Parallel Combinator Reduction: Some Performance Bounds

Mike Joy

Department of Computer Science,
University of Warwick,
COVENTRY,
CV4 7AL,
UK

e-mail: msj@dcs.warwick.ac.uk

Tom Axford

School of Computer Science,
University of Birmingham,
PO Box 363,
BIRMINGHAM,
B15 2TT,
UK

e-mail tha@cs.bham.ac.uk

ABSTRACT

A parallel graph reduction machine simulator is described. This performs combinator reduction and can simulate various different parallel reduction strategies. A number of functional programs are examined, and experimental results presented comparing the amount of parallelism obtainable using explicit divide-and-conquer with the maximum amount of parallelism available in the programs.

Keywords: functional programming, shared-memory parallel processing, combinators, graph reduction

1. Introduction

In recent years functional languages [8] have gained popularity, since translating a formal specification of a problem to an executable functional program is (relatively) straightforward. Imperative language constructs model closely the hardware structures of conventional (von Neumann) architectures. Functional languages, however, were not designed with any architecture in mind, and as a result, implementations of functional languages on present-day machines tend to be relatively slow. The question of how to speed up functional language implementations is therefore one which must be addressed in order for their benefits fully to be realised.

One way of running a program faster is to execute it on several processors simultaneously. Functional languages contain no notion of "state", and many synchronization problems present with parallel imperative programs are absent from parallel declarative programs. Thus simultaneous execution of sections of a functional program on different processors should present relatively few problems.

There are several different approaches [16] to the problems of parallelising the evaluation of a functional program. Most of these fall into one of two categories. Firstly, the high-level program may be annotated explicitly to instruct how the parallelism should be achieved. Secondly, the source program can be analysed automatically using techniques such as strictness analysis in order to determine which sections of the program may be executed concurrently. We have written a simulator for a shared-memory multiprocessor which can model various strategies for parallelising a functional program using combinator graph reduction.

Our interest is in the implementation of functional languages using combinator graph reduction, and in particular in the extent to which parallel evaluation can be exploited. To this end we have been investigating the generic properties of a fixed translation scheme. The transformation from a functional program to a combinator graph is a complex procedure which can be accomplished using various algorithms, and it was therefore important to deal with a default translation strategy.

We introduce the concept of *saturation parallelism*. Given a translation of a functional program to a combinator graph, there are sections of the graph which can be evaluated simultaneously; if a parallel machine evaluates *all* such sections in parallel, then we describe that evaluation strategy as saturation parallelism. For an idealized machine with unlimited processors and zero communication costs, this strategy will provide a measure of the minimum time necessary to evaluate that program, and thus provide a base-line against which the performance of other parallel evaluation strategies can be measured.

This paper reports the results of experiments with different strategies for parallel evaluation. We have run some benchmark programs using (a) explicitly introduced parallelism and (b) saturation parallelism. The examples we have chosen are such that explicit parallelism can be introduced in a natural fashion.

2. Combinator Graph Reduction

The particular implementation method for functional languages which interests us is *combinator graph reduction*. We can think of a combinator as a simple function. Using combinators to evaluate a functional program involves first of all taking that program, thought of as a sugared lambda-expression, and "abstracting" the variables. All lambda expressions are thus removed and combinators introduced. So a program containing names and value-bindings is translated to an equivalent expression containing no names, and the resulting expression contains only combinators, together with basic data. An implementation which did not abstract names would need to keep track of the partially-evaluated expressions associated with each name, a significant communication overhead.

This expression can then be considered as a graph, where the application of a function to an argument is represented as a binary node, and an atomic datum or a basic combinator is a leaf node. The graph may not necessarily be a tree, as sharing of common subgraphs may take place, and in the case of recursive functions, that sharing may cause cycles to be introduced into the graph.

The program can be evaluated by *reducing* the graph, that is, repeatedly rewriting sections of the graph until the graph becomes a datum which is the result of the program. For instance, if we have a unary operator applied to an argument, then the binary node representing that application will be overwritten by the result of the application.

2.1. Sets of Combinators

Some preliminary observations on our choice of combinators are appropriate at this point. We work with combinators as presented originally by Curry [3] , although this formalism is not the only way of describing combinatory logic (*categorical combinators*, for instance, create a powerful theory which has also been used as a basis for graph reduction machines). For the purposes of this paper we identify two principal classes of combinators. Firstly, combinators which form a (small) fixed set, such as Turner's combinators [20,21] , and secondly combinators which form an extensible set and which may be created when required, such as Supercombinators [11] .

The former class has the advantage of being simple to use and to reason about, but yields combinator expressions which are relatively large and complex [10] . The latter class requires sophisticated abstraction techniques, and typically combinators will represent compiled functions which can be run on conventional architectures fairly efficiently. In [5] Hartel presents statistics indicating that, using various metrics, the speed of evaluation (measured by combinator reduction steps) of this latter class of combinators compared with a fixed set is better by typically less than an order of magnitude.

A benefit of having a fixed combinator set is that an architecture can (in theory, at least) be devised so that the combinators are implemented in hardware. Novel coding techniques [13,15] can also be employed to reduce the space wastage inherent in a naïve implementation of such a fixed combinator set. Most current work relating to parallel combinator reduction however assumes an extensible combinator set.

We have examined combinator reduction involving various fixed combinator sets. In this paper we consider one particular choice of combinators, the set originally introduced by Turner [20,21] . They are defined in Figure 1 (α, β, γ, δ represent arbitrary expressions), where the reduction rule for each combinator is given.

