an expression e is practically sequential if all subexpressions of e are either sequentially ordered or are too fine grained to warrant evaluating in parallel.

In the next chapter we return to the discussion of the compilation process. The heuristic algorithms used by the compiler are based on the results of this analysis.

Chapter 4

Automatic Partitioning of Functional Programs

In this chapter we describe the second phase of our compilation process. A program which has been transformed into a set of refined supercombinators is partitioned into a set of **serial combinators**. Each serial combinator specifies the behavior of a single task and determines the granularity of the parallel computation. The analytical treatment of parallelism in chapter 3 has provided the basis for the partitioning algorithms described in this chapter.

4.1 Heuristics

We now briefly describe the heuristics used by the compiler to partition the program. While these heuristics are based upon the analysis of chapter 3, the amount of information available to the compiler is far less than is assumed in the analysis. Even if the compiler had sufficient information, the complexity of solving for T_p for the various cases discussed above would make the compilation time prohibitively long.

A more complete description of the algorithms used to partition the program can be found in section 4.3.

4.1.1 Determining Execution Times

The analysis of section 3.4 assumed that the execution times of various expressions in a program were known. These execution times are in general uncomputable. In order for a compiler to partition a program based on this analysis, it will have to rely on methods aimed at *estimating* how long an expression will take to evaluate. The methods fall under the following categories:

1. User supplied information: The compiler can use information supplied by the user in estimating the time required to evaluate a given expression. Some possibilities are:

   - Explicit specification of execution times: The user may be able to specify how long a given expression will take to execute. In many cases it would be sufficient for each function *definition* in the program to be annotated with the expected execution time of a call to that function. To be accurate, the expected execution time of an invocation of each function would have to be independent of the value of the arguments. Otherwise, the user may be forced to give an expected execution time for each function *application* in the program.

     Based on these user-supplied execution time estimates, the compiler can decide which expressions are worth evaluating in parallel and the order in which parallel tasks for various expressions should be created.

   - Specification of program behavior: The user may be able to annotate the program with information on how the program will behave. For example the user may be able to specify, for each recursive function in the program, how the depth of the recursion depends on the values of the arguments.

     If the execution time of a given function is dependent upon the value of its arguments, the decision to decompose an expression involving that function may have to be made at run-time (this is what the Stardust interpreter [27] does). The compiler would have to generate code that would use the values of the arguments to aid the dynamic scheduler in deciding where to evaluate each invocation of

the function.

2. Execution Profiles: Supplied with information gained from previous program runs, the compiler may be able to estimate the execution time of various function invocations in the current run. This requires some consistency in execution times over the range of inputs. The program must also be executed often enough to compensate for the expense of taking performance measurements during the first run and for increased execution time of the first run due to poor (or non-existent) partitioning.

3. Compiler Analysis: The compiler may perform a complexity analysis on a program without any external assistance (such as annotations or profiles). Naturally the analysis would have to be rather simple.

Since the goal of this research is to investigate the viability of *automatically* partitioning functional programs, we use the third method listed above to estimate execution times. The complexity analysis used in the compiler is painfully simple and consists of the following:

- The complexity of each expression that does not include a recursive function call can be determined by adding up the cost of the instructions (e.g. memory references, arithmetic operations, etc.) required to compute its value. In this way an accurate estimate of execution times of arithmetic expressions and other simple non-recursive expressions can be produced.

- The compiler has to assume that the complexity of a call to recursive functions is essentially *infinite*. Therefore, a separate task will be created to evaluate a recursive call. Unfortunately, calls to recursive functions that take a very short time to evaluate may be sent to remote processors. Since all invocations of recursive functions are considered to have the same execution costs (namely ∞), the ability to reduce the total execution time by ordering the spawning of remote tasks in an optimal way is lost.

As a heuristic, the execution times of recursive function invocations can be ordered by the size of the body of the function. If each pass through the body of a recursive function f takes more time than each pass through a recursive function g then we assume that an invocation of f takes longer than an invocation of g. This only works, of course, if the depth of the recursion of all functions is

roughly equivalent.

If the expression in the function position is not the name of a user-defined function or primitive operation then it may not be obvious whether or not the function is recursive. It may be a bound variable or some other higher-order expression. In this case, an *abstract interpretation* of the program can be performed to determine the nature (recursive or non-recursive) of the function. This is analogous to the use of abstract interpretation to detect sharing in functional programs described in chapter 2.

4.1.2 Heuristic Partitioning Based on Imperfect Information

Using the simple complexity heuristic described above, the expected execution time of a given expression is either some finite value or ∞. Most of the expressions that do not involve recursion are simple arithmetic expressions that are not complex enough to warrant creating a task for. In the analysis presented in section 3.4, it was clear that any expression whose complexity is less than C_{loc} should be executed locally as part of a larger task. Expressions can be broken down into three categories:

1. Those expressions with complexity less than C_{loc}.
2. Those expressions with a finite complexity greater than C_{loc}.
3. Those expressions with an infinite complexity.

Using this limited information, the compiler has to determine which expression (aside from the ones with complexity less than C_{loc}) to compute locally and to decide upon an order in which to spawn those expressions that are being executed remotely.

In most multiprocessors, the cost C_{loc} of creating a task on a remote processor is generally far greater that the complexity of any non-recursive expression found in a program. Therefore, the expressions will generally fall into two classes, those with complexity less than C_{loc} and those with infinite complexity. In order to simplify the heuristics, we assume that only these two classes of expressions exist.

Primitive Operations

When a strict primitive operation, such as $(e_1 + e_2)$, is encountered in a program, the compiler has to decide which, if any, operand should be executed on a remote processor. According to the analysis of section 3.4.1, if any operand is evaluated remotely then it should be the *least* complex argument. The compiler relies on the heuristic complexity analysis and information about C_{loc} and C_{lat} to determine if either inequality 3.7 or 3.8 is satisfied. If so, then the primitive operation is worth partitioning.

If the primitive operator is the conditional operator, the heuristic decomposition methods can be applied recursively to each of the predicate, consequent, and alternate components of the conditional. However, neither the consequent nor the alternate can evaluated in parallel with the predicate (nor each other).

Function Applications

In a function application, any strict argument whose complexity has been determined to be less than C_{loc} is computed locally. Since the strict arguments with sufficient complexity to be spawned remotely usually have the same complexity, namely ∞, the order in which they will be spawned will be determined by order in which they are needed in the body of the function.

In many cases, none of the arguments in a function application will have an execution time less than C_{loc}. In this case, one of the arguments should be evaluated locally. Since all of the arguments will generally have a complexity of ∞, the argument that is needed first will be evaluated locally.

Determining the order in which the arguments to a function application are needed within the body of the function can be difficult, especially in the presence of higher order functions. A sophisticated analysis, called *path analysis*, has been developed by Bloss and Hudak [6] to determine this order.

4.2 Serial Combinators

Since we are decomposing a functional program into tasks that will each be executed on a single processor, we need to completely describe the behavior of

each task in such a way that a compiler can generate code for it. In this section, we assign a procedural description to the work that a task performs. One way to view this procedural description is as an intermediate representation of the source program in which explicit synchronization between parallel components of the program has been included.

The procedural description of each sequential piece of the program is expressed as a **serial combinator**. A serial combinator is a function whose body contains constructs for creating and synchronizing the execution of tasks. The body of each serial combinator is executed sequentially.

Each call to a serial combinator creates a new task, along with a new node in the program graph to hold the state information for that task task. Like any combinator, a serial combinator does not require a hierarchical environment structure. Maintaining a hierarchical environment structure on a multiprocessor could require interprocessor communication to resolve variable references, since the environment may be spread over a number of processors. Every variable accessed by a serial combinator, however, can be found in the local environment represented by its activation record.

A serial combinator is defined to be a function with the following properties:

1. It is a combinator.

2. Its body is practically sequential and consists of:

   - expressions that are either sequential or too fine-grained to decompose,
   - constructs that explicitly create tasks by invoking other serial combinators,
   - constructs for initiating the evaluation of non-strict arguments, and
   - constructs for suspending evaluation until the values computed by remote tasks arrive.

3. It is the largest possible function that satisfies properties 1 and 2. That is, its body could not occur as a subexpression within the body of another serial combinator.

The third property listed above reflects the fact that the program should be partitioned into as few serial combinators as possible without reducing the potential for exploiting useful parallelism. If a practically sequential expression is decomposed into several serial combinators, its execution time will be adversely affected by the overhead of the serial combinator calls. Since each serial combinator specifies the work to be performed by a single task, some unnecessary overhead may be incurred by the task scheduler if there are a greater number of serial combinator invocations than necessary.

Even though serial combinators control the execution order by creating tasks and initiating the evaluation of delayed expression, they must preserve the termination properties of normal order reduction. In section 4.2.5 we discuss how delayed serial combinator applications are also represented as nodes in the program graph and are used to preserve the lazy semantics of ALFL.

4.2.1 Serial Combinators and Tasks

Serial combinators were chosen to specify the behavior of a task because they are natural extensions of the functions used in uniprocessor graph reduction. On a uniprocessor system, graph reduction provides the mechanism for lazy evaluation and maintaining shared expressions via the creation of nodes in the graph. On a multiprocessor, the evaluation of serial combinators via graph reduction accomplishes all of the following:

1. It supports lazy evaluation via the creation of nodes representing delayed expressions in the same manner as uniprocessor graph reduction.

2. It supports sharing of expressions via the manipulation of arcs between nodes, also in the same way as uniprocessor graph reduction.

3. It provides a representation for the state of a task via the nodes in the graph. The state information that must be contained in each node includes

    - local variables and information concerning their state (either unevaluated or evaluated),
    - an instruction pointer, and

- synchronization information (whether the task is executing or suspended, and which values if any are needed before the task can resume).

4. It provides a multi-threaded dynamic chain as a mechanism for parallel activations of serial combinators to return values to the task that spawned them. The dynamic links for many currently executing functions may point to the same activation record.

Multiprocessor serial combinator reduction subsumes all the functionality of uniprocessor graph reduction. Thus, *every* function in the partitioned program will be a serial combinator and every serial combinator will generate a node in the graph, whether or not explicit synchronization or lazy evaluation is required. In chapter 5 a new model of graph reduction is discussed that lifts this requirement. A detailed description of how multiprocessor graph reduction is actually implemented can be found in chapters 6 and 8.

4.2.2 Constructs for creating tasks and synchronization

Refined supercombinators, like functions in ALFL, are represented in a standard parse-tree form call LIF, for *Lambda Intermediate Form.*[1] The representation of serial combinators, however, must make the evaluation order of their bodies explicit. Serial combinators are represented in *Serial Combinator Intermediate Form* or SCIF. SCIF is a parse tree representation that includes constructs for creating tasks, synchronizing tasks, and initiating the evaluation of delayed expressions.

These added constructs are the *spawn*, *wait*, and *demand* constructs. For ease of explanation, we will represent serial combinators using S-expression syntax much like that of Lisp. Each construct is described below.

The demand construct

The *demand construct* has the form

(demand ($v_1 \ldots v_n$)

[1] LIF is very similar to another intermediate representation for functional programs called FLIC [59].

body)

and indicates that the values of variables $v_1 \ldots v_n$ should be demanded in parallel. The evaluation of *body* begins as soon as the values of $v_1 \ldots v_n$ have been demanded and does *not* wait for the values to return. Because serial combinators preserve the termination properties of lazy evaluation, we must be certain, using strictness analysis, that the values of $v_1 \ldots v_n$ will be needed at some point in the computation.

Demanding the value of a variable may create a task to evaluate a delayed expression. Thus, the order in which variables occur within a demand construct is significant. Using the heuristics of section 4.1, the variables are ordered according to the complexity of the expressions that they are bound to (if that can be determined) and when their values are needed within *body*.

The wait construct

A *wait construct* has the form

(`wait` ($v_1 \ldots v_n$)
body)

and indicates that the values of $v_1 \ldots v_n$ must be available before the evaluation of *body* can begin. If the values of $v_1 \ldots v_n$ are still being computed by remote processors then the evaluation of the serial combinator is suspended. Evaluation is resumed when the needed values have returned. Each of $v_1 \ldots v_n$ must have already been demanded or spawned (see below). Although the evaluation of a serial combinator call may be suspended, the local processor is free to execute any other available task. The order in which the variables occur in a wait construct is irrelevant.

The spawn construct

The *spawn construct* has the form

(`spawn` ((v_1 exp_1) ... (v_n exp_n))
body)

and specifies that each expression exp_i should be evaluated by creating a new task, along with a corresponding node in the program graph. Every function call

that creates a node in the graph must be an invocation of a serial combinator. Therefore, each exp_i must be a serial combinator call. When exp_i becomes evaluated, the value returned by the corresponding task is bound to the variable v_i. Since the evaluation of exp_i has already commenced, v_i should not occur in a demand construct at any point in *body*. The evaluation of *body* proceeds without blocking on the values of $v_1 \ldots v_n$ and thus each v_i must occur within a wait construct when its value is needed.

The spawn construct is the *only* way to specify the creation of a node. Therefore all serial combinator applications must occur within a spawn construct. If the compiler determines that one of the serial combinator applications in the spawn construct should be evaluated locally (based on the analysis and heuristics of section 3.4), the corresponding task will be executed on the local processor without invoking the dynamic scheduler. In this case, the variable-expression pair in the spawn construct is marked "local". An example of the use of the spawn and wait constructs is:

```
(spawn (((v1 (f x y)) (local v2 (g x y))))
   (wait (v1 v2)
     (+ v1 v2)))
```

The order in which variable-expression pairs occur in a spawn construct is significant and indicates the order in which the corresponding tasks are created. The order of the variable-expression pairs is determined using the heuristics of section 4.1. Expressions within a spawn construct may contain references to variables bound within the same spawn construct.

In the above example, it may seem strange that the variable `v2`, which was computed locally, occurs in a wait construct. The evaluation of `(g x y)` may suspend, and it is possible for the value of `v1` to become available before the value of `v2`. If this happens, we do not want the evaluation of `(+ v1 + v2)` to begin when `v1` arrives, but rather to wait until the values of both variables are available.

The let construct

A final construct that is of less interest but is still useful is the *let construct.* It has the form

(let ((v_1 exp_1) ... (v_n exp_n)
  body)

and is similar to spawn with the exception that each exp_i is an expression that can be executed immediately as part of the current task on the local processor *without* invoking graph reduction. This means that exp_i is restricted to an application of a primitive operator that requires no activation record. Each exp_i will be evaluated in a conventional manner (using registers, temporary locations, etc.) on the local processor.

The evaluation of *body* continues as soon as the values have been computed. Naturally, no v_i need appear in a demand or a wait. The let construct is primarily used to show that certain arguments in a serial combinator application are not worth evaluating in parallel but should still be evaluated immediately. An example of the use of the let construct is

```
(let ((v1 (+ x y)) (v2 (* y z)))
   (spawn ((local v3 (f v1 v2)))
      (wait (v3)
        (+ v3 2))))
```

The order in which variable-expression pairs occur in a let construct is irrelevant since all the expressions will be evaluated sequentially.

It is worth noting that the constructs described above can be viewed of as *annotations* to expressions written in the lambda-calculus (or some equivalent functional notation). As such, these constructs could be provided to allow a programmer to specify explicitly the desired parallel behavior of his program. Research into *para-functional programming* [30,35,31] has explored the possibility of including these kinds of annotations in functional languages.

4.2.3 The placement of spawns and demands

In section 4.3, we present the algorithm for translating refined supercombinators into serial combinators. In this section, we discuss the placement of the synchronization constructs in a serial combinator.

To maximize the amount of useful parallelism, the serial combinator invocations should be spawned as soon as it is known that their values are needed. Likewise, variables bound to non-strict arguments should be demanded as soon as it is known that their values are needed. For the spawns and demands to occur as soon as possible during execution, the spawn and demand constructs in the SCIF tree representation of each serial combinator definition must occur as *high* in the SCIF tree as possible. That is to say that each expression spawned and each variable demanded is lifted, or *hoisted*, up the tree from where it originally occurred to the first place in which it can be determined that it will be needed.

This is a different issue from the lifting of expressions out of tasks, described in section 3.5. Since a serial combinator specifies the behavior of a single task, lifting an expression within a serial combinator body does not move the expression out of the task in which it would ordinarily occur.

An expression is first determined to be safely evaluable at at one of two places:

1. *At the branches of a conditional*: Before the predicate of a conditional is evaluated, those expressions or variables that occurred in either branch (but not both branches) of the conditional cannot be spawned or demanded. Once the predicate has been evaluated, all serial combinator calls whose values will be needed in the appropriate branch can immediately be spawned. For example, suppose the following expression comprises the body of a function in LIF:

   ```
   (f x y) -> (g x y) + (h x y), 1
   ```

   where `f`, `g`, and `h` are all sufficiently complex to occur in spawn constructs and the values of `x` and `y` are available. The corresponding SCIF expression could be (ignoring the need for a wait construct),

   ```
   (spawn ((local v1 (f x y)))
      (if v1 (spawn ((v2 (g x y)) (local v3 (h x y)))
                 (+ v2 v3))
             1))
   ```

   Notice that the calls to `g` and `h` cannot be hoisted any higher without being premature.

2. *At the top of serial combinator definitions*: If there are serial combinator applications that will always be evaluated within a serial combinator body then these applications should occur in a spawn construct at the top of the serial combinator definition. Likewise any variable that may be bound to a delayed expression and will eventually be referenced should occur in a demand construct at the top of the serial combinator definition. In the previous example, the spawn construct containing the call to `f` occurred at the top of the serial combinator body.

4.2.4 The placement of waits

In order to minimize the time that tasks spend waiting, each serial combinator should accomplish as much as possible before encountering a wait construct. Therefore wait constructs should occur as *low as possible* in the SCIF tree for each serial combinator. In contrast to spawns and demands, a wait construct containing a set of variables should therefore be hoisted as *little* as possible above the place where the variables are first referenced.

The most obvious place for a wait construct is *immediately* before a variable reference, where it would contain only that variable. Suppose that the variable reference occurs within an arithmetic expression (or any other expression that does not utilize graph reduction). Such an expression should be evaluated using the registers and stack of the local processor to hold the operands and intermediate values. If the expression

```
(+ (* x y) (- z w))
```
(4.1)

occurs within the body of a function and and only the values of `x` and `y` are known to be available then `z` and `w` will have to occur in a wait construct. A straightforward translation of the expression into SCIF would be

```
(+ (* x y) (- (wait (z) z) (wait (w) w)))
```

However, there are two problems with this SCIF expression:

1. *Intermediate values*: If evaluation of the expression has to suspend until the value of `z` or `w` has arrived then the value of `(* x y)` may already have been computed and would need to be saved. There are several ways to accomplish the saving of intermediate values during suspension:

- The activation record for the serial combinator containing the expression is made large enough to contain all intermediate values that could possibly exist during suspension. This could result in a significant waste of space.
- An area of memory is set aside for storing intermediate values during suspension. As the execution of each serial combinator invocation is resumed, the corresponding intermediate values have to be located and restored into the registers. The cost of maintaining such a mechanism could be significant.

In either case, the occurrence of a wait in an arithmetic expression (or any other primitive operation) adds a significant overhead to the context switch during suspension.

2. *Multiple Suspensions*: After the value of `z` had returned, the task evaluating the expression would resume executing. However, before any useful computation could be performed, the task would *again* be suspended if the value of `w` had not yet arrived. Therefore, the execution of the task would have been suspended and resumed *twice* even though resuming execution after the value of `z` returned provided no benefit.

A better translation of expression 4.1 into SCIF would be

```
(wait (z w) (+ (* x y) (- z w)))                    (4.2)
```

Once the evaluation of the arithmetic part of expression 4.2 begins, all the needed values are available and execution will not have to suspend. This means that no intermediate value will have to be preserved during a suspension. In addition, only one wait construct (and thus only one suspend/resume) is sufficient to evaluate expression 4.2.

In general the execution times of expressions involving only primitive operations are short. Therefore, the lifting of a wait construct out of such an expression would not cause a task to wait much earlier than necessary.

The only places a wait construct can occur without creating the problem of storing intermediate values during suspension is at the top of the body in a serial combinator definition and at a conditional. For example,

```
f x y -> g x, h y
```

can be translated into SCIF as follows:

```
(spawn ((local v1 (f x y)))
  (wait (v1)
     (if v1 (spawn ((local v2 (g x)))
                 (wait (v2)
                    v2))
            (spawn ((local v3 (h y)))
                    (wait (v3)
                       v3))) ))
```

Not all conditionals should be allowed to contain wait constructs, however. If a conditional expression is nested within another expression, a wait construct might create the problem of storing intermediate values. For example, if

```
(* (+ x y) (if w z 2))
```

has to block on the value of z, a translation of the expression into

```
(wait (x y w)
   (* (+ x y)
      (if w (wait (z)
                  z)
            2)))
```

would require storing the value of (+ x y) during suspension. One solution would be to transform the original expression such that the conditional is not nested:

```
w -> (x+y)*z, (x+y)*2
```

The SCIF version would be

```
(wait (x y w)
  (if w (wait (z) (* (+ x y) z))
        (* (+ x y) 2)))
```

However, such a transformation can only occur if the conditional will always be executed (i.e the expression is strict with respect to the conditional).

The other alternative is to treat a nested conditional expression as an invocation of the serial combinator IF, defined as follows:

```
IF p c a == (wait (p)
               (if p (demand (c)
                         (wait (c)
                           c))
                     (demand (a)
                         (wait (a)
                           a))
```

The SCIF version of the previous example would be

```
(spawn ((local v1 (IF w z 1)))
  (wait (v1 x y)
     (* (+ x y) v1)))
```

A wait construct may occur within a conditional expression that is nested within another conditional expression. For example,

```
(if x 1 (if y 2 3))
```

can safely be translated into

```
(wait (x)
   (if x 1
         (demand (y)
             (wait (y)
                (if y 2 3)))))
```

We are guaranteed that no intermediate value will be generated by the predicate of the outermost conditional.

For the rest of this chapter, we will assume that all conditionals that should not contain wait constructs have already been translated into calls to the IF combinator. This will simplify our presentation of the translation of refined supercombinators into serial combinators.

4.2.5 The creation of delayed expressions

In graph reduction, the delayed evaluation of an expression is represented by a node in the program graph. The node serves as a closure, containing all the information that will be needed when the expression is ready to be evaluated.

We have not yet described how the creation of nodes representing the delayed evaluation of expressions is specified in serial combinators. The evaluation of an expression is delayed when the expression occurs as a non-strict argument in a serial combinator application. As mentioned in section 4.2.2, every serial

combinator application must occur in a spawn construct. The arguments in these applications are assumed to be non-strict unless they are simply variables that have already been bound to expressions in spawn, demand, or let constructs. Given the expression

```
f (g 1) 2
```

if f is not strict in its first argument, the SCIF version of the above expression would be

```
(spawn ((local v1 (f (g 1) 2)))
   (wait (v1)
      v1))
```

Since (g 1) was not explicitly spawned, a node in the graph is created to represent its delayed evaluation. If the value of the corresponding bound variable in the body of f is demanded then evaluation of (g 1) will commence.

If, however, f is strict in its first argument (and g is sufficiently complex) then the SCIF version of the above expression would be

```
(spawn ((v1 (g 1)) (local v2 (f v1 2)))
   (wait (v2)
      v2))
```

In this case, we can see that the evaluation of (g 1) has already started by the time f is called. In chapter 6 we describe the mechanism for passing unevaluated (or currently evaluating) arguments to serial combinators.

Lazy creation of delayed expressions

In his dissertation, Hughes [37] describes an optimization that reduces the cost of representing delayed expressions by insuring that at most one node is created for each non-strict argument in a function invocation. In conventional graph reduction, several nodes may be created to represent a non-strict argument. For example, in the serial combinator expression

```
(spawn ((v1 (f (g (h x 1) (h y 2)) 3)))
    v1)
```

where f is not strict in its first argument, the straightforward way to create a delayed expression for (g (h x 1) (h x 1) y) would be to create a node representing a delayed invocation of g and two other nodes representing the delayed invocations of h. Figure 4.1 illustrates how three nodes are used to represent

the delayed expression `(g (h x 1) (h y 2))`. We call this optimization *lazy*

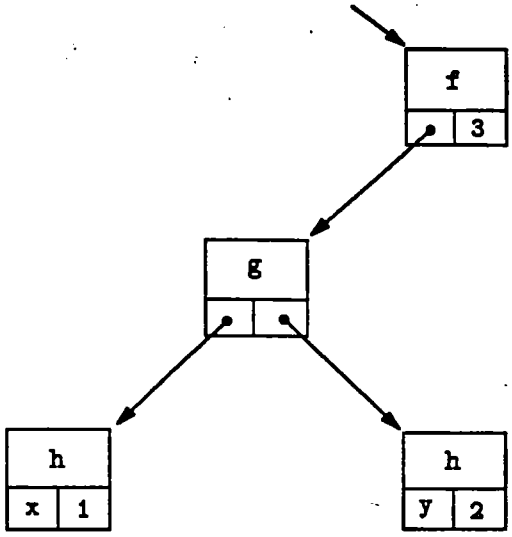


Figure 4.1: Multiple nodes representing a delayed expression

creation.[2] A new serial combinator `foo` is defined as follows:

```
foo x y == g (h x 1) (h y 2);
```

Our original expression becomes `(f (foo x y) 3)`.

Only one node is needed to represent the delayed expression `(foo x y)` as shown in figure 4.2. Only if the value of `(g (h x 1) (h y 2))` is needed will

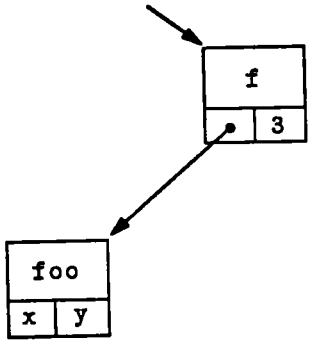


Figure 4.2: A single node representing an arbitrarily large delayed expression

[2]Hughes left it unnamed.

the subgraph pictured in figure 4.1 be constructed. In general, a subgraph representing a delayed expression may be arbitrarily large. Thus lazy creation can provide a significant savings.

4.3 Serial Combinator Generation

4.3.1 The Serialize algorithm

The *serialize* algorithm transforms an LIF expression into an SCIF expression, taking an LIF expression e and returning a tuple:

$$\langle e',\ s,\ w,\ d,\ l,\ C \rangle$$

where

- e' is an SCIF expression which may contain spawn, wait, demand, and let constructs.
- s is a set of variable-expression pairs that should occur in a spawn construct before e' is evaluated.
- w is a set of variables that should occur in a wait construct before e' is evaluated.
- d is a set of variables that should occur in a demand construct before e' is evaluated.
- l is a set of variable-expression pairs that should occur in a let construct before e' is evaluated.
- C is a set of new serial combinator definitions that were generated by *serialize*.

We define *serialize* later in this section.

The *serialize_prog* algorithm transforms a program—represented as a set of refined supercombinators—into a set of serial combinators. Here is the definition of *serialize_prog*(P) where P is a refined supercombinator program:

1. Let $P = \{\ F_1\ x_{11} \ldots x_{1k_1} == e_1;$
   $\ldots$
   $F_n\ x_{n1} \ldots x_{nk_n} == e_n;$
   `result` $e\ \}$
2. For each i, $1 \leq i \leq n$, let $\langle e_i',\ s_i,\ w_i,\ d_i,\ l_i,\ C_i \rangle = serialize(e_i)$
3. Let $\langle e',\ s,\ w,\ d,\ l,\ C \rangle = serialize(e)$
4. For each i, $1 \leq i \leq n$, redefine each F_i to be a serial combinator:

$$F_i\ x_{i1} \ldots x_{ik_1} == order\_constructs(e_i', s_i, w_i, d_i, l_i)$$

   where *order_constructs* creates an expression that contains the necessary spawn, wait, demand, and let constructs for s_i, w_i, d_i, and l_i, respectively. The body of the expression (after these constructs) is e_i'. *Order_constructs* is defined later in this section.

5. Let C_P be the set containing the definition of each serial combinator F_i generated in the previous step. In other words, C_p contains the serial combinator version of each function defined in the source (or refined supercombinator) program.
6. Let $e'' = order\_constructs(e', s, w, d, l)$
7. The serial combinator version of the program P consists of the set of new combinator definitions described by

$$C_p \cup \bigcup_{i=1}^{n} C_i$$

   and the result expression e''.

In simple terms, *serialize* decomposes a function application or binary expression as follows:

1. For each strict argument of sufficient complexity, *serialize* creates a serial combinator definition and puts a call to that serial combinator into a spawn construct.
2. Each strict argument that is not worth spawning is placed in a let construct.

3. For each non-strict argument, a new serial combinator definition is created and a call to that serial combinator is substituted into the expression. Since this call is not explicitly spawned, a node in the graph is created to represent its delayed evaluation (see section 4.2.5).

Here is the formal definition of *serialize*. It takes an expression e and performs the following actions:

- If e is a constant c then return $\langle c, \{\}, \{\}, \{\}, \{\}, \{\}\rangle$
- If e is a bound variable x then return $\langle x, \{\}, \{x\}, \{x\}, \{\}, \{\}\rangle$
- If e is a conditional (`if` e_1 e_2 e_3) then:

  1. Let $\langle e_1', s_1, w_1, d_1, l_1, C_1\rangle = serialize(e_1)$
     $\langle e_2', s_2, w_2, d_2, l_2, C_2\rangle = serialize(e_2)$
     $\langle e_3', s_3, w_3, d_3, l_3, C_3\rangle = serialize(e_3)$
  2. Let $d' = d_1 \cup (d_2 \cap d_3)$
  3. Let $e' =$ (`if` e_1'
     $order\_constructs(e_2', s_2, (w_2 - w_1), (d_2 - d'), l_2)$
     $order\_constructs(e_3', s_3, (w_3 - w_1), (d_3 - d'), l_3)$)
  4. Return $\langle e', s_1, w_1, d', l_1, (C_1 \cup C_2 \cup C_3)\rangle$

- If e is a strict binary operation, say (`+` e_1 e_2), perform the following steps. We assume, using the complexity measure T defined in chapter 3, that $T(e_1) \leq T(e_2)$. Otherwise substitute e_1 for e_2 and vice versa.

  1. Let $\langle e_1', s_1, w_1, d_1, l_1, C_1\rangle = serialize(e_1)$
     $\langle e_2', s_2, w_2, d_2, l_2, C_2\rangle = serialize(e_2)$
  2. If

     $$T(e_2) \leq T(e_1) + C_{lat} \text{ and } C_{loc} + C_{lat} \leq T(e_2))$$

     or

     $$T(e_1) + C_{lat} \leq T(e_2) \text{ and } C_{loc} \leq T(e_1)$$

     then e_1 can be evaluated in parallel with e_2.[3] Therefore:

     (a) Create a new identifier V

[3]These are just inequalities 3.7 and 3.8 from chapter 3.

(b) If task lifting (see section 3.5) is *not* performed then

i. Let $e_1'' = order\_constructs(e_1',\ s_1,\ w_1,\ d_1,\ l_1)$. Thus e_1'' consists of the expression e_1' preceded by the spawns contained in s_1, the waits contained in w_1, and so on.

ii. Define a new combinator F_1 as follows:

$$F_1\ a_1 \ldots a_n = e_1''$$

where $a_1 \ldots a_n$ are the free variables in e_1'. Let C be the singleton set containing this new combinator definition.

iii. Let $e' = (+\ V\ e_2')$

iv. Let p be the variable-expression pair $(V\ \ (F_1\ a_1 \ldots a_n))$.

v. Return $\langle e',\ (s_2 \cup \{p\}),\ (w_2 \cup \{V\}),\ d_2,\ l_2,\ C_1 \cup C_2 \cup C\rangle$

(c) If task lifting *is* performed then

i. Let w be the set of variables in w_1 that occur free in e_1' but not in any expression in l_1. Let $w' = w_1 - w$. The value of each element of w' is needed in either s_1 or l_1. Therefore w' must also be lifted out of e_1.

ii. Define a new combinator F_1 as follows:

$$F_1\ b_1 \ldots b_m = (\texttt{wait}\ w\ e_1')$$

where $b_1 \ldots b_m$ are the free variables in e_1' including the variables bound in s_1 and l_1. Let C be the singleton set containing this new combinator definition.

iii. Let $e' = (+\ V\ e_2')$

iv. Let p be the variable-expression pair $(V\ \ (F_1\ b_1 \ldots b_m))$.

v. Return $\langle e',\ (s_1 \cup s_2 \cup \{p\}), (w' \cup w_2 \cup \{V\}),\ (d_1 \cup d_2),\ (l_1 \cup l_2),\ (C_1 \cup C_2 \cup C)\rangle$

3. Otherwise, $(+\ e_1\ e_2)$ should not be decomposed. Return

$$\langle (+\ e_1'\ e_2'),\ (s_1 \cup s_2),\ (w_1 \cup w_2),\ (d_1 \cup d_2),\ (l_1 \cup l_2),\ (C_1 \cup C_2)\rangle$$

- If e is an application $(e_0\ e_1 \ldots e_n)$ then

1. For each i, $0 \leq i \leq n$, let $\langle e'_i,\ s_i,\ w_i,\ d_i,\ l_i,\ C_i \rangle = serialize(e_i)$
2. Define the following sets:

$$S = \{i \mid e_0 \text{ is strict in its } i\text{th argument}\}$$
$$P = \{i \mid i \in S \text{ and } T(e_i) < C_{loc}\}$$
$$Q = \{j \mid j \in S \text{ and } T(e_j) \geq C_{loc}\}$$
$$R = \{k \mid k \notin S,\ k \leq n\}$$

3. Let V_i be a new identifier for each $i \in S$ and let V also be a new identifier.
4. If lifting is *not* performed then:
   (a) For each $j \in (Q \cup R)$, let $e''_j = order\_constructs(e'_j, s_j, w_j, d_j, l_j)$ and define a new combinator F_j:

$$F_j\ a_{j1} \ldots a_{jn_j} = e''_j$$

   where $a_{j1} \ldots a_{jn_j}$ are the free variables in e_j. Let C be the set of all the new combinator definitions.
   (b) Let $l = \{(v_j\ e'_j) \mid j \in P\}$
   (c) For each $j \in Q$, let p_j be the variable-expression pair $(v_j\ (F_j\ a_{j1} \ldots a_{jn_j}))$. Also, let p be the variable-expression pair $(V\ (v_0\ x_1 \ldots x_n))$ where, for $1 \leq i \leq n$,

$$x_i = \begin{cases} v_i & \text{if } i \in S \\ (F_i\ a_{i1} \ldots a_{in_i}) & \text{if } i \in R \end{cases}$$

   (d) Let $s = \{p_j \mid j \in Q\} \cup \{p\}$
   (e) Let $w = \bigcup_{j \in P} w_j$, and let $d = \bigcup_{j \in P} d_j$.
   (f) Return $\langle V,\ s,\ w,\ d,\ l,\ (\bigcup_{i=1}^{n} C_i) \cup C \rangle$
5. Otherwise, if lifting *is* performed then:
   (a) For each $j \in Q$ let w'_j be the set of variables in w_j that occur free in e'_j but not in any expression in l_j. Also, for each $j \in Q$ define a new combinator F_j:

$$F_j\ b_{j1} \ldots b_{jm_j} = (\texttt{wait}\ w'_j\ e'_j)$$

where $b_{j1} \ldots b_{jm_j}$ are the free variables in e'_j. Let C_Q be the set of these new combinator definitions.

