Parallel Implementations of Functional Programming Languages

S. L. PEYTON JONES*
University College London, Gower Street, London WC1E 6BT

One of the most attractive features of functional programming languages is their suitability for programming parallel computers. This paper is devoted to discussion of such a claim. Firstly, parallel functional programming is discussed from the programmer's point of view. Secondly, since most parallel functional language implementations are based on the concept of graph reduction, the issues raised by graph reduction are discussed. Finally, the paper concludes with a case study of a particular parallel graph reduction machine and a survey of other parallel architectures.

Received November 1988

1. INTRODUCTION

It is now very nearly as easy to build a parallel computer as to build a sequential one, and there are strong incentives to do so: parallelism seems to offer the opportunity to improve both the absolute performance level and the cost/performance ratio of our machines. It is important to improve both factors. Mass-produced 8-bit microprocessors offer excellent performance for their cost, but their absolute level of performance is too low for most applications. On the other hand, very few applications are prepared to ignore cost altogether in their search for performance.

A large number of multiprocessors are now commercially available, including bus-based shared-memory architectures (such as Sequent's Symmetry), loosely-coupled distributed-memory ensembles (such as transputer systems or the Intel Hypercube), and even a hybrid of the two (such as the Meiko Computing Surface).

Alas, in stark contrast with these developments in hardware, programming a parallel machine seems to be very much harder than programming a sequential one, except for relatively simple algorithms. One of the most attractive features of *functional programming languages* is their suitability for programming parallel computers. This paper is devoted to a discussion of this claim.

First of all, we discuss parallel functional programming from the programmer's point of view. Assuming that the reader has some knowledge of *graph reduction* on which most parallel functional language implementations are based, we proceed to a discussion of some implementation issues raised by graph reduction. (The paper by Augustsson and Johnsson earlier in this issue gives an introduction to graph reduction.) The paper concludes with a case study of a particular parallel graph reduction machine, a brief survey of other similar machines.

2. PARALLEL FUNCTIONAL PROGRAMMING

2.1 Why programming in parallel imperative languages is hard

Writing a large imperative program is quite difficult, even for a sequential computer. It is more difficult still if it has to take advantage of a parallel system, for reasons which include the following:

(1) The programmer has to conceive of a parallel algorithm which meets the specification.
(2) The algorithm then has to be mapped onto the abstractions provided by the programming language. This entails
   - identifying a number (often a fixed number) of sequential activities, called tasks or processes, which will be run concurrently;
   - defining interfaces between these tasks, which allow them to synchronise and communicate without hazards.

   Any data shared between tasks has to be specially protected by the programmer to prevent hazards. Some systems prevent all such sharing, in which case information has to be shared by explicit data transmission between tasks.
(3) In some systems (e.g. Occam on transputers), the programmer is also responsible for mapping each task, T, onto a processor, and ensuring that the processor is physically connected to all processors running tasks with which the task T communicates.
(4) One detail to which the programmer does not generally have access is the *scheduling policy*, by which each processor determines which task it will run next. Whilst this leaves more freedom for the implementation, the programmer is responsible for establishing that the resulting ensemble of tasks is free from deadlock, and meets the original specification *under all possible interleavings of task execution*. This proof of correctness may be very considerably harder than the corresponding sequential case (which is already very difficult in general). Nevertheless, it appears to be easier than requiring the programmer to specify a precise execution schedule and prove correctness with respect to this.

2.2 Why programming in parallel functional languages is easier.

Of these considerations, *only the first applies to parallel functional programming*. To illustrate this claim, consider the following parallel functional program:

```
par_sum n = dsum 1 n
dsum lo hi = if (hi = lo) then hi
             else (dsum lo mid)
                       +
             (dsum (mid+1) hi)
             where
             mid = (lo+hi)/2
```

The function par_sum is defined in terms of dsum (short for 'doubly-recursive sum'). The latter function works by splitting the problem in half, and computing each half separately, before combining the result together: a classic divide-and-conquer algorithm.