Combinator	*Reduction Rule*		
S	S α β γ	$>_{\mathrm{S}}$	α γ (β γ)
K	K α β	$>_{\mathrm{K}}$	α
I	I α	$>_{\mathrm{I}}$	α
B	B α β γ	$>_{\mathrm{B}}$	α (β γ)
C	C α β γ	$>_{\mathrm{C}}$	α γ β
S′	S′ α β γ δ	$>_{\mathrm{S'}}$	α (β δ) (γ δ)
B′	B′ α β γ δ	$>_{\mathrm{B'}}$	α β (γ δ)
C′	C′ α β γ δ	$>_{\mathrm{C'}}$	α (β δ) γ
Y	Y α	$>_{\mathrm{Y}}$	α (Y α)

Figure 1: The set of combinators used

These yield a number of graph rewrite rules, given in Figure 2:

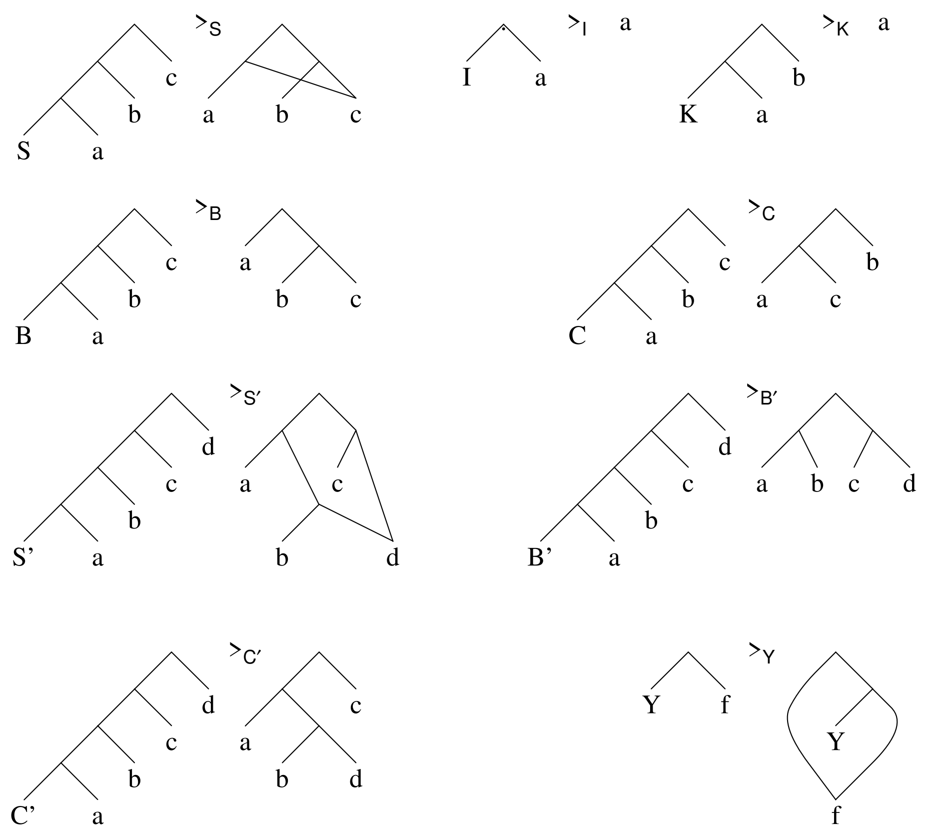

Figure 2: The Graph Rewrite Rules

2.2. Parallelism

A number of projects have recently examined parallel graph reduction for functional languages. Some of these use an extensible set of combinators, such as GRIP, MaRS, Alfalfa and the Evaluation Transformer model. Several are not based on combinators, such as Alice, Flagship, the <ν,G> Machine, the ABC-Machine and Rediflow.

Given a large combinator graph, it is likely that many nodes in the graph can be rewritten concurrently. Due to the small granularity of the combinators, combinator graph reduction is a candidate for massive parallelism [6] . New generations of computers, such as custom-made VLSI architectures and Neural Machines place us within sight of an efficient parallel graph reduction machine.

Unfortunately, it is not always easy (or necessarily even possible) to decide which graph nodes *need* to be rewritten, and which do not (such as the unused arm of a conditional). The technique of lazy evaluation on a single processor will minimise the amount of work necessary to reduce a graph, but does not lend itself to identifying opportunities for parallelism on a multiprocessor.

Various strategies for parallel graph reduction have been examined, such as load balancing amongst the available processors [19] and strictness analysis [7] . Given a specific fixed combinator set, we examined the time efficiency of executing a program using two different strategies:

(i) saturation parallelism (rewriting sections of graph in parallel *whenever* it is possible to do so), and

(ii) by introducing parallelism explicitly (i.e. only in places chosen by the programmer).

Saturation parallelism is in general not a practical technique because it involves an unbounded number of processors. It is, however, useful since it gives us an indication as to the maximum amount of parallelism available in a program, against which we can measure how close to optimal speedup "real" strategies come. Explicit parallelism, on the other hand, is a powerful technique if used correctly. Its main disadvantage is that the onus is placed upon the programmer to code a program and identify the parallelism in it correctly.