(b) For each $j \in R$, let $e''_j = order\_constructs(e'_j,\ s_j,\ w_j,\ d_j,\ l_j)$ and define a new combinator F_j:

$$F_j\ a_{j1} \ldots a_{jn_j} = e''_j$$

where $a_{j1} \ldots a_{jn_j}$ are the free variables in e_j. Let C_R be the set of these new combinator definitions.

(c) Let $l = \{(v_j\ \ e_j) \mid j \in P\} \cup (\bigcup_{j \in S} l_s)$

(d) For each $j \in Q$, let p_j be the variable-expression pair

$$(v_j\ \ (F_j\ \ b_{j1} \ldots b_{jm_j}))$$

Also, let p be the variable-expression pair

$$(V\ \ (v_0\ \ x_1 \ldots x_n))$$

where, for $1 \leq i \leq n$,

$$x_i = \begin{cases} v_i & \text{if } i \in S \\ (F_i\ \ a_{i1} \ldots a_{in_i}) & \text{if } i \in R \end{cases}$$

(e) Let $s = \{p_j \mid j \in Q\} \cup (\bigcup_{i \in S} s_i)\ \cup p$

(f) Let $w = (\bigcup_{j \in P} w_j)\ \cup\ (\bigcup_{i \in Q}(w_i - w'_i))\ \cup\ \{V\}$ and let $d = \bigcup_{j \in S} d_j$.

(g) Return $\langle V,\ s,\ w,\ d,\ l,\ (\bigcup_{i=1}^{n} C_i) \cup C_Q \cup C_R \rangle$

4.3.2 Order_constructs and Top_sort

Order_constructs is defined as follows:

$$order\_constructs(e,\ s,\ w,\ d,\ l) = \ \texttt{(demand}\ d\ \ top\_sort(e, s, w, l))$$

It creates a demand construct containing the variables in d. The body of the demand construct is an SCIF expression generated by *top_sort*.

Top_sort takes a list of spawns, waits, and lets and creates an SCIF expression with the appropriate constructs. Since expressions in a let construct may reference variables bound in a spawn construct and vice versa, the let, spawn, and wait constructs have to be (topologically) sorted so that variables are bound before they are used.

Top_sort(e, s, w, l) behaves as follows:

- If $s = \{\}$ and $l = \{\}$ then let $e' =$ (`wait` w e) and return e'.
- Otherwise, define the following sets

$$
\begin{aligned}
s' = \{(v_i\ exp_i) \mid\ & (v_i\ exp_i) \in s \text{ and } exp_i \text{ contains no} \\
& \text{occurrence of a variable bound in } s\} \\
l' = \{(v_j\ exp_j) \mid\ & (v_j\ exp_j) \in l \text{ and } v_j \text{ occurs free in an} \\
& \text{expression in } s'\} \\
w' = \{v_k \mid\ & v_k \in w \text{ and } v_k \text{ occurs free in an} \\
& \text{expression in } l'\}
\end{aligned}
$$

- Let e' = top_sort(e, (s-s)', (w-w'), (l-l')) and return

```
(wait w'
   (let l'
      (spawn order_spawns(s', e')
         e'))))
```

The procedure *order_spawns*(s', e') arranges the variable-expression pairs in s' in the order in which the variables are referenced in e' (see section 4.1.2).

Consider the following example of the use of *order_constructs*. If

$s = \{$`(v1 (f x y)) (v2 (g v4 2)) (v3 (f 2 3))`$\}$
$l = \{$`(v4 (+ x y))`$\}$
$w = \{$`v1, v2, v3, x`$\}$
$d = \{$`x`$\}$

then *order_constructs*$(s, w, d, l,$ `v5`$)$ will return

```
(demand (x)
  (spawn ((v1 (f x y)) (v3 (f 2 3)))
     (wait (x)
       (let ((v4 (+ x y)))
          (spawn ((v2 (g v4 2)))
             (wait (v1 v2 v3)
                (+ v1 (* v3 v2)))))))
```

4.3.3 The Clean-up Phase

A serial combinator expression generated by *serialize* may contain redundant spawn or wait constructs. For example, the LIF expression

```
(if (= x 1)
    (if (= y 1) (+ x y) 1)
    2)
```

would be translated into

```
(demand (x)
   (wait (x)
      (if (= x 1)
          (demand (y)
             (wait (y)
               (if (= y 1)
                   (demand (x)
                      (wait (x)
                         (+ x y)))
                   1)))
           2)))
```

The last phase, called the *clean-up phase*, of the translation of refined super-combinators into serial combinators is a tree walk over the serial combinator bodies (generated by *serialize*) to remove redundant demands and waits. It is a simple pre-order tree walk and the details are left to the reader. The cleaned-up version of the above serial combinator expression would be

```
(demand (x)
   (wait (x)
      (if (= x 1)
          (demand (y)
             (wait (y)
               (if (= y 1)
                   (+ x y)))
                   1)))
           2)))
```

The clean-up phase also attaches the label `local` to a variable-expression pair in a spawn construct if appropriate. Any spawn construct satisfying the following conditions will have its first variable-expression pair modified with the `local`.

- The spawn construct is immediately followed by a wait construct.

- The first variable bound in the spawn construct occurs in the wait construct.

Since evaluation will have to suspend until the first expression in the spawn construct becomes evaluated, that expression should be executed locally.

For example, the LIF expression `(+ (f x y) (g x y))` would be translated by *serialize* into

```
(spawn ((v1 (f x y)) (v2 (g x y)))
   (wait (v1 v2)
      (+ v1 v2)))
```

This expression would be translated by the clean-up phase into

```
(spawn ((local v1 (f x y)) (v2 (g x y)))
   (wait (v1 v2)
      (+ v1 v2)))
```

If the first variable bound in the spawn construct does not occur in the wait list, no variable bound in the spawn construct can occur in the wait list. This is because the first variable bound in the spawn construct is the first variable whose value is required in the body (see *order_spawns*).

4.4 Examples

Here are two examples of the translation from ALFL programs into Serial Combinators. The serial combinator code is the actual output of the compiler (before code-generation).

The first example is a divide and conquer factorial program:

```
{ pfac l h == l=h->l, { pfac1 mid == pfac l mid +
                                    pfac (mid + 1) h;
                        result pfac1} ((l+h)/2);
  result pfac 1 10;
  }
```

The only place where significant amount of parallelism occurs is in the body of pfac1 in which pfac is invoked twice in parallel.

The refined supercombinator version is:

```
{ pfac l h == l = h -> l, pfac1 l h ((l + h) / 2);
  pfac1 l h mid == pfac l mid + pfac (mid + 1) h;
  result pfac 1 10
  }
```

and the serial combinator version is:

```
{ pfac l h == (demand (l h )
                (wait (l h )
                  (if (= l h)
                     l
                     (let ((v1 (/ (+ l h)  2) ) )
                       (spawn ((local v2 (pfac1 l h v1) ) )
                         (wait (v2)
                            v2))))))


 pfac1 l h mid == (demand (mid l h)
                    (wait (mid)
                      (let ((v4 (+ mid 1)))
                        (spawn ((v15 (pfac v4 h))
                                (local v3 (pfac l mid)))
                          (wait (v3 v5)
                            (+ v3 v5))))))


 result pfac 1 10
 }
```

Even though pfac contains a spawn construct, the spawned expression is evaluated locally and thus no attempt is made at exploiting (non-existent)

parallelism. Only in the body of pfac1 is parallelism exploited by spawning two expressions simultaneously. One of the spawned expressions is evaluated locally.

The second example is Quicksort. The ALFL version is:

```
{ qs L ==
   L=[] -> [],
           { split X acc ==
              X=[]->acc,
                    (hd X) < (hd L) ->
                       split (tl X) [(hd X)^(hd acc),
                                     hd (tl acc))],
                       split (tl X) [(hd acc),
                                     (hd X)^(hd (tl acc))];
            result { qs1 res ==
                       qs (hd res) ^^
                       ((hd L) ^ qs:(hd (tl res)));
                     result qs1; } (split (tl L) [[],[]]);
            };
 result qs [5, 1, 3, 0, 17, 34, 12];
}
```

in which it is assumed that the operators hd, tl, and ^(the cons operator) are strict in all their arguments and can be accomplished without invoking graph reduction. In addition, ^^(the append operator) is assumed to use the strict version of ^.[4]

The refined supercombinator version of this program is:

```
{ qs L == l=[] -> [],  qs1 L (split L (tl L) [[],[]]);
   qs1 L res == qs (hd res)  ^^  ((hd L) ^ qs (hd (tl res)))
   split L X acc ==
      X=[] -> acc,
              hd X < hd L ->
                 split L (tl X) [(hd X)^(hd acc), hd (tl acc)]
                 split L (tl X) [(hd acc), (hd X)^(hd (tl acc))];
   result qs [5, 1, 3, 0, 17, 34, 12];
  }
```

and the serial combinator version is

[4]The strict versions of cons and append are nonstandard in lazy functional languages. In order to exploit the parallelism in the above manner, either a strictness analysis for lists [39,76] or user-supplied information is required.

```
{ qs L ==
    (demand (L)
     (wait (L)
      (if (= L [])
          []
          (let ((v5 (tl L)) (v6 (^ [] (^ [] []))))
             (spawn ((v8 (split L v5 v6))
                     (local v9 (qs1 L v8)) )
                  (wait (v9)
                     v9))))))


  qs1 L res ==
   (demand (res L)
     (wait (res)
       (let ((v10 (hd res)))
           (spawn ((v11 (qs v10)))
              (let ((v13 (hd (tl res))))
                 (spawn ((v14 (qs v13)))
                    (wait (L)
                      (let ((v12 (hd L)))
                        (wait (v14)
                           (let ((v15 (^ v12 v14)))
                              (spawn ((local v16 (append v11 v15)))
                                 (wait (v16)
                                    v16)))))))))))
```

```
{ qs L ==
    (demand (L)
     (wait (L)
      (if (= L [])
          []
          (let ((v5 (tl L)) (v6 (^ [] (^ [] []))))
             (spawn ((v8 (split L v5 v6))
                     (local v9 (qs1 L v8)) )
                  (wait (v9)
                     v9))))))


  qs1 L res ==
   (demand (res L)
     (wait (res)
       (let ((v10 (hd res)))
           (spawn ((v11 (qs v10)))
              (let ((v13 (hd (tl res))))
                 (spawn ((v14 (qs v13)))
                    (wait (L)
                      (let ((v12 (hd L)))
                        (wait (v14)
                           (let ((v15 (^ v12 v14)))
                              (spawn ((local v16 (append v11 v15)))
                                 (wait (v16)
                                    v16)))))))))))
```

```
split L X acc ==
  (demand (X acc)
    (wait (X)
      (if (= X [])
          (wait (acc )
             acc)
          (demand (L)
            (wait (L)
              (let ((v18 (hd X)) (v19 (hd L)))
                 (if (< v18 v19)
                     (wait (acc)
                        (let ((v20 (tl X))
                              (v21 (^ (^ (hd X) (hd acc))
                                      (^ (hd (tl acc)) []))))
                          (spawn ((local v22 (split L v20 v21)))
                            (wait (v22 )
                              v22))))
                     (wait (acc)
                       (let ((v23 (tl X))
                             (v24 (^ (hd acc)
                                     (^ (^ (hd X)
                                           (hd (tl acc))) []))))
                          (spawn ((local v25 (split L v23 v24)))
                             (wait (v25)
                                v25)))))))))))


  result (let ((v26 (^ 5 (^ 1 (^ 3 (^ 0 (^ 17
                           (^ 34 (^ 12 [])))))))))
           (spawn ((local v27 (qs v26)))
              (wait (v26)
                 v26)))
  }
```

Since the function `split` is practically sequential, all spawns in its body are local. Most of the useful parallelism is exploited in the body of `qs1`.

Chapter 5

A Heterogeneous Graph Reduction Model

In the previous chapter, we stated that every function call in the program had to be an invocation of a serial combinator and had to create a node in the program graph. This is because a node is the only kind of activation record available in graph reduction. For many serial combinator invocations, however, the full power of graph reduction—supporting lazy evaluation, sharing, and parallelism—is not needed.

In this chapter we describe a *heterogeneous evaluation model* that incorporates both graph reduction and conventional stack-based evaluation. A modification to the compiler, and an extension to the SCIF representation of serial combinators, is presented in order to target the compiler to this evaluation strategy.

5.1 Motivation

The graph reduction model, while extremely powerful and general, fails to exploit a particular *strength* of current multiprocessor architectures: The hardware and instruction set of each processor have been optimized for the execution of sequential programs written in first-order, call-by-value programming languages. The organization of these machines is centered around the use of registers for performing primitive operations on data and the use of a stack to

provide a mechanism for executing procedure calls. Although it may be argued that we would be better off designing processors specifically targeted for graph-oriented evaluation (as in [11,17,51]) our desire is to build a system that can exploit the strengths of the current machines.

We have already seen that many parts of a functional program may be practically sequential. It is also the case that in some of the practically sequential expressions, applicative order evaluation preserves the termination properties of normal order evaluation. Any expression that exhibits both these properties can therefore be efficiently executed in a conventional manner, utilizing only the stack and registers of the host processor (although higher-order functions may require the creation of closures in a heap).

The serial combinator execution strategy described in chapter 4 was not a completely pure graph reduction model. Expressions, such as arithmetic operations, involving only primitive (strict) operators are evaluated without graph reduction (and were thus placed in let constructs). Instead, they are evaluated using a sequence of hardware-supported instructions to move data into registers and perform primitive operations.

Each serial combinator call causes the creation of a new node in the program graph to serve as an activation record for that call. Even if the serial combinator call is to be evaluated sequentially, graph reduction is required to evaluate the call. Using graph reduction to evaluate a sequential function call incurs overhead costs not usually incurred by conventional stack-based evaluation. The overhead results from:

1. *Storage Management*: The use of expensive techniques for allocating and reclaiming activation records in a graph.

2. *Graph Manipulation*: The transformation of the graph by manipulating arcs and updating nodes.

If graph reduction is not required to evaluate a particular serial combinator invocation, the total execution time can be reduced by allocating the activation record on a stack. The serial combinator would be evaluated by executing a sequence of conventional stack-oriented instructions.

5.2 Heterogeneous Evaluation Model

We would like to minimize overhead by using mechanisms for parallelism and lazy evaluation only when necessary. In doing so, our evaluation strategy becomes becomes a hybrid of three evaluation models, which have varying degrees of power and accompanying overhead. Listed in decreasing order of power and overhead cost, these models are:

1. Parallel Graph Reduction
2. Sequential Graph Reduction
3. Sequential Stack-based evaluation

In chapter 4 we described the use of the first two models in serial combinator execution but not the least powerful (and least expensive) model.

Graph reduction provides two services that conventional stack-based execution does not, namely heap (graph space) allocation of closures and a return mechanism for parallel function applications. Heap allocation of closures is necessary to support lazy evaluation via the construction of delayed expressions. Closures are also constructed to represent partial function applications (i.e. higher-order functions).

A serial combinator invocation may require a value that is being computed on a remote processor. Therefore, the system must provide a mechanism for suspending evaluation of the serial combinator until the needed value is available. Activation records must be created in a heap, not on the stack, in order to preserve the suspended state of the invocation. The processor must be free to use the stack for other purposes while the serial combinator call is suspended.

A serial combinator may call several other serial combinators in parallel. Because of the nondeterminism in the order in which parallel function invocations complete, there must be a mechanism to provide synchronization between functions that are returning values and the functions that called them. This is usually accomplished by maintaining a number of status bits that are modified when values return. The suspension and resumption of a function invocation depend on the state of these bits. The overhead involved in setting and testing the status bits is significant.

In many functional programs there are sequences of function calls that contain no parallelism; in these cases, heap allocation of activation records and the use of status bits for synchronization is unnecessary. The activation records for these function calls should be allocated on a stack. The stack provides a sequential, deterministic return mechanism at very low cost. A heap is still needed, however, to support lazy evaluation and higher order functions used in other parts of the program.

We say a function is *stack executable* if:

- It calls at most one function at a time; that is, it never makes several function calls in parallel.
- It never "forks" a function call. That is, it never proceeds without waiting for a value of a function call to return.
- It only calls functions that are themselves stack executable.

Determining if a function, f, is stack executable in a first order language involves solving a simple recursive set equation. In the higher order case, f may make calls to "unknown" functions and a collecting interpretation (or some other sophisticated flow analysis) would be needed to determine if the unknown function is itself stack executable.

A complete application of a stack executable function can be executed on the stack only if all of the arguments have already been evaluated. If an application of a stack executable function contains an unevaluated argument, the activation record must be allocated in the heap since evaluation of the function call may have to be suspended.

Our compiler currently solves the set equation for stack executable functions in the *first order* case, giving up when it encounters an unknown function call. In the next section, we discuss how stack executable serial combinators are found and transformed.

5.2.1 Modifying Serial Combinators

After the compiler has generated a set of serial combinator definitions based solely on the graph reduction model, the SCIF representation of the serial combinators is modified for the use of the heterogeneous evaluation model. This

new representation for serial combinators is called *extended* SCIF. An extended SCIF parse tree, in addition to having the constructs described in section 4.2.2, may contain a *stack-spawn construct.* This construct is of the form,

(`stack-spawn` ($(v_1\ exp_1)$ ... $(v_n\ exp_n)$)
body)

and indicates that each expression exp_i, which is a serial combinator call, can be evaluated immediately on the local processor by creating an activation record for it on the stack. The value returned by this conventional evaluation of exp_i is bound to the variable v_i. The stack-spawn construct is similar to the let construct, with the exception that an activation record is required for each expression. Generally, there will be only a single variable-expression pair in a stack-spawn construct, since multiple pairs would indicate that several serial combinator calls are being evaluated sequentially.

Each exp_i in the stack-spawn construct represents a serial combinator call and must satisfy the conditions under which it can be evaluated using the stack. These conditions are

- that the serial combinator is stack executable, and
- that all arguments in the call have already been evaluated.

The first step in creating a new set of serial combinator definitions that can be executed on the stack is to determine which serial combinators are stack executable. In terms of a serial combinator f expressed in SCIF, the requirements stated in the previous section can be expressed as follows:

1. Every spawn construct within f contains exactly one variable-expression pair.
2. A wait construct immediately follows every spawn construct and contains the variable bound in the spawn construct.
3. The expression in each spawn construct must be a call to a stack executable serial combinator and all the arguments in the call must already be evaluated.

Any variable-expression pair in a spawn construct that satisfies the first two conditions will already have been modified with the word `local` by the clean-up tree walk described in section 4.3.3. Any occurrence of a spawn construct that

satisfies the first, second, and third conditions above is a *stack executable spawn*, whether or not it occurs within a serial combinator that is stack executable.

For any program represented by serial combinators, we would like to find the set S of stack executable serial combinators. Obviously, the third condition stated above requires the solution of a recursive set equation. In order for a function f to be an element of S, all functions called by f must be elements of S. Suppose $stack(f, S)$ indicates, given S, whether the function f satisfies the conditions stated above. If so, then

$$S = \{f \mid stack(f, S) = true\}$$

A fixpoint iteration method is used to solve for S. The initial set S^0 consists of all serial combinators in the program.

$$S^{i+1} = \{f \mid stack(f, S^i) = true\}$$

When a fixpoint is reached, i.e. $S^{j+1} = S^j$ for some value of j, then we have solved the set equation and $S = S^j$.

The definition of *stack* is

$$stack(f, S) = stack\_walk(body(f), S)$$

where $body(f)$ returns the body of the definition of f.

Stack_walk traverses the SCIF version of the *body* of f to determine if it is stack executable. The arguments to *stack_walk* are a node n in the $SCIF$ tree and the set of stack executable functions. It assumes that the formal parameters to f have already been evaluated. $Stack\_walk(n, S)$ behaves as follows:

1. If n represents a constant, a variable, or a binary operation then

   $$stack\_walk(n, S) = true$$

   According to the definition of serial combinators in chapter 4, none of these expressions can contain any synchronization constructs.

2. If n represents a conditional, `(if p c a)`, then

   $$stack\_walk(n, S) = stack\_walk(c, S) \wedge stack\_walk(a, S)$$

   According to the definition of serial combinators, the predicate p cannot contain any synchronization constructs.

3. If n is a let construct, (`let` $((v_1\ exp_1)\ \ldots\ (v_n\ exp_n))\ body$), or if n is a demand construct, (`demand` $((v_1 \ldots v_n)\ body$), then

$$stack\_walk(n, S) = stack\_walk(body, S)$$

Neither a let construct nor a demand construct affects whether a function is stack executable. This is because neither construct creates any unevaluated expressions or executes any expressions in parallel.

4. If n is a spawn construct of the form:

(`spawn` $((v\ (g\ e_1 \ldots e_k))$
(`wait` $(v_1 \ldots v_n)$
$body))$

then

$$\begin{aligned} stack\_walk(n, S) = \ & (v \in \{v_1, \ldots, v_n\}) \wedge \\ & (g \in S) \wedge \\ & (\text{each } e_i \text{ is an identifier}) \wedge \\ & stack\_walk(body, S) \end{aligned}$$

In this case, if some e_i were not an identifier then it would represent a delayed serial combinator application.

5. If n is a spawn construct that is not of the above form then

$$stack\_walk(n, S) = false$$

6. If n is a wait construct of the form, (`wait` $(v_1 \ldots v_n)\ body$), then

$$stack\_walk(n, S) = stack\_walk(body, S)$$

since each v_i in the wait construct is assumed to have been evaluated.

For each stack executable function f, *two* definitions are generated. The first definition specifies the behavior of the combinator when a call is to be evaluated using graph reduction. This is necessary in case some of the arguments in a call represent either a delayed expression or an expression currently being evaluated. In either case, synchronization constructs such as demand and wait have to occur within the body of f in order to use the values of these arguments.

The second definition of f specifies its behavior when executed on the stack. Since all arguments to f are assumed to be evaluated no demand construct is needed in f. Additionally, since all expressions in spawn constructs must be calls to other stack executable serial combinators, all spawns are converted to stack-spawn constructs, and thus no wait construct is needed in f.

Every serial combinator in the program, whether stack executable or not, is transformed into extended SCIF so that calls to stack executable combinators may be, if appropriate, executed on the stack. Regardless of whether a combinator f is stack executable, any stack executable spawn within the body of f can be converted to a stack-spawn.

As a simple example, consider the factorial function (written in ALFL):

```
fac x == x=0 -> 1, x * fac (x-1);
```

The SCIF version of `fac` would be:

```
fac x == (demand (x)
             (wait (x)
                (if (= x 0)
                    1
                    (let ((v1 (- x 1)))
                       (spawn ((local v2 (fac v1)))
                          (wait (v2)
                             (* x v2)))))))
```

Since the spawn in the body of `fac` is a stack executable spawn and `fac` is stack executable, the two definitions for `fac` in extended SCIF are:

```
g_fac x == (demand (x)
               (wait (x)
                  (if (= x 0)
                      1
                      (let ((v1 (- x 1)))
                         (stack-spawn ((v2 (s_fac v1)))
                             (* x v2)))))))

s_fac x == if (= x 0)
               1
               (let ((v1 (- x 1)))
                  (stack-spawn ((v2 (s_fac v1)))
                      (* x v2)))
```

`g_fac` is the version that utilizes graph reduction and `s_fac` is the stack exe-

cutable version. Any call to `fac` with arguments that are either unevaluated or currently evaluating must be a call to `g_fac`. Within `g_fac` the argument to the recursive call is already evaluated, thus `g_fac` calls `s_fac`.

5.3 Higher Order Functions and Closures

We have already discussed the difficulty in deciding whether a higher order serial combinator is stack executable. Our compiler assumes that a call to an unknown function in the body of a higher order serial combinator is not stack executable. However, if a combinator *returns* a function as its value, the combinator may still be stack executable.

When a function is returned as a value, a closure must be created to contain the environment in which the body function will eventually be executed. In graph reduction, this closure is represented by a node in the program graph, in the same way that activation records are. If a stack executable combinator call returns a function as its value, the activation record for the combinator will be created on the stack. The node representing the result of the call will be created in the graph.

5.4 Tail Recursion

Even if a serial combinator invocation is not stack-executable, the compiler may be able to avoid allocating a new node to represent its activation record. If the value that a function f returns is simply the result of a call to a function g, then the activation record for a call to f can be overwritten with the activation record for the call to g. This is the familiar *tail recursion* optimization, which saves heap space as well as the time required to allocate and reclaim an extra node.

Tail recursion is expressed in extended SCIF by a *tail-spawn construct*, which is of the form,

(`tail-spawn` *exp*)

where *exp* is a serial combinator application. This means that the current activation record can be overwritten with the activation record for *exp*. The

evaluation of *exp* proceeds immediately.

A spawn construct can be converted to a tail-spawn construct if it is of the form

```
(spawn ((v exp))
    (wait (v)
        v))
```

Such a spawn construct can be converted to a stack-spawn construct containing *exp*.

Consider the definition of `f` in the following program:

```
{ f x y ==  g x = h y -> 1, f (x+1) (y+1);
  g x == ...
  h x == ...
  result f 1 2;
 }
```

The SCIF version of `f` would be:

```
f x y == (spawn ((local v1 (g x)) (v2 (h y)))
             (wait (v1 v2)
                (if (= v1 v2)
                    1
                    (spawn ((local v3 (f (+ x 1) (+ y 1))))
                       (wait (v3)
                           v3))))
```

Since `f` is not stack executable, the best that the compiler can do is to convert the last spawn into a tail-spawn.

```
f x y == (spawn ((local v1 (g x)) (v2 (h y)))
             (wait (v1 v2)
                (if (= v1 v2)
                    1
                    (tail-spawn (f (+ x 1) (+ y 1))))))
```

5.5 An example

For an example of how a program, represented as a set of serial combinators in SCIF, is transformed into a new set of serial combinators in extended SCIF we refer to the quicksort example in section 4.4.

The SCIF version of the program is given on page 107. The extended SCIF version of quicksort, in which stack-spawns and tail-spawns occur, is given below. Notice that `split` is the only stack executable combinator in the program. The names of the combinators that are evaluated using graph reduction are prefixed by "g_". The names of the combinators that are executed on the stack are prefixed by "s_".

```
{
      --- Graph Reduction Functions ---
  g_qs L ==
    (demand (L)
       (wait (L)
          (if (= L [])
              []
              (let ((v5 (tl L)) (v6 (^ [] (^ [] [])))
                 (spawn ((v8 (g_split L v5 v6)))
                     (tail-spawn (g_qs1 L v8)))))))


  g_qs1 L res ==
     (demand (res L)
        (wait (res)
           (let ((v10 (hd res)))
             (spawn ((v11 (g_qs v10)))
                (let ((v13 (hd (tl res))))
                   (spawn ((v14 (g_qs v13)))
                      (wait (L)
                        (let ((v12 (hd L)))
                          (wait (v14)
                             (let ((v15 (^ v12 v14)))
                                (tail-spawn
                                  (g_append v11 v15)))))))))))))
```

```
g_split L X acc ==
  (demand (X acc)
     (wait (X)
        (if (= X [])
            (wait (acc)
               acc)
            (demand (L)
               (wait (L)
                 (let ((v18 (hd X)) (v19 (hd L)))
                   (if (< v18 v19)
                      (wait (acc)
                         (let ((v20 (tl X))
                               (v21 (^ (^ (hd X) (hd acc))
                                       (^ (hd (tl acc))
                                          []))))
                            (stack-spawn
                                 ((v22 (s_split L v20 v21)))
                               v22)))
                      (wait (acc)
                         (let ((v23 (tl X))
                               (v24 (^ (hd acc)
                                       (^ (^ (hd X)
                                             (hd (tl acc)))
                                          []))))
                            (stack-spawn
                                 ((v25 (s_split L v23 v24)))
                               v25)))))))))


     --- Stack Functions ---
s_split L X acc ==
     (if (= X [])
         acc
         (let ((v26 (hd X)) (v27 (hd L)))
             (if (< v26 v27)
                 (let ((v28 (tl X))
                       (v29 (^ (^ (hd X) (hd acc))
                               (^ (hd (tl acc)) []))))
                    (stack-spawn ((v30 (s_split L v28 v29)))
                       v30))
                 (let ((v31 (tl X))
                       (v32 (^ (hd acc)
                               (^ (^ (hd X) (hd (tl acc))) []))))
                    (stack-spawn ((v33 (s_split L v31 v32)))
                       v33)))))
```

```
  result (let ((v34 (^ 5 (^ 1 (^ 3 (^ 0 (^ 17
                          (^ 34 (^ 12 [])))))))))
           (tail-spawn (g_qs v34)))
}
```

Chapter 6

Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor

In the preceding chapters we described the translation of an ALFL program into a set of serial combinators that specify the behavior of parallel tasks. Up to this point, the compilation has been relatively architecture-independent. Only the communication costs C_{loc} and C_{lat} were considered when generating serial combinators. The final phase of compilation, *code generation* for serial combinators, must be targeted toward a particular architecture.

In this chapter we describe an implementation, called **Alfalfa**, of a heterogeneous graph reducer on the Intel iPSC hypercube multiprocessor and discuss the generation of code for this system. The code that is generated is a mix of

- conventional instructions using the registers and stack of each processor in the machine, and

- instructions for specifying manipulation of the graph. These instructions are routines provided by Alfalfa.

Before describing how code is generated from serial combinators, we describe how Alfalfa is implemented on the Intel iPSC.

6.1 The Intel iPSC

The Intel iPSC (for "intel Personal Super Computer") is a loosely-coupled MIMD multiprocessor that can be configured with up to 128 Intel 80286 microprocessors. Each processor has its own private memory, and there is no shared memory. The processors are linked via a hypercube network—each processor sits at a vertex of an n dimensional hypercube. In the iPSC, the value of n can vary between 0 and 7 depending on the number of available processors.

There are 2^n processors in an n dimensional hypercube. Each processor has n neighboring processors with which it can communicate directly. All communication between non-neighboring processors is routed through intermediate processors. The only means of communication in the iPSC is the passing of messages. The operating system provides the user with a few communication primitives such as send, blocking receive, and non-blocking receive [41].

On each processor board, there are n Ethernet chips, each of which is responsible for point to point communication with one of the processor's neighbors. The longest distance a message must travel within the iPSC is through n links (or "hops"). Message routing is performed by the operating system and is transparent to the programmer.

Unfortunately, the current version of the iPSC suffers from a large communication overhead, both in terms of message latency and of cost to a processor for sending a message.

Although the iPSC consists of a set of processors with disjoint memories, graph reduction requires the use of a single, global program graph. The Alfalfa system described below supports a global graph space via message passing. In chapter 8, the implementation of a heterogeneous graph reduction system on a shared-memory multiprocessor is described, in which message passing is not needed to implement the global graph.

6.2 The Alfalfa System

The Alfalfa system is responsible for distributing the program graph over the processors in the iPSC and performing the execution of serial combinator code.

It is replicated on each processor and is solely responsible for reducing the portion of the graph residing in the local store of that processor. The major components of the Alfalfa system, pictured in figure 6.1, are the graph reducer, message handler, dynamic scheduler, and storage manager.