The slightly surprising feature of this program is that it contains no overt reference to a parallel execution strategy. What, then, makes is a parallel program? The difference becomes clear when par_sum is contrasted with a sequential version of the same program:

```
seq_sum n = if(n = 1)
            then 1
            else n+(seq_sum (n-1))
```

This version of sum can only be executed sequentially, because the data dependencies within the expression being evaluated force the additions to take place one after the other. In other words, par_sum *is a parallel program because the data dependencies within it permit the concurrent evaluation of several parts of the expression.*

As in any programming language, a parallel algorithm is essential. There is simply no substitute for the creative work of a programmer in algorithm development. (Actually, in the example, it is conceivable that a clever compiler could make the transformation from seq_sum to par_sum, but almost all real problems are harder than this.)

Notice, however, that in contrast to parallel imperative programming:

(1) *No new language constructs are required to express parallelism*, or synchronisation and communication between concurrent tasks. The concurrency is entirely implicit, and new tasks can be spawned dynamically if the machine has spare capacity to execute them.

(2) *No special measures need be taken to protect data shared by concurrent tasks.* For example, the expression mid is safely shared by the two operands of the multiplication in dsum. In this case mid would probably be computed before the concurrent tasks began, but no special measures would be required even if the shared object was itself the result of a large computation carried out concurrently.

(3) *It is no more complicated to reason about the correctness of a parallel functional program than of a sequential one*, because all the same techniques work, and no new constructs have been added to the language. Furthermore, *deadlock cannot arise*, except as a result of self-dependency (for example: let a = a+1 in a). Expressions which depend on their own value are meaningless, and can often be detected by a compiler.

(4) *The results of the program are determinate;* that is, it is not possible for the results to vary from run to run, depending on extraneous factors such as scheduling policies.

In summary, we suggest that functional programming languages offer a medium in which the programmer can express the essential features of a parallel algorithm, without simultaneously having to detail the solution to a number of lower-level problems.

2.3 Resource allocation and scheduling: the crucial issue

How can we characterise the essential differences between a parallel imperative program and a parallel functional program? Here is one attempt:

> a parallel imperative program specifies in detail many *resource-allocation decisions* which the parallel functional program does not mention at all.

Examples of such resource–allocation issues have already been mentioned, such as: the placement of tasks and data over the processors and memory provided by the machine; the communication mechanisms between tasks; the dynamic generation and control of new tasks; storage allocation and reclamation.

This list suggests analogies in the history of computer science. For example, in the beginning all programs were written in assembly language, and compilers were distrusted because they were unlikely to do as good a job of register allocation as a human programmer. Now we recognise that

- the performance penalty of automatic register allocation is relatively slight;
- progamming at a higher level of abstraction (arbitrary numbers of named storage locations rather than a fixed number of volatile registers) allows us to build vastly more complex programs than we would otherwise be able to contemplate.

To take another example, overlays were invented to allow programs to be executed which could not fit completely in memory at one time. The programmer was responsible for bringing in overlays as required, laying them over code that was no longer needed. Nowadays we take paged virtual memory systems for granted, again tolerating slight performance loss in return for a higher level of abstraction (a large virtual memory space). Actually, the complexity of today's systems is such that one could imagine a paging system outperforming a manual overlay system, particularly if the processor and memory is being shared by more than one user.

A large majority of research in parallel processing focuses (quite properly) on developments of the low-level imperative programming technology. Much of this research is directed towards developing classes of algorithms for which it is possible to pre-determine the resource allocation decisions, and extremely effective parallel systems have been demonstrated. A recent example is the 1000–1023 factor speedups achieved for scientific applications by the Sandia National Laboratories on a 1024-processor hypercube.[1]

Other work is directed towards the development of better programming abstractions (for example: a transputer which provides transparent routing messages between tasks regardless of physical connectivity), in exchange for some loss in performance.

Functional programming languages start at the other end of the spectrum, and offer a very higher level of abstraction to the parallel programmer. At present, so few parallel implementations of functional languages exist that it is impossible to offer a clear experimental basis for claims of expressiveness and performance, but this situation is changing. Nevertheless, history suggests that performance costs become less and less important while the benefits of abstraction become more and more evident.

2.4 Annotations

The high level of abstraction offered by functional languages places heavy demands on the compile-time and run-time strategies by which the mchine allocates it resources. Apart from developing good automatic resource-allocation strategies, it is natural to ask whether it is possible to provide the parallel functional programmer with the ability to control some of the resource–management decisions. One popular proposal for achieving this is by means of *annotations*.

An annotation is a *meaning-preserving* decoration of the program text; that is, the program would deliver the same results even if all the annotations were deleted. An annotation constitutes a piece of advice from the programmer to the compiler about some aspect of resource allocation.