We have a upper bound on the speed of a parallel evaluation of a combinator graph, namely that with saturation parallelism. We have also introduced explicit parallelism by using a simple divide-and-conquer approach (see section 6 below). Other more sophisticated attempts at achieving parallelism would be expected to yield a performance between the upper bound of saturation parallelism and that given by our explicit parallelism.

3. The Graph Reduction Machine

The simulator takes as input GCODE [12] , which is a textual description of a graph representing a functional program. GCODE contains lambdas, combinators, basic operators (delta combinators), integers, real numbers, and sum-product domains. The GCODE is generated from FLIC [17] which is minimally sugared lambda-calculus. The input is then compiled to abstract variables and introduce combinators [20,21] . Recursion is handled by the "Y" combinator. The resulting graph is acyclic, and the reduction rule for the "Y" combinator preserves this property.

Reduction of the graph takes place by rewriting several parts of it concurrently. The usual serial lazy evaluation strategy is embedded within this parallel reduction strategy, in the sense that at any stage in the reduction process the next reduction step in the lazy evaluation strategy will always be one of the several concurrent reduction steps in our parallel strategies.

3.1. Architecture

The simulated machine has a fixed number of *processors*, on which may run a variable number of *processes*. Each process rewrites a section of graph until one of the following conditions is reached:

- that section becomes a datum (integer, floating point number, or sum-product);
- that section cannot evaluate further (when an operator is presented with too few arguments);
- that process is destroyed by another process.

A specific process will not recursively evaluate elements of a sum-product domain, but will spawn further processes if required. A process contains (inter alia) two *stacks*, a *status* value and a *priority* value.

Each processor is either allocated a single process and is *Busy*, or is awaiting a process and is *Free*. A process may have status *Running* if in the process of rewriting a section of graph, *Suspended* if awaiting the result of another process before evaluation may proceed, or *Waiting* if it is able to continue running but has not been allocated to a processor.

A process is allocated a "priority", which is either *Possible* or *Needed*. A *Needed* process is one which is known to be required to be evaluated by the lazy evaluation strategy. A *Possible* process is one which *may* require evaluation.

Three queues of processes are maintained; one contains the *Running* processes, one the *Waiting* processes, and one the *Suspended* processes. The elements of each queue are ordered by process priority.

A "master process" controls the allocation of processes to processors, and will perform the administration of the processes and their allocation to processors, as follows:

- Each queue has its elements ordered by *Priority*, with *Needed* processes at the head of the queue.
- When a process in the *Running* queue completes execution, that process ceases to exist and the processor allocated to it becomes *Free*.
- If a process becomes *Suspended*, it is placed in the *Suspended* queue and the processor allocated to it becomes *Free*.
- If data becomes available to a *Suspended* process causing it to be able to continue evaluating, it gains status *Waiting* and is moved to the *Waiting* queue.

- A processor which is *Free* is allocated to the process at the head of the *Waiting* queue. If the *Waiting* queue is empty that processor simply remains *Free*.
- If all processors are *Busy*, and one of the *Running* processes has priority *Possible*, and one of the *Waiting* processes has priority *Needed* (which we may assume to be at the head of the *Waiting* queue), then that *Possible* process is swapped with the head of the *Waiting* queue.

The machine runs in time measured by "cycles". In one cycle a process will perform one reduction step, and the master process will handle any tasks which this may generate. At the start of each cycle the master process ensures that each processor which can be allocated a process is allocated one, and that each *Running* process is known to be able to perform a reduction step. Each processor then causes the process allocated to it to perform one single combinator reduction.

At the start of a program execution, after the combinator graph has been created, a single process is created with priority *Needed*, which will begin to rewrite the graph using the standard lazy algorithm. Where necessary it will create other processes with priority *Needed*, or indicate to the master process that other processes with priority *Possible* should have priority updated to *Needed*. Further processes may be created as dictated by the parallel evaluation strategy.

3.2. Lazy Evaluation

It will be clear from the above description that the number of *Needed* graph reduction steps will be precisely the number of steps necessary to evaluate a program using lazy evaluation on a single processor. Thus the number of cycles taken for the multiprocessor simulator to evaluate a program will be no more than the number of steps to evaluate a program using lazy evaluation. Our system of Queues guards against the danger of speculatively offshipped processes being evaluated instead of required processes and thereby causing performance degradation.

3.3. Shared Memory

For the purposes of this exercise we have deliberately left out of our calculations overheads associated with communication between processors. Since we wished to examine the effect of the idealised situation where unrestricted parallelism was available the effect of introducing extra metrics into the calculations would have been a distraction. The assumption of a shared-memory multiprocessor was felt to be the simplest way to simulate such an idealised machine. Indeed, the machine resources we had available to use were only just sufficient to perform the experiments reported here, and simulation of a distributed processor would undoubtedly have been infeasible.

3.4. Overheads

Much of the workload relating to moving processes between processors is left in the control of the master process. In a worst-case example, this process could do constant amount of work for *each* processor *each cycle*. By implementing the master process so that its functions are distributed about the processors, this work would then involve constant extra time per cycle. The process has been implemented in the way described for reasons of improving the clarity of the software.

4. The Parallel Evaluation Strategies

We consider two parallel evaluation strategies. These are described in terms of the creation of new processes and their allocation to specific processors; subsequent management of processes is as described above.