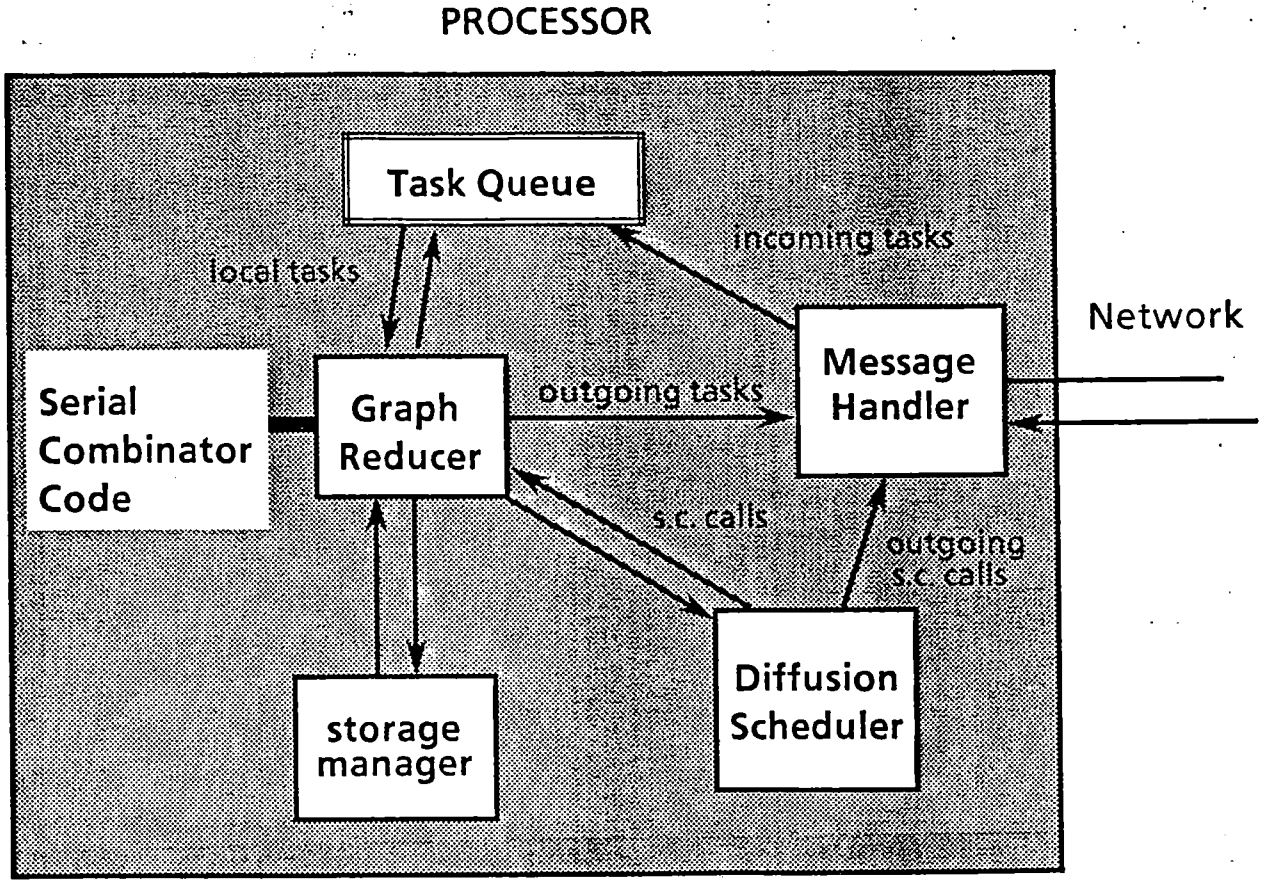


Figure 6.1: The Alfalfa system

Each component of the system provides routines that will be called by serial combinators to support the distributed execution of a program. Since we will be discussing code generation for Alfalfa in section 6.4, we describe the routines, written in C, that are called by the serial combinators.

6.2.1 Data Structures

The synchronization constructs (spawns, waits, etc.) that serial combinators contain are simply calls to routines in the graph reducer component that per-

form the necessary transformations on the graph. Before discussing how these operations are performed we describe the data structures involved.

Since ALFL is an untyped language (and no type inference is performed by the compiler) the Alfalfa system must provide run-time type checking. In order to do so, we provide each value with a bit indicating its type. In this system, the type bit is the rightmost (least significant) bit. A value can be one of two base types:

1. *Integer*: A value can be a four byte integer, in which case the rightmost bit of that value is 0. The actual value represented in such a way is the value of the four byte integer shifted to the right one bit. The advantage of using the rightmost bit as the type bit (as is the case in at least one Lisp system [53]) is that addition and subtraction can be performed directly on integers without modifying the type bits of the operands or result.[1]

2. *Extend*: An extend data type can itself be one of several (non-integer) types. The value of the extend is actually the address of a block of bytes (on the same processor) containing one of the following:

   - A floating point number
   - A "cons" cell
   - An array or vector
   - A closure representing a partial application

   For type checking, the rightmost bit of an extend is always set to 1. In order to use this value as the address of a block of data, the rightmost bit is zeroed before the address is used.

In either case, a value occupies four bytes and is defined to be of type CINT as follows:

```
typedef long int CINT;
```

A node in the graph is defined as follows:

```
typedef struct nodetype { char state;
                          valuecont value;
```

[1]Currently `true` and `false` are represented as integers. This is a loophole in the dynamic type checking system that will be corrected in future implementations.

```
    child args[MAXCHILDREN];
    pointer requests[MAXREQUESTS];
    unsigned requestcount : 8;
    bitfield waitmask;
    bitfield evalfield;
    char refcount;
}
node;
```

and is a block of bytes that contains the following fields:

- *State*: Either "unevaluated", "pending" (which means that the serial combinator invocation represented by the node is in the process of being evaluated), or "evaluated".

- *Value*: If the node has been evaluated then its value field contains the result. Otherwise the value field is a pointer to code that specifies the computation to be performed when the value of the node is requested. This data type is defined in C as follows:

```
typedef union valueconttype { CINT value;
                              int (*cont)();
                            }
                           valuecont;
```

- *Args*: A vector containing the arguments to the serial combinator invocation represented by the node. Each argument is either a base value or a pointer to another node in the graph. The value of any bound variable created in the body of the serial combinator (via a spawn or let construct) will also be contained in the args vector. Each element of the args vector is of type `child`, defined as follows:

```
typedef union childtype { pointer point;
                          CINT value;
                        } child;
```

- *Requests*: A list of other nodes that have requested the value of the node. Because of sharing, there may be several outstanding requests.

- *Evalfield*: A bitfield indicating the status of each element in the args vector. If the ith bit of this bitfield is 1, then the ith argument has already been evaluated, and the ith element of the args vector contains a value. Otherwise the ith argument is a pointer to another node representing a delayed or currently evaluating expression.

- *Waitmask*: A bitfield indicating which arguments must be evaluated before the evaluation of the node can proceed. The evaluation of a node must be suspended until for every 1 bit in the waitmask there is a corresponding 1 bit in the evalfield.

- *Refcount*: The reference count of the node, for storage reclamation purposes.

A pointer to a node in the graph, defined by

```
typedef struct pointertype { char pe;
                             char argnum;
                             struct nodetype *node;
                           } pointer;
```

has three fields:

- *Processing element* (pe): The address of the processor upon which the node resides. Since there are at most 128 processors in the iPSC, 8 bits are more than sufficient to represent a processor address.

- *Node*: The address of the node on its host processor. This is a full 32 bit address.

- *Argnum*: An index into the node's args vector. This is only relevant when the pointer is contained in a return-task (see below). Even if this field is unused in a pointer, there is no waste of space since there are 8 bits left over in the processor address field of the pointer (the 80286 requires that all structures be aligned at 16-bit word boundaries).

The C procedure call,

```
write_point(p, pe, n, argnum);
```

instructs that fields of the pointer `p` are to be updated with the processor address `pe`, the node address `n`, and the index `argnum`.

In figure 6.2 a node is illustrated with its various fields identified. This node

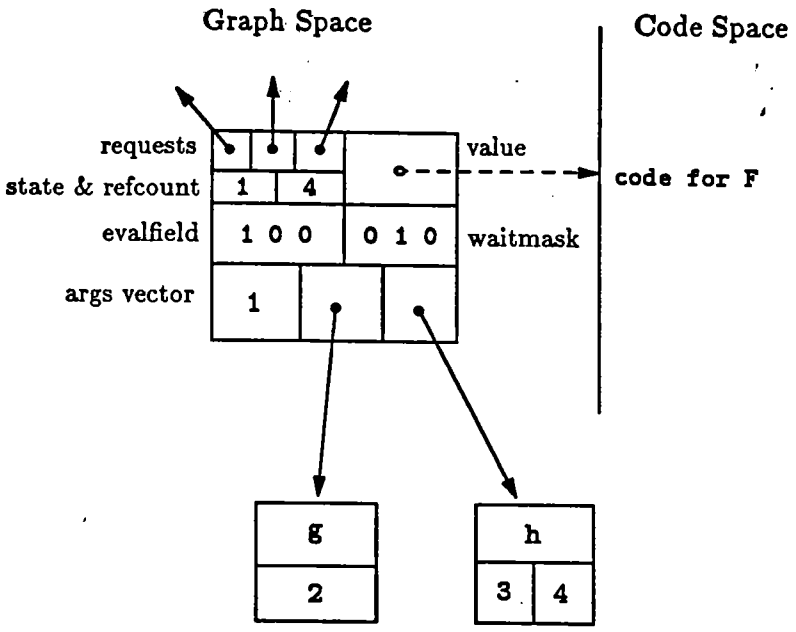


Figure 6.2: A node in the program graph

is created by a spawn construct,

```
(spawn ((v1 (f 1 (g 2))))
   ....)
```

where

```
f x y == (if (= x 1)
             (demand (y)
                (spawn ((v2 (h 3 4)))
                   (wait (y)
                      ...)
             ...)
```

The state of the node reflects the execution of `f` after the wait construct in the body of `f` has been executed.

A *task* is an instruction to the graph reducer to perform an operation on the program graph.[2] There are two kinds of tasks:

- An *eval-task* contains pointers to a target node and a source node. It

[2]In Alfalfa a task is a piece of data specifying some work to be done. This use of the word *task* is different from the use in previous chapters.

indicates that the value of the target node is being requested by the source node.

```
typedef struct evaltasktype { pointer targetnode;
                              pointer sourcenode;
                            } evaltask;
```

The sourcenode pointer serves as a *return pointer* to use when the value of the target node has been evaluated.

- A *return-task* contains a value and pointer to a node. It indicates that the value is being returned as the value of one of the arguments to the node. The pointer's argnum field indicates which element of the node's args vector is to be updated.

```
typedef struct returntasktype { pointer returnnode;
                                CINT value;
                              } returntask;
```

A task, being either a return-task or eval-task is defined as follows:

```
typedef struct tasktype { char type;
                          union { evaltask eval;
                                  returntask return;
                                } field;
                        } task;
```

A *task queue* resides on each processor in the system and contains a list of tasks to be performed. Program execution proceeds by repeatedly removing tasks from the task queue and performing the action specified. Each task in a processor's task queue will affect only nodes residing in the local store.

6.2.2 The Graph Reducer

The mechanisms in the graph reducer module that support the transformation of the program graph are described below.

Distribution of serial combinator and Alfalfa code

The C routines generated by the compiler from serial combinators are compiled by the iPSC's C compiler and linked with the compiled C routines of the Alfalfa system. The resulting executable image is then replicated onto every processor in the iPSC. This has two desirable results:

1. The code for each serial combinator is available on every processor. Thus, Alfalfa does not have to worry about migration of code during execution. On the iPSC, the executable image is broadcast to each processor before execution.

2. The image is loaded onto each processor at the same address. Therefore, an address of a serial combinator routine is consistent across all processors. This means that a closure can be passed from processor to processor without modifying its code pointer.

Graph Initialization

Before execution begins, a collection of nodes representing the initial graph is created. This initial graph corresponds to the `result` expression in the serial combinator version of the program and is allocated on a single "root" processor.

To start execution, an eval-task is created for the root node in the graph and placed in the task queue of the root processor. The graph reducer on *each processor* removes tasks, if present, from the local task queue. The code executed by each processor is (very roughly):

```
task_evaluation()
{ task *c;
  for(;;) {
     process_messages();
     c = pop(taskqueue);
     while (c == NIL) {
          process_messages();
          c = pop(taskqueue);
          }
```

```
    switch(c->type) {
       case EVALTASK:
          { node *n;
             n = c->field.evaltask.targetnode.node;
             eval_node(n, c->returnode);
             }
           break;

       case RETURNTASK:
       { node *n;
          int argnum;
          n = c->field.returntask.returnnode.node;
          argnum = (int) c->targetnode->argnum;
          n->args[argnum].value = c->value;
          n->evalfield = n->evalfield | (1 << argnum);
          possibly_resume(n);
          }
        break;
       }
    }
}
```

In C, | is the bitwise OR operator. Each of the routines used above will be defined shortly.

Evaluating a node

The evaluation of a node n commences when an eval-task whose target is n is encountered on the task queue. The evaluation proceeds as follows:

- If n's state is "unevaluated", then the code pointed to by n's value field is executed (this code is serial combinator code generated by the compiler).

  Before executing the serial combinator code, the node must be updated in the following way:

  - The *state* of the node is set to "pending".
  - The address of the node that requested the value of n is put in n's requests list. The address of the requesting node is contained in the sourcenode field of the eval-task.

- If n's state is "pending", then the address of the requesting node is placed in n's requests queue. Since n is already in the process of being evaluated, no further action is taken.
- If n's state is "evaluated", then the value in n's value field is immediately returned to the requesting node via a return-task (see below).

The routine `eval_node` is defined as follows:

```
task_evalnode(n, retpoint)
node *n;
pointer *retpoint;
{
  switch (n->state) {
    case UNEVALUATED:
          add_request(n,retpoint);
                   /* add the return pointer
                      to n's request list */
          n->state = PENDING;  /* modify the state */
          (*(n->value.cont))(n);  /* Jump to the code for n */
          break;

    case EVALUATED:
          return_value(retpoint, n->value.value);
                    /* return n's value to the
                       requesting node */
          break;

    case PENDING:
          add_request(n,retpoint);
                  /* just add the return pointer to
                     n's request list */
          break;
    }
}
```

Returning a value

When the serial combinator code for a node n has been executed, a return-task is created to return the resulting value v to each requesting node. In addition, n's state is changed to "evaluated" and its value field is overwritten with v.

```
return_value(point, val)
pointer *point;
CINT val;
{ n->value.value = val;
   n->state = EVALUATED;
   if (point->pe == localpe) then {
      /* the value is returned to a local node */
      task *c;
      c = allocate_task();
      c->type = RETURNTASK;
      c->field.returntask.returnnode = point->node;
      c->value = val;
      push(c, taskqueue);
      }

   else {
     /* invoke the message handler to send a message containing
        a return task to the appropriate processor */
    send_return(point, val);
    }
 }
```

Requesting a value

The C routine `getvalue` is used to demand the value of an argument to a node. The procedure call

```
getvalue(n, i);
```

instructs the graph reducer to demand the value of the `i`th argument of node `n`. If the `i`th bit of `n`'s evalfield is 1, no action is taken. Otherwise, an eval-task is created to demand the value of the node pointed to by the `i`th element of `n`'s args vector. Naturally, the eval-task is placed in the task queue of the processor on which the demanded node lies.

```
getvalue(n, argnum)
node *n;
int argnum;
{ if ((n->evalfield & (1 << argnum)) == 0) then
        create_eval_task(n->args[argnum], n, argnum);
  }
```

(In C, `&` is the bitwise `AND` operator.)

Suspending evaluation of a node

During the evaluation of n the value of one of its arguments may be needed (i.e. may have occurred in a wait construct) but not yet be available. That the argument is not available is indicated by the corresponding bit in n's evalfield being 0.

The value of the argument must have previously been demanded via a call to `getvalue` and the evaluation of n must suspend until the needed value returns. Before evaluation of n is suspended, its waitmask field is updated so that the bit corresponding to each needed argument is set to 1. Furthermore, the value field of n is updated to point to a *continuation* that will be executed when evaluation of n resumes. Once these two simple (single instruction) operations have been performed the graph reducer is free to execute the next task on the local task queue.

Updating and resuming a node

When the graph reducer encounters a return-task for n, it updates the appropriate element of n's args vector with the specified value. It then sets the corresponding bit in n's evalfield to 1. In order for the evaluation of n to resume, every 1 in n's waitmask must have a corresponding 1 in n's evalfield. If this condition is satisfied, then the evaluation of n is resumed by executing the code (continuation) pointed to by n's value field. The routine `possibly_resume` checks to see if the evaluation of a node can resume.

```
possibly_resume(n)
node *n;
{
  if (n->waitmask == (n->waitmask & n->evalfield)) then
   (*(n->value.cont))(n);
}
```

One of the key features of the Alfalfa system is that suspending and resuming of serial combinator invocations is very inexpensive.

Creating Subgraphs

The execution of a spawn construct in a serial combinator causes the creation of a new subgraph. The subgraph is allocated in the store of a processor chosen by the dynamic scheduler. An eval-task is created for the root node of the new subgraph and placed in the task queue of the processor on which the subgraph resides.

Because of *lazy creation* of delayed expressions (described in section 4.2.5, each subgraph representing a spawned expression has a height of at most two, consisting of a root node and possibly some nodes representing delayed arguments. None of the arguments themselves can have any descendants.

The serial combinator code may specify that a subgraph should be created locally. The code will contain a call to the storage allocator to allocate space in the local store for the new subgraph. How the various fields of the new nodes are initialized is also determined by the serial combinator code. The C statement,

```
n = local_new_node(f);
```

instructs the graph reducer to create a node on the local processor whose value is a pointer to the code for the C routine `f`. The variable `n` is assigned the address of the new node.

If the dynamic scheduler chooses to create a new subgraph on a remote processor, space for a *template* of the subgraph is allocated in the *outgoing message buffer* of the local processor. Once the fields of the nodes in the template subgraph have been initialized, the template serves as a complete *descriptor* of the actual subgraph that will be created on the remote processor. The C call,

```
n = new_node_in_buf(f)
```

instructs the graph reducer to create a node in the message buffer. The variable `n` is assigned the address, within the message buffer, of the new node. This address is used to update the fields of the new node.

Once the descriptor has been written into the buffer the procedure call

```
build_eval(newpe, localpe, n, argnum)
```

causes a message containing the descriptor to be sent to the processor `newpe`. When the message is received, the descriptor is copied into `newpe`'s graph space

and an eval-task for the root node of the new subgraph is placed in `newpe`'s task queue. The eval-task includes a return pointer that contains the processor `localpe`, the node `n`, and the index `argnum` into `n`'s args vector.

A small example

Given the serial combinator definitions,

```
f x y == (spawn ((v1 (g x)) (local v2 (h y)))
            (wait (v1 v2)
               (+ v1 v2)))


g x == (if (= x 1)
           2
           ( ... ))


h y == (if (= y 2)
           1
           ( ... ))
```

figure 6.3 illustrates the evaluation of a node, labeled **n1**, corresponding to the combinator expression `(f 1 2)`.

6.2.3 The Dynamic Scheduler

The dynamic scheduler is responsible for distributing the execution of a program throughout the processors of the iPSC in order to exploit the parallelism within the program. The algorithms used by the dynamic scheduler come under a class of algorithms known as *diffusion scheduling algorithms* and are described in detail in chapter 7. When a new subgraph is to be created the dynamic scheduler uses diffusion scheduling to decide where to allocate the subgraph. The most important aspect of diffusion scheduling is that a processor can only create a new subgraph on *neighbors*—processors that it can communicate with directly.

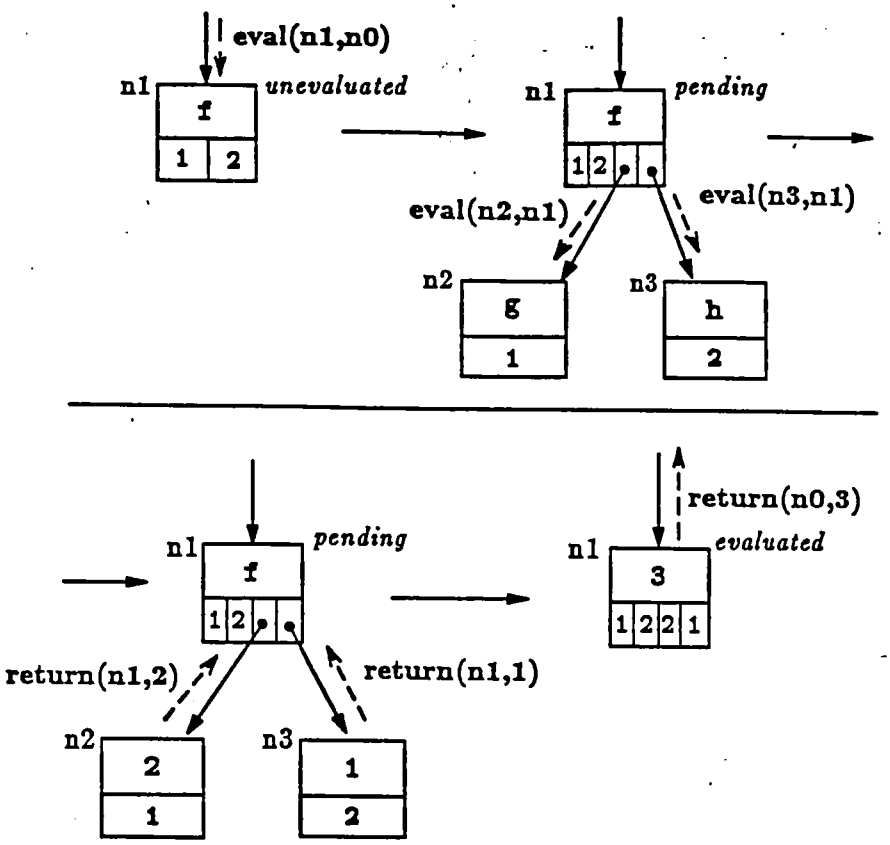


Figure 6.3: Example of serial combinator reduction in Alfalfa

6.2.4 Message Handler

The message handler provides the interface between each processor and the network. It creates messages that are required to support the actions of the graph reducer and dynamic scheduler.

Message passing support for graph operations

When a task is sent to a remote processor, it is encapsulated into a "reducer" message. This is necessary when:

1. A node needs the value of another node on a remote processor,or
2. A node becomes evaluated and returns its value to a node on a remote processor, or

3. A new subgraph is to be created on a remote processor. In this case the message contains a descriptor as described above. The descriptor is a block of bytes that corresponds precisely to the way the subgraph will be represented in the graph. When the message is received, each node in the descriptor is copied into the graph space of the receiving processor (at address supplied by the local storage allocator). In the descriptor, any pointer to a node in the new subgraph must contain a *relative* address since the absolute address of each new node was unknown when the descriptor was created. Therefore, after the descriptor has been copied into the graph space of the receiving processor all such pointers have to be relocated.

In each case, the graph reducer invokes routines in the message handler module to create the appropriate messages.

Message passing support for dynamic scheduling

The message handler provides the dynamic scheduler with the capability to make decisions based on the state of surrounding processors. Each processor must be able to report its local state to the surrounding processors. The message handler provides routines to create these processor-state messages (see chapter 7 for the details).

Incoming messages

Naturally, the message handler is also responsible for handling incoming messages. When a message is received, the message handler removes the message from the message buffer. It determines the type of the message (either a reducer message or a processor-state message) and makes the contents of the message available to the appropriate module (either the graph reducer or the dynamic scheduler).

Distributed reference counting also requires message passing to perform reference counting operations (such as increment and decrement) across processor boundaries. A third type of message (called a "storage" message) is used to perform these operations.

6.2.5 Storage Manager

The storage manager on each processor is responsible for allocating and reclaiming nodes in the local portion of the graph space. The storage reclamation scheme must maintain the integrity of the distributed graph while minimizing the amount of storage wasted.

In the Alfalfa system, free nodes are allocated from free-lists on each processor. Although no compaction of memory is performed, fragmentation of the address space does not significantly hinder the computation, because all nodes are the same size. When space for a node is required by the graph reducer, the first element of the free-list is used.

A separate free-list is kept for extends. When an extend is required, the free-list is traversed until the first free extend of sufficient size is found (this corresponds to the first-fit method for memory allocation).

Nodes that are no longer needed are reclaimed using a distributed reference counting scheme. Unfortunately, this scheme requires a significant amount of communication to maintain the integrity of the graph [29]. The advantage of distributed reference counting lies in its simplicity and that it requires no synchronization between processors. Most mark-and-sweep algorithms require *all* processors to halt while garbage collection is being performed.

Using constant size nodes creates the potential for a significant waste of space; serial combinator invocations represented by these nodes may have a widely varying number of arguments. The most straightforward implementation would use nodes with args vectors that are large enough to hold any possible serial combinator invocation. Alfalfa's approach is to allocate nodes with args vectors that are large enough to accommodate most, but not all, serial combinator calls. Generally an args vector with a handful of elements is sufficient. For those serial combinator invocations that require a larger args vector, we use an `extend` to hold additional arguments (and an additional evalfield and waitmask). This situation is pictured in figure 6.4.

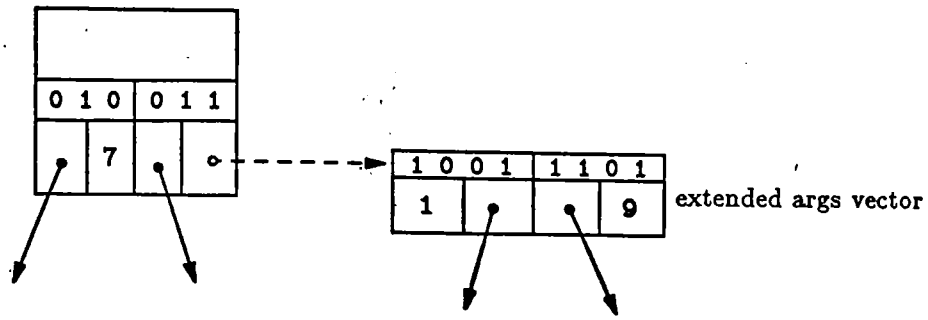


Figure 6.4: Using an extend to increase the size of the args vector

6.3 Address sharing for vertical parallelism

Suppose the following serial combinator expression is being evaluated on processor p_0.

```
(spawn ((v1 (f 1 2)) (v2 (g v1 2)))
    (wait (v2)
       (+ v2 1)))
```

Notice that the argument `v1` in the application of g in the spawn construct is bound to `(f 1 2)` in the same spawn construct. In this case vertical parallelism is being exploited, since the expression `(f 1 2)` is being evaluated at the same time as `(g v1 2)`. While it may be desirable to be able to exploit such vertical parallelism, it creates an implementation problem on a loosely-coupled multiprocessor.

A node n representing the call `(f 1 2)` is created. In order for g to be able to refer to its first argument, it must be passed the address of n. The problem lies in making the address of n available to g. If n is created on a remote processor p_1, then n's address is not immediately available to p_0. Figure 6.5 illustrates this situation. An obvious, brute force solution to this problem is for p_1 to send a message specifying n's address back to p_0. Only after this message is received by p_0 can `(g v1 2)` be spawned. The drawback to this approach is that an extra message must be sent by the p_1 and that the evaluation of `(g v1 2)` is delayed. This is clearly an obstacle to exploiting the vertical parallelism between `(f 1 2)` and `(g v1 2)`.

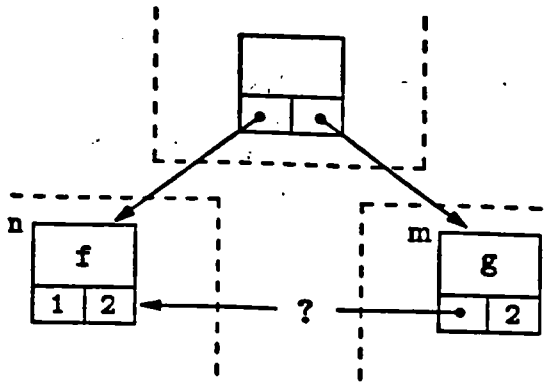


Figure 6.5: Problem in determining the address of a node created remotely

A more sophisticated and probably more efficient solution would be for p_0 to create a symbolic address for n and to pass that address to g. This symbolic address would have to be mapped to the real address for n by p_1 (using a hash table or some other method). The drawback to this approach is the need to perform the mapping each time a symbolic address is used by a remote processor. If much sharing of symbolic addresses occurs, the mapping costs may become significant. If future distributed architectures include associative memories for performing such mapping (analogous to a translation look-aside buffer), this method may be a reasonable solution.

The solution used by Alfalfa is called *address sharing.*[3] In this method, a processor p reserves a block of free nodes for exclusive use by each of its neighbors. It also provides each neighbor with a list of the addresses of these free nodes. Thus, a neighbor is guaranteed to be able to create a node on p at each of those addresses.

In the above example, p_0 can specify the address of n on p_1. The message that p_0 sends to p_1 includes the address in p_1's memory where n should be allocated. Thus, g can be passed the real address for n without any delay. This particular technique of providing the address of a remote argument in a combinator application is called *forward address sharing.*

[3]A simplified version of this idea was initially suggested by Simon Peyton Jones during an informal conversation.

In order to support address sharing, each processor p must make sure that its neighbors have a supply of addresses of free nodes in p's local memory. It does so by keeping a count of how many nodes in its memory are reserved by each neighboring processor p_i. Each time p_i instructs p to create a node at a specified address, p decrements the count for p_i. When that count gets below a certain threshold, p invokes its local storage allocator to reserve a new set of nodes for use by p_i and sends p_i a list of the addresses of these nodes. Often, the message containing a new list of addresses can be appended to another message being sent to p_i. The threshold at which p sends p_i a new supply of addresses must be large enough to ensure that p_i does not run out before it receives the new supply.

Another problem arises in implementing vertical parallelism. In the previous example, suppose that the invocation of `g` is evaluated on processor p_2. When the node n representing `(f 1 2)` becomes evaluated, the resulting value needs to find its way to p_2 to be used by `g`.

The most obvious way to accomplish this would be for `g` to generate an eval-task for n. A message containing the eval-task would be sent by p_2 to p_1. Even though p_0 had already sent a message to p_1 specifying the creation and evaluation of n, *another* message must be sent in order for the value of n to be used. This situation is pictured in figure 6.6.

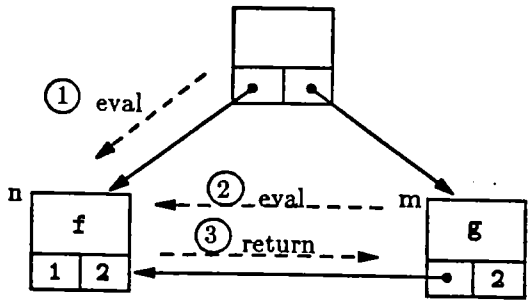


Figure 6.6: Three messages are required for `g` to get `f`'s value

A more efficient solution uses address sharing in the following way: When p_0 sends the message to p_1 specifying the creation of n and the generation of an eval-task for n, the *return pointer* in that eval-task should not point to the

parent node on p_0, but rather to the node m representing `(g v1 2)` on p_2. Address sharing is needed to determine the address of m to use as the return pointer. In the above example, the parent node on p_0 does not need the value of `(f 1 2)`. Using address sharing, at least one message is saved. Figure 6.7 shows how only two messages are required for `g` to get the value of the call to `f`. The technique of using address sharing to provide a return address in an

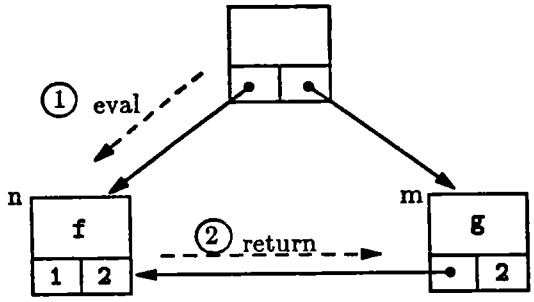


Figure 6.7: Vertical parallelism using address sharing

eval-task is called *backward address sharing.*

There is a synchronization problem that arises, although rarely, when backward address sharing is used. This synchronization problem is similar to the problems that arise in distributed reference counting [29]. In our example, it is possible for the evaluation of `(f 1 2)` to complete and for the result to be sent, ostensibly, to the node m representing `(g v1 2)` *before* m is actually created. In other words, the message containing the return-task destined for m arrives before the message specifying the creation of m. This situation is illustrated in figure 6.8. In this case, the first element of the args vector of an *empty* node will be updated. When m is finally created, that element may be incorrectly overwritten.

The solution to this synchronization problem is rather simple. We previously defined three states that a node could be in: "unevaluated", "pending", and "evaluated". Since two bits are required to represent the state field in a node, a fourth state can be included without additional cost. This fourth state is "empty" and indicates that the node is unused. Therefore, when a return-task is encountered that points to an empty node, the return-task is put back onto

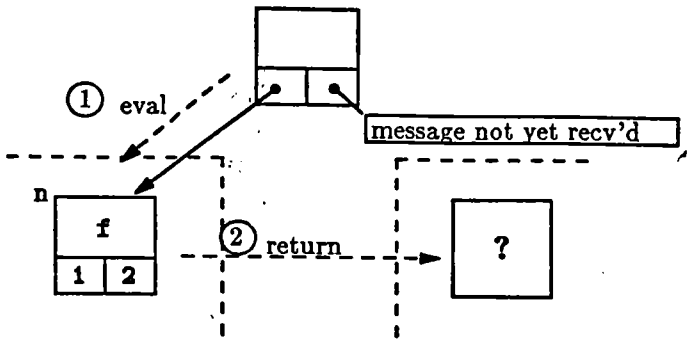


Figure 6.8: Synchronization problem in address sharing

the end of the task queue. In order to implement this solution, when any node is reclaimed, the state field of the node must be reinitialized to "empty".