Thus, annotations provide an opportunity to improve (or degrade!) the performance of the proram without the danger of introducing bugs. Unlike the program itself, appropriate annotations are likely to depend on the particular system architecture which is being used to execute the program.

As an example of a possible form of annotation, consider the following version of dsum:

```
dsum lo hi = if (hi = lo) then hi
             else (dsum lo mid)
                  {!} + {!}
             (dsum (mid+1) hi)
             where
             mid = (lo+hi)/2
```

Here, the annotation {!} expresses the programmer's intention that the arguments of the + may be computer in parallel. (If this is the default behaviour for +, then an annotation may be provided to *inhibit* parallel evaluation. Reasons why this might be useful will be discussed later.)

Hudak describes an annotation-based programming methodology recent article in *Computer*.[2]

3. PARALLEL GRAPH REDUCTION

We have already seen that functional programming languages allow programs to be written, in which the parallelism is implicit in the data dependencies of the program. How should we implement a parallel functional language?

Graph reduction is a particularly attractive execution model for parallel systems, for the following reasons.

- There is no sequential concept of 'program counter' in the model. Graph reduction is by its very nature decentralised and distributed.
- Reductions can take place concurrently in many places in the program graph, and the relative order in which reduction are scheduled cannot affect the result that is delivered.
- All communication and synchronisation between concurrent activities is mediated through the graph. At any stage, the current state of the computation is represented by the state of the graph.

A parallel graph reduction implementation is naturally more complicated than a sequential one, and we discuss the issues raised by parallel graph reduction in the following sections. (The paper by Augustsson and Johnsson earlier in this issue gives an introduction to graph reduction.)

3.1 The underlying architecture

We assume a MIMD multiprocessor architecture with a number of *processors* and *memories* connected together with some kind of communication network. The memories hold the graph, and together provide a *global virtual address space*, so that any part of the graph can be accessed from any processor.

MIMD machines are typically divided into closely-coupled shared-memory architectures and loosely coupled distributed-memory architectures. Both, however, are able to support a global address space for the graph so, from the point of view of graph reduction, the only difference between the two is the distribution of access latency to different parts of the address space. For the purposes of this paper we will therefore treat both uniformly. The only extra assumption we make is that each processor has some local memory to which it can make particularly rapid access, which holds for almost all parallel architectures.

The access latency distribution can be visualised conveniently using a *latency diagram* in which memory access latency is plotted on the horizontal axis, and memory size on the vertical axis. Figure 1 gives a possible latency diagram for a bus-based multiprocessor, in which a small amount of memory is available with a short latency, while all the rest is available with a longer latency. Figure 2 shows a possible latency diagram for a hypercube-connected network, in which the amount of accessible memory increases exponentially with latency, until the 'waist' of the cube is reached, when it decreases exponentially again. Latency diagrams allow some of the salient characteristics of a multiprocessor to be expressed in a compact and easily assimilated form. Other crucial characteristics, such as network bandwidth, remain to be expressed separately.

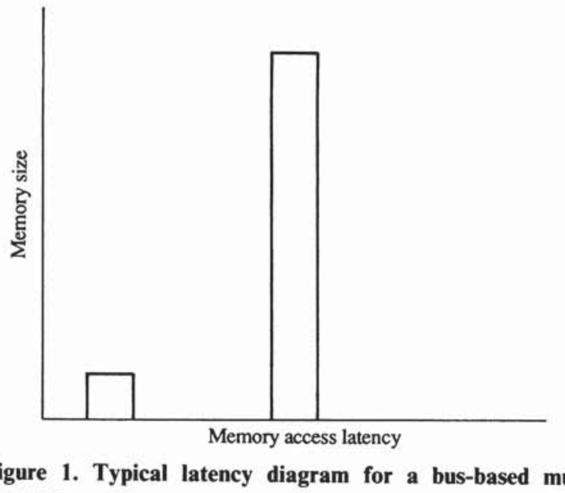

Figure 1. Typical latency diagram for a bus-based multiprocessor.