4.1. Strategy 1: Saturation Parallelism

The maximum set of processes is constructed. The set of available processes is unbounded, and processes are dynamically constructed according to the following rules. Initially, each reducible sub-expression of the original graph is assigned a process. During evaluation of the program, whenever a new application node is created, it is checked to see whether it represents a reducible expression, in which case a process is assigned to it. These application nodes are formed after combinators `S`, `B`, `S′`, `B′` and `C′` [20,21] have been reduced.

For instance, consider the following combinator expression (α, β, γ, δ and ε represent arbitrary expressions):

$$\mathtt{B}\ (\mathtt{B}\ \alpha\ \beta)\ (\mathtt{C}\ \gamma\ \delta)\ \varepsilon$$

After one reduction step, fired by the leftmost B, we get

$$\mathtt{B}\ \alpha\ \beta\ (\mathtt{C}\ \gamma\ \delta\ \varepsilon)$$

In this expression both the B and C combinators have sufficient arguments. The B-reduction will be performed by the same process as the previous reduction (and on the same processor), and a new process (with priority *Possible*) will be created to handle the C-reduction. This latter process will then be placed in the *Waiting* queue, and consequently reduced by another processor when one becomes *Free*. If the parent process in due course attempts to perform the reduction it has offshipped, the parent process will be *Suspended* until the child has performed that reduction.

This results in a large set of processes, most of which will have priority *Possible*, and many of which will be short-lived. The overheads of managing large numbers of processes are high, since many of them will return results which are not required; this strategy is intended as a benchmark against which we can measure the extent to which other strategies exploit available parallelism. We obtain a figure which is the minimum time (measured in cycles) for the graph to evaluate fully.

4.2. Strategy 2: Explicit Parallelism

Initially, only one process is created, as described in the previous section. Subsequently processes are flagged so that they will be queued to run on their parent processor *only*. Conceptually, each processor has its own private triple of *Running*, *Waiting* and *Suspended* queues.

Whenever a section of code has to be sent to a different processor (when the combinator PAR [18] is encountered) a process is created and offshipped to another processor with empty *Running* and *Waiting* queues. PAR is defined by the rule given in Figure 4.

Combinator	*Reduction Rule*
PAR	PAR α β $>_{\mathrm{PAR}}$ β

Figure 3: The combinator PAR

Expression α is offshipped (if possible) to another processor; PAR is typically used when α is a subexpression of β. If such a processor does not exist the process is placed in a *Waiting* queue for the first such processor to become available. If such a processor never becomes available, the spawned process will eventually be performed by its parent processor.

All processes are executed on a single processor, unless annotated to be offshipped to another process. A process assigned to a specific processor will only be evaluated on that processor, as will all processes spawned by that process unless explicitly offshipped. In high-level code segments below this is indicated by the annotation offship.

For example, consider the following expression being evaluated by process 1:

$$\mathtt{S}\ \mathtt{PAR}\ (+\ \beta)\ \alpha$$

After one reduction step by process 1 this yields

$$\mathtt{PAR}\ \alpha\ (+\ \beta\ \alpha)$$

After one more by the same process we get

$$+\ \beta\ \alpha$$

where + and β will be evaluated by process 1 and α by process 2 (say). Process 2 will be queued to run on a different processor to that running process 1. If β is fully reduced before α has been fully reduced, then a *Needed* process will be created to reduce α (on the same processor as process 1) even though another process (of priority *Possible*) still exists to evaluate α.

PAR is similar to the annotation {P} in [14] but is appropriate to a shared-memory implementation.

5. Worked Example

In this section we examine one functional program in detail, and isolate opportunities for parallelism within it. The program we have chosen is *Fibonacci numbers*, given by

```
fib n = if n >= 1 then (fib (n-1) + fib (n-2)) else 1
```

A lambda-expression representing this Fibonacci function is:

```
Y λfib λn IF 1 (+ (fib (− n 1)) (fib (− n 2))) (>= n 1)
```

Note the operator IF † is defined by

IF *false-case true-case condition*

for consistency with the data structures in FLIC, and that all operators are *Curried*, that is, in prefix form. When abstraction has taken place, we arrive at the combinator expression:

```
Y (C′ (S′ (IF 1)) (S′ (S′ +) (C B (C − 1)) (C B (C − 2))) (C >= 1))
```

5.1. Saturation Parallelism

Using the name FIB to denote this expression, and applying it to argument 10, we have in Figure 3 the following reduction sequences. Each line denotes the expression after *one* reduction step, and the symbols in the first and second columns respectively indicate the machine "cycle" and the identity of the processor which performs the reduction.

† Actually, CASE 2

Cycle	Proc.	Expression
1	1	`Y (C′ (S′ (IF 1)) (S′ (S′ +) (C B (C − 1)) (C B (C − 2))) (C >= 1)) 10`
2	1	`C′ (S′ (IF 1)) (S′ (S′ +) (C B (C − 1)) (C B (C − 2))) (C >= 1) FIB 10`
3	1	`S′ (IF 1) (S′ (S′ +) (C B (C − 1)) (C B (C − 2)) FIB) (C >= 1) 10`
4	1	`IF 1 (S′ (S′ +) (C B (C − 1)) (C B (C − 2)) FIB 10) (C >= 1 10)`
4	2	`IF 1 (S′ (S′ +) (C B (C − 1)) (C B (C − 2)) FIB 10) (>= 10 1)`
5	1	`IF 1 (S′ + (C B (C − 1) FIB) (C B (C − 2) FIB) 10) (>= 10 1)`
5	3	`IF 1 (S′ + (C B (C − 1) FIB) (C B (C − 2) FIB) 10) FALSE`
5	4	`IF 1 (S′ + (B FIB (C − 1)) (C B (C − 2) FIB) 10) FALSE`
5	1	`IF 1 (S′ + (B FIB (C − 1)) (B FIB (C − 2)) 10) FALSE`
6	2	`S′ + (B FIB (C − 1)) (B FIB (C − 2)) 10`
6	3	`+ (B FIB (C − 1) 10) (B FIB (C − 2) 10)`
6	4	`+ (FIB (C − 1 10)) (B FIB (C − 2) 10)`
7	5	`+ (FIB (C − 1 10)) (FIB (C − 2 10))`
7	6	`+ (FIB (− 10 1)) (FIB (C − 2 10))`
8	5	`+ (FIB (− 10 1)) (FIB (− 10 2))`
8	6	`+ (FIB 9) (FIB (− 10 2))`
9	1	`+ (FIB 9) (FIB 8)`