A similar problem arises in forward address sharing when a message containing an eval-task for a node reaches a processor before the message specifying the creation of that node. When an eval-task is encountered for an empty node, the eval-task is also put back into the task queue.

Although forward address sharing is implemented in the current implementation of Alfalfa, backward sharing is not. In future versions, backward sharing will certainly be implemented.

In a serial combinator expression in which only horizontal parallelism is exploited no address sharing is required. For example, if a serial combinator whose invocation is represented by a node n contains the expression,

```
(spawn ((v1 (f 1 2)) (v2 (g 3 4)))
   (wait (v1 v2)
       (+ v1 v2)))
```

the local processor does not need to know the address of either of the nodes created for the invocations of `f` and `g`. If these nodes are created on remote processors, the message sent to each remote processor instructs it to create a node at any address it chooses and to return the value of the node to n.

6.3.1 Forward address sharing in serial combinators

Before code is generated for each serial combinator, the forward address sharing required for each expression in a spawn construct is made explicit. In order to

indicate that a subgraph representing an expression in a spawn construct should be created using address sharing, each spawn construct in a serial combinator is modified so that each variable-expression pair requiring address sharing is modified by the word `address`. For example, the expression,

```
(spawn ((v1 (f 1 2)) (v2 (g v1 2)))
    ...)
```

is transformed into

```
(spawn ((address v1 (f 1 2)) (v2 (g v1 2)))
    ...)
```

To determine which expressions should be spawned using address sharing, a tree walk is performed on the SCIF tree for each serial combinator. For each variable-expression pair in a spawn construct that is not modified by `local`, the following action is taken: If the variable in the variable-expression pair is referred to either in the same spawn construct or in the body of the spawn construct *before* the variable occurs in a wait construct, the variable-expression pair is modified by `address`.

Address sharing routines in Alfalfa

The C procedure `get_address(newpe)` returns the address of a free node on the remote processor `newpe`. The C call,

```
build_eval_address(newpe, address, localpe, n, argnum);
```

causes the message handler to send a message to processor `newpe` that includes:

- the descriptor of a new subgraph (already written into the outgoing message buffer),
- the address to be used for the root node of the new subgraph, and
- a return pointer to the processor `localpe`, the node `n`, and the index `argnum` into `n`'s args vector.

It is analogous to the routine `build_eval` described in section 6.2.2 except that the message contains the address of the root node.

6.4 Code Generation for Alfalfa

Now that we have described the Alfalfa system, we discuss how serial combinators are translated into code that can be executed on Alfalfa. Two important points about the code generation phase are:

- The code generated by the compiler is C, not native machine code. The C code is then compiled by the iPSC's C compiler. This approach facilitated the timely development of a research prototype like Alfalfa.
- All graph reduction and multiprocessing routines provided by the Alfalfa system are invoked by the serial combinator code only when needed. They can be thought of as library routines. Therefore, any segment of serial combinator code that requires no graph reduction or multiprocessing support will be identical to the code that would be generated for an applicative order, sequential program.

In order to discuss the important aspects of code generation for serial combinators in a reasonable amount of space, we are forced to omit an immense amount of detail about the code generation phase (and the Alfalfa system in general). Some of the detail that is omitted concerns generating code to support:

- dynamic type checking
- extends for vectors and floating point numbers,
- closures representing partial function applications
- distributed reference counting
- error detection and recovery
- statistics gathering

Instead, we concentrate on how code is generated for the various synchronization constructs that occur in serial combinators.

6.4.1 Code generation for graph reducible combinators

Since the activation record for each invocation of a *graph reducible* combinator (i.e. a combinator that is not *stack executable*) is a node in graph, the C

routine corresponding to each serial combinator takes a single argument. This argument is the address of the node representing the activation record. The location of each bound variable in the serial combinator expression is specified by an offset into the args vector of the corresponding node.

In section 6.2.2 we described how resuming a suspended node required that the address of a continuation be stored in the node's value field. This continuation specifies the behavior of the node when it resumes. Each continuation, like the top-level C routine representing a serial combinator, takes the address of the node as an argument.

The first stage of code generation for a serial combinator is to break the combinator definition down into the set of continuations it represents. Each wait construct in the serial combinator indicates that a continuation must be generated to represent the behavior of the rest of the combinator. Therefore, each wait construct in the serial combinator is modified so that the body of the wait construct specifies the *name* of the continuation to be called when the needed values return. The continuation itself will be compiled into a C routine. For example, in the serial combinator definition

```
f x y ==  (spawn ((local v1 (g x y)) (v2 (g 3 4)))
              (wait (v1)
                 (let ((v3 (+ v1 5)))
                   (wait (v2)
                      (+ v2 v3)))
```

the expression after each wait construct must be converted into a continuation. The definition now is represented as a list that contains the top-level expression and the definitions of the continuations.

```
f x y == [ (spawn ((local v1 (g x y)) (v2 (g 3 4)))
              (wait (v1)
                 fcont1)),
           fcont1: (let ((v3 (+ v1 5)))
                     (wait (v2)
                        fcont2)),
           fcont2: (+ v2 v3) ]
```

The serial combinator definition is now a list that includes the top expression and some continuation definitions. As mentioned above, the C routines corresponding to `f`, `fcont1`, and `fcont2` will all take the same argument, namely

the address of the node representing the activation of `f`.

The compiler uses the procedure *C_combinator* to generate code for a combinator definition in which the continuations have been made explicit. The procedure $emit(e_1, \ldots, e_n)$ writes its arguments, in left-to-right order, into some output file.

$$
\begin{aligned}
&C\_combinator(f\ x_0 \ldots x_n\ ==\ [body, c_1 : exp_1, \ldots, c_k : exp_k]) = \\
&\qquad emit(f,\ \text{“(n)”}) \\
&\qquad emit(\text{“}\texttt{node *n; \{}\text{”}) \\
&\qquad \text{let}\ \ sym\_table = [\langle 0, uneval\rangle/x_0, \ldots, \langle n, uneval\rangle/x_n] \\
&\qquad\qquad cont\_table = [exp_1/c_1, \ldots, exp_k/c_k] \\
&\qquad\qquad \{\langle c'_1, sym\_table_1\rangle, \ldots \langle c'_m, sym\_table_m\rangle\} = \\
&\qquad\qquad\qquad C\_body(exp, sym\_table) \\
&\qquad emit(\text{“}\}\text{”}) \\
&\qquad C\_contin(cont\_table, \langle c'_1, sym\_table_1\rangle, \ldots, \langle c'_m, sym\_table_m\rangle)
\end{aligned}
$$

The symbol table *sym_table* is used to keep track of the index into the args vector for each bound variable and the state of the each variable (unevaluated or evaluated). Initially, the state of each bound variable (formal parameter) in the symbol table is unknown, and the compiler assumes that it is unevaluated. In addition to creating this symbol table, *C_combinator* emits the header for a definition of a C routine that takes a single argument `n` of type `(node *)`. A table *cont_table* associates each continuation name with its definition.

The routine *C_body* (which we will define shortly) generates code for the expression representing the body of a combinator or continuation. Initially, *C_body* is called on the top level expression in a combinator definition. Since the top level expression may end with a wait construct, *C_body* returns the names of the continuations that could possibly be called next. There may be several possibilities depending on how the conditionals in the top level expression are resolved during execution. A symbol table is associated with each continuation returned by *C_body* and provides the necessary information about bound variables when generating code for the continuation. The procedure that

directs the code generation for each continuation is *C_contin*.

$$
\begin{array}{l}
C\_contin(cont\_table, \langle c_1, sym\_table_1\rangle, \ldots, \langle c_m, sym\_table_m\rangle) = \\
\quad \text{for } \; i = 1 \ldots m \\
\qquad emit(c_1, \text{ ``(n)''}) \\
\qquad emit(\text{``}\texttt{node *n; \{}\text{''}) \\
\qquad \text{let } \; body_i = cont\_table[\![c_i]\!] \\
\qquad\quad \{\langle c_{i1}, sym\_table_{i1}\rangle, \ldots \langle c_{in}, sym\_table_{in}\rangle\} = \\
\qquad\qquad\qquad C\_body(body_i, sym\_table_i) \\
\qquad emit(\text{``}\}\text{''}) \\
\qquad C\_contin(cont\_table, \langle c_{i1}, sym\_table_{i1}\rangle, \ldots, \langle c'_{im}, sym\_table_{im}\rangle)
\end{array}
$$

Since the compilation of the body of each continuation by *C_body* may return the names of other continuations, *C_contin* calls itself recursively to generate the code for those continuations.

We now define the procedure *C_body*, which generates code for the body of each combinator and continuation. It takes two arguments, an expression and a symbol table describing the bound variables in the expression. We describe *C_body*'s behavior for each type of expression.

Constants

If the expression is a constant, *C_body* generates code to return the value of the constant.

$$
\begin{array}{rl}
C\_body(c, sym\_table) = & emit(\text{``}\texttt{return\_value(n,}\text{''}, \; c, \; \text{``}\texttt{);}\text{''}) \\
& return(\{\})
\end{array}
$$

The routine `return_value` has already been defined on page 135.

Bound variables

If the body of a combinator is a reference to a bound variable, then the compiler generates code to return the value of the corresponding element of the args

vector of the current node.

$C\_body(x, sym\_table) =$
  let $< index, state >= sym\_table[\![x]\!]$
  $emit($"`return_value(n, n->args[`", $index$, "`].value);`")
  $return(\{\})$

Binary operations

In section 4.2.4 we stated that no binary operation could contain any synchronization construct. Therefore, when *C_body* encounters a binary operation, such as (+ e_1 e_2), it uses the procedure *C_exp* to convert the expression directly into C syntax and instructs the graph reducer to return the value of the expression. *C_exp* converts an expression in SCIF (without any synchronization constructs) to C by generating the code for the expression in infix notation. Any variable reference in an expression passed to *C_exp* is generated as an index into the args vector in the same manner as *C_body*. *C_exp* will not be defined in more detail.

$C\_body((+\ e_1\ e_2), sym\_table) =$ $emit($"`return_value(n,`"$)$
  $C\_exp((+\ e_1\ e_2), sym\_table)$
  $emit($"`);`"$)$
  $return(\{\})$

Conditionals

When *C_body* encounters a conditional, it calls *C_exp* to generate the code for the predicate. Code for each branch of the conditional is emitted by calling *C_body* recursively. The continuation names, along with the corresponding symbol tables, of both branches are returned since any one of the continuations

from the consequent or alternate may be executed.

$$
\begin{aligned}
&C\_body((\texttt{if } e_1\ e_2\ e_3), sym\_table) = \\
&\quad emit(\text{“}\texttt{if (}\text{”}) \\
&\quad C\_exp(e_1, sym\_table) \\
&\quad emit(\text{“}\texttt{) then \{}\text{”}) \\
&\quad \text{let } S = C\_body(e_2, sym\_table) \\
&\quad emit(\text{“}\texttt{\} else \{}\text{”}) \\
&\quad \text{let } S' = C\_body(e_3, sym\_table) \\
&\quad emit(\text{“}\texttt{\}}\text{”}) \\
&\quad return(S \cup S')
\end{aligned}
$$

The sets S and S' returned by *C_body* contain continuation-symbol table pairs as previously described.

Demand constructs

When *C_body* encounters a demand construct, it performs the following actions:

1. It looks up in the symbol table each variable that occurs in the demand construct. This is done to determine the index into the args vector for that variable and the state of the variable.
2. If the state of the variable is "unevaluated", then code is generated to get the value of the corresponding element of the args vector.
3. *C_body* is then called recursively to generate code for the body of the demand construct.

$$
\begin{aligned}
&C\_body((\texttt{demand } (v_1 \ldots v_k)\ body), sym\_table) = \\
&\quad \text{for } i = 1 \ldots k \\
&\quad\quad \text{let } \langle index_i, state_i \rangle = sym\_table[\![v_i]\!] \\
&\quad\quad \text{if } (state_i = uneval) \text{ then} \\
&\quad\quad\quad emit(\text{“}\texttt{getvalue(n,}\text{”}, index_i, \text{“}\texttt{);}\text{”}) \\
&\quad \text{let } S = C\_body(body, sym\_table) \\
&\quad return(S)
\end{aligned}
$$

The C routine `getvalue` was defined on page 136.

Wait constructs

When *C_body* encounters a wait construct it performs the following actions:

1. It looks up, in the symbol table, the variables that occur in the wait construct to determine their offsets in the args vector.
2. It emits code to modify the current node's waitmask such that the bits corresponding to the above variables are on.
3. It emits code that modifies the value field of the current node to point to the continuation named in the body of the wait construct.
4. Finally, it emits a call to `possibly_resume` in case the needed arguments have already been evaluated.

$C\_body(($`wait` $(v_1 \ldots v_k)\ cont\_name), sym\_table) =$
  let $\langle index_i, state_i \rangle = sym\_table[\![v_i]\!]$, for $1 \leq i \leq k$
    $sym\_table' = sym\_table[\langle index_1, eval \rangle / v_1, \ldots, \langle index_k, eval \rangle / v_k]$
  $emit($"`n->waitmask =`", $\sum_{i=1}^{k} 2^{index_i}$, "`;`"$)$
  $emit($"`n->value = `", $cont\_name$, "`;`"$)$
  $emit($"`possibly_resume(n);`"$)$
  $return(\{\langle cont\_name, sym\_table' \rangle\})$

As an example of the code generated for a wait construct, consider the following serial combinator definition:

```
f x y z ==  (if (= y 6)
                (demand (x z)
                    (wait (x z)
                          (+ x z)))
                1)
```

Figure 6.9 shows the code generated for `f` and a node resulting from the serial combinator invocation `(f (g 1) 6 (g 2))`, where `g` is some other serial combinator. Figure 6.10 shows the code generated for `fcont1`, the continuation that is called after the values of the bound variables `x` and `z` are available. The node pictured is about to resume executing by jumping to the code for `fcont1`.

In chapter 4, we noted that it seemed strange that variable being evaluated locally should have to occur in a wait construct. For example, in the SCIF expression

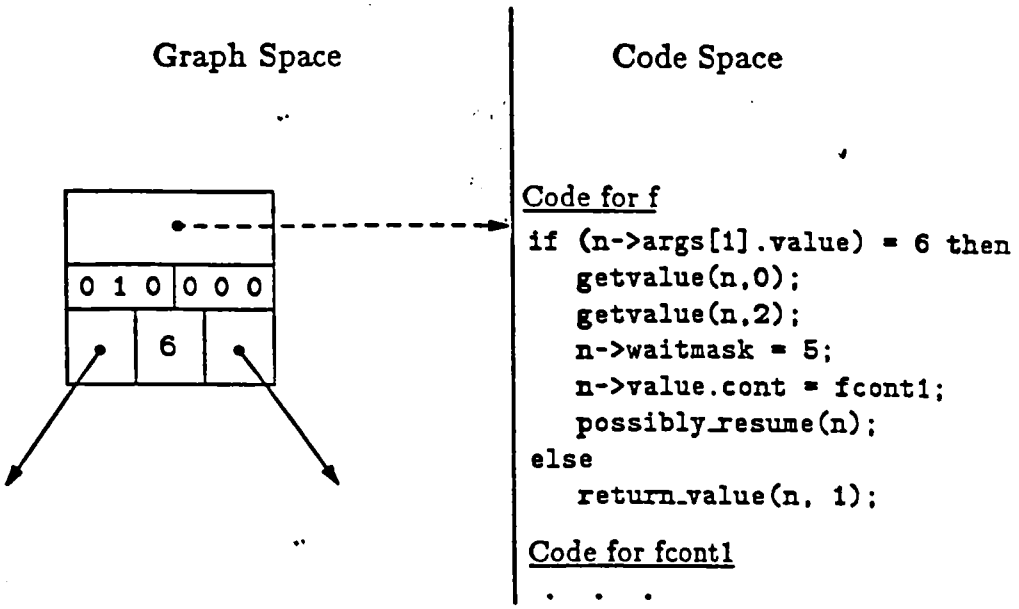


Figure 6.9: A node about to execute the serial combinator code

```
(spawn ((local v1 (f x y)) (v2 (g a b)))
   (wait (v1 v2)
      (+ v1 v2)))
```

it seems unnecessary for `v1` to occur in the wait construct. However, it is necessary for the following reason: The wait construct is the only way to specify which bits in a node's waitmask are to be set. Even though `v1` is being computed locally, it is possible for the value of `v2` to return while `v1` is still being computed (if, for example, the call to `f` suspends). If this happens, and the bit corresponding to `v1` in the waitmask is not set, then the node will resume prematurely. Even though `v1` is being evaluated locally, the corresponding bit in the waitmask must be set. Thus, `v1` must occur in a wait construct.

Let constructs

When encountering a let construct, *C_body* performs the following actions:

1. It counts the number of variables in the symbol table in order to find the index of the next available slot in the args vector.

2. For each variable-expression pair in the let construct, an index is assigned in the args vector and the state of the variable is set to "evaluated". This

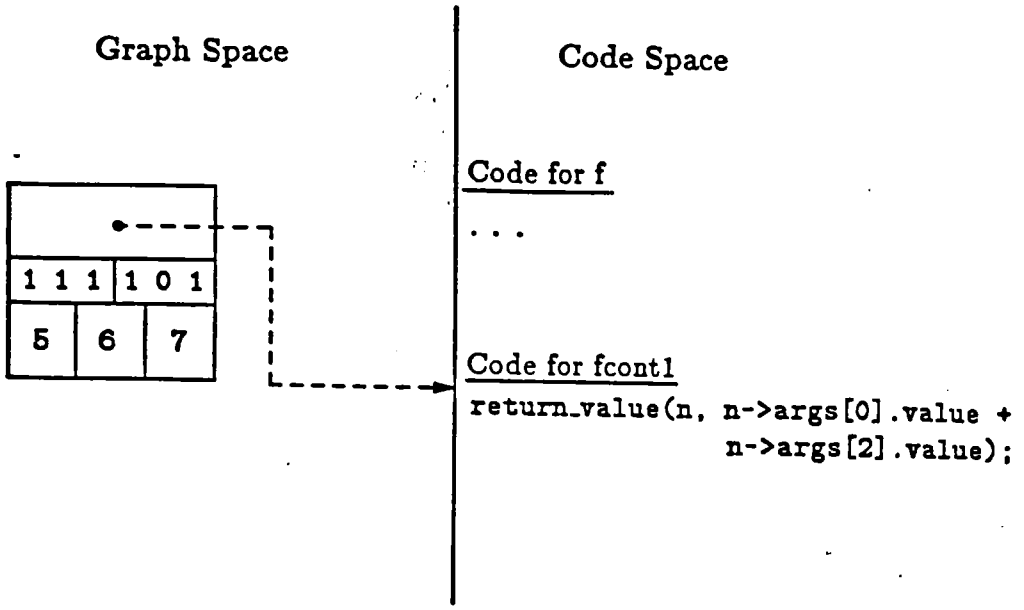


Figure 6.10: A suspended node about to resume executing

index-state pair is added to the symbol table.

3. Code for each expression in the let construct is generated by calling *C_exp*.
4. Code is emitted that updates the evalfield of the node to reflect that the new slots in the args vector contain values.
5. *C_body* is called recursively to generate code for the body of the let construct.

$C\_body((\texttt{let}\ ((v_1\ exp_1)\ldots(v_k\ exp_k))\ body), sym\_table) =$
  let $p = num\_vars(sym\_table)$
    $sym\_table' = sym\_table[\langle p+1, eval\rangle/v_1, \ldots, \langle p+k, eval\rangle/v_k]$
  for $i = 1 \ldots k$
    $emit(\text{“}\texttt{n->args[}\text{”}, p+i, \text{“}\texttt{d].value =}\text{”})$
    $C\_exp(exp_i, sym\_table')$
    $emit(\text{“;”})$
  $emit(\text{“}\texttt{n->evalfield = n->evalfield |}\text{”}, \sum_{j=p+1}^{p+k} 2^j, \text{“;”})$
  let $S = C\_body(body, sym\_table')$
  $return(S)$

(In C, the operator | denotes the bitwise logical or operator.)

Spawn construct

When encountering a spawn construct, *C_body* performs the following actions:

1. It determines the index of the next available slot in the args vector.
2. For each variable-expression pair in the spawn construct, *C_body* assigns an index to the variable and initializes the state of the variable to "unevaluated". This index-state pair is added to the symbol table.
3. *C_body* emits code to create the nodes representing the expressions in the spawn construct.
4. *C_body* is then called recursively to generate code for the body of the spawn construct.

```
C_body((spawn (p_1 ... p_k) body), sym_table) =
    let  q = num_vars(sym_table)
         sym_table' = sym_table[ ⟨q + 1, uneval⟩/v_1, ...,
                                 ⟨q + k, uneval⟩/v_k]
    for  i = 1 ... k
         if p_i = (local v_i exp_i) then
            local_node(exp_i, sym_table')
         else if p_i = (address v_i exp_i) then
            emit("newpe = choose_pe();")
            emit("address = get_address(newpe);")
            node_in_buf(exp_i, sym_table')
            emit("build_eval_address(newpe, address,",
                 "localpe, n,", i, ");")
         else
            node_in_buf(exp_i, sym_table')
            emit("build_eval(newpe, localpe, n,", i, ");")
    let S = C_body(body, sym_table')
    return(S)
```

Both *local_node* and *node_in_buf*, given an expression and a symbol table, generate code that causes a new subgraph to be created.

If a variable-expression pair in a spawn construct is modified by the word `local` then *local_node* is used to generate the appropriate code. *Local_node* is (roughly) defined as follows:

local_node(*exp*, *sym_table*) =
    let $(fe_0 \ldots e_k) = exp$
        newnode = a new identifier
    emit("{ `node *`", *newnode*, ";")
    emit(*newnode*, "= `local_new_node(`", *f*, "`);`")
    for $i = 0 \ldots k$
        if $e_i = x$ where x is a constant or bound variable then
            emit(*newnode*, "`->args[`", i, "`] =`")
            C_exp(x, *sym_table*)
            emit(";")
        else if $e_i = (f_i\ e_{i1} \ldots e_{in})$ then
            local_node($(f_i\ e_{i1} \ldots e_{in})$, *sym_table*)
        assign_evalfield(*newnode*, $e_0, \ldots, e_k$, *sym_table*)
        emit("}")

The new subgraph representing the expression will be allocated locally.

Otherwise, *node_in_buf* will generate code to create descriptor for the new subgraph in the outgoing message buffer. *Node_in_buf* is (roughly) defined as

follows:

$node\_in\_buf(exp, sym\_table) =$
  let $(f e_0 \ldots e_k) = exp$
    $newnode$ = a new identifier
  $emit($"{ node *", $newnode$, ";")
  $emit(newnode,$ "= new_node_in_buf(", f, ");")
  for $i = 0 \ldots k$
    if $e_i = x$, where x is a constant or bound variable, then
      $emit(newnode,$ "->args[", i, "] =")
      $C\_exp(x, sym\_table)$
      $emit($";")
    else if $e_i = (f_i\ e_{i1} \ldots e_{in})$ then
      $node\_in\_buf(newnode, i, (f_i\ e_{i1} \ldots e_{in}), sym\_table)$
  $assign\_evalfield(newnode, e_0, \ldots, e_k, sym\_table)$
  $emit($"}")

The procedure $assign\_evalfield(newnode, e_0, \ldots, e_k, sym\_table)$ generates code that initializes the evalfield of *newnode* to the appropriate value based on whether each argument, e_i, is evaluated or not. Each e_i is one of the following:

- A constant: In this case the ith bit of *newnode*'s evalfield is set to 1.
- A bound variable: In this case the ith bit is set to the value of the corresponding bit of the evalfield of the n, the node representing the current serial combinator.
- A serial combinator application representing delayed expression: The ith bit of the evalfield is set to 0.

Stack-spawn constructs

When encountering a stack-spawn construct, *C_body* performs the following actions:

1. It determines the index of the next available slot in the args vector.
2. For each variable-expression pair in the stack-spawn construct, *C_body* assigns an index to the variable and initializes the state of the variable is

"evaluated". This index-state pair is added to the symbol table.

3. Each expression in a stack-spawn construct is a call to a stack-executable combinator. Thus *C_body* simply generates code for a C function call.
4. *C_body* emits code that updates the evalfield of the node. The bit corresponding to each new variable is set to 1.
5. *C_body* is called recursively to generate code for the body of the stack-spawn construct.

$C\_body((stack\_spawn\ ((v_1\ exp_1)\ldots(v_k\ exp_k))\ body), sym\_table) =$

let $p = num\_vars(sym\_table)$
$sym\_table' = sym\_table[\langle p+1, eval\rangle/v_1, \ldots, \langle p+k, eval\rangle/v_k]$
for $i = 1 \ldots k$
let $(f_i\ e_{i1} \ldots e_{in}) = exp_i$
$emit($"`n->args[`", $p+i$, "`d].value =`"$)$
$emit(f_i,$ "`(`"$)$
$C\_exp(e_{i1}, sym\_table')$
$emit($"`,`"$)$
$C\_exp(e_{i2}, sym\_table')$
$emit($"`,`"$)$
$\vdots$
$C\_exp(e_{in}, sym\_table')$
$emit($"`)`"$)$
$emit($"`n->evalfield = n->evalfield |`", $\sum_{j=p+1}^{p+k} 2^j$, "`;`"$)$
let $S = C\_body(body, sym\_table')$
$return(S)$

Tail-spawn constructs

We will not describe code generation for a tail-spawn construct in detail, but rather discuss the most important aspects of it.

Each field of the node representing the "old" invocation is overwritten by the information about the "new" invocation. Some of the arguments in the new invocation, however, may depend on the values of the old arguments. For example, in the following serial combinator definition,

generated that saves the old argument residing in the jth slot of the args vector into a temporary location and writes the jth new argument into the args vector. This heuristic was chosen in the hope that by removing the argument with the most incoming arcs from the graph, the largest number of cycles in the graph are broken.

5. Steps 2–4 are repeated until no new arguments remain to be written into the args vector.

Figure 6.11 demonstrates how the dependency graph is used to generate the tail-spawn code the following serial combinator:

```
f x y == (tail-spawn (g y x 6))
```
(6.2)

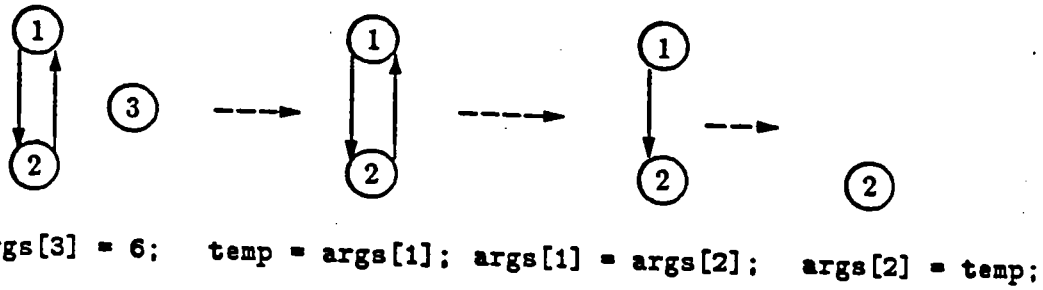


Figure 6.11: A dependency graph for the arguments in a tail-spawn

As a trivial optimization, if the ith new argument is identical to the ith old argument then no code is generated to update the ith slot in the args vector.

6.4.2 Code generation for stack executable combinators

Each stack executable combinator definition is simply translated into a C routine. Since the iPSC's C compiler will generate code that utilizes a stack to hold the activation records for each call, the translation from extended SCIF into C for is straightforward.

$$
\begin{aligned}
C\_stack(f\ x_0 \ldots x_n == body) = \ & emit(f,\ \text{“(”},\ x_0,\ \ldots,\ x_n,\ \text{“)”}) \\
& emit(\text{“CINT”},\ x_0,\ \ldots,\ x_n,\ \text{“; \{”}) \\
& C\_stack\_body(body) \\
& emit(\text{“\}”})
\end{aligned}
$$

C_stack_body generates code for the body of a stack executable combinator. Each bound variable in the combinator definition corresponds to a variable with the identical name in the C routine. Thus no symbol table is required. The procedure *C_stack_exp* converts an expression in SCIF to C.

If the expression is a constant or variable, then *C_stack_body* generates C code to return the value of the expression as the result of the combinator call.

$$C\_stack\_body(c) = emit(\text{“}\texttt{return(}\text{”},\ c,\ \text{“}\texttt{)}\text{”})$$

$$C\_stack\_body(x) = emit(\text{“}\texttt{return(}\text{”},\ x,\ \text{“}\texttt{)}\text{”})$$

If the expression is a binary operation, *C_stack_exp* is called to generate the corresponding C code.

$$\begin{aligned} C\_stack\_body(\texttt{+}\ e_1\ e_2) = \ & emit(\text{“}\texttt{return(}\text{”}) \\ & C\_stack\_exp(\texttt{+}\ e_1\ e_2) \\ & emit(\text{“}\texttt{);}\text{”}) \end{aligned}$$

C_stack_body generates C code for a conditional by using *C_stack_exp* to generate code for the predicate and recursively calling *C_stack_body* for the consequent and alternate.

$$\begin{aligned} C\_stack\_body(\texttt{if}\ e_1\ e_2\ e_3) = \ & emit(\text{“}\texttt{if (}\text{”}) \\ & C\_stack\_exp(e_1) \\ & emit(\text{“}\texttt{) then \{}\text{”}) \\ & C\_stack\_body(e_2) \\ & emit(\text{“}\texttt{\} else \{}\text{”}) \\ & C\_stack\_body(e_3) \\ & emit(\text{“}\texttt{\}}\text{”}) \end{aligned}$$

C_stack_exp is used to generate code for each expression in a let construct, and

C_stack_body is called recursively on the body.

$$
\begin{aligned}
&\mathit{C\_stack\_body}((\text{let } ((v_1\ exp_1)\ldots(v_k\ exp_k))\ body)) = \\
&\qquad \mathit{emit}(\text{“\{ CINT”}, v_1, \ldots, v_k, \text{“;”}) \\
&\qquad \text{for } i = 1\ldots k \\
&\qquad\quad \mathit{emit}(v_i, \text{“=”}) \\
&\qquad\quad \mathit{C\_stack\_exp}(exp_i) \\
&\qquad\quad \mathit{emit}(\text{“;”}) \\
&\qquad \mathit{C\_stack\_body}(body) \\
&\qquad \mathit{emit}(\text{“\}”})
\end{aligned}
$$

A stack-spawn construct is handled in a similar manner. Each expression in the stack-spawn construct is translated into a C function call. The code for the arguments in each call is generated by *C_stack_exp*. *C_stack_body* is recursively called on the body of the stack-spawn.

$$
\begin{aligned}
&\mathit{C\_stack\_body}((\mathit{stack\_spawn}\ ((v_1\ exp_1)\ldots(v_k\ exp_k))\ body)) = \\
&\qquad \mathit{emit}(\text{“\{ CINT”}, v_1, \ldots, v_k, \text{“;”}) \\
&\qquad \text{for } i = 1\ldots k \\
&\qquad\quad \text{let } (f_i\ e_{i1}\ldots e_{in}) = exp_i \\
&\qquad\quad \mathit{emit}(v_i, \text{“=”}, f_i, \text{“(”})\mathit{C\_stack\_exp}(e_{i1}) \\
&\qquad\quad \mathit{emit}(\text{“,”}) \\
&\qquad\quad \mathit{C\_stack\_exp}(e_{i2}) \\
&\qquad\quad \mathit{emit}(\text{“,”}) \\
&\qquad\quad \vdots \\
&\qquad\quad \mathit{C\_stack\_exp}(e_{in}) \\
&\qquad\quad \mathit{emit}(\text{“) ;”}) \\
&\qquad \mathit{C\_stack\_body}(body) \\
&\qquad \mathit{emit}(\text{“\}”})
\end{aligned}
$$

6.5 An example

As an example of how code is generated for a program, consider the following (strange) factorial program.

```
{ sfac 1 h == 1=h->1, h * sfac 1 (h-1);
```

```
pfac l h == l=h->l,
            { pfac1 mid ==  sfac l mid * pfac (mid + 1) h;
              result pfac1} ((l+h)/2);
result pfac 1 10;
}
```

Notice that this program consists of a serial factorial function, `sfac`, and an unusual divide and conquer factorial, `pfac`. `Pfac` contains a call to `sfac` as well as a recursive call to `pfac`. The serial combinator version of the above program is:

```
{
 --- graph reduction functions ---
 g_sfac l h == (demand (l h)
                 (wait (l h)
                   (if (= l h)
                       l
                       (let ((v3 (- h 1)))
                         (stack-spawn ((v4 (s_sfac l v3)))
                           (* h v4))))))


  g_pfac l h == (demand (l h )
                  (wait (l h )
                    (if (= l h)
                        l
                        (let ((v1 (/ (+ l h)  2) ) )
                          (tail-spawn (g_pfac1 l h v1))))))


 g_pfac1 l h mid ==
      (demand (l mid h)
        (wait (l mid)
          (spawn ((v5 (g_sfac l mid)))
            (let ((v6 (+ mid 1)))
              (wait (h )
                spawn ((<local> v7 (g_pfac v6 h)))
                  (wait (v5 v7)
                    (* v5 v7)))))))
```

```
--- stack functions ---
s_sfac l h == if (= l h)
              l
              (let ((v3 (- h 1)))
                (stack-spawn ((v4 (s_sfac l v3)))
                  (* h v4)))

result g_pfac 1 10
}
```

The C routines that the compiler generates for the above serial combinators are listed below. Much of the actual C code has been edited out, especially the code that accomplishes reference counting, type checking, and so on.

```
G_SFAC(n)
node *n;
{
getvalue(n, 0);
getvalue(n, 1);
n->waitmask =  3;
n->value.cont = G_SFAC_CONT1;
possibly_resume(n);
}


G_SFAC_CONT1(n)
node *n;
{
if ((n->args[0].value == n->args[1].value)) then
  return_value(n, n->args[0].value);
else {
   n->evalfield = n->evalfield | 4;
   n->args[2].value = n->args[1].value - 1;
   n->evalfield = n->evalfield | 8;
   n->args[3].value = S_SFAC(n->args[0].value,
                             n->args[2].value);
   return_value(n, n->args[1].value *
                   n->args[3].value);
  }
}
```

```
G_PFAC(n)
node *n;
{
getvalue(n, 0);
getvalue(n, 1);
n->waitmask =  3;
n->value.cont = G_PFAC_CONT1;
possibly_resume(n);
}


G_PFAC_CONT1(n)
node *n;
{
if (n->args[0].value == n->args[1].value) then
   return_value(n, n->args[0].value);
else {
   n->evalfield = n->evalfield | 4;
   n->args[2].value = (n->args[0].value + n->args[1].value) / 2;
   n->value.cont = G_PFAC1;
   n->numargs = 3;
   n->evalfield =  7;
   (*n->value.cont)(n);
   }
}


G_PFAC1(n)
node *n;
{
getvalue(n, 0);
getvalue(n, 2);
getvalue(n, 1);
n->waitmask =  5;
n->value.cont = G_PFAC1_CONT1;
possibly_resume(n);
}
```

```
G_PFAC1_CONT1(n)
node *n;
{
{ node* NEWNODE1;
  unsigned newpe;
  newpe = choose_pe();
  NEWNODE1 = new_node_in_buf(G_SFAC);
  NEWNODE1->args[0].value = n->args[0].value;
  NEWNODE1->args[1].value = n->args[2].value;
  NEWNODE1->evalfield =  3;
  build_eval(newpe, NEWNODE1, localpe, n, 3);
  }
n->evalfield = n->evalfield | 16;
n->args[4].value = n->args[2].value + 1;
n->waitmask =  2;
n->value.cont = G_PFAC1_CONT2;
possibly_resume(n);
}


G_PFAC1_CONT2(n)
node *n;
{
{ node* NEWNODE2;
NEWNODE2 = local_new_node(G_PFAC)
NEWNODE2->args[0].value = n->args[4].value;
NEWNODE2->args[1].value = n->args[1].value;
NEWNODE2->evalfield = 3;
write_point(&(n->args[5].point), localpe, NEWNODE2, NIL);
getvalue(n, 5);
}
n->waitmask =  40;
n->value.cont = G_PFAC_CONT3;
possibly_resume(n);
}


G_PFAC_CONT3(n)
node *n;
{
return_value(n, n->args[3].value * n->args[5].value);
}
```

```
CINT S_SFAC_(L, H)
CINT L, H;
{
if (L == H) then
   return(L);
else {
    CINT V3, V4;
    V3 = H - 1;
    V4 = S_SFAC(L, V3);
    return(H * V4);
  }
}
```

In the next chapter, we present a number of dynamic scheduling heuristics used by Alfalfa, along with a set of benchmarks that demonstrate the effectiveness of these algorithms.