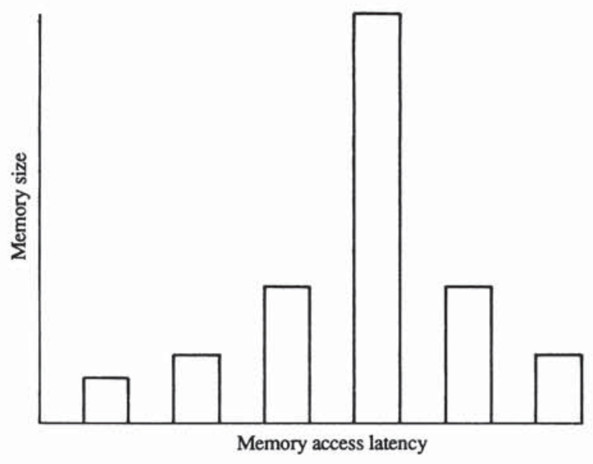

Figure 2. Typical latency diagram for a hypercube multi-processor.

3.2 The parallel graph reduction model

A *task* is a sequential computation whose purpose is to reduce a particular sub-graph to normal form. At any moment there may be many tasks running in a parallel graph reduction machine; this collection of tasks is called the *task pool*. A processor in search of work fetches a new task from the task pool and executes it. A particular physical processor may execute a single task at a time, or may split its time between a number of tasks; tasks can be thought of as *virtual processors*.

Initially there is only one task, whose job is to evaluate the whole program. During its execution, this task may encounter a sub-graph whose value will be required in the future. In this case it may place a pointer to the sub-graph in the task pool, where it is available for execution by other processors; we call this *sparking* a child task.

If the parent task requires the value of the sparked sub-graph while the child task is computing it, the parent becomes *blocked*. When the child task completes the evaluation of the sub-graph, the parent task is *resumed*.

There are two ways of organising the blocking/resumption process. In the *notification model*, the parent is blocked if the child task has not completed when the parent requires its value, or if it has not started work (presumably because no processor was free to execute it). When the child task completes it *notifies* the parent, which causes the parent to be resumed. The same applies to any other task which is by then awaiting the value of the sub-graph. The advantage of this model is that a parent can go to sleep with a notification count, awaiting notification from several children before it is resumed.

In the *evaluate-and-die model*, when the parent requires the value of a sub-graph which it has sparked a child to evaluate, the parent *simple evaluates the sub-graph just as if it had never created the child*. There are then three cases to consider:

- The child has completed its work, and the sub-graph is now in normal form. In this case the parent's evaluation is rather fast, since it degenerates to a fetch of the value.
- The child is currently evaluating the sub-graph. In this case the parent is blocked, and resumed when the child completes evaluation. Upon resumption the parent sees the sub-graph in its normal form, and continues as in the previous case.
- The child has not started work yet. In this case there is no point in blocking the parent, and it can proceed to evaluate the sub-graph. The child task is still in the task pool, but it has become an *orphan*, and can be discarded.

The advantage of this model is that blocking only occurs when the parent and child actually collide. In all other cases, the parent's execution continues unhindered.

Notice that in either case the blocking/resumption mechanism is the *only* form of inter-task communication and synchronisation. Once an expression has been evaluated to normal form, then arbitrarily many tasks can inspect it simultaneously without contention.

3.3 Storage management

Any graph-reduction machine requires a storage-management system which allocates storage for new graph nodes, and reclaims store which is no longer referenced by the graph. Parallel machines present a particular challenge for such garbage-collection techniques. Parallel garbage collection is a subject beyond the scope of this paper, but there is a wide and continuing literature on the subject (to give some recent examples, Appel[3] deals with closely-coupled systems; Deb[4] and Rudalics[5] deal with distributed systems. Cohen[6] gives a survey).

4. GENERATING AND CONTROLLING PARALLELISM

In a parallel machine it is natural to be concerned about whether enough parallelism will be generated. In practice, it seems that a more serious problem is that of getting swamped in parallelism.[7] Consider a divide-and-conquer algorithm, such as dsum above. If the system is capable of evaluating the two recursive calls concurrently, the number of tasks may then explode exponentially, and the machine's store may overflow with partially-completed computations.

In other words, there is an important resource-management problem that is concerned with deciding when a new task should be sparked and run. There are several questions to be answered here:

- will the result of the task be useful?
- is the computation large enough to justify the administration associated with a task?
- is there enough spare capacity to run the task in parallel?
- which task from the pool should an idle processor choose to run next?