Figure 4: Fibonacci

Now, the first three cycles can be performed before `FIB` has been given argument 10. Recursive calls can therefore take place in parallel, reducing the number of cycles from 9 to 6. The behaviour of this function when allowed unlimited processors is indeed $6n+O(1)$ cycles to evaluate `fib n`.

5.2. Explicit Parallelism

In order to introduce parallelism *explicitly* into the Fibonacci function, we use the `PAR` combinator, as defined above. The lambda expression we then have is:

```
Y λfib λn IF 1 ((λx PAR x (+ (fib (− n 1)) x)) (fib (−n 2))) (>= n 1)
```

This lambda expression is slightly larger than the previous one, and more combinators will be introduced by the abstraction algorithm. By a similar analysis to that in Figure 3, 17 cycles are required for each level of recursion, assuming that sufficient processors are available.

5.3. Analysis

Much of the saturation parallelism is redundant. For instance, the two arms of the IF will always be spawned, and when `fib` is given small arguments, there is significant redundant computation here. Also, some of the spawned processes are very short-lived (for instance, those numbered 2 and 6 in the table above). It can be argued that the communication costs for such processes make their creation and destruction undesirable. We conclude that explicit parallelism gives an acceptably efficient allocation of tasks to processors *in this particular case*, given the set of combinators we have elected to use.

6. Test Programs

In this section we present the results of running several programs under both strategies. The examples have been chosen so that each has significant divide-and-conquer parallelism inherent within it.

6.1. Divide-and-Conquer

The technique of *divide-and-conquer* is a well-known method of explicitly introducing parallelism [1] to certain classes of problem. In [2] a model of list processing is presented suitable for divide-and-conquer techniques, implementing a list as a binary tree. In the examples below which use lists this model is employed.

6.2. Lists

The primitive functions for the list processing model are:

(i) **[]** is the empty list.

(ii) **singleton x** (or alternatively **[x]**) is the list which contains a single element **x**.

(iii) **concatenate s t** (or alternatively **s++t**) is the list formed by concatenating the lists **s** and **t**.

(iv) **split s** is a pair of lists obtained by partitioning the list **s** into two parts. It is defined only if **s** contains at least two elements. Both lists returned are non-empty. **split s** is defined by **split (concatenate s t) = (s, t)**

(v) **length s** (or alternatively **#s**) is the number of elements of **s**.

(vi) **element s** is the only element present in the singleton list **s**. This function is undefined for lists containing more or less than one element.

Note that we model lists with ++ as a primitive list constructor, rather than `cons`.

The examples described in the following section all have natural opportunities to introduce parallelism. Those functions which employ lists in their definitions are assumed to use the divide-and-conquer list paradigm described above, where the trees implementing those lists are balanced.

6.3. Test Examples

Each of the examples we used is presented here in pseudocode. The actual programs were hand-coded in FLIC. The first three have been specified as functions which take an integer argument and would be expected to exhibit time of execution linear in their argument. The fourth, quicksort, should have time quadratic in its argument. The fifth, adaptive quadrature integration, similar to an example in [4] , takes no arguments.

In the following examples, the annotation offship has been used instead of PAR to make the code more easily intelligible, where

f x (offship y)	is equivalent to	PAR y (f x y)
	and	
x ++ offship y	is equivalent to	PAR y (x ++ y)

```
Map n = map double [1..2^n]
          where
          double x = x + x
          map f [] = []
          map f [x] = [f x]
          map f (l ++ r) = map f l ++ offship (map f r)
```

Figure 5: Map

```
Reverse n = reverse [1..2^n]
               where
               reverse [] = []
               reverse [a] = [a]
               reverse (a ++ b) = reverse b ++ offship (reverse a)
```

Figure 6: Reverse

```
Hanoi n = hanoi n 0 1 2
          where
          hanoi 0 a b c = []
          hanoi n a b c = hanoi (n-1) a c b ++ [(a, b)] ++ offship (hanoi (n-1) c b a)
```

Figure 7: Towers of Hanoi

```
Quicksort n
  = quicksort [2^n..1]
        where
        quicksort [] = []
        quicksort [n] = [n]
        quicksort s
          = quicksort a1 ++ offship (quicksort a2)
                where
                (a1, a2) = reduce f (map (g  (median (first s) (last s))) s)
                g medn x = if (x > medn) then ([x], []) else ([], [x])
                f (s1, s2) (t1, t2) = (s1++t1, s2++t2)
                first [n] = n
                first (s++t) = first s
                last [n] = n
                last (s++t) = last t
                map f [] = []
                map f [x] = [f x]
                map f (l ++ r) = map f l ++ offship (map f r)
                median m n =  (m + n) div 2
                reduce p [n] = n
                reduce p (s ++ t) = p (reduce p s) (offship (reduce p t))
```