Chapter 7

Dynamic Scheduling in Alfalfa

Having described the Alfalfa system, we present empirical results on its performance using a variety of dynamic scheduling algorithms. All of these algorithms fall into the category of algorithms that we call *diffusion scheduling.*

7.1 Diffusion Scheduling

Diffusion scheduling is a class of algorithms for dynamically scheduling tasks on loosely-coupled (distributed-memory) multiprocessors. The only assumption that need be made about the multiprocessor network is that it is *connected*, which means that each processor can communicate (directly or indirectly) with any other processor. Most distributed memory multiprocessor networks are *sparse*; each processor can communicate directly with only a few of the processors in the system. A processor's *nearest neighbors* are those processors it can communicate with directly.

Diffusion scheduling is completely decentralized. Each processor is responsible for deciding whether to execute a task locally or to send it to another processor. Both the decision of *whether* to allocate work on another processor and *which processor* to allocate the work on are made locally. These two components of diffusion scheduling are called the *transfer policy* and the *location policy*, respectively [18].

In the diffusion scheduling algorithms we have tested on Alfalfa, the information available to a processor is restricted to local load information and

possibly load information about its nearest neighbors.[1] Thus each processor in Alfalfa has a limited view of the state of the system. A processor can only allocate work onto a nearest neighbor, and cannot directly affect the state of a non-neighboring processor. A processor only sends a task to a neighbor when it believes that the neighbor has a lower work load. At the start of the computation, work is generally allocated to a small number of processors. As the parallelism of the computation increases, the work diffuses over the other processors. Instead of leaping across the network, the computation diffuses from heavily loaded areas of the network to less loaded areas.

In a multiprocessor such as the Intel iPSC, in which communication is extremely expensive, the diffusion scheduling algorithms we use have two attractive features:

1. Since each processor need only be aware of the load of its nearest neighbors, the amount of communication required to keep the information up to date is relatively small. A less restrictive scheduling algorithm that required a processor to keep information about a greater number of processors would probably require more communication.

2. Since each processor can allocate work only on neighboring processors, most communication (for sending and returning values, for example) occurs between neighbors.

However, our diffusion scheduling algorithms are less sensitive to changes in the computational demands of a program than less restrictive scheduling methods. If a few of processors are suddenly saturated by an explosive growth in parallelism, they are less able to send some of the work to remote processors that may be underutilized. This may lead to a poorer distribution of work through the system.

The diffusion scheduling used by Alfalfa is similar in spirit to the scheduling used in the in the Rediflow multiprocessor [48] (which has only been simulated to date). The results of simulation experiments involving *fixed* combinator reduction using diffusion scheduling were reported previously by Hudak and Goldberg [32].

[1] In general, diffusion scheduling algorithms may allow each processor to possess information about an arbitrary number of processors.

7.1.1 Processor Load

In diffusion scheduling, the transfer and location policies often depend on the load of neighboring processors, which can be measured in several ways:

- *CPU utilization*: A processor may keep track of how busy it has been over a certain period of time. This may or may not be a reasonable measure of how busy it will be in the future.

- *Space utilization*: The amount of occupied storage in a processor can be used as indicator of a processor's load. However, there are two disadvantages to doing so:

  1. There may be no relationship between the amount of storage a task uses and its demands on the cpu.

  2. Space utilization is dependent on the storage reclamation method used. If a mark/sweep reclamation algorithm is used then space utilization would provide a very poor indicator of processor load because the reclamation process doesn't start until memory is nearly full. Much of memory may be occupied by dead objects. A reference-counting method would enable space utilization to provide a better measure of load since unneeded structures are collected immediately. However, cyclic structures will remain uncollected and may distort the measure of storage usage.

  Even so, statistics collected during storage reclamation, either through garbage collection or reference counting, may provide a basis for a reasonable space utilization metric.

- *Number of waiting tasks*: The number of tasks that a processor has yet to execute may provide a reasonable load measure. Generally waiting tasks are queued and the length of the queue provides this measure. However, if the computational demands of the tasks vary greatly, the accuracy of such a measure will suffer. In programs in which there is a reasonable distribution of task execution times, the queue length of waiting tasks should provide a reasonable measure.

In Alfalfa, a processor's load is measured by the number of waiting tasks in its queue.

7.1.2 Transfer Policies

A processor decides whether to allocate work on another neighbor based on load information about itself and possibly its neighbors. If the processor's load is lower than the reported load of its neighbors, it will choose to perform the work locally.

If one of its neighbors appears to have a lower load, the processor may choose to perform the work locally anyway. This can happen if the difference between the neighbor's load and the local load is not enough to warrant incurring communication overhead.

In several of the diffusion algorithms we tested in Alfalfa, the transfer policy relied on only local load information. Each processor decided whether or not to allocate work remotely based solely on the length of its task queue. In other algorithms we tested, load information about its neighbors is made available to each processor.

7.1.3 Location Policies

Once a processor has decided not to execute a task locally, it must choose a neighbor to send the task to. If no load information is available about a processor's neighbors then the processor will choose a neighbor based on some local policy. Two examples of such a policy would be to choose neighboring processors randomly or in a round-robin fashion. If load information about each neighbor is available, the processor generally chooses the neighbor with the lowest load. Of course, one could invent a diffusion policy that did not choose the least loaded neighbor, but rather included other information in the decision [32].

Several of the diffusion algorithms we tested on Alfalfa did not require load information about neighbors for their transfer policies. No communication, other than that needed for the computation, was required to perform dynamic scheduling. The location policy in each case was to allocate work to neighbors

in a round-robin fashion.

The diffusion algorithms that used neighbor load information for their transfer policies also used the same information for their location policies. In each case, the location policy was to allocate work on the least loaded neighbor.

7.1.4 Information Policies

A diffusion strategy that requires communication between neighboring processors has to specify an *information policy* [54]. The information policy determines how load information about a processor is made available to its neighbors. Examples of information policies are:

1. *Polling*: When a processor has a task that could be performed remotely, it polls some or all of its neighbors to see what their current load is. Then, based on the transfer and location policy, it chooses a processor to execute the task. Unfortunately, a processor could continually be polled even though its load has not changed significantly. This would result in wasted communication.

2. *Update*: Each processor automatically notifies its neighbors about its current load. An *update policy* must specify when load information should be sent. Two choices are:

   - *Time-Dependent Updates*: A processor sends load messages at regular time intervals during the computation. If the time interval is too short, an update message may be sent when the load of the processor has not changed significantly since the last update. If the interval is too long, neighboring processors may remain unaware of drastic changes in a processor's load.
   - *State-Dependent Updates*: Update messages are sent only when a processor's load changes significantly. Of course, the update policy must define what constitutes a significant change in the load. If the policy is too sensitive to changes then too many update messages will be sent, causing unnecessary communication overhead. If the policy is too insensitive to variations in a processor's load, the neighbors will possess incorrect information much of the time.

In the diffusion algorithms that require communication, Alfalfa uses a state-dependent update policy. We varied the update policy in our empirical studies to find an appropriate sensitivity to load fluctuations.

7.2 Communication in the Intel iPSC

In chapter 6 we mentioned that communication overhead in the Intel iPSC is extremely high. The following statistics describe the message passing performance of the iPSC.

Message latency

The message latency (the time between when the send is initiated and when the message is received at the destination) between neighbors is dependent on the size of the message as follows:

Message Size (bytes)	Latency (milliseconds)
2	1.281
500	1.992
1000	2.711
1500	4.398
2000	5.111
3000	7.514

For messages that are not sent between neighbors, each additional hop adds significantly to the message delay. For a two byte message, the message latency as a function of path length is shown in the following table:

No. of Hops	Latency (milliseconds)
1	1.275
2	1.830
3	2.335
4	2.935
5	3.482

As indicated in the table, each hop adds 0.552 milliseconds to the message delay for a two byte message.

Processor overhead

In chapter 4 we showed that processor overhead generally has a greater effect on the overall performance of a parallel computation than message latency. Unfortunately, the iPSC suffers from significant processor overhead for sending messages.

Message Size (bytes)	Overhead (milliseconds)
2	1.6770
500	2.3530
1000	3.0550
1500	4.7330
2000	5.4450
3000	7.7980

Surprisingly, the processor overhead is greater than the message delay. A message arrives at its destination while the sending processor is still executing the message passing routines!

Another unfortunate aspect of the iPSC is that a processor incurs substantial overhead when *forwarding* a message. Each processor along the path of a two byte message incurs an overhead of 0.780 milliseconds. Again, the processor overhead involved in forwarding a message is greater than the additional message delay.

7.3 Application Programs

We tested Alfalfa on five programs. They were:

1. *Parallel Factorial* (`pfac`): This is a simple divide and conquer algorithm for computing factorial. The granularity of the corresponding serial combinator program is very fine. If one considers the tree structure of a divide and conquer algorithm, the only operation performed at the leaves of parallel factorial is to return the value of one of the arguments. At each vertex in the tree, a few arithmetic operations are performed, along with two recursive calls:

```
{ pfac l h == l=h -> l, { pfac1 mid ==  pfac l mid + pfac (mid + 1) h;
                          result pfac1;
                          } ((l+h)/2);
  result pfac 1 1000;
  }
```

(For experimental purposes, we are taking the sum of all the numbers rather than the product.)

This is the serial combinator version of `pfac`:

```
{ --- Graph Reduction Functions ---
  pfac l h == (demand (l h)
                 (wait (l h)
                   (if (= l h)
                        l
                        (let ((v1 (/ (+ l h)  2)))
                           (tail-spawn (pfac1 l h v1))))))

  pfac1 l h mid == (demand (l mid h)
                     (wait (l mid)
                       (spawn ((v3 (pfac l mid)))
                         (let ((v4 (+ mid 1)))
                           (wait (h)
                             (spawn ((local v5 (pfac v4 h)))
                               (wait (v3 v5)
                                 (+ v3 v5)))))))))

  result pfac 1 1000
  }
```

2. *Eight Queens* (queens): This program finds all solutions to the eight queens problem by performing a parallel search through possible board configurations. Because of memory limitations in the iPSC, the program was executed using a seven-by-seven chess board. The program uses the functions `mkv`, `upd`, and `sel` to create a vector, update a vector (functionally), and select an element of a vector, respectively.

```
{ nqueens dim ==
    { boardsafe board row col oldcol ==
            oldcol < 0->true,
                       safe (sel board oldcol) oldcol row col ->
                                 boardsafe board row col (oldcol-1),
                                 false;

      safe oldrow oldcol newrow newcol ==
           oldrow=newrow->
                    false,
                    abs (newrow - oldrow) = (newcol-oldcol) -> false,
                                                               true;
```

```
      queens board row col ==
            boardsafe board row col (col-1) ->
                        (col+1)=dim ->1,
                         all_queens (upd board col row) 0 (col+1),
                        0;

      all_queens board row col ==
           row = dim ->
                  0,
                  queens board row col + all_queens board (row+1) col;

      result all_queens (mkv dim) 0 0;
       };

 result nqueens 7;
}
```

The queens program decomposes nicely for multiprocessor execution. Large numbers of serial combinator calls may be evaluated in parallel, and each serial combinator has a substantial sequential component. This can be seen in the sequential stack-based definition of `boardsafe` in the serial combinator version of the program. The figures in sections 7.4 and 7.5 indicate that the granularity of the serial combinators is sufficiently coarse to provide good performance on multiprocessors with very high communication overhead. Here is the serial combinator version of queens:

```
{   --- graph reduction functions ---
  nqueens dim == (demand (dim)
                   (wait (dim)
                     (let (v1 (mkv dim))
                       (tail-recurse (all_queens dim v1 0 0)))))

  all_queens dim board row col ==
          (demand (row dim)
            (wait (row dim)
              (if (= row dim)
                  0
                  (demand (board col)
                    (spawn ((v3 (queens dim board row col)))
                      (let ((v4 (+ row 1)))
                        (wait (board col)
                          (spawn
                              ((local v5 (all_queens dim board v4 col)))
                            (wait (v3 v5)
                              (+ v3 v5)))))))))))
```

```
queens dim board row col ==
        (demand (board row col)
          (wait (col)
            (let ((v6 (- col 1)))
              (spawn ((local v7 (boardsafe board row col v6)))
                (wait (v7)
                  (if v7
                      (demand (dim)
                         (wait (dim)
                           (if (= (+ col 1)  dim)
                               1
                               (wait (board row)
                                 (let ((v8 (upd board col row))
                                       (v9 (+ col 1)))
                                   (tail-recurse
                                       (all_queens dim v8 0 v9)))))))
                      0))))))

boardsafe board row col oldcol ==
     (demand (oldcol)
       (wait (oldcol)
         (if (< oldcol 0)
             true
             (demand (board row col)
              (wait (board)
                (let ((v13 (sel board oldcol)))
                  (wait (row col)
                   (stack-spawn ((v14 (s_safe v13 oldcol row col)))
                     (if v14
                         (let ((v15 (- oldcol 1)))
                           (stack-spawn
                               ((v16 (s_boardsafe board row col v15)))
                             v16))

                         false)))))))))

--- stack functions ---
s_safe oldrow oldcol newrow newcol ==
     (if (= oldrow newrow)
         false
        (let ((v11 (- newrow oldrow))
              (v12 (abs v11)))
          (if (= v12 (- newcol oldcol))
              false
              true)))
```

```
s_boardsafe board row col oldcol ==
      (if (< oldcol 0)
          true
          (let ((v13 (sel board oldcol)))
            (stack-spawn (v14 (s_safe v13 oldcol row col)))
              (if v14
                  (let ((v15 (- oldcol 1)))
                    (stack-spawn
                         ((v16 (s_boardsafe board row col v15)))
                       v16))
                  false)))

result nqueens 7
}
```

3. *Adaptive Quadrature* (quad): This is a numerical algorithm for approximating the area under a curve. The interval of interest is partitioned into subintervals whose areas are approximated by trapezoids. In order to increase accuracy, the subintervals have varying widths based on the shape of the curve. In sections where the curve is relatively straight (i.e. has a small second derivative), the subintervals may be large. The subintervals are smaller in sections where the curve is less well-behaved.

   An adaptive algorithm (so called because the method adapts to the data) is appropriate in this case since a constant subinterval method would either waste computation in straight sections of the curve or sacrifice accuracy in other sections.

   Adaptive quadrature is highly parallel. However, the computation of the area of each trapezoid is a very simple operation, and thus the granularity of the computation is relatively fine. Nevertheless, the performance of Alfalfa when executing this program was quite good.[2]

```
{ Area F left right == ((F left + F right)/2) * (right - left);
```

[2]Because of problems in Alfalfa's implementation of floating point operations, we actually used integer arithmetic to implement fixed point operations.

```
Found F left mid right oldval ==
     { Found1 newleft newright. ==
          abs ((newleft+newright) - oldval) < 0.5 ->
              newleft+newright,
              Found F left ((left+mid)/2) mid newleft +
              Found F mid ((mid+right)/2) right newright;
       result Found1;
       } (Area F left mid) (Area F mid right);


Quad F left right == Found F left ((left+right)/2)
                          right (Area F left right);


Foo x == ((((x - 6) * x) + 3) * x) - 2;

result Quad Foo 0.0 10.0;
}
```

Here is the serial combinator version of the quad program:

```
{
  --- Graph Reduction Functions ---
  foo x == (demand (x)
              (wait (x)
                (- (* (+ (* (- x 6) x) 3) x)  2)))


  quad f left right ==
         (demand (f left right)
            (wait (f left right)
              (spawn ((address v19 (area f left right)))
                (let ((v18 (/ (+ left right)  2)))
         (tail-recurse (found f left v18 right v19))))))

  found f left mid right oldval ==
      (demand (f left mid right oldval)
         (wait (f left mid right)
            (spawn ((local v21 (area f left mid))
                    (v22 (area f mid right)))
               (wait (oldval v21 v22)
                  (tail-recurse
                       (found1 f left mid right oldval v21 v22))))))
```

```
area f left right == (demand (f left right)
                       (wait (f)
                         (spawn ((local v24 (f left))
                                 (v25 (f right)))
                           (wait (v24 v25 right left)
                             (* (/ (+ v24 v25)  2)
                                (- right left))))))


found1 f left mid right oldval newleft newright ==
       (demand (newleft newright oldval)
          (wait (newleft newright oldval)
            (if (< (abs (- (+ newleft newright)  oldval))  0.5)
                (+ newleft newright)
                (demand (left mid right)
                  (wait (left mid right)
                    (let ((v26 (/ (+ left mid)  2))
                          (v28 (/ (+ mid right)  2)))
                      (spawn
                          ((local v27 (found f left v26 mid newleft))
                           (v29 (found f mid v28 right newright)))
                         (wait (v27 v29)
                           (+ v27 v29)))))))))


--- stack functions ---
s_foo x == (- (* (+ (* (- x 6) x) 3) x)  2)))

result quad foo 0 10
}
```

4. *Matrix Multiplication* (`matmult`): This program performs standard matrix multiplication. Each matrix is represented as a vector of vectors.

```
{
dotprod row col elt size ==
    elt=size -> 0,
          ((sel row elt) *
           (sel col elt)) +
          dotprod row col (elt+1) size;


rowmult row colvec elt size ==
    elt=size -> 0,
      dotprod row (sel colvec elt) 0 size +
      rowmult row colvec (elt+1) size;
```

```
rowcolmult rowvec colvec elt size ==
 elt=size->0,
    rowmult (rowvec-elt) colvec 0 size +
    rowcolmult rowvec colvec (elt+1) size;


result rowcolmult (create_matrix 30) (create_matrix 30) 0 30;
}
```

We used `create_matrix` to create and initialize a matrix (to avoid having to do it explicitly).

Here is the serial combinator version of matrix multiplication:

```
{
 --- graph reduction functions ---
 rowcolmult rowvec colvec elt size ==
       (demand (elt size)
         (wait (elt size)
           (if (= elt size)
               0
               (demand (rowvec colvec)
                 (wait (rowvec)
                   (let ((v30 (sel rowvec elt))
                         (v32 (+ elt 1)))
                     (wait (colvec)
                       (spawn ((local v31 (rowmult v30 colvec 0 size))
                           (v33 (rowcolmult rowvec colvec v32 size)))
                         (wait (v31 v33)
                           (+ v31 v33)))))))))))


rowmult row colvec elt size ==
     (demand (elt size)
       (wait (elt size)
         (if (= elt size)
             0
             (demand (row colvec)
               (wait (colvec)
                 (let ((v34 (sel colvec elt))
                       (v36 (+ elt 1)))
                   (wait (row)
                     (spawn ((local v35 (dotprod row v34 0 size))
                      (v37 (rowmult row colvec v36 size)))
                      (wait (v35 v37)
                        (+ v35 v37)))))))))))
```

```
dotprod row col elt size ==
     (demand (elt size)
        (wait (elt size)
          (if (= elt size)
              0
              (demand (row col)
                (let ((v38 (+ elt 1)))
                  (wait (row col)
                    (stack-spawn ((v39 (s_dotprod row col v38 size)))
                      (+ (* (sel row elt)
                            (sel col elt))
                         v39)))))))))


c1 == (let ((v40 (create_matrix 30))
            (v41 (create_matrix 30)))
        (tail-recurse (rowcolmult v40 v41 0 2)))


--- stack functions ---
s_dotprod row col elt size ==
      (if (= elt size)
          0
          (let ((v38 (+ elt 1)))
            (stack-spawn ((v39 (s_dotprod row col v38 size)))
              (+ (* (sel row elt)
                    (sel col elt))
                 v39))))


result c1
}
```

Although matrix multiplication has a high degree of parallelism, the execution of `matmult` requires the distribution of copies of the rows and columns of the matrices. The statistics presented in sections 7.4 and 7.5 demonstrate that this has a significant negative effect on Alfalfa's performance because of its high communication costs. When this program is run on a single processor, no copying of the rows and columns of the initial matrices is required.

5. *Quicksort*: This programs sorts a list of numbers using the well-known quicksort algorithm [26]. The parallelism exhibited by quicksort is only on the order of $\log n$, where n is the number of elements in the list (quicksort's expected sequential time is $0(nlogn)$ and its expected parallel execution

time is $0(n)$). Unfortunately, due to memory limitations on Alfalfa, we were unable to run quicksort with a sufficiently large list to exploit a large amount of parallelism.

```
{ qs L ==
    L=[] -> [],
          { split X acc ==
             X=[]->acc,
                   (hd X) < (hd L) ->
                      split (tl X) [(hd X)^(hd acc),
                                    hd (tl acc))],
                      split (tl X) [(hd acc),
                                    (hd X)^(hd (tl acc))];
             result { qs1 res ==
                         qs (hd res) ^^
                         ((hd L) ^ qs:(hd (tl res)));
                      result qs1; } (split (tl L) [[],[]]);
             };
result qs (create_list 200);
}
```

The serial combinator version of `quicksort` has already been presented in section 5.5

As we shall see, Alfalfa performed quite well on the first three programs. Using a variety of diffusion methods, significant (although not linear) speedup over the single-processor case was achieved. For the reasons mentioned above, Alfalfa's performance on `matmult` was unimpressive.

There are, of course, an infinite number of programs that Alfalfa could be tested on. These five programs had the desirable properties of being easy to write, easy to implement and were sufficiently different to provide a reasonable study into Alfalfa's performance. As Alfalfa becomes used by a greater number of people, a greater variety of programs will be executed on it.

We were also only able to test a few different diffusion algorithms. A large number of reasonable candidates had to be ignored. As it were, with only five diffusion algorithms being tested, we still had to run Alfalfa almost 500 times to gather the data presented here. We may not have found the best diffusion strategy. However, a substantially more complete study of diffusion techniques would have been impossible given the time and manpower limitations of this

study. [3]

7.4 Non-Communicating Diffusion Scheduling

This section describes Alfalfa's performance on the five application programs using three non-communicating diffusion scheduling algorithms: *simple round-robin*, *dependent round-robin*, and *ratio round-robin*. These algorithms have identical location policies; their transfer policies differ, however.

7.4.1 Simple Round-Robin Diffusion

The simple round-robin transfer policy dictates that a new task (serial combinator invocation) is sent to a neighboring processor whenever the local task queue length L surpasses a fixed threshold T. The value of T is independent of the number of processors in the system. Figure 7.1 plots Alfalfa's execution time (in milliseconds) as a function of the number of processors used. All five programs were tested using simple round-robin diffusion and a number of different T values. The results demonstrate the following points:

- A large T value reduces the amount of parallelism exploited in a system with a large number of processors. However, in most cases a high value for T performed better in systems with only a few processors than a low T value. This is because a high T value reduces the number of tasks sent between busy processors on a small system.
- For programs such as `pfac` and `quad` with many fine grain tasks, even a high value for T does not prevent degradation on a two processor system. `Queens`, `matmult` and `quicksort` perform reasonably on two processors when T is large.
- On a system with many (e.g. 32) processors, a low T value performs surprisingly well. This behavior seems to be program-independent and implies that even simplistic diffusion strategies will work as long as there are sufficient processors.

[3] In chapter 8, we also present a large amount of data about executing serial combinators on a shared memory multiprocessor.

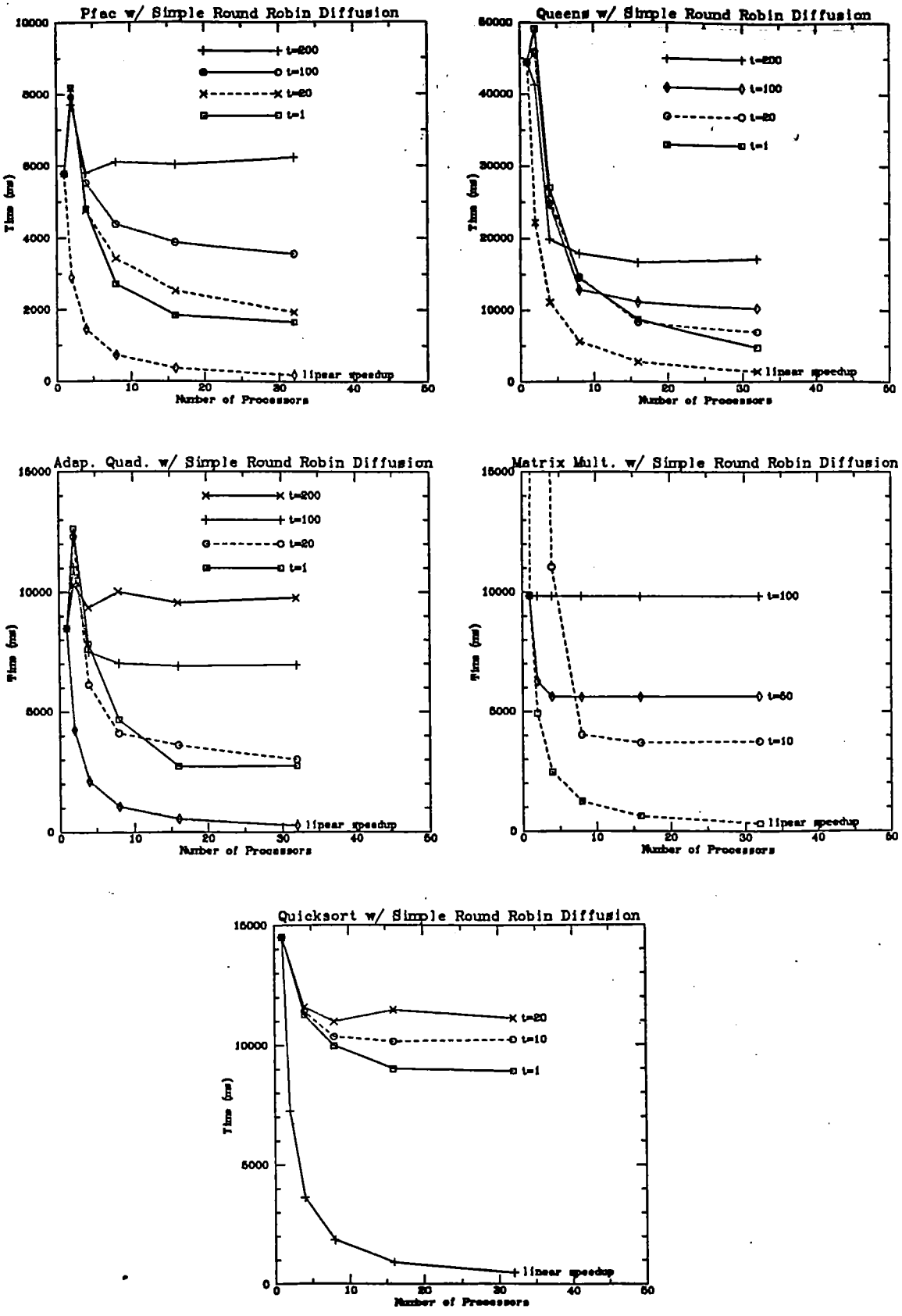


Figure 7.1: Alfalfa's performance using simple round-robin diffusion

- Only `matmult` performed poorly with $T = 1$ (in fact, it performed so poorly it could not be plotted on the same scale as the others in figure 7.1). This is because the high cost of sending large messages in order to distribute copies of the rows and columns of the matrices.

  Although `queens` and `pfac` have substantially different granularity, they both performed best on 32 processors when $T = 1$. Simple round-robin diffusion seemed to be more sensitive to message size (and thus message cost) than to the granularity of the program.

- No single value for T was appropriate for all sizes of hypercubes. If simple round-robin diffusion is used in a system, the T value should be adjusted to the size of the system.

In addition to execution times, we are interested in how well the work was distributed among the processors in the system. Table 7.1 lists the distribution performance for selected values of T for each program on 16 processors. The average number of tasks executed, nodes created, reducer messages sent ("Red"), and storage messages sent ("Strg") by the processors are listed as a function of their distance from the root processor (the processor which began the computation). From these distribution figures, we observe that:

- As expected, the work distribution was better when a small T was used than a large T. Naturally, a small T caused a greater number of messages to be sent.

- In systems with at least 16 processors the better distribution performance associated with a low T value outweighed the additional message cost. Only in `queens` did a high threshold ($T = 20$) give as good speedup as a low threshold ($T = 1$). When $T = 20$ the work was well distributed through the system while the number of messages was relatively small.

- Although `matmult` with $T = 1$ was distributed better than any other program, the performance was extremely poor. Again, this is because of the large amount of data that had to be sent between processors.