4.1 Is the task useful?

Some tasks are more useful than others, but there is a particularly important distinction between tasks whose results will *certainly* be needed and tasks whose results *may* be needed. This distinction is elaborated in the following sections.

4.1.1 Conservative and speculative parallelism

Sometimes it is clear that the value of a sub-graph will *certainly* be useful in the future, and there is no risk of wasting computing resources if we always spark a task to evaluate such sub-graphs. For example, if a task is evaluating the expression $(E_1 + E_2)$, where E_1 and E_2 are arbitrary expressions, it may choose to evaluate E_1 itself, and to spark a child task to evaluate E_2, whose value will certainly be needed later. We call this *conservative* parallelism.

Under a conservative parallelism regime, it is possible to spark a task whenever a function is applied to an argument whose value will certainly be required. A considerable amount of research has been focused on compile-time analysis techniques, called *strictness analysis methods*, which automatically derive this information for each function from the program text. In the case of first-order languages with no data structures, effective methods of strictness analysis have been known for some while,[8] but recent work has extended these methods to higher-order languages or data structures.[9-11]

For some programming paradigms, it may be useful to be able to start parallel computations whose results *may or may not* be useful. For example, a search program may try several alternatives in parallel, and pick the first to terminate. The work done on the other alternatives is wasted, but there is no way to tell which alternative is the useful one at the outset. We call this *speculative* parallelism, because parallel activities are initiated on the basis of speculation about the usefulness of their results.

Certain language constructs make speculation unavoidable. An example is the bottom-avoiding non-deterministic choice operator, amb, which takes two arguments, evaluates them in parallel, and returns the first one to terminate. It is called 'bottom-avoiding' because if one of its arguments fails to terminate but the other does terminate, the terminating argument will be returned. (In the jargon of programming language semantics, the value of a non-terminating computation is called 'bottom'.) If both arguments terminate, then it is clearly indeterminate which will terminate first, because it depends on the machine's scheduling policy. The amb operator makes speculation unavoidable, because once one argument of amb has terminated, the other turns out to have been speculative. Unfortunately, it is impossible to predict which of the two tasks is the speculative one until after the event!

Speculation can be desirable even in completely deterministic contexts. Suppose one function is producing a list whose initial segment is consumed by another. The producer does not know how many list elements will be consumed, so the only conservative strategy is to produce just one element at a time, when it is demanded by the consumer. Unfortunately, this is a completely sequential strategy. To gain parallelism, the producer could speculatively evaluate some elements ahead of the consumer, but these speculative tasks will then have to be killed when the consumer finally discards the rest of the list (remember, we are supposing that the consumer uses only an initial segment of the list).

Another example of the need for speculation concerns language support for interrupts. If a functional operating system begins evaluation of an expression, and the user presses the interrupt key, the evaluation of the program turns out to have been speculative.

It is clear that some support for speculation is desirable. Unfortunately it is also expensive, as the next section shows.

4.1.2 The perils of speculation

Speculative parallelism is considerably harder to implement than conservative parallelism.

- Conservative tasks should always be run in preference to speculative tasks. For example, in the conditional expression if E_1 then E_2 else E_3 the expression E_2 and E_3 could be sparked speculatively whilst the condition E_1 was evaluated. This certainly gains parallelism, but if the machine spends all its resources evaluating E_2 and E_3 it will never make progress! Clearly the conservative task evaluating E_1 should be run at a higher priority.
- Speculative tasks may become conservative. For example, if E_1 evaluated to True, then the speculative task evaluating E_2 would now be conservative, and its priority should be increased to reflect this fact. Furthermore, the priority of some of the tasks it has spawned should also be increased, namely those ones whose evaluation is essential for the evaluation of E_2; and so on.
- Speculative tasks may become *irrelevant*, such as the task evaluating E_3 in the example. Such irrelevant tasks should be killed; and their children, and *their* children, and so on.
- Speculative tasks should be scheduled fairly so that, for example, a search program explores all the alternatives at roughly equal rates. This requirement significantly complicates the scheduler, and adds run-time overhead.

To suggest how hard some of these priority-changing operations are, consider the special case of killing irrelevant tasks, which amounts to reducing their priority to zero. Killing irrelevant tasks may at first appear to be a similar problem to that of garbage collection of memory, but it is much more difficult for several reasons.[12]

Firstly, an irrelevant task is not merely passively harmful like unreferenced storage. It is actively pernicious, consuming processing resources, allocating more store, and (worst of all) spawning further irrelevant tasks. There is an obvious danger that it will breed new tasks faster than the task-killer can kill them off.