Figure 8: Quicksort

```
func x = 4.0*x^3 + 3.0*x^2 + 2.0*x + 5.0

area f left right
        = (right-left) * ((f left) + (f right))/ 2.0

check f left right estimate tolerance newleft newright =
        if (estimate – newestimate)^2 < tolerance^2
        then newestimate
        else (trapezoid f left mid newleft (tolerance / 2.0)) +
                (offship (trapezoid f mid r newright (tolerance / 2.0)))
                where
                newestimate = newleft + newright
                mid = (left+right) / 2.0

trapezoid f left right estimate tolerance
        = check f left right estimate tolerance arealeft arearight
                where
                mid = (left+right) / 2.0
                arealeft = area f left mid
                arearight = area f mid right

integrate f left right tolerance
        = trapezoid f left right (area f left right) tolerance

Integration = integrate function 0.0 2.0 0.01
```

Figure 9: Adaptive Quadrature Integration

7. Results

We now give some statistics relating to the above examples, and discuss their importance.

7.1. Maximum Speedup

We look first at the relation between the maximum possible speedup (as measured by saturation parallelism), and that produced by introducing explicit parallelism. For the purposes of this section, we assume an unbounded set of processors.

Figure 10 contains statistics relating to the execution time (measured in cycles) of the programs detailed above. Column 2 gives the actual time for the program to run (*n*, where appropriate, is the argument to the function). Columns 3 and 4 detail the value of *t* (from column 2) for the two strategies. The final column gives the ratio of columns 3 and 4, and represents the fraction of the available parallelism utilised by the "explicit" strategy.

Example	Speed	Saturation	Explicit	Efficiency
Map	$tn + O(1)$	8	226	0.035
Reverse	$tn + O(1)$	8	133	0.060
Hanoi	$tn + O(1)$	11	126	0.087
Quicksort	$tn^2 + O(n)$	1	23.5	0.043
Integration	t	179	3224	0.056

Figure 10: constant of linearity

A combinator expression derived from a lambda expression will, by definition, contain a number of combinators together with other operators mentioned in the lambda expression. In order to evaluate the expression, some or all of the combinators and operators will, at some time during the evaluation, be presented with their required number of arguments. As an expression evaluates, sub-expressions will be created which can therefore be reduced concurrently (for instance, after an `S` combinator has been fired).

The principal reason for the high speed of saturation parallelism is the detection of *all* of these concurrent sub-expressions. When parallelism is explicitly described, only some of the processes which can be evaluated in parallel are in fact offshipped.

Our results indicate that a speedup typically of more than an order of magnitude can be accomplished by successfully detecting such sub-expressions, compared to the inherent divide-and-conquer parallelism in the original functional program. It will be apparent that many of the sub-expressions will have a very short lifetime, and therefore such parallelism could only be usefully captured on an architecture with small process creation and communication overheads.

Suppose, on the other hand, we had an implementation where our functional program is coded so that the defined functions are implemented as basic operators in their own right. Then we would expect no such extra parallelism to exist. The speedup effected by saturation parallelism is a feature of the granularity of the combinators and operators to which the functional program is compiled.

7.2. Utilisation of a Fixed Processor Set

We now look at the performance given a fixed set of processors. We use two measures. Firstly speedup, namely

lazy_reduction_steps / cycles

(where *lazy_reduction_steps* is the number of reductions where pure lazy evaluation is performed on a single processor) and secondly "efficiency", being

number of processors × *time taken*

where the "time taken" for the 1-processor case is 1. In Figures 11 and 13 the columns represent the speedup for particular numbers of processors, and Figures 12 and 14 the efficiency. The example programs were run with a selection of arguments, where appropriate.

7.2.1. Saturation Parallelism

	Number of Processors			
	2	16	128	1024
Map 1	1.19	2.75	12.93	18.43
Map 5	1.25	3.34	24.83	48.13
Map 9	1.44	8.88	73.73	286.72
Reverse 1	1.27	3.22	9.22	9.22
Reverse 5	1.51	7.52	32.77	32.77
Reverse 9	1.80	13.12	93.31	249.86
Hanoi 1	1.15	2.35	8.83	11.26
Hanoi 5	1.38	7.63	25.09	45.06
Hanoi 9	1.42	8.62	68.61	240.64
Quicksort 2	1.14	2.26	9.60	50.18
Quicksort 3	1.18	2.62	10.88	40.96
Quicksort 4	1.21	2.69	10.37	45.23
Integration	1.15	5.14	27.39	94.21

Figure 11: speedup (saturation parallelism)

	Number of Processors			
	2	16	128	1024
Map 1	1.68	5.82	9.90	55.56
Map 5	1.60	4.79	5.16	21.28
Map 9	1.39	1.80	1.74	3.57
Reverse 1	1.57	4.97	13.88	111.06
Reverse 5	1.32	2.13	3.91	31.25
Reverse 9	1.11	1.22	1.37	4.10
Hanoi 1	1.74	6.81	14.50	90.94
Hanoi 5	1.45	2.10	5.10	22.73
Hanoi 9	1.41	1.86	1.87	4.26
Quicksort 2	1.75	7.08	13.33	20.41
Quicksort 3	1.69	6.11	11.76	25.00
Quicksort 4	1.65	5.95	12.34	22.64
Integration	1.74	3.11	4.67	10.87

Figure 12: time × number of processors (saturation parallelism)

With the saturation strategy, and where the size of problem being solved is sufficiently large, speedup is roughly half the number of processors employed. In other cases, the speedup factor is typically 20% of the number of processors available.