- Of all the programs, `queens` had the lowest message/task ratio. Not surprisingly, it performed the best of all the programs. Many of the tasks

PFAC: T = 1

Dist	Tasks	Nodes	Red	Strg
0	393.0	186.0	183.0	102.0
1	313.3	151.8	149.8	79.8
2	250.3	124.7	118.3	59.7
3	183.0	97.0	87.5	38.3
4	118.0	70.0	56.0	17.0

PFAC: T = 20

Dist	Tasks	Nodes	Red	Strg
0	650.0	281.0	240.0	164.0
1	420.0	196.5	124.5	75.8
2	267.0	147.3	48.0	10.2
3	16.5	12.0	7.5	0.0
4	0.0	0.0	0.0	0.0

QUEENS: T = 1

Dist	Tasks	Nodes	Red	Strg
0	2296.0	1120.0	640.0	348.0
1	2063.0	1024.0	567.5	291.3
2	1840.7	916.3	496.7	252.3
3	1659.0	849.5	436.0	198.0
4	1454.0	729.0	394.0	195.0

QUEENS: T = 20

Dist	Tasks	Nodes	Red	Strg
0	2307.0	1075.0	611.0	384.0
1	2663.5	1309.8	410.5	227.3
2	1945.2	974.2	235.5	116.2
3	1233.0	654.3	98.5	11.5
4	118.0	65.0	12.0	0.0

QUEENS: T = 100

Dist	Tasks	Nodes	Red	Strg
0	5202.0	2406.0	650.0	520.0
1	4889.8	2444.5	260.8	130.8
2	820.2	442.8	65.5	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: T = 1

Dist	Tasks	Nodes	Red	Strg
0	685.0	313.0	310.0	184.0
1	487.8	234.8	234.8	126.5
2	335.8	169.8	169.8	83.0
3	206.3	112.8	112.8	46.8
4	91.0	61.0	61.0	15.0

ADAPTIVE QUAD: T = 20

Dist	Tasks	Nodes	Red	Strg
0	1089.0	441.0	324.0	265.0
1	933.3	470.3	125.3	59.0
2	124.2	76.8	29.5	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: T = 100

Dist	Tasks	Nodes	Red	Strg
0	2735.0	1111.0	512.0	512.0
1	708.0	418.0	128.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

MATRIX MULT: T = 1

Dist	Tasks	Nodes	Red	Strg
0	234.0	117.0	116.0	58.0
1	230.5	115.5	115.5	57.5
2	233.0	116.3	116.3	58.3
3	234.5	117.0	117.0	58.8
4	230.0	116.0	116.0	57.0

MATRIX MULT: T = 10

Dist	Tasks	Nodes	Red	Strg
0	500.0	247.0	12.0	9.0
1	414.3	207.3	6.3	3.0
2	190.7	95.3	3.0	1.5
3	105.3	53.3	1.3	0.0
4	0.0	0.0	0.0	0.0

MATRIX MULT: T = 50

Dist	Tasks	Nodes	Red	Strg
0	1942.0	969.0	4.0	4.0
1	445.0	223.0	1.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

QUICKSORT: T = 1

Dist	Tasks	Nodes	Red	Strg
0	326.0	128.0	147.0	72.0
1	172.5	71.8	88.0	37.5
2	119.2	53.3	49.5	17.0
3	75.5	34.0	29.8	10.0
4	76.0	33.0	20.0	7.0

QUICKSORT: T = 10

Dist	Tasks	Nodes	Red	Strg
0	452.0	175.0	146.0	79.0
1	274.5	115.5	62.5	27.5
2	90.7	43.2	16.0	1.5
3	3.8	2.0	1.3	0.0
4	0.0	0.0	0.0	0.0

QUICKSORT: T = 20

Dist	Tasks	Nodes	Red	Strg
0	668.0	249.0	163.0	107.0
1	328.5	149.3	32.0	3.5
2	21.2	9.7	1.7	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

Table 7.1: Work distribution using simple round-robin diffusion

(serial combinator invocations) in queens are automatically executed locally without invoking the diffusion scheduler (because of the use of local spawn constructs in the program).

7.4.2 Dependent Round-Robin Diffusion

Dependent round-robin diffusion is identical to simple round-robin, except that the threshold task queue length is dependent on the dimension d of the hypercube multiprocessor. When a new task needs to be executed, it is sent to a neighboring processor iff

$$L \geq 2^{(E-(C*d))}$$

where E and C are constants and L is the task queue length. The values chosen for E and C determine the diffusion properties of the computation. A large E will cause more tasks to be executed locally than a small E. If E is large, the potential speedup is reduced. In a system with a small number of processors, though, less wasted communication occurs from sending tasks between busy processors. A large value of C will cause the queue length threshold to decrease quickly as processors are added.

The definition of the threshold was chosen rather arbitrarily. Another way of expressing the above formula for the threshold is

$$2^E/P^C$$

where P is the number of processors (equal to 2^d). Figure 7.2 presents the performance measurements for dependent round-robin diffusion and highlights the following points:

- For each program, fixed values of E and C performed well over a range of hypercube sizes.
- In the programs that generated many tasks, a high value of E (either 12 or 15) and a relatively high value of C (generally 3) worked best. When the number of processors is low the threshold is very high and most tasks are executed locally. The large value of C means that as processors are added, the threshold decreases quickly and the computation gets well distributed.

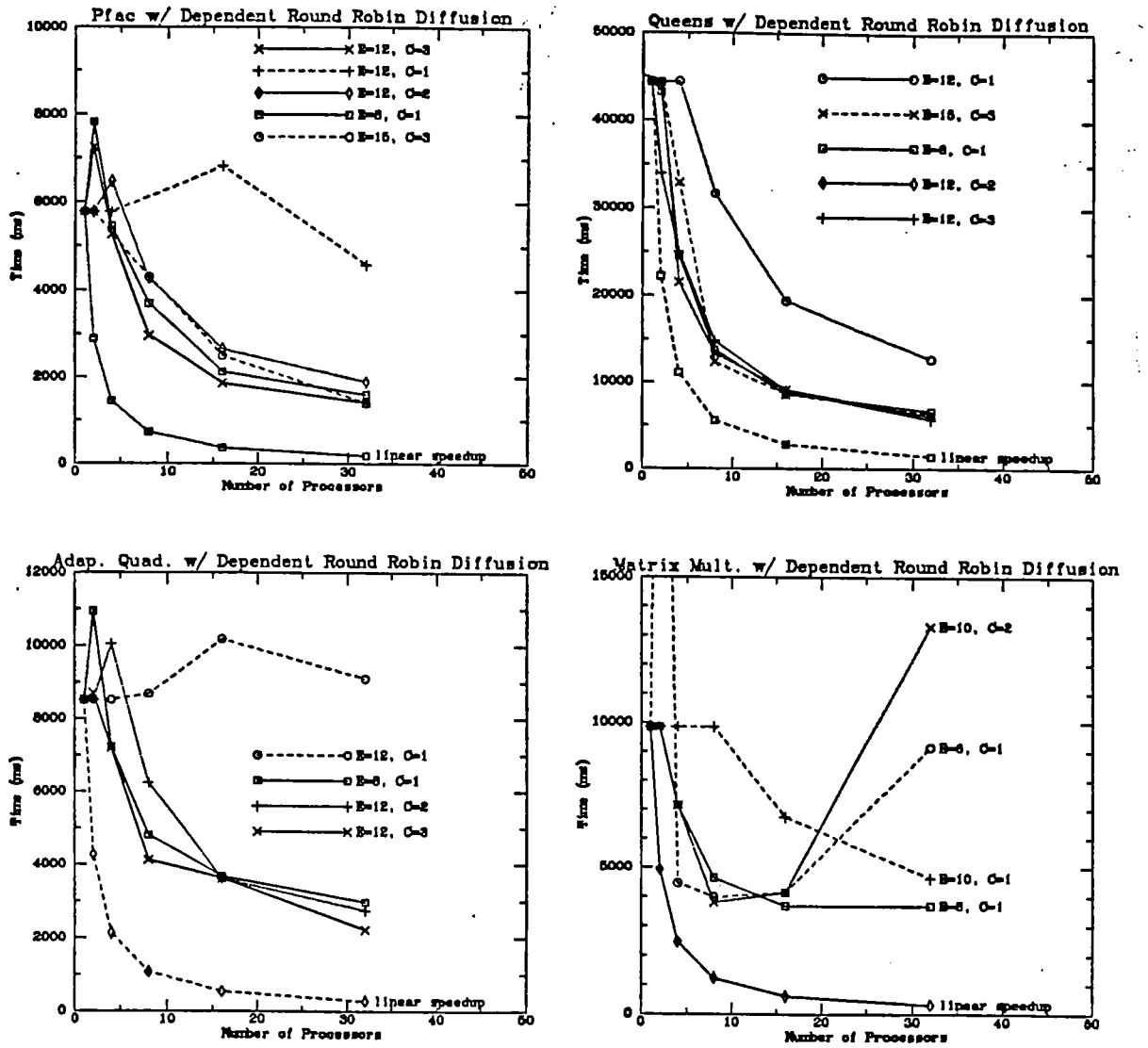


Figure 7.2: Alfalfa's performance using dependent round-robin diffusion

- In `matmult` a small C value ($C = 1$) performed best. The message sizes are apparently too large to permit a low threshold even for large numbers of processors.

Table 7.2 lists the distribution performance for selected values of of E and C for each program on 16 processors.

7.4.3 Ratio Round-Robin Diffusion

One of the problems with both the simple and the dependent round-robin methods is that no parallelism is exploited until the queue length on the root processor has surpassed the threshold. If the threshold (either T or $2^{E-(C*d)}$) is large there may be a substantial period at the start of the computation in which only the root processor is computing. Reducing the threshold to avoid this problem may be unacceptable if the machine has relatively few processors.

Ratio round-robin diffusion causes a task to be sent to a neighboring processor when the *ratio* of t_l, the number of tasks that the scheduler decided to execute locally, to t_r, the number of tasks sent to remote processors, is greater than some threshold value. As in dependent round-robin, the value of the threshold depends on the number of processors. A new task is sent to a neighbor iff

$$\frac{t_l}{t_r} \geq 2^{E-(C*d)}$$

The first task is always to a neighboring processor.

This has the advantage of spreading the computation out from the root processor sooner than the other diffusion methods described so far. However, the ratio t_l/t_r may not be nearly as accurate an indication of a processor's load, since it does not account for tasks received from other processors. Figure 7.3 presents the results of using ratio round-robin diffusion. Ratio round-robin diffusion performed better on a small number of processors than dependent round-robin diffusion. Overall, the performance was similar to dependent round-robin. Table 7.3 lists the distribution performance for various values of E and C. The distribution figures indicate that:

- For `pfac`, `queens`, and `quad`, the work distribution using ratio round-robin diffusion was very similar to the work distribution using dependent

PFAC: E = 12 C = 1

Dist	Tasks	Nodes	Red	Strg
0	2490.0	1001.0	488.0	488.0
1	377.0	249.5	122.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

PFAC: E = 12 C = 2

Dist	Tasks	Nodes	Red	Strg
0	618.0	271.0	238.0	157.0
1	391.3	182.8	136.8	81.3
2	265.5	142.2	62.5	21.8
3	55.5	36.0	16.5	0.0
4	0.0	0.0	0.0	0.0

PFAC: E = 12 C = 3

Dist	Tasks	Nodes	Red	Strg
0	431.0	202.0	199.0	113.0
1	334.5	160.5	158.5	86.0
2	242.3	122.0	119.0	58.7
3	170.0	92.0	83.5	34.8
4	95.0	55.0	55.0	20.0

QUEENS: E = 12 C = 1

Dist	Tasks	Nodes	Red	Strg
0	11034.0	5226.0	582.0	582.0
1	4662.0	2403.8	145.5	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

QUEENS: E = 12 C = 2

Dist	Tasks	Nodes	Red	Strg
0	2293.0	1070.0	619.0	386.0
1	2372.8	1169.8	430.8	232.0
2	1976.3	986.7	269.3	136.2
3	1398.0	732.8	136.5	34.5
4	448.0	241.0	34.0	0.0

QUEENS: E = 12 C = 3

Dist	Tasks	Nodes	Red	Strg
0	2396.0	1152.0	670.0	381.0
1	2085.3	1037.8	566.3	288.0
2	1815.5	905.8	485.2	244.5
3	1634.3	837.3	427.3	193.5
4	1515.0	754.0	393.0	200.0

ADAPTIVE QUAD: E = 12 C = 1

Dist	Tasks	Nodes	Red	Strg
0	4855.0	2157.0	540.0	540.0
1	178.0	156.5	135.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: E = 12 C = 2

Dist	Tasks	Nodes	Red	Strg
0	972.0	394.0	301.0	242.0
1	820.5	411.5	120.5	59.0
2	217.7	123.0	30.7	1.2
3	1.8	1.3	0.8	0.0
4	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: E = 12 C = 3

Dist	Tasks	Nodes	Red	Strg
0	690.0	310.0	307.0	188.0
1	461.0	223.5	223.5	118.8
2	318.0	163.7	163.7	77.2
3	233.0	124.0	124.0	54.5
4	193.0	101.0	101.0	46.0

MATRIX MULT: E = 6 C = 1

Dist	Tasks	Nodes	Red	Strg
0	276.0	135.0	10.0	8.0
1	400.8	200.8	6.3	2.8
2	205.3	102.7	3.7	1.8
3	124.5	62.5	2.0	0.8
4	113.0	57.0	1.0	0.0

MATRIX MULT: E = 8 C = 1

Dist	Tasks	Nodes	Red	Strg
0	588.0	291.0	10.0	8.0
1	468.5	234.5	5.0	2.3
2	159.5	79.8	2.2	1.0
3	75.8	38.3	0.8	0.0
4	0.0	0.0	0.0	0.0

MATRIX MULT: E = 10 C = 1

Dist	Tasks	Nodes	Red	Strg
0	2401.0	1198.0	5.0	5.0
1	330.3	165.8	1.3	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

Table 7.2: Work distribution using dependent round-robin diffusion

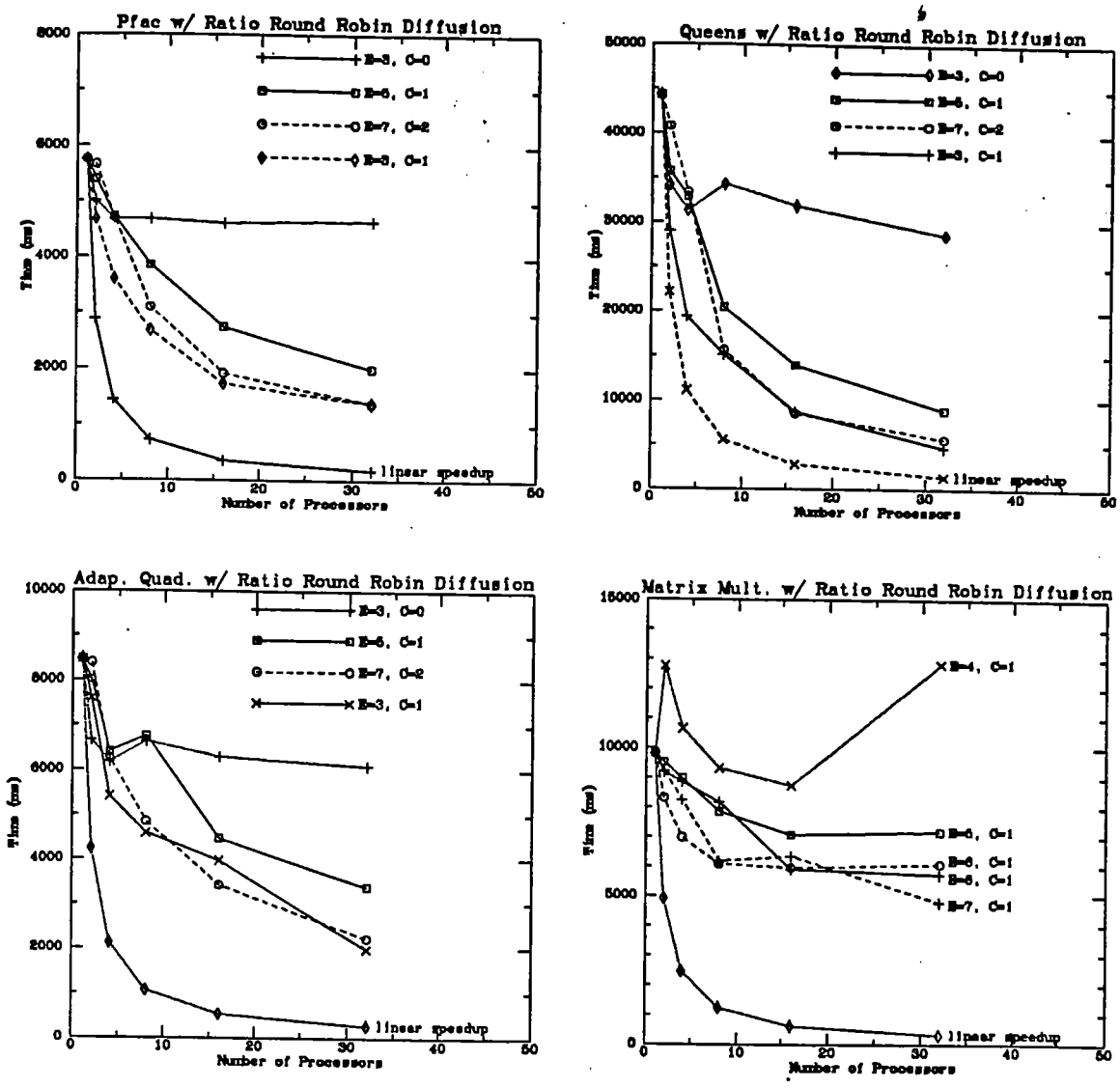


Figure 7.3: Alfalfa's performance using ratio round-robin diffusion

PFAC: E = 3 C = 0

Dist	Tasks	Nodes	Red	Strg
0	2879.0	1402.0	89.0	82.0
1	265.8	139.8	27.3	6.8
2	9.3	6.3	3.3	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

PFAC: E = 3 C = 1

Dist	Tasks	Nodes	Red	Strg
0	336.0	161.0	160.0	87.0
1	287.3	140.8	140.8	73.3
2	246.0	123.7	123.7	61.2
3	212.0	109.0	109.0	51.5
4	189.0	97.0	97.0	46.0

PFAC: E = 5 C = 1

Dist	Tasks	Nodes	Red	Strg
0	1311.0	616.0	151.0	115.0
1	442.3	220.8	72.3	36.5
2	128.7	69.3	29.0	9.5
3	36.0	20.5	9.5	2.3
4	2.0	2.0	2.0	0.0

QUEENS: E = 3 C = 0

Dist	Tasks	Nodes	Red	Strg
0	18626.0	9176.0	334.0	304.0
1	2193.8	1122.0	109.3	29.5
2	370.7	190.3	19.3	4.7
3	14.3	8.8	3.3	0.0
4	0.0	0.0	0.0	0.0

QUEENS: E = 3 C = 1

Dist	Tasks	Nodes	Red	Strg
0	2169.0	1069.0	607.0	319.0
1	2053.3	1023.5	569.8	288.0
2	1853.0	915.3	516.3	269.3
3	1686.5	864.3	454.0	206.0
4	1436.0	729.0	390.0	184.0

QUEENS: E = 5 C = 1

Dist	Tasks	Nodes	Red	Strg
0	7677.0	3709.0	555.0	407.0
1	3197.3	1594.0	287.3	148.3
2	1229.3	630.7	130.7	49.3
3	420.8	222.3	52.3	14.3
4	157.0	83.0	19.0	5.0

ADAPTIVE QUAD: E = 3 C = 0

Dist	Tasks	Nodes	Red	Strg
0	3701.0	1803.0	116.0	105.0
1	423.5	219.5	37.5	11.0
2	27.8	16.5	5.8	0.3
3	1.3	0.8	0.3	0.0
4	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: E = 3 C = 1

Dist	Tasks	Nodes	Red	Strg
0	657.0	300.0	300.0	178.0
1	469.3	227.0	227.0	121.0
2	332.8	168.8	168.8	82.0
3	225.8	120.8	120.8	52.5
4	133.0	79.0	79.0	27.0

ADAPTIVE QUAD: E = 5 C = 1

Dist	Tasks	Nodes	Red	Strg
0	1338.0	640.0	175.0	116.0
1	701.5	341.5	101.0	59.8
2	210.3	112.7	47.0	16.0
3	39.5	24.5	14.0	2.3
4	3.0	3.0	3.0	0.0

MATRIX MULT: E = 3 C = 1

Dist	Tasks	Nodes	Red	Strg
0	237.0	118.0	117.0	59.0
1	232.3	116.3	116.3	58.0
2	232.7	116.3	116.3	58.2
3	232.8	116.3	116.3	58.3
4	229.0	115.0	115.0	57.0

MATRIX MULT: E = 5 C = 1

Dist	Tasks	Nodes	Red	Strg
0	251.0	124.0	39.0	21.0
1	226.3	113.8	38.3	18.5
2	225.7	113.0	37.7	18.7
3	232.3	116.3	38.8	19.3
4	283.0	139.0	43.0	24.0

MATRIX MULT: E = 7 C = 1

Dist	Tasks	Nodes	Red	Strg
0	1241.0	608.0	45.0	35.0
1	342.0	172.0	20.0	9.0
2	131.8	66.8	8.5	3.3
3	71.3	36.3	4.8	1.8
4	37.0	19.0	3.0	1.0

Table 7.3: Work distribution using ratio round-robin diffusion

round-robin diffusion.

- For `pfac`, while the distribution with ratio round-robin diffusion was better than dependent round-robin, the message load was significantly higher. This accounts for ratio round-robin's poorer execution time performance.

7.5 Communicating Diffusion Scheduling

Alfalfa was tested on the five application programs using two communicating diffusion algorithms— *simple communicating diffusion* and *dependent communicating diffusion.* Both of these diffusion strategies have identical location policies. Once the transfer policy has determined that a task should be sent to a neighboring processor, the neighbor with the smallest reported queue length will be chosen.

7.5.1 Simple Communicating Diffusion

In simple communicating diffusion, the update policy is as follows: The load of a processor is reported to its neighbors whenever its task queue length L differs by at least a factor of two from the previously reported value[4] *and* either the old value or new value is greater than some threshold M. Since a processor's task queue length will often fluctuate between 0 and 1 or 1 and 2, M should be large enough to prevent messages from being sent due to minor changes in load. A large value of M means that fewer update messages will be sent by lightly loaded processors. The factor-of-two requirement was chosen somewhat arbitrarily, and arose from the observation that small changes in queue length are more significant on a lightly loaded processor than on a heavily loaded processor. The factor-of-two requirement reduces the number of update messages sent by heavily loaded processors.

When a new task needs to be executed, a neighboring processor p is chosen iff

$$L_p \leq L - R$$

where:

[4]We refer to this restriction as the *factor-of-two requirement.*

- L is the local task queue length,
- R is a constant, and
- L_p, the reported load of p, is the lowest reported load of all neighboring processors.

A large R value will cause fewer tasks to be sent to neighboring processors than a small R value.

The execution times using simple communicating diffusion are plotted in figure 7.4. These results indicate that:

- Low values for R performed best on all programs. However, except for `queens`, small values of R for small numbers of processors did poorly. Note the similarity with simple round-robin diffusion.
- On 32 processors, the best simple communicating diffusion policy performed no better than the best non-communicating policy. This reinforces the observation that a large number of processors will allow a simple diffusion policy to perform well.
- Performance was relatively insensitive to the value of M. Once the number of tasks on the local task queue surpassed M, the factor-of-two requirement provides a much greater constraint on the number of update messages sent. Not surprisingly, performance was much more sensitive to R.
- When $R = 20$, an M value of 50 tended to do better than an M value of 20. Apparently, the extra information provided by a lower M value was not worth the extra communication required.

Table 7.4 lists the distribution performance for various values of R and M on 16 processors. The number of system messages ("sys") used to perform the load updates is listed, as well as the other types of messages.

The distribution statistics indicate that:

- The distribution of work was determined primarily by the value of R.
- Even for small values of M, the number of system messages per processor was low. This would indicate that the task queue length of each processor rarely changes significantly (as defined by the factor-of-two requirement).

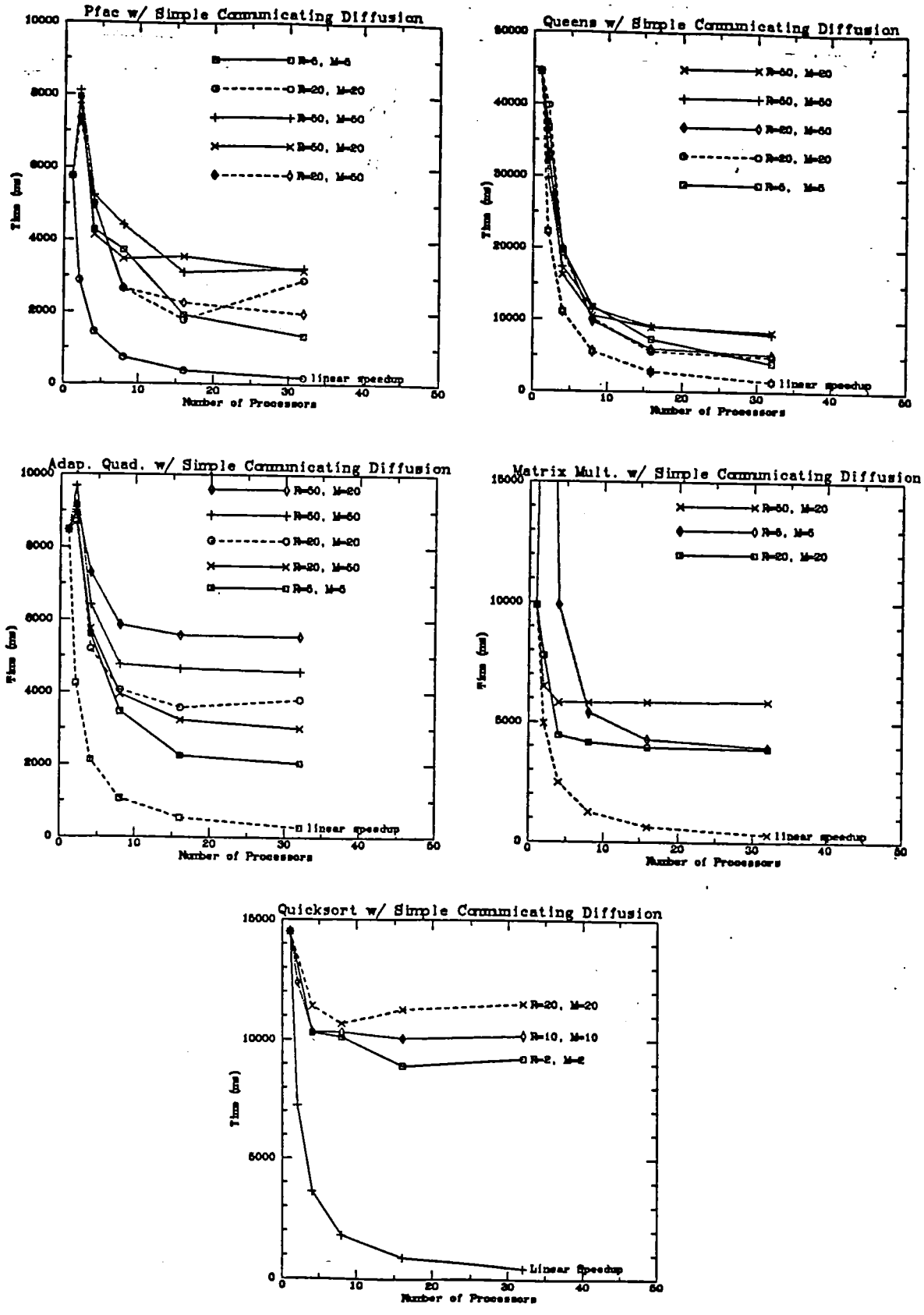


Figure 7.4: Alfalfa's performance using simple communicating diffusion

PFAC: R = 5 , M = 5

Dist	Tasks	Nodes	Red	Strg	Sys
0	376.0	175.0	110.0	68.0	13.0
1	308.8	144.8	109.3	64.3	10.8
2	257.3	123.7	87.0	48.5	9.5
3	185.0	108.0	58.5	13.8	9.0
4	103.0	71.0	39.0	0.0	4.0

PFAC: R = 50 , M = 50

Dist	Tasks	Nodes	Red	Strg	Sys
0	1150.0	453.0	250.0	247.0	6.0
1	506.8	249.3	131.8	70.0	2.0
2	136.8	91.5	46.2	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

PFAC: R = 50 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	1180.0	455.0	270.0	270.0	8.0
1	571.3	294.3	117.8	50.3	5.5
2	88.8	61.2	33.5	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

PFAC: R = 20 , M = 50

Dist	Tasks	Nodes	Red	Strg	Sys
0	624.0	255.0	214.0	164.0	2.0
1	402.0	181.0	147.0	93.5	2.3
2	277.8	157.5	70.8	16.8	0.0
3	24.8	18.8	12.8	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

QUEENS: R = 20 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	2507.0	1148.0	229.0	220.0	4.0
1	2183.5	1042.5	208.5	153.5	4.5
2	1895.2	958.8	182.8	80.2	8.0
3	1504.0	796.3	152.0	31.8	7.3
4	1054.0	585.0	116.0	0.0	6.0

QUEENS: R = 20 , M = 50

Dist	Tasks	Nodes	Red	Strg	Sys
0	2144.0	1003.0	238.0	188.0	4.0
1	2080.5	1004.5	256.5	164.0	3.0
2	2029.3	1016.8	238.3	117.0	2.7
3	1532.0	805.0	189.0	55.5	2.5
4	912.0	499.0	114.0	14.0	2.0

QUEENS: R = 50 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	4184.0	1891.0	402.0	402.0	6.0
1	3509.5	1734.8	241.0	140.5	4.5
2	1910.0	1001.8	93.7	0.0	6.3
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: R = 5 , M = 5

Dist	Tasks	Nodes	Red	Strg	Sys
0	641.0	275.0	194.0	142.0	22.0
1	506.5	227.5	134.0	92.8	16.3
2	334.3	175.0	95.3	39.8	12.7
3	192.8	114.3	74.8	19.5	7.3
4	123.0	91.0	69.0	5.0	2.0

ADAPTIVE QUAD: R = 20 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	1185.0	451.0	282.0	282.0	10.0
1	706.8	343.3	161.3	90.8	6.0
2	256.2	157.8	61.5	1.0	0.3
3	4.5	3.0	1.5	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

ADAPTIVE QUAD: R = 20 , M = 50

Dist	Tasks	Nodes	Red	Strg	Sys
0	952.0	362.0	233.0	230.0	4.0
1	678.5	327.5	138.5	81.0	0.0
2	316.8	185.2	53.5	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

MATRIX MULT: R = 5 , M = 5

Dist	Tasks	Nodes	Red	Strg	Sys
0	253.0	124.0	9.0	7.0	2.0
1	271.8	135.3	8.8	5.0	0.5
2	243.0	121.7	6.0	2.8	0.0
3	231.0	116.5	2.5	0.3	0.0
4	0.0	0.0	0.0	0.0	0.0

MATRIX MULT: R = 20 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	809.0	402.0	7.0	6.0	0.0
1	487.0	243.5	3.0	1.5	0.0
2	160.8	80.8	0.8	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

QUICKSORT: R = 2 , M = 2

Dist	Tasks	Nodes	Red	Strg	Sys
0	327.0	124.0	99.0	58.0	31.0
1	165.8	69.0	57.8	24.5	17.0
2	139.3	61.0	39.8	14.5	8.5
3	67.3	32.3	21.3	3.5	3.8
4	14.0	9.0	9.0	0.0	0.0

QUICKSORT: R = 20 , M = 20

Dist	Tasks	Nodes	Red	Strg	Sys
0	502.0	186.0	123.0	82.0	2.0
1	375.0	165.0	38.5	10.8	1.0
2	17.8	9.7	7.2	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

Table 7.4: Work distribution using simple communicating diffusion

- Modifying the values of M makes relatively little difference in the distribution performance or in the number of messages sent. Again, this is because of the factor-of-two requirement.

7.5.2 Dependent Communicating Diffusion

Both the transfer and the update policies in dependent communication diffusion depend on the number of processors in the system. The update policy dictates that an update message is sent to neighboring processors when the local task queue length L changes by a factor of two and either the new value or the old value of L is greater than

$$2^{(E_u-(C_u*d))}$$

where E_u and C_u are constants and d is the dimension of the hypercube. In this case, a large E_u will cause fewer update messages to be sent than a small E_u.

A task is sent to a neighboring processor p iff, for constants E and C,

$$l_p \leq L - 2^{(E-(C*d)}$$

and l_p is the lowest reported load of all neighboring processors. Figure 7.5 illustrates the performance of the programs for various values of E, C, E_u (labeled `UE` in the graph), and C_u (labeled `UC`). The following aspects of the performance of dependent communicating diffusion are worth mentioning:

- High values of E (10 or 12) and C (around 3) performed the best. In a system with many processors, a small difference between the task queue length of a processor and of its neighbor should cause a task to be sent to that neighbor.When there are fewer processors, however, the difference between the load of a processor and the load of a neighbor should be large before a task is be sent.

- A high E_u value and a low C_u value performed best. This indicates that even in the presence of a large number of processors, the number of update messages should be kept relatively small.