Secondly, the possibility of sharing in the graph complicates matters. Suppose that a conservative task is blocked awaiting the evaluation of a sub-graph which just happened to be shared by a speculative task which got there first; then if the speculative task is killed the conservative task will wait for ever.

A promising approach to the problem is to link the killing of irrelevant tasks with the ordinary garbage collector. Any task which is evaluating a garbage graph node (that is, a node unreachable from the root of the original graph) must certainly be irrelevant, and can be killed.[13] Unfortunately, this does not identify all irrelevant tasks. There are at least two possible reasons why an irrelevant task might be evaluating a non-garbage node:

- It may be that the subgraph is being retained for possible later evaluation; though it is possible to argue in this case that the task is not entirely irrelevant after all.
- Alternatively, it may be that the subgraph is still attached via a *space leak*, such as an unevaluated

expression of the form (fst (E_1, E_2)). In this expression, E_2 is garbage since the selector fst selects the first element of the pair, but the garbage collector will not usually realise this fact.[14]

There are a number of further objections to this task-deletion mechanism. Firstly, there may be a considerable delay between a piece of data becoming garbage and its identification as such, during which the irrelevant task can proceed unchecked. Secondly, the method does not generalise to handle other changes of priority, such as a priority upgrade. Thirdly, garbage data cannot be re-allocated until all irrelevant tasks which might refer to it have been killed.

An obstacle for more sophisticated schemes is that there is no direct reference from a task to those it has spawned, so the task-deletion mechanism has no direct way of finding the irrelevant children. Hudak[15] proposes a scheme which addresses this issue directly, by attaching to each graph node a list of all the nodes which it has sparked. His method thereby solves all of the difficulties mentioned above, with the exception of the 'space leak' problem, but the complexity and overheads involved are quite substantial, and the algorithm has not been implemented.

4.1.3 Conclusion

It seems clear that speculative parallelism is sometimes desirable, but it is not at all clear how best to implement it.

One approach, taken by Henderson and Jones in their work on functional operating systems,[16] is to ignore the problem! They gave speculative tasks the same priority as conservative tasks, and made no attempt to kill irrelevant tasks. Their non-deterministic construct was a bottom-avoiding list-merging operator, so that the alternative discarded by one non-deterministic selection would always be one of the alternatives in the next selection. As a result, an irrelevant task would invariably become speculative again, and hence the fact that it had never been killed off did not matter.

A similar approach is taken by the MIT dataflow group. Speculative tasks are always allowed to run to completion, regardless of whether their results are actually required or not: 'on our dataflow machine this phenomenon is reflected in the sometimes unnerving behaviour that an answer may be printed long before termination is reported'.[17] (In their case, though, it is essential to complete even the speculative tasks, in case two assignments are made to the same element of an I-structure, indicating a run-time error.)

These solutions are hardly general, and it seems inevitable that substantially increased overheads are associated with general speculative parallelism. This is an area where further work is required.

Some systems allow the programmer to control the generation of new tasks by means of annotations, rather than relying entirely on automatic inferencing techniques. This may have to be carried out with some care, since the system may assume that such tasks are conservative. If the programmer inadvertently initiates a speculative task which turns out to be irrelevant, the machine may deliver the results more slowly, or even run out of space before completion. Annotations may not be entirely benign!

4.2 Is the task big enough?

There is always some administrative overhead associated with the creation of a task, so it would be desirable to avoid creating tasks which only perform a very small amount of computation. Furthermore, the more tasks there are, the greater the amount of (expensive) communication and synchronisation that will take place. This is saying no more than the common observation that large-grain parallelism is easier to implement efficiently than fine-grain parallelism.

The two main strategies that have been suggested to address this problem are compile-time analysis and programmer annotation.

4.2.1 Compile-time complexity analysis

Under this approach, the compiler tries to analyse the program to determine how much work is involved in evaluating each part, and only generates code to create new tasks when the work involved is sufficient to justify it. In general this is impossible, since the answers may depend on the input data.