For the first three functions, where the number of steps necessary fully to evaluate the functions are small, the amount of available speculative parallelism is also small, and the parallel efficiency is high. Quicksort, however, requires a relatively large number of reduction steps, and there is ample opportunity for speculative parallelism. We therefore, not surprisingly, find that a much greater proportion of the processors are at any time engaged on evaluating unrequired sections of the graph.

For the function Reverse, we see identical speedups when 128 and 1024 processors are used, for arguments 1 and 5. These indicate that at most 128 processors were required.

Since allocation of processes to processors is via a queue on a "first come first served" basis, there is no distinction made at run-time between different processes of priority *Possible*. For instance, Quicksort 3 has marginally less speedup than Quicksort 2. This is simply a result of unrequired speculative processes happening to be fired, and spawning children, before required ones; thus the *Waiting* process queue becomes overfull of unrequired processes. This behaviour will appear to a programmer to be non-deterministic.

7.2.2. Explicit Parallelism

	Number of Processors				
	2	4	8	16	32
Map 3	1.15	1.32	1.65	1.65	1.65
Map 6	1.49	1.96	2.35	2.72	3.14
Map 9	1.85	3.21	4.49	6.77	8.96
Reverse 3	1.22	1.43	1.60	1.60	1.60
Reverse 6	1.61	2.32	2.90	3.44	4.32
Reverse 9	1.90	3.45	4.90	8.03	11.50
Hanoi 3	1.16	1.26	1.26	1.26	1.26
Hanoi 6	1.43	1.77	2.08	2.30	2.46
Hanoi 9	1.81	3.02	4.17	5.87	7.26
Quicksort 3	1.07	1.27	1.45	1.73	1.73
Quicksort 6	1.31	1.80	2.34	2.78	3.20
Quicksort 9	1.49	2.48	4.54	6.13	9.47
Integration	1.67	2.51	3.58	4.98	6.30

Figure 13: speedup (explicit parallelism)

	Number of Processors				
	2	4	8	16	32
Map 3	1.74	3.03	4.85	9.70	19.39
Map 6	1.34	2.04	3.40	5.88	10.19
Map 9	1.08	1.25	1.78	2.36	3.57
Reverse 3	1.64	2.80	5.00	10.00	20.00
Reverse 6	1.24	1.72	2.76	4.65	7.41
Reverse 9	1.05	1.16	1.63	1.99	2.78
Hanoi 3	1.72	3.17	6.35	12.70	25.40
Hanoi 6	1.40	2.26	3.85	6.96	13.01
Hanoi 9	1.10	1.32	1.92	2.73	4.41
Quicksort 3	1.87	3.15	5.52	9.25	18.50
Quicksort 6	1.53	2.22	3.42	5.76	10.00
Quicksort 9	1.34	1.61	1.76	2.61	3.38
Integration	1.20	1.59	2.23	3.21	5.08

Figure 14: time × number of processors (explicit parallelism)

With explicit parallelism, we would expect poor performance when the number of processors exceeds the available parallelism in the program. This is indeed the case. However, with relatively few processors, parallel efficiency is high.

We obtain a speedup of typically 50% of the number of processors utilised. These figures can be compared with those obtained in [9] where Hudak and Goldberg report processor utilisation and speedup for a divide-and-conquer factorial on a distributed memory machine. A direct comparison cannot be made, for two reasons; they use a metric which includes communication costs for their simulated machine configured as a torus with up to 128 processors, and their algorithm for offshipping processes is simply the divide-and-conquer paradigm. Their heuristic produces a speedup factor typically 50% of the number of processors used. For our simulated machine, when the size of the problem being evaluated is large compared to the number of available processors, and where the problem naturally allows divide-and-conquer parallelism, we also obtain a speedup of around 50% of the number of processors utilised.

It is dangerous to utilise massively parallel evaluation strategies with a small number of processors, unless some mechanism is in place to prevent the machine being swamped with unrequired tasks. The Queue method we use for this exercise ensures that the Needed processes will always be evaluated in preference to speculative Possible processes, and the speedup factor, which will always be at least 1, will be a measure of the efficiency with which ones offshipment algorithm accurately detects the correct subgraphs to evaluate in parallel.

8. Conclusions

We have examined combinator graphs derived from a number of benchmark programs, and obtained experimental results relating to running the program on a parallel machine. Our aim was to compare the number of parallelism opportunities and the idealized speedup factors when using various different strategies for introducing parallel evaluation. Many problems lend themselves naturally to divide-and-conquer parallelism. We have assumed that the problems are sufficiently large to ensure efficient utilization of our machine.

We have demonstrated that by using a "fine-grained" set of combinators one can compensate for the granularity of the operators used, and the consequent large number of basic reduction steps needed to evaluate a function, by employing massive parallelism. We have also confirmed that explicitly introducing divide-and-conquer parallelism with a small number of processors, and using the same machine, enables a speedup approaching linear in the number of processors used. Our machine provides a vehicle for modelling parallelism at both small and large scale.