The distribution results for dependent communicating diffusion are shown in table 7.5. They show that the distribution of work is also relatively insensitive

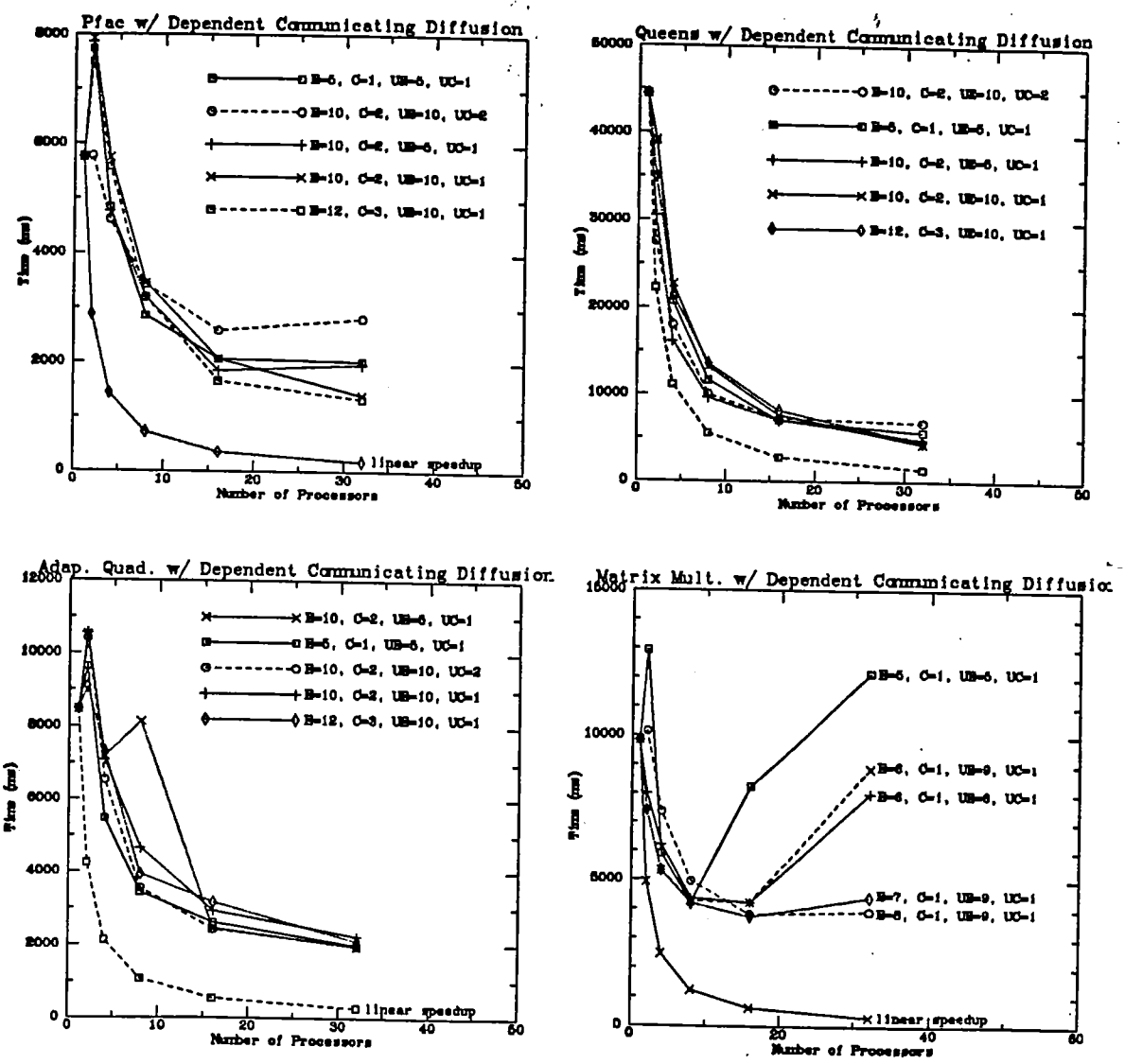


Figure 7.5: Alfalfa's performance using dependent communicating diffusion

PFAC: $E = 5$ $C = 1$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	313.0	142.0	119.0	74.0	16.0
1	262.0	125.0	106.0	59.0	12.0
2	247.8	123.2	104.5	53.0	15.3
3	232.8	122.3	111.8	50.0	15.3
4	219.0	129.0	125.0	43.0	6.0

PFAC: $E = 10$ $C = 2$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	395.0	178.0	139.0	89.0	23.0
1	303.0	146.0	125.0	68.0	13.3
2	257.2	127.2	90.2	46.5	16.5
3	178.3	96.8	62.8	23.8	15.0
4	135.0	87.0	71.0	16.0	10.0

PFAC: $E = 10$ $C = 2$ $E_u = 10$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	493.0	212.0	203.0	136.0	2.0
1	353.5	168.0	160.5	89.0	1.5
2	234.0	119.3	108.0	51.7	0.0
3	161.0	92.5	56.0	16.0	0.0
4	43.0	29.0	15.0	0.0	0.0

QUEENS: $E = 5$ $C = 1$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	1711.0	821.0	351.0	210.0	40.0
1	1779.3	891.5	401.8	199.0	43.8
2	1850.0	925.7	419.3	209.0	38.2
3	1973.0	985.0	426.5	214.8	47.8
4	1862.0	960.0	446.0	194.0	36.0

QUEENS: $E = 10$ $C = 2$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	1814.0	887.0	302.0	171.0	24.0
1	1713.8	852.5	314.8	161.8	27.8
2	1861.2	918.2	360.5	192.7	40.7
3	1895.0	960.8	392.5	183.0	35.3
4	2266.0	1192.0	364.0	123.0	75.0

QUEENS: $E = 10$ $C = 2$ $E_u = 10$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	2364.0	1149.0	384.0	225.0	4.0
1	1949.3	943.5	404.8	233.5	2.0
2	1648.7	816.5	400.7	208.2	3.0
3	2109.5	1103.3	362.0	132.5	1.0
4	1191.0	606.0	279.0	129.0	2.0

ADAP. QUAD: $E = 5$ $C = 1$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	529.0	235.0	206.0	132.0	30.0
1	386.3	186.8	151.3	82.0	22.5
2	340.3	173.0	132.0	63.2	29.5
3	300.5	155.5	127.5	58.5	24.3
4	249.0	141.0	117.0	42.0	23.0

ADAP QUAD: $E = 10$ $C = 2$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	573.0	229.0	148.0	131.0	39.0
1	492.8	227.8	141.3	89.3	33.5
2	352.0	182.3	107.3	47.3	28.7
3	209.8	123.3	66.8	15.0	16.5
4	72.0	56.0	40.0	0.0	11.0

MATRIX MULT: $E = 5$ $C = 1$ $E_u = 5$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	191.0	94.0	59.0	31.0	8.0
1	218.5	109.0	46.0	23.3	5.8
2	254.5	127.5	38.8	19.2	4.7
3	205.8	102.8	34.3	17.3	4.8
4	307.0	155.0	29.0	13.0	6.0

MATRIX MULT: $E = 6$ $C = 1$ $E_u = 9$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	712.0	351.0	20.0	15.0	0.0
1	465.0	232.5	7.5	3.8	0.0
2	191.7	96.7	1.7	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

MATRIX MULT: $E = 7$ $C = 1$ $E_u = 9$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	300.0	147.0	10.0	8.0	0.0
1	467.5	234.5	6.5	2.5	0.0
2	195.5	97.8	3.5	1.7	0.0
3	94.8	47.3	1.3	0.8	0.0
4	0.0	0.0	0.0	0.0	0.0

MATRIX MULT: $E = 8$ $C = 1$ $E_u = 9$ $C_u = 1$

Dist	Tasks	Nodes	Red	Strg	Sys
0	580.0	287.0	8.0	7.0	0.0
1	415.0	207.5	4.0	2.0	0.0
2	193.2	96.8	1.8	0.7	0.0
3	80.8	40.8	0.8	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

Table 7.5: Work distribution using dependent communicating diffusion

to the number of system messages. The transfer policy dictated by E and C has a much greater affect on load distribution than the update policy, E_u and C_u.

7.6 Comparing the Diffusion Methods

In figure 7.6 the best performance of each diffusion method for each program is plotted. The most striking aspect of these results is that, at least in Alfalfa, there is surprisingly little difference between the best performances of the various diffusion algorithms on 32 processors. However, certain patterns do develop when the programs are considered on a case-by-case basis.

- `Pfac`: For `pfac` the round-robin diffusion strategies worked as well as the communicating strategies. A large number of fine-grained tasks are generated during execution. Each processor sends enough tasks to its neighbors to spread the computation through the system sufficiently. Comparing the distribution figures, the task distribution for `pfac` using non-communicating methods was as good as the communicating methods. In either case, the fine-grained nature of `pfac` could be seen in the high ratio of messages to tasks when the queue length thresholds were low.

- `Queens`: For `queens` the communicating methods performed significantly better than the non-communicating methods in systems with sixteen processors or less. The distribution figures show that a far greater number of messages were sent using the non-communicating diffusion methods. Apparently, the queue length thresholds were surpassed in the non-communicating strategies when all the processors in the system were heavily loaded. The few update messages in the communicating strategies served to increase the number of tasks executed locally when the overall system load was high.

- `Adaptive Quad`: For this program, the communicating diffusion algorithms performed better than the non-communicating in systems of less than 32 processors. Although the distribution performance was worse using the communicating methods, fewer messages were sent.

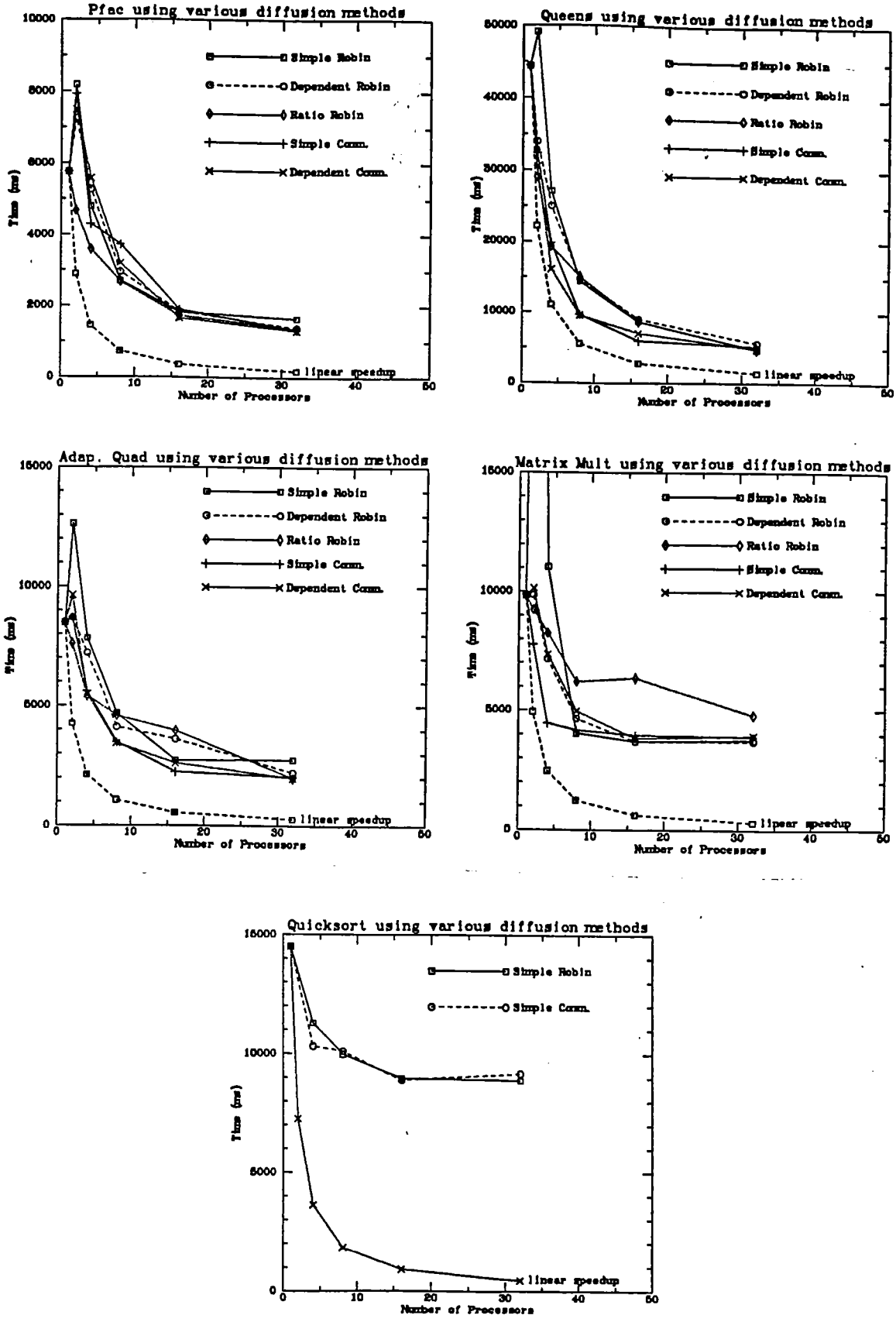


Figure 7.6: Comparing diffusion methods for each program

- `Matmult`: On small numbers of processors, the communicating diffusion policies clearly outperformed the non-communicating policies, although neither did particularly well. On 16 processors, the performance of the communicating and the non-communicating methods were about the same. While the task distribution for the non-communicating methods was better, the communicating methods caused fewer messages to be sent.

- `Quicksort`:Both non-communicating and communicating diffusion performed equal poorly on `quicksort`. Only on four processors did the communicating method perform better. As with the other programs, the non-communicating method provided better work distribution while the communicating method caused fewer messages to be sent.

In every case, the number of update messages required to implement the communicating diffusion strategies was small, compared to the number of system messages sent and the number of tasks executed. This may be because the factor-of-two requirement overly restricted the flow of load information between neighbors. Processors may have been forced to rely on out-of-date information about their neighbors.

7.7 Conclusions

Our experiments were designed for two purposes:

1. To find effective dynamic scheduling techniques for executing serial combinators on distributed memory multiprocessors, and
2. To gauge the effectiveness of our approach to automatic partitioning of functional programs for this class of architectures.

7.7.1 Choosing a Diffusion Policy

We saw that the communicating diffusion policies performed better than the non-communicating policies in systems with small numbers of processors. In other words, the communicating diffusion methods performed better on systems where all the processors became saturated with work. If a program is very large

or if many programs are running simultaneously, communicating diffusion will probably perform better on large multiprocessor systems as well.

The communicating strategies performed better because fewer messages were sent by each processor even though the non-communicating policies had better task distribution. In a loosely-coupled system where communication is cheap, the non-communicating methods may actually perform better due to better task distribution.

When the number of processors was large, the non-communicating methods performed as well as the communicating methods. In a large system in which most of the processors are not heavily loaded it is less likely for a task to be sent by one busy processor to another busy processor.

Related Analytical Results

A fair amount of analytical work has been performed to determine appropriate dynamic load balancing strategies for loosely coupled multiprocessors [14,54]. A recent analytical study of dynamic load balancing performed at the University of Washington [18] concluded that, on average, a simple and inexpensive diffusion strategy will perform just a well as a sophisticated diffusion strategy. The scheduling methods that were analyzed, however, were somewhat different from those used by Alfalfa. Even so, Alfalfa has provided strong empirical evidence to support the conclusion.

7.7.2 Alfalfa's Performance

Even though the compiler goes to great lengths to make serial combinators as coarse-grained as possible, the performance of Alfalfa was still largely program dependent. As it turned out, the granularity of the source program affected Alfalfa's performance *less* than the existence of large shared data structures that needed to be replicated throughout the system or data structures whose access is inherently sequential.

Considering the extremely high communication costs of the Intel iPSC, Alfalfa performed quite well on three of the five programs: `pfac`, `queens`, and `quad`. Each of these programs had little, if any, shared data and each exhibited

a large amount of inherent parallelism. On `queens`, Alfalfa performed especially well; the compiler was able to make good use of the practically sequential portions of the program. The benefits of the serial combinator approach were illustrated in the low message to task ratio in the statistics for `queens`.

On `matmult` Alfalfa's performance was disappointing. Although the execution time was reduced significantly on a small number of processors, the amount of additional parallelism exploited as processors were added was minimal. Unlike the three program that performed well, this program required a significant amount of data to be sent between processors. The figures at the beginning of the chapter indicate that the communication costs in the Intel iPSC are roughly proportional to the size of the messages. This is probably responsible for the inability of Alfalfa to exploit the parallelism in `matmult` on a large scale.

A solution to this problem would be to explicitly partition the matrix and fully distribute it when the program is loaded. One way to accomplish this would be to provide the programmer with the ability to specify how the data in a program should be partitioned (as in para-functional programming [30,35,31]). Alternatively, it may be possible to devise compiler algorithms to perform *data partitioning* automatically. This is beyond the capability of our serial combinator compiler.

Alfalfa performed the worst on `quicksort`, apparently because the available parallelism was limited due to the small list of numbers was sorted. Other researchers have reported that they achieved reasonable speedups for quicksort using larger lists[38].

Chapter 8

Buckwheat: Graph Reduction on a shared memory multiprocessor

Buckwheat is an implementation of a heterogeneous graph reducer on the Encore Multimax, a shared memory (tightly coupled) multiprocessor [19]. In many ways, Buckwheat is similar to Alfalfa. We therefore present a brief description of Buckwheat that covers only those aspects that differ from Alfalfa.

8.1 The Encore Multimax

The Encore Multimax is a bus-based shared memory multiprocessor. Buckwheat was implemented on a system that contained twelve processors. Each processor is a 10 MHz National Semiconductor NS32032 microprocessor. Two processors fit onto a card called the Dual Processor Card and share a cache. The memory resides on cards called Shared Memory Cards, each of which can hold four megabytes of storage. Any location in memory can be accessed by any processor over a very fast bus called the Nanobus, which has a data transfer rate of 100 Megabytes per second. Each shared memory card has a cycle time of 320 nanoseconds and can send or receive up to eight bytes of data with each bus transaction.

An important feature of the shared memory in the Multimax is that any

byte can be used as a lock (for enforcing mutual exclusion, etc.). Atomic test-and-set instructions set and reset these locks.

The operating system, UMAX 4.2, is a multiprocessor extension of Unix that supports multiprogramming as well as multiprocessing. Processes are scheduled according to processor availability: If a single process creates new processes by invoking the `fork()` routine, the new processes are able to run in parallel with parent process. In order for the parent and child processes to share data, the parent must explicitly declare an area of memory to be shared before it creates the child processes.

A call to `fork()` creates a heavyweight process (a full Unix process containing all the state needed to execute independently). Encore has also provided a multi-tasking library containing routines for creating lightweight processes (unfortunately, this library was not available when Buckwheat was being implemented).

Currently, there is no way to explicitly map Unix processes onto processors to ensure that the processes will run in parallel. However, it has been our experience that if the number of active processes in the system is less than the number of processors, each active process will be executed on a different processor. When executing Buckwheat, we were careful to ensure that the number of active processes did not surpass the number of available processors.

Given this, the distinction between a process and the processor executing it is unimportant. In the following sections we describe the organization of Buckwheat on the each of the Multimax *processors*, even though we really mean *processes*. On a shared-memory multiprocessor that allowed the user to map processes to processors, the following description of Buckwheat would be completely accurate.

8.2 Shared Memory Graph Reduction

In graph reduction, the program graph logically resides in a single graph space. Thus, a shared-memory (tightly-coupled) multiprocessor is the most natural architecture on which to implement graph reduction.

In Alfalfa, the graph reduction system on a loosely-coupled multiprocessor,

a great deal of overhead was required to implement a single graph space that encompassed the individual memories in the system. This overhead (in both space and time) included:

1. An extra field in each node pointer to specify a processor address as well as a node address in the memory of that processor (see section 6.2.1).
2. Message passing between processors in order to perform transformations on the distributed program graph (see section 6.2.4).
3. Using diffusion scheduling algorithms to choose remote processors to allocate work on. These algorithms may involve a significant amount of communication or computation or both (chapter 7).
4. Message passing to support distributed reference-counting storage reclamation (see section 6.2.5).

On the Multimax, however, this overhead is unnecessary for the following reasons:

1. Since the graph lies in a single address space supported by the hardware, a 32-bit address is sufficient to access any node in the graph.
2. Any processor can access any component of the program graph. Therefore, no communication between processors needs to occur when a transformation is performed on the graph. Naturally, access to any part of the graph being mutated must be restricted to the processor performing the mutation.

   Similarly, no interprocessor communication is required to perform reference counting. Only mutual exclusion to a node's reference count is required when the reference count is being modified.
3. In Buckwheat, the processors are *self-scheduled.* That is, when a processor becomes free, it takes a task from a shared task queue and performs the action dictated by the task. No processor needs to be aware of the state of any other processor in the system. Scheduling based on shared queues is discussed in detail in section 8.3.

Buckwheat executes serial combinators generated by the serial combinator compiler described in chapters 4 and 5. Although code generation for Buck-

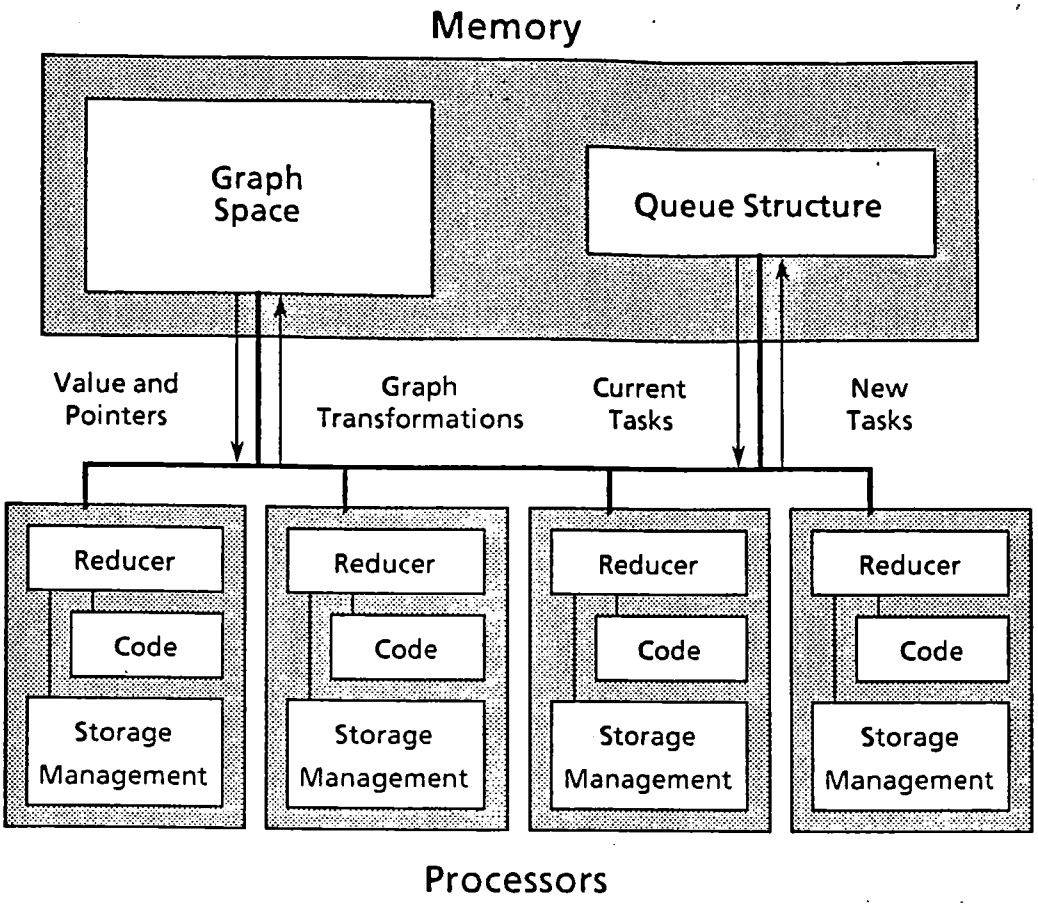


Figure 8.1: The Buckwheat system

wheat differs from code generation for Alfalfa, the differences are relatively minor and do not significantly affect the manner in which the program graph is transformed during execution. No further discussion of code generation for Buckwheat is presented here.

8.2.1 System Organization

The Buckwheat implementation is essentially a simplified version of Alfalfa. The components of the Alfalfa system that support message passing and diffusion scheduling are unnecessary in Buckwheat. Thus, Buckwheat contains neither a message handler nor a dynamic scheduler.

The organization of buckwheat is pictured in figure 8.1. Each processor

has a private copy of the graph reducer module, serial combinator code, and storage manager. Even though the Multimax has a single physical memory, multiple copies of these module allow the processors to execute the routines without memory contention. Of course, there may still be contention for the bus. However, because the Nanobus is so fast and the processors have caches, the effect of bus contention is minimal.

The graph space and task queue structure reside in a shared area of memory. In its simplest form, the queue structure consists of a single queue from which all processors access tasks to be executed. A more sophisticated task queue structure is described in section 8.3.

Although each processor has a copy of the storage management routines, the graph structures (nodes, tasks, and extends) are allocated in shared memory. Like Alfalfa, Buckwheat keeps free nodes, tasks, and extends on a free-list. If all free structures were kept on a single shared free-list, significant contention for the head of the free-list would arise. Buckwheat uses the simple distributed free-list scheme described in section 8.4.

8.2.2 Node Representation

The graph structures in Buckwheat are identical to those in Alfalfa with two exceptions:

1. A node pointer is a standard (32 bit) pointer.
2. Each node contains an additional byte that serves as a lock for mutual exclusion.

8.3 Queue-based Scheduling

Processor scheduling is accomplished by maintaining a central queue structure which every processor accesses. The simplest approach would be for every processor to take tasks from and add tasks to the same shared queue. When a processor accesses the queue, a lock is set to prevent any other processor accessing the queue simultaneously.

A shared queue is a source of contention, which increases as the number of

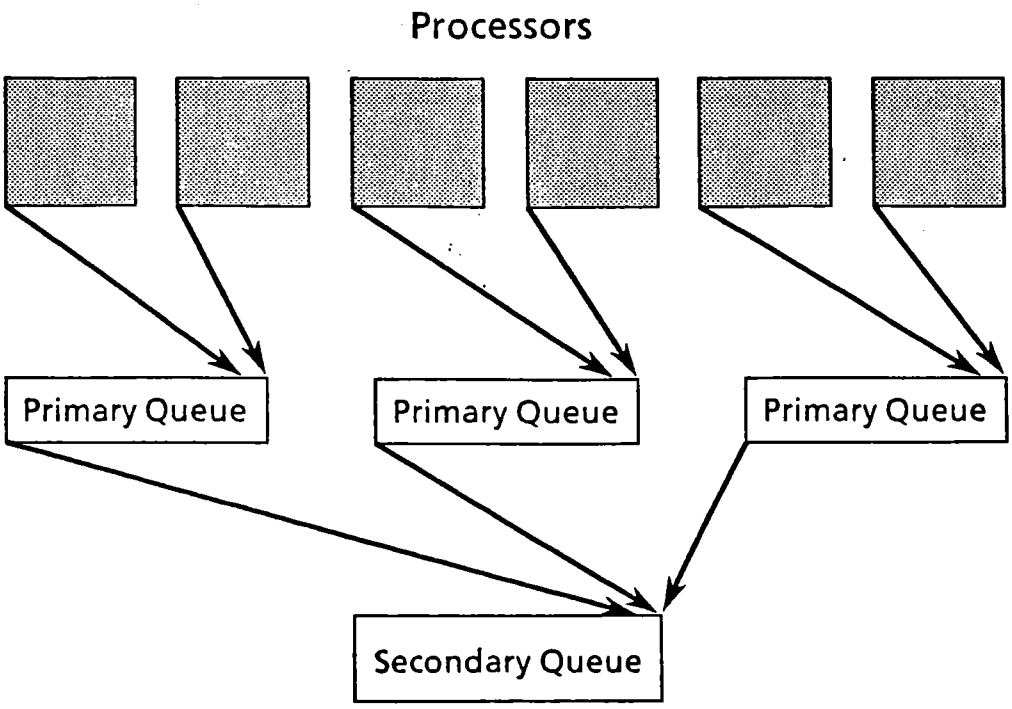


Figure 8.2: Buckwheat's two-level queue structure

processors in the system grows. Unless the hardware supports efficient access to a central queue (as in the Ultracomputer at NYU [23]), it is often necessary to modify the queue structure to prevent contention.

The solution we have implemented for Buckwheat is a *two-level* queue structure illustrated in figure 8.2. A processor can directly access a task queue, called a *primary queue*, that it shares with a small number of other processors. There may be many primary queues in the system. Each primary queue has a rather small fixed size. We define the set of processors accessing a single primary queue to be a *primary cluster*.

If a processor is ready to execute a task and its primary task queue is empty, it can access another queue, called the *secondary queue*, which is shared among all the processors in the system. Similarly, if a processor attempts to put a task onto its primary queue and its primary queue is full, the task is put onto the secondary queue. Since all processors can access the secondary queue, the *secondary cluster* consists of the whole system.

Notice that the two-level queue structure is a special case of a general *hierarchical* queue structure in which there may be many levels of queues. A queue structure with many levels may be appropriate in a system with huge numbers of processors. One would then have tertiary clusters, quaternary clusters, and so on. On the 12-processor Multimax, the two-level queue is quite adequate.

There are several advantages to a two-level queue structure over a single shared queue:

1. Since a primary queue is shared by a relatively small number of processors, the contention for the queue is reduced.

2. The secondary queue is a simple method for spreading tasks from a busy primary cluster to other primary clusters. The cost of the extra indirection needed to access the secondary queue is only incurred by idle processors in idle primary clusters or when a primary cluster becomes very busy. If the size of the primary queue is chosen appropriately, the vast majority of queue accesses will be to primary queues.

Each primary cluster has the same number of processors, if possible. If the total number of processors is not a multiple of the number of clusters, then the clusters are arranged so that they have nearly equal numbers of processors.

A different solution to the problem of contention for a single task queue would be a single level queue structure with many primary queues and no secondary queue. If a processor finds that its primary queue is empty, then it has to find a non-empty primary queue to retrieve a task from. If a processor tries to add a task to a full queue, it then has to find a non-full queue to add the task to. This method has the disadvantage that a processor may have to access many queues before it finds one with an available task (or room for a new task).

Another way to reduce contention for a single task queue is for each processor to remove several tasks at once, thus reducing the frequency of queue access. This is a special case of the two-level queue structure, in which a primary cluster consists of a single processor.

Another strategy (which may prove to be better than ours) would consist of a number of primary queues, but no secondary queue, in which each processor places tasks in a queue other than the one from which it removes tasks. Tasks

would become well distributed without having contention for a single shared queue.[1]

In section 8.5 we compare the performance of a single shared task queue to the performance of a two-level queue structure. Various numbers of processors per primary cluster and various sizes of primary queues were used in the experiments.

8.4 Storage Management for Shared Memory

The storage management module allocates new nodes, tasks, and extends from free-lists. Unfortunately, a single shared free-list would cause contention if many processors tried to allocate new structures simultaneously. The solution used by Buckwheat is for each processor to maintain its own free-list. At the start of the computation, the free graph space is divided among the processors. Each processor can access only its own free-list when it needs a new structure or reclaims an unused structure.

Buckwheat does not use a hierarchical free-list scheme analogous to its hierarchical task queue structure for two reasons:

1. We can expect the creation of nodes, pointers, and extends to be roughly proportional to the number of tasks executed by a processor. If the hierarchical task queue structure does a good job distributing tasks among the processors, the sizes of the free-lists of the processors should remain roughly equal.

2. The amount of parallelism exploited by Buckwheat is determined by the task distribution among the processors and not by the relative sizes of the free-lists. Unless a processor exhausts its free-list, Buckwheat's performance should not be affected by imbalances in the sizes of the free-list.

[1]This idea was recently suggested by Paul Hudak, long after our experiments with Buckwheat were completed.

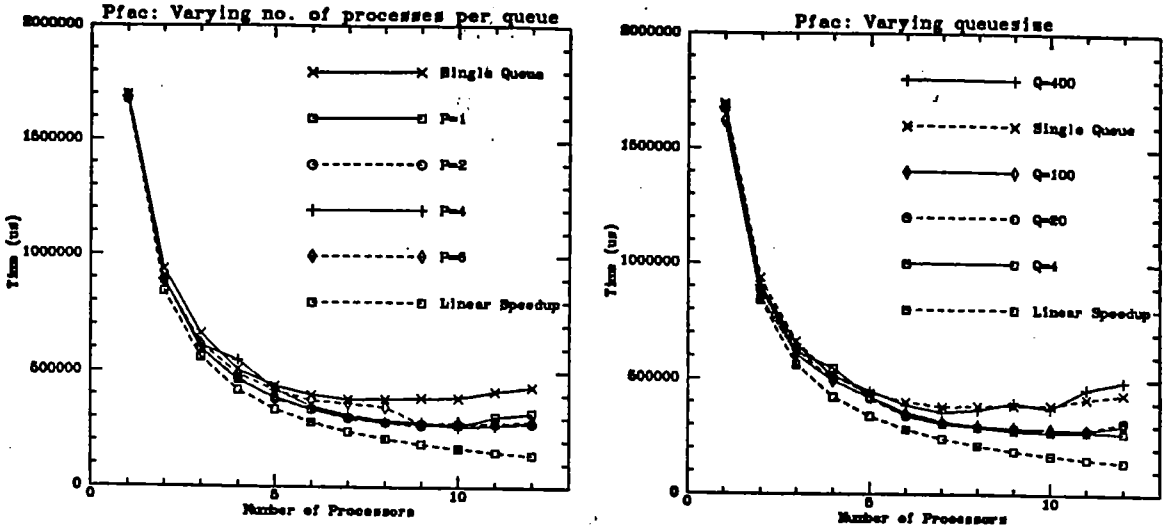


Figure 8.3: The execution times for `pfac` on Buckwheat

8.5 Execution Results

The five applications programs (`pfac`, `queens`, `quad`, `matmult` and `quicksort`) were executed on Buckwheat. Almost 600 runs were performed to measure

1. The performance of Buckwheat using a single shared task queue, and
2. The effect of using a two-level queue structure. The number of processors in a primary cluster as well as the sizes of the primary queues were varied in order to find the best task queue configuration.

Figures 8.3 through 8.7 plot the execution times (in microseconds) for the five programs as a function of the number of processors used.

8.5.1 Finding the Appropriate Cluster Size

In each figure, the graph on the left plots the execution times using a single shared task queue. It also plots the execution times using a two-level queue structure for various values of P, the number of processors in a primary cluster. The size Q of each primary queue was fixed at 10 tasks.