Nevertheless, some simple approximations may eliminate the worst offenders. For example, the approach taken by Goldberg[18] to estimate the execution cost of evaluating an expression is as follows:

- if the expression is a simple arithmetic expression, then its cost can be computed rather simply;
- if the expression involves a call to a known function, then inspection of the function body can yield the execution cost provided that the function is non-recursive, and calls only known non-recursive functions, and so on;
- if the expression involves a call to a known recursive function it is hard to find a compile-time bound for the evaluation cost, so assume an infinite cost;
- similarly, if the expression involves a call to a function which is a free variable of the expression, the evaluation cost will depend on the function bound to this variable, so assume an infinite cost.

The approximations are rather coarse, and most expressions encountered in practice are attributed an infinite cost.

Better estimates may be possible, but they are likely to carry some run-time overhead. For example, the cost of adding up the elements of a list is proportional to the length of the list. Even if the compiler can figure out this fact, how is the run-time system to find the length of the list? Either every list has to carry round its length in an easily accessible place, or the length of the list has to be computed dynamically. Each carries such heavy overheads that the whole exercise is unlikely to improve the overall performance.

It seems unreasonable to expect very substantial progress in this area, but solid experimental evidence is lacking.

4.2.2 Programmer annotations

Alternatively, programmer annotations can supply directives to the compiler that particular expressions are or are not of sufficient size to merit a task, possibly based on a criterion computed at run-time. For example, consider the dsum function from Section 2.2. At each stage the problem is divided in two, and initially is is sensible for new tasks to be generated. Nevertheless, when the difference between the arguments becomes sufficiently small, it would be better to refrain entirely from sparking new tasks, even if there were still idle processors available.

For example, consider the following version of the dsum example.

```
dsum lo hi
= if (hi = lo) then hi
  else (dsum lo mid)
  + {! worth_sparking}
  (dsum (mid+1) hi)
  where
  mid = (lo+hi)/2
  worth_sparking = (hi−lo) > 50
```

The annotation {! worth_sparking} is intended to indicate to the run-time system that it is worth sparking a subtask for the second argument of the + if the value of worth_sparking is true, that is if the difference between hi and lo is sufficiently large. When the difference becomes small enough, no further tasks are sparked.

The point at which sparking should be stopped depends on the value of the arguments in a way which it would be ambitious to expect the compiler to figure out. There appears to be no substitute for some programmer insight here, together with some fine tuning based on performance measurements.

It is possible to take this idea even further. A good compiler can produce near-optimal sequential machine code for functions such as dsum, so further improvements could be made by compiling two versions of dsum, one which sparked new tasks, and one which was compiled as for a sequential machine. Which of the two is used could depend, as before, on a programmer annotation which took into account the values of the parameters.

4.3 Is there any spare capacity?

There is no point in sparking a new task if the machine is already saturated with work. An ideal situation would be for the problem to break into one independent task for each processor, each of which is executed sequentially in the processor's local memory, before the results are finally combined together. Whilst the run-time scheduling system is unlikely to achieve this ideal in practice, it does suggest that the system should refrain from creating new tasks when the machine load is greater than some threshold.

Such a strategy has a number of desirable effects. Firstly, it reduces the resources required to add and remove tasks from the task pool. Secondly, it makes collisions between tasks less likely because once the machine has become sufficiently loaded tasks cease creating children with which they may subsequently collide. Thirdly, it increases locality, because tasks cease creating children which may be executed on another processor. In effect, the grain size is dynamically increased.

It is worth noting that the evaluate-and-die model of synchronisation (see Section 4.1) can easily accommodate this strategy, whereas matters are more complicated for the notification model. If the notification model is used, and a potentially sparkable sub-graph is not sparked because the machine load is high, this decision has to be communicated to the code which consumes the value of the sub-graph. If not, the parent task may indefinitely await notification from a child task that does not exist. If the evaluate-and-die model is used, sparks may freely be discarded without informing the consumer at all.

Limiting the machine load in this way depends on each processor having a reasonable measure of the current machine load. Possible measures include the size of the task pool, the utilisation of memory, and the utilisation of the processors over the recent past. Unfortunately, none of these measures necessarily relate directly to the amount of work required to process all the tasks in the local pool. For example, the machine could appear to have plenty of work in the task pool, but it might largely consist of small tasks.