Our results show that there is potentially a large amount of parallelism present in functional programs written in the divide-and-conquer style, but the problem size must be larger than the number of processors if good speedups are to be possible. As we have ignored the overheads of process creation and interprocessor communication, our figures are essentially upper bounds on the parallelism that can be achieved in this way. Nevertheless, the explicit divide-and-conquer parallelism is large-grain parallelism, particularly in the earlier stages of the division part of divide-and-conquer, and not likely to be much affected by overheads. On the other hand, the extra parallelism obtained with saturation parallelism is typically fine-grain parallelism, and probably realisable only if process creation overheads are very small indeed.

The present analysis is applicable only to shared-memory architectures. Further work remains to be done on using a similar approach for distributed memory architectures, and on larger and more realistic applications using the divide-and-conquer paradigm.

References

1. L. Acker and D. Miranker, “On Parallel Divide-and-Conquer,” Report TR-91-27, Department of Computer Sciences, University of Texas at Austin, Austin, TX (1991).

2. T.H. Axford and M.S. Joy, “List Processing Primitives for Parallel Computation,” *Computer Languages,* 19, 1, pp. 1-17 (1993).

3. H.B. Curry, W. Craig, and R. Feys, *Combinatory Logic, volume 1,* North-Holland, Amsterdam, NL (1958).

4. M.I. Greenberg, “An Investigation into Architectures for a Parallel Packet Reduction Machine,” Technical Report UMCS-89-1, Department of Computer Science, University of Manchester, Manchester, UK (1989). PhD Thesis.

5. P.H. Hartel, “Performance of Lazy Combinator Graph Reduction,” *Software - Practice and Experience,* 21, 3, pp. 299-329 (1991). Also Report D-27, Fakulteit Wiskunde en Informatika, Universiteit van Amsterdam (1989).

6. W.D. Hillis and G.L. Steele, “Data Parallel Algorithms,” *Communications of the ACM,* 29, 12, pp. 1170-1183 (1986).

7. G. Hogen, A. Kindler, and R. Loogen, “Automatic Parallelization of Lazy Functional Programs” in *ESOP’92 - 4th European Symposium on Programming, Rennes, France,* ed. B. Krieg-Brückner, pp. 254-268, Springer-Verlag, Berlin, DE (1992). Lecture Notes in Computer Science 582.

8. P.R. Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” *ACM Computing Surveys,* 21, 3, pp. 359-411 (1989).

9. P.R. Hudak and B. Goldberg, “Experiments in Diffused Combinator Reduction” in *ACM Symposium on Lisp and Functional Programming, Austin, TX,* pp. 167-176 (1984).

10. P.R. Hudak and B. Goldberg, "Serial Combinators: "Optimal" Grains of Parallelism" in *Functional Programming Languages and Computer Architecture,* ed. J.-P. Jouannaud, pp. 382-399, Springer-Verlag, Berlin, DE (1985). Lecture Notes in Computer Science 201; ISBN 3-540-15975-4; Proceedings of. Conference at Nancy.

11. R.J.M. Hughes, "Super-Combinators" in *Conference Record of the 1980 LISP Conference, Stanford, CA,* pp. 1-11, ACM, New York (1982).

12. M.S. Joy and T.H. Axford, "GCODE: A Revised Standard Graphical Representation for Functional Programs," *ACM SIGPLAN Notices,* 26, 1, pp. 133-139 (1991). Also Research Report 159, Department of Computer Science, University of Warwick, Coventry (1990) and Research Report CSR-90-9, School of Computer Science, University of Birmingham (1990).

13. J.R. Kennaway and M.R. Sleep, "Director Strings as Combinators," *ACM Transactions on Programming Languages and Systems,* 10, 4, pp. 602-626 (1988). Also University of East Anglia Report SYS-C87-06.

14. P.W.M. Koopman, M.C.J.D van Eekelen, and M.J. Plasmeijer, "Operational Machine Specification in a Functional Programming Language," Technical Report no. 90-21, Department of Informatics, University of Nijmegen, Nijmegen, NL (1990).

15. K. Noshita and T. Hikita, *The BC-Chain Method for Representing Combinators in Linear Space,* Department of Computer Science, Denkitusin University, Tokyo, JP (1985).

16. S.L. Peyton Jones, "Parallel Implementations of Functional Programming Languages," *The Computer Journal,* 32, 2, pp. 175-186 (1989).

17. S.L. Peyton Jones and M.S. Joy, "FLIC - a Functional Language Intermediate Code," Research Report 148, Department of Computer Science, University of Warwick, Coventry, UK (1989). Revised 1990. Previous version appeared as Internal Note 2048, Department of Computer Science, University College London (1987).

18. P. Roe, "Some Ideas on Parallel Functional Programming" in *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989,* ed. K. Davis and R.J.M. Hughes, pp. 338-352, Springer-Verlag, London, UK (1990). British Computer Society Workshops in Computing Series; ISBN 3-540-19609-9.

19. H. Seidl and R. Wilhelm, "Probabilistic Load Balancing for Parallel Graph Reduction" in *Proceedings IEE Region 10 Conference,* pp. 879-884, IEEE, New York (1989).

20. D.A. Turner, "A New Implementation Technique for Applicative Languages," *Software - Practice and Experience,* 9, pp. 31-49 (1979).

21. D.A. Turner, "Another Algorithm for Bracket Abstraction," *Journal of Symbolic Logic,* 44, 3, pp. 67-70 (1979).