In every program, the two-level queue structure performed better than a

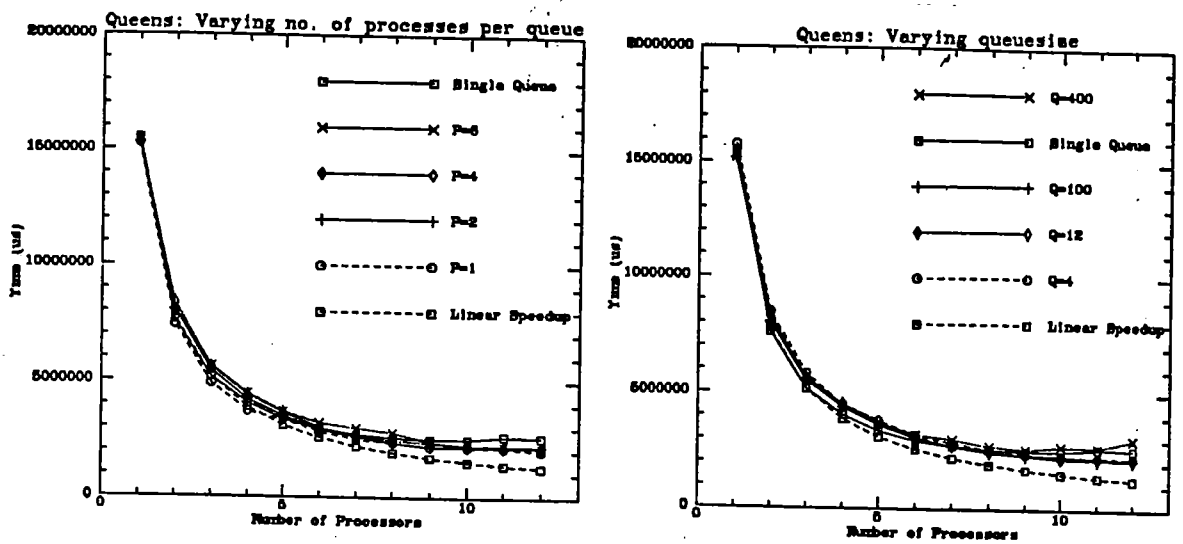


Figure 8.4: The execution times for queens on Buckwheat

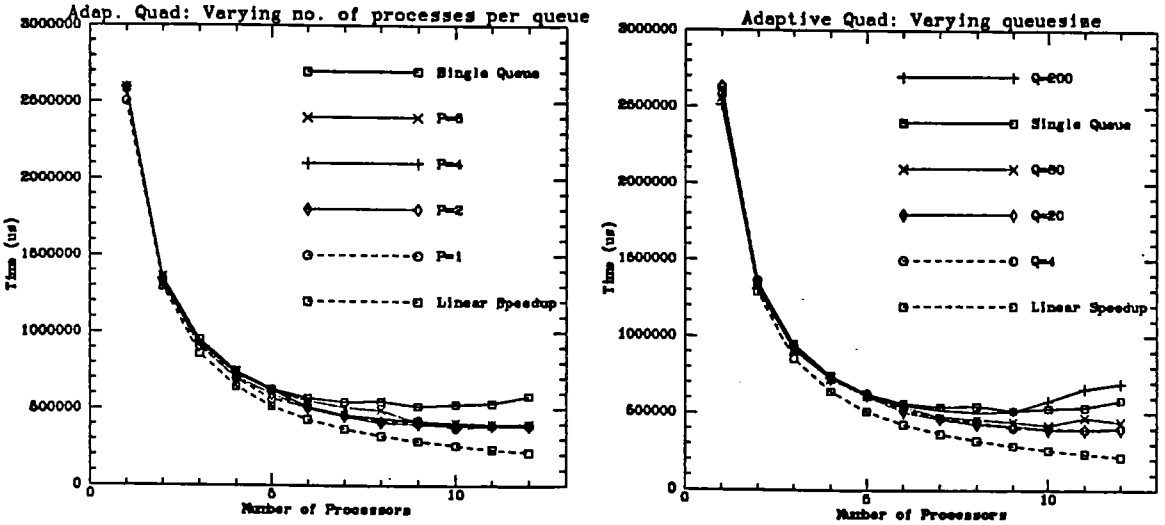


Figure 8.5: The execution times for quad on Buckwheat

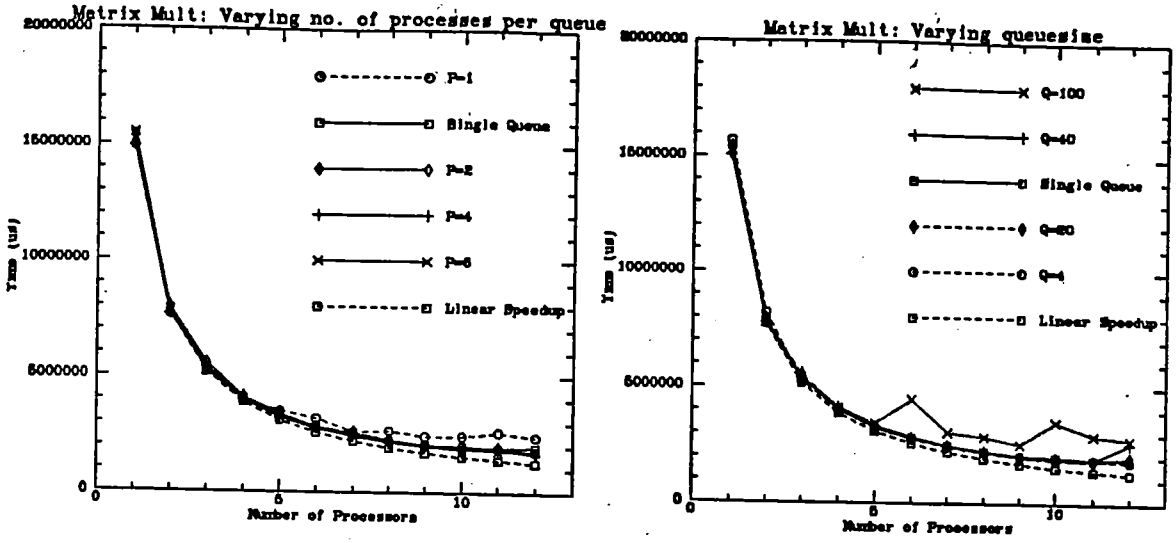


Figure 8.6: The execution times for `matmult` on Buckwheat

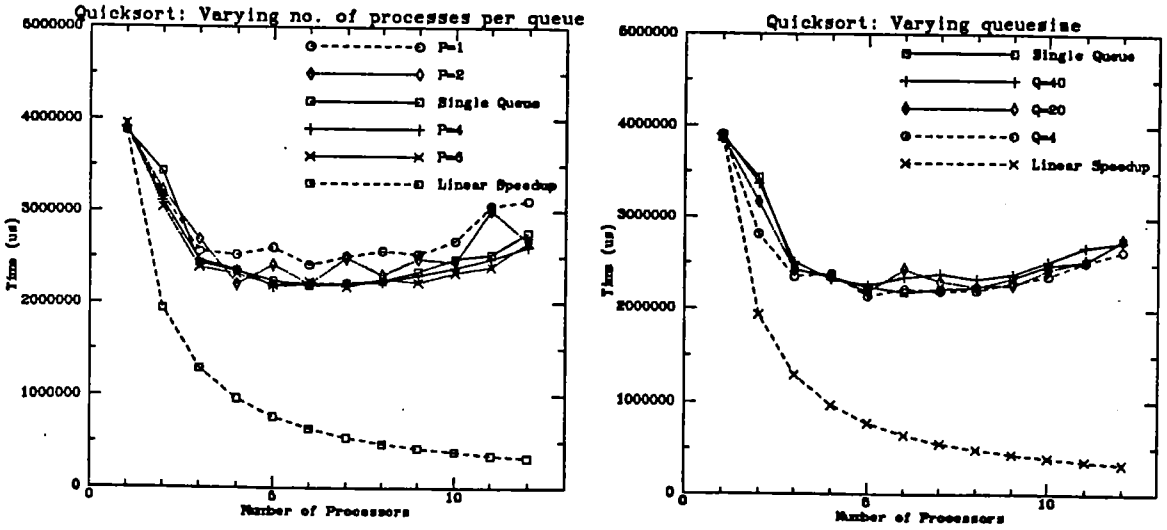


Figure 8.7: The execution times for `quicksort` on Buckwheat

single queue. For small numbers of processors the difference was small, but as the number of processors grew, contention for the single queue actually caused the execution time to *increase*. The two level queue structure significantly reduced the effect of contention for task queues.

With $Q = 10$ the two-level queue structure performed very well over the range of values for P. In `pfac`, `matmult` and `quicksort`, the performance when $P = 1$ was poorer than for other values for P. With only one processor per primary queue, parallelism can only be exploited by having tasks spill over onto the secondary queue. One would expect the task distribution to suffer. In almost every program, however, the performance with $P = 1$ was still superior to the single queue case.

A P value of 4 provided the best performance over all the programs. Surprisingly, this proved to be program-independent (although $P = 2$ performed just about as well). Having found an appropriate number of processors per cluster, it remained to find the best primary queue size for Buckwheat.

8.5.2 Determining Primary Queue Size

In each of figures 8.3 through 8.7, the graph on the right plots the execution times for various values of Q, the size of the primary queues. The number of processors P per cluster was fixed at four. In every case, a large value of Q performed poorly. When Q is large, fewer tasks spill over to the secondary queue and task distribution is poor. The smaller Q values performed much better; the execution times for values of Q under twenty were very similar, although a Q value of 4 (the smallest value of Q we tested) performed the best. Again, the best Q value seemed to be program-independent.

8.5.3 Task Distribution

Table 8.1 lists the task distribution for each program for various value of P. The value of Q is fixed at 10. Notice that the task distributions were nearly insensitive to the number of processors per cluster. In every program except `quicksort`, the work load was well distributed for every value of P. Apparently, the only affect of changing the value of P was to change the amount of

PFAC (Q=10)

PID	Single	P=1	P=2	P=4	P=6
0	330	349	320	374	367
1	299	306	283	382	362
2	279	300	319	316	349
3	358	310	363	324	348
4	341	348	288	299	335
5	361	350	338	355	377
6	338	348	402	316	324
7	297	369	377	325	296
8	338	383	378	330	342
9	340	311	305	348	303
10	369	320	324	320	286
11	361	317	314	322	322

QUEENS (Q=10)

PID	Single	P=1	P=2	P=4	P=6
0	2485	2188	2353	2394	2440
1	2634	2130	2553	2846	2448
2	2231	2440	2398	2323	2408
3	2322	2194	2647	2723	2493
4	2803	2875	2466	2609	2727
5	2637	2803	2743	2192	2472
6	2275	2688	2754	2514	2737
7	2244	2417	2496	2228	2390
8	2706	2754	2321	2318	2329
9	2648	2237	2570	2378	2239
10	2315	2811	2137	2521	2273
11	2395	2158	2257	2649	2739

QUAD (Q=10)

PID	Single	P=1	P=2	P=4	P=6
0	470	426	404	424	401
1	436	441	460	458	498
2	504	417	424	491	434
3	467	485	527	436	495
4	450	522	511	431	500
5	458	526	436	497	434
6	504	489	504	470	430
7	507	444	431	487	433
8	465	508	521	487	532
9	504	402	559	549	417
10	404	434	428	437	517
11	411	514	403	441	517

MATMULT (Q=10)

PID	Single	P=1	P=2	P=4	P=6
0	825	945	865	878	816
1	860	782	792	923	943
2	902	823	808	805	854
3	888	820	831	813	849
4	841	1061	882	867	806
5	831	1016	874	793	936
6	820	983	859	760	846
7	907	621	824	784	806
8	839	819	914	820	807
9	790	595	948	863	797
10	844	809	839	935	945
11	868	941	779	974	810

QUICKSORT (Q=10)

PID	Single	P=1	P=2	P=4	P=6
0	85	136	232	139	101
1	119	154	64	33	1
2	115	73	123	48	1
3	145	180	128	35	1
4	98	2	71	88	1
5	98	140	74	70	2
6	137	173	157	103	295
7	129	170	77	73	170
8	102	128	119	186	274
9	138	52	137	239	306
10	143	104	130	222	199
11	113	110	110	186	71

Table 8.1: Task distribution: Varying number of processors per cluster

contention for the primary queues.

Only `quicksort`'s performance showed some dependence on P. Large P values had poorer task distributions than small P values. Surprisingly, the larger P values resulted in shorter execution times. The reasons for this behavior are not clear.

Table 8.2 lists the task distribution for the five programs for various values of Q. The value of P is fixed at four.

Buckwheat's performance was much more dependent on the value of Q than the value of P. If Q is too large, tasks stay on the large primary queues without spilling over onto the secondary queue. In this case, tasks cannot migrate from busy clusters to idle clusters and the work distribution suffers.

If there are many tasks in the system, a large Q would be acceptable. However, if the inherent parallelism in a program is low, a large Q value hurts performance. This is why `pfac`, `queens`, and `quad` could tolerate a higher value for Q then the other programs. For all the programs, however, $Q = 4$ performed quite well in terms of task distribution and execution time.

8.6 Conclusions

8.6.1 Buckwheat's Performance

With the exception of `quicksort`, all the programs performed well. `Matmult` had the greatest reduction in execution time over the sequential case. This is a marked contrast to its performance on Alfalfa and emphasizes that its poor performance on Alfalfa was caused by the high cost of sending large amounts of data in messages. This is an indication that some programs may be inherently more efficient on one type of multiprocessor architecture than another.

`Quicksort` performed poorly on Buckwheat. Unlike `matmult`, `quicksort`'s performance on Buckwheat was almost as poor as it was on Alfalfa. This is another indication of the limited parallelism available in our experiments with `quicksort`.

The other programs, `pfac`, `queens`, and `quad`, showed almost linear speedup. This reinforces the conclusion reached in the previous chapter: Programs ex-

PFAC (P=4)

PID	Single	Q=4	Q=20	Q=100	Q=400
0	330	369	360	346	802
1	299	304	334	394	26
2	279	340	322	325	23
3	358	330	310	352	32
4	341	300	311	361	31
5	361	302	323	319	40
6	338	348	348	366	36
7	297	366	340	331	34
8	338	356	364	312	762
9	340	338	334	292	688
10	369	321	345	286	740
11	361	337	320	327	797

QUEENS (P=4)

PID	Single	Q=4	Q=100	Q=400	Q=800
0	2485	2325	2255	3358	5840
1	2634	2487	2790	1950	64
2	2231	2926	2394	1846	46
3	2322	2659	2715	1992	47
4	2803	2331	2177	1901	57
5	2637	2314	2213	1649	67
6	2275	2253	2145	1713	66
7	2244	2582	2556	1671	56
8	2706	2555	2729	3304	6125
9	2648	2529	2708	3446	6531
10	2315	2352	2762	3608	5380
11	2395	2382	2251	3257	5416

QUAD (P=4)

PID	Single	Q=4	Q=80	Q=200	Q=400
0	470	465	472	753	1001
1	436	383	384	217	76
2	504	426	459	254	78
3	467	495	479	232	84
4	450	460	373	221	79
5	458	537	449	170	104
6	504	482	414	240	69
7	507	482	492	199	88
8	465	520	511	859	1023
9	504	428	569	796	953
10	404	500	526	785	998
11	411	430	480	882	1055

MATMULT (P=4)

PID	Single	Q=4	Q=20	Q=100	Q=200
0	825	807	995	1465	2201
1	860	828	809	91	1
2	902	885	816	109	1
3	888	800	799	89	1
4	841	846	828	749	1
5	831	855	801	675	1
6	820	894	825	653	1
7	907	985	860	751	1
8	839	828	867	1495	2022
9	790	814	780	1364	2005
10	844	831	917	1314	1931
11	868	842	918	1460	2049

QUICKSORT (P=4)

PID	Single	Q=4	Q=20	Q=40
0	85	128	219	180
1	119	127	33	1
2	115	135	30	1
3	145	142	26	1
4	98	133	26	1
5	98	115	23	1
6	137	138	25	1
7	129	121	15	1
8	102	125	267	340
9	138	96	244	319
10	143	92	245	329
11	113	70	269	247

Table 8.2: Task distribution: Varying the size of primary queues

hibiting a large amount of inherent parallelism without using large centralized data structures can be automatically partitioned into serial combinators so that they perform well over a variety of multiprocessors.

8.6.2 Comparing Buckwheat and Alfalfa

Buckwheat's performance was clearly better than Alfalfa's. In addition to achieving greater speedup, the shared memory allowed a straightforward implementation of a logically shared graph space. Even if the iPSC had a significantly lower communication overhead than it does, the cost of supporting a single graph space would still have had a negative affect on its performance. Buckwheat was much simpler to implement and debug and proved to be far more reliable than Alfalfa.

This discussion does not deal with the aspect of *extensibility* of multiprocessor architectures. It may be the case that in a large shared-memory multiprocessor, the memory contention involved in accessing the shared graph space may cause a severe degradation in the performance of graph reduction. This has been a widely researched topic and there are several current implementations and prototypes of large shared-memory (or partially shared-memory) multiprocessors (such as the NYU Ultracomputer [22], the IBM RP3 [63], the BBN Butterfly [71], and the BBN Monarch [64] under development).

Chapter 9

Related Work, Future Work, and Conclusions

9.1 Related Work

Our work has benefited greatly from a large amount of research on sequential evaluation of functional programs as well as parallel evaluation. The following research projects (related to our work) have played a significant role in the use and implementation of functional programs.

9.1.1 AMPS

The *Applicative Multi-Processing System* [50,49] developed at the University of Utah was one of the first implementation designs for parallel graph reduction. Although no real multiprocessor implementation of AMPS was built, many of the ideas were tested via simulation. The most significant difference between the AMPS design and our implementation is that AMPS used the functions defined in the source program to determine the granularity of the computation. No sophisticated compiler was used to automatically decompose a functional program into tasks of the appropriate granularity. Even so, AMPS can be considered a direct precursor to the implementation described in this dissertation.

A multiprocessing system designed later at Utah was called *Rediflow*. As its name suggests, it attempted to incorporate the graph reduction and data flow

evaluation models. Much of Rediflow centered on a hardware design of an appropriate multiprocessor, rather than the development of compiler techniques. It used a dynamic task scheduling technique similar to the diffusion scheduling method described in chapter 7. No implementation of Rediflow has been built to date.

9.1.2 The ALICE project

ALICE, for *Applicative Language Idealized Computing Engine*, is a multiprocessing system built at Imperial College, London [17]. It is designed specifically to support parallel graph reduction and execute supercombinators (although the compiler described in this dissertation could be adapted to generate code for ALICE). Each processor in ALICE consists of an Inmos Transputer and accesses a shared graph space.

9.1.3 The GRIP project

GRIP [62], for *Graph Reduction in Parallel*, is a bus-based, shared memory multiprocessor for performing graph reduction. A prototype has recently been completed at University College London. Like ALICE, it is intended to execute supercombinators. Its distinguishing feature is its use of intelligent memory units (IMUs). IMUs are microprogrammable and are capable of performing graph operations as well as supporting garbage collection. This allows a greater number of processors to be connected to the bus without causing excessive bus contention. GRIP, like ALICE, may serve as a good target machine for the compiler described here.

9.1.4 Cobweb

Cobweb [24] is a parallel machine designed to perform combinator reduction. The architecture of the machine is tailored for the use of wafer-scale integration. It consists of a large number of processing elements on a wafer. The arrangement of the processors gives the machine its name. For combinator reduction, an abstract machine called Norman was designed to be supported by

Cobweb. Initially, Norman was not a parallel graph reduction model. Parallelism could only be achieved by executing many programs at once. Recently, the abstract machine has been extended to be able to reduce a single fixed combinator program in parallel [46].

9.1.5 The G-machine

The G-machine [44,43,2] is an abstract machine for sequential graph reduction developed at Chalmers University, Sweden. It provides an intermediate representation for the compilation of lazy functional languages [42,3] for conventional machines. A hardware implementation of the G-machine is currently being constructed at the Oregon Graduate center [51]. In either case, the G-machine is designed to execute a variant of supercombinators. Lambda lifting [45] was developed for use in the Chalmers compiler.

The G-machine is a stack-based graph reducer. Its main similarity to the work described in this dissertation is in the use of *programmed graph reduction* in which, as in serial combinators, the body of a function is a set of instructions specifying transformations on the graph. In other graph reduction systems, a function application is automatically expanded into a graphical representation of the body of the function.

9.1.6 The SKIM machines

SKIM I and SKIM II are microcoded processors specifically designed for efficient combinator reduction [11,70]. Unlike the G-machine, the SKIM machines are designed to reduce graphs involving only fixed combinators. Both machines were built at Cambridge University. Perhaps the most interesting aspect of the SKIM II machine is its use of one-bit reference counts in pointers to reduce the overhead associated with reference counting.

9.1.7 NORMA

NORMA (for Normal Order Reduction MAchine) [66] is a hardware implementation for a sequential fixed combinator reduction. It was built at the Burroughs

Austin Research Center. Like the SKIM machines, it was microprogrammable and could conceivably be reprogrammed to execute supercombinators.

9.1.8 Other partitioning methods for functional programs

A program partitioning algorithm [65] has been developed by Sarkar and Hennessy at Stanford for programs written in single-assignment languages (such as SISAL [55] and VAL [56,1]). The execution model is called *macro-dataflow*, and resembles a dataflow model with coarser units of computation. The specification of a single task, called a `macro-actor`, is analogous to a serial combinator in that its granularity is made as large as possible without sacrificing useful parallelism. Macro-actors, however, result from partitioning programs written in an applicative order, first order functional language, and it is a fundamental property of macro-actors that their evaluation never suspends once it starts. While this may reduce the synchronization overhead, it is not clear how to apply the partitioning algorithm to programs written in lazy or higher order functional languages.

Another partitioning method [27] was developed at Carnegie-Mellon for a lazy functional language called Stardust. The primary difference between this work and ours is that the partitioning is performed at *run-time* based on execution time estimates provided by the programmer.

9.2 Conclusions

The question that this dissertation attempted to answer was:

> Is it feasible to execute conventional functional programs on currently available multiprocessors so that a significant reduction in the execution time is achieved?

The results of the experiments described in chapters 7 and 8 have demonstrated that the answer is "Yes... and no" for the following reasons:

- For many programs, automatic partitioning and scheduling can be very effective on current implementations on both loosely-coupled and tightly-couple multiprocessors. These programs have the following properties:

1. They exhibit a significant amount of parallelism. Even if the parallelism seems to be fine-grained, the compilation and evaluation methods described here were still able to cause a significant reduction in their execution times when executed on a multiprocessor.

   The granularity of serial combinators still appeared to be somewhat program dependent. The eight-queens program performed better than the parallel factorial program for this reason. In either case though, the serial combinators had a sufficiently coarse granularity to perform well on both types of multiprocessors.

2. They do not contain large shared data structures. Our compilation process performed automatic decomposition of the expressions in a program, not the data. Thus, on loosely-coupled multiprocessors, programs such as matrix multiplication incurred significant overhead because of the large amounts of data transmitted between processors.

   The parallel factorial, 8-queens, and adaptive quadrature programs all exhibited both of these properties and performed well on both types of multiprocessors.

- For programs exhibiting a large amount of parallelism and containing large shared data structures, automatic partitioning and scheduling performs well for shared-memory multiprocessors, but apparently not for loosely-coupled processors with high communication costs. Either automatic or programmer directed *data partitioning* is required to make use of programs with large shared data structures. Matrix multiplication is an example of a program that worked very well on a shared-memory machine but poorly on a loosely-coupled multiprocessor.

We conclude that the serial combinator approach has proved to be successful for exploiting implicit parallelism in programs without large shared data structures. More work remains to be done on data partitioning.

Our experiments also provided insight into issues of dynamic scheduling. For loosely-coupled multiprocessors, simple dynamic scheduling algorithms proved to be just as effective as more sophisticated ones requiring more communication. On shared-memory multiprocessors, the reduction in memory-contention

provided by a multiple task queue structure resulted in significantly improved performance over the use of a single shared task queue.

Although our goal was to evaluate functional programs efficiently on a variety of conventional multiprocessors, our experiments have led us to believe that a multiprocessor that provides hardware support for a shared graph space is most appropriate. This support could be be in the form of a physically shared memory or a collection of memory modules that are accessible to every processor via a sophisticated network. On loosely-coupled multiprocessors, the difficulty of building (and debugging) a functional language implementation, as well the overhead incurred during execution, makes these machines less desirable.

9.3 Future Work

There are a number of areas worth researching to improve the performance of serial combinator reduction. One area is improving *data partitioning and distribution.* We will be taking two approaches:

1. Increasing the sophistication of the serial combinator compiler: We intend to investigate the use of semantic-based analysis such as abstract interpretation for data partitioning.

2. Including constructs in our functional language that allows the user to specify the decomposition of the data (as well as expressions): This style of programming has been called *para-functional programming* [30,35,31]. We expect to modify the compiler and Alfalfa to accommodate programs that include programmer supplied annotations for program decomposition and distribution.

In addition to working on methods for effective data partitioning, we also will be using semantics-based analyses to improve the low level behavior of distributed graph reduction. Some areas of application are:

1. Program analysis for reducing the number of synchronization operations in serial combinators: In many cases, a serial combinator has to check to see if an argument is already evaluated when a sophisticated compile-time analysis (such as path analysis [6]) could indicate that the check is

unnecessary.

2. Modifying the definition of serial combinators to allow a single task to evaluate several disjoint expressions to increase the grain size. This involves implementing multiple value returns either directly in Alfalfa or using tupling of values. Currently, several disjoint expressions cannot be compiled into a single serial combinator.

3. Using semantics based analyses to improve the algorithm for determining whether an activation record should be allocated on the stack or in the heap. This would constitute a higher-order extension to escape analysis for closures.

Bibliography

[1] W.B. Ackerman and J.B Dennis. *VAL – A value-oriented algorithmic language, preliminary reference manual.* Lab. for Comp. Sci. MIT/LCS/TR-218, MIT, June 1979.

[2] L. Augustsson. A compiler for Lazy ML. In *Proc. 1984 ACM Conf. on LISP and Functional Prog.*, pages 218–227, August 1984.

[3] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1987.

[4] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[5] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[6] A. Bloss and P. Hudak. Path semantics. In *Proc. of the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, April 1987.

[7] A. Bloss and P. Hudak. Variations on strictness analysis. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 132–142, Cambridge, Massachusetts, August 1986.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N.D. Jones, editors, *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1985.

[9] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, N.J., 1941.

[10] C. Clack and S.L. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag LNCS 201, September 1985.

[11] T. Clarke, P. Gladstone, and Norman A. MacLean. SKIM — The S, K, I reduction machine. In *The 1980 LISP Conference*, pages 128–135, Stanford University, August 1980.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, pages 238–252, ACM, 1977.

[13] H.K. Curry and R. Feys. *Combinatory Logic.* Noth-Holland Pub. Co., Amsterdam, 1958.

[14] G. Cybenko. *Dynamic Load Balancing for Distributed Memory Multiprocessors.* Technical Report 87-1, Department of Computer Science, Tufts University, Jan 1987.

[15] L. Damas and R. Milner. Principle type schemes for functional languages. In *9th ACM Sym. on Prin. of Prog. Lang.*, ACM, August 1982.

[16] J. Darlington, P. Henderson, and D.A. Turner, editors. *Functional Programming and its Applications.* Cambridge University Press, Cambridge, England, 1982.

[17] J. Darlington and M. Reeve. Alice: a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Functional Programming Languages and Computer Architecture*, pages 65–76, ACM, October 1981.

[18] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5):662–675, May 1986.

[19] *Mutimax Technical Summary.* Encore Computer Corporation, Marlborough, MA, 1986.

[20] B. Goldberg. Detecting sharing of partial applications in functional programs. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*, pages 408–425, Springer Verlag LNCS 274, September 1987.

[21] B. Goldberg and P. Hudak. Alfalfa: distributed graph reduction on a hypercube multiprocessor. In *Proceedings of the Santa Fe Graph Reduction Workshop*, pages 94–113, Los Alamos/MCC, Springer-Verlag LNCS 279, October 1986.

[22] A. Gottlieb. *An Overview of the NYU Ultracomputer Project.* Ultracomputer Note #100, Courant Inst. of Mathematical Science, New York University, April 1987.

[23] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Trans. on Comp.*, C-32(2):175–189, February 1983.

[24] C.L. Hankin, P.E. Osmon, and M.J. Shute. COBWEB: a combinator reduction architecture. In *Functional Programming Languages and Computer Architecture*, pages 99–112, Springer-Verlag LNCS 201, September 1985.

[25] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[26] C.A.R. Hoare. Quicksort. *Computing J.*, 5(4):10–15, April 1962.

[27] D. Hornig. *Automatic Partitioning and Scheduling on a Network of Personal Computers.* PhD thesis, Carnegie-Mellon Unversity, November 1984.

[28] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[29] P. Hudak. *Object and Task Reclamation in Distributed Applicative Processing Systems.* PhD thesis, University of Utah, July 1982.

[30] P. Hudak. Para-functional languages for parallel and distributed computing. In *Proceedings of Spring COMPCON '87*, pages 334–337, IEEE, February 1987.

[31] P. Hudak. Para-functional programming. *Computer*, 19(8):60–71, August 1986.

[32] P. Hudak and B. Goldberg. Experiments in diffused combinator reduction. In *Proc. 1984 ACM Conf. on LISP and Functional Prog.*, pages 167–176, ACM, August 1984.

[33] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture*, pages 382–388, Springer-Verlag LNCS 201, September 1985.

[34] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In *11th ACM Sym. on Prin. of Prog. Lang.*, pages 121–132, ACM, January 1984.

[35] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proc. 12th Sym. on Prin. of Prog. Lang.*, pages 243–254, ACM, January 1986.

[36] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 97–109, January 1986.

[37] R.J.M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Oxford University, July 1983.

[38] R.J.M. Hughes. Personal communication. January 1988.

[39] R.J.M. Hughes. Strictness detection in non-flat domains. In H. Ganzinger and N.D. Jones, editors, *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1986.

[40] R.J.M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. 1982 ACM Conf. on LISP and Functional*

Prog., pages 1–10, ACM, August 1982.

[41] *iPSC User's Guide — Preliminary.* Intel Corporation, July 1985.

[42] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1987.

[43] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction*, pages 58–69, June 1984.

[44] T. Johnsson. *The G-machine: an abstract machine for graph reduction.* Technical Report, PMG, Dept. of Computer Science, Chalmers Univ. of Tech., February 1985.

[45] T. Johnsson. Lambda Lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203, Springer-Verlag LNCS 201, September 1985.

[46] R.J. Karia. *An investigation of Combinator Reduction on Mutiprocessor Architectures.* PhD thesis, Department of Computer Science, Westfield College, University of London, January 1987.

[47] R.M. Keller. *FEL Programmer's Guide.* AMPS TR 7, University of Utah, March 1982.

[48] R.M. Keller, F.C.H. Lin, and J. Tanaka. Rediflow multiprocessing. In *Proc. Compcon Spring 84*, pages 410–417, February 1984.

[49] R.M. Keller, G. Lindstrom, and S. Patil. *An Architecture for a Loosely-Coupled Parallel Processor.* Technical Report UUCS-78-105, Univ. of Utah, October 1978.

[50] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *AFIPS*, pages 613–622, AFIPS, June 1979.

[51] R.B. Kieburtz. The G-machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture*, pages 400–413, Springer-Verlag LNCS 201, September 1985.

[52] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, Department of Computer Science, May 1988.

search Parallel Processor Prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, IEEE, August 1985.

[64] R.D. Rettberg. Parallel processing at Bolt Beranek and Newman. Lecture Notes. Presented at NYU, November 1987.

[65] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 202–211, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986.

[66] Mark Scheevel. NORMA: A graph reduction processor. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 212–219, Cambridge, Massachusetts, August 1986.

[67] M. Schonfinkel. Uber die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305, 1924.

[68] D. Scott and C. Strachey. *Towards a Mathematical Semantics for Computer Languages.* Tech. Monograph PRG-6, Programming Research Group, University of Oxford, 1971.

[69] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[70] W.R. Stoye, J.W. Clarke, and A.C. Norman. Some practical methods for rapid combinator reduction. In *Proceeding of the ACM Conference on LISP and Functional Programming*, pages 159–166, ACM, August 1984.

[71] R. Thomas, W. Crowther, and R. Gurwitz. *Benchmark results for a 256-node Butterfly parallel processor.* Technical Report Rep. 6355, BBN Laboratories, Cambridge, Mass., August 1986.

[72] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16, Springer-Verlag LNCS 201, September 1985.

[73] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.

[74] D.A. Turner. *SASL language manual.* Technical Report, University of St. Andrews, 1976.

[75] D.A. Turner. The semantic elegance of applicative languages. In *Functional Programming Languages and Computer Architecture*, pages 85–92, ACM, 1981.

[76] P.L. Wadler. Strictness analysis on non-flat domains (byy abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[77] C.P. Wadsworth. The relation between computational and denotational properties for Scott's models of the lambda-calculus. *SIAM J. Comp.*, 5(3):488–521, September 1976.

[78] C.P. Wadsworth. *Semantics and Pragmantics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.