The load information can either be computed centrally by a system management process and broadcast to all processors (as is done in the GRIP multiprocessor, described below), or computed in a distributed way. In the Flagship machine, for example, the load average is computed at an extremely low level, in the interconnection network itself.[19]

Alternatively, each processor can use an estimate its *own* load to control whether or not it creates new tasks, relying on a separate mechanism to reduce its load if other processors are idle by migrating its tasks to them.

Even given a good measure of load, the load-limiting strategy can lead to the following undesirable situation: a task may refrain from creating a sub-task when the machine is loaded but, if the machine subsequently runs out of work, the sub-task might have proved very useful.

This is another area in which careful experiments are required to discover effective strategies.

4.4 Effect of scheduling strategy

At any moment there may be many tasks awaiting execution, and only a finite number of processors available to execute them. Which task should be scheduled first, and how important is this decision?

To a first approximation, assuming conservative parallelism, it makes no difference which task is selected for execution, since all tasks are doing essential work. Nevertheless, scheduling policy does have some important effects when the number of tasks exceeds the number of processors available to execute them:

- Execution of long sequential task might be postponed until near the end of the computation, whereas it would be better to overlap its execution with other tasks by starting it earlier.
- Some strategies may breed sub-tasks faster than others. Fast-breeder strategies are good when the machine is unloaded, and bad when the load increases.

Eager *et al.* [20] elegantly demonstrate that, assuming conservative parallelism, under *any* scheduling policy the processor utilisation for N processors must be larger than $A/(N+A-1)$, where A is the speed-up factor that would be obtained by executing the same program with an unbounded number of processors. If $A \gg N$ this lower bound approaches 1, which reassures us that scheduling policy is comparatively unimportant for programs with a high average level of parallelism. Furthermore, provided $A > N$, the processor utilisation must be greater than 50%, which provides a surprisingly good bound for programs whose average level of parallelism is as small as the number of processors.

Using an idea proposed by Burton and Sleep,[21] the Manchester dataflow group have discovered an effective strategy for controlling the explosion of parallelism in divide-and-conquer algorithms.[7] If the task scheduled next for execution is the *most recently created task*, then the tree of tasks is explored in a *depth-first* manner. In contrast, scheduling the *least recently created task* first gives rise to *breadth-first* exploration of the tree. The depth-first strategy minimises the number of partially-completed tasks, and hence optimises for space usage, while the breadth-first strategy seeks maximum parallelism. The Manchester group advocate dynamically changing the scheduling strategy depending on the load of the machine, using breadth-first when the load is low, and depth-first when the load increases. They call this mechanism a *throttle*.

5. LOCALITY AND THE PLACEMENT OF TASKS

Having now considered how parallelism is generated and controlled, we now move on to discuss issues raised by the underlying physical machine architecture.

In considering how to map a parallel implementation of graph reduction onto a given parallel architecture, one issue dominates all the others: *how can a high degree of spatial locality be achieved simultaneously with a high degree of processor utilisation*? In order to reduce the communication bandwidth required to manageable proportions it is essential that as much communication as possible is very local, namely between a processor and its local memory. There is of course one trivial way to achieve perfect locality, by executing all tasks on one processor, an observation which points out the tension between locality and processor utilisation.

Three major strategies influence the achievable locality and processor utilisation, which will be discussed in turn in the succeeding sections:

- Data placement strategy.
- Task placement strategy.
- Caching strategy.

For sufficiently well-behaved problems it is in principle possible for the programmer or compiler to plan a static task and data placement strategy. For many parallel computers (such as array processors and systolic architectures), this is an absolute requirement. However, as Section 2 has discussed, the attraction of parallel functional programming is exactly that this sort of static planning is not required, and indeed it would often be an impossibly complex (and input-data-dependent) problem.

Hence we concern ourselves exclusively with *adaptive* policies, which take decisions on the basis of information about the current system state, rather than being decided statically in advance. There is a wide range of possible strategies, ranging from very simple ones based on limited information, to complex policies which attempt to make near-optimal decisions based on detailed information about the system state.

It may also be possible to use programmer annotations to control data and task placement decisions.[2]

5.1 Data placement

The simplest data placement strategy is to allocate new data objects locally to the processor doing the allocation.

In principle, a more sophisticated scheme might allocate data on a processor which was subsequently going to work on that data, but this requires prior knowledge of the task placement strategy, which itself may vary in response to run-time factors. This kind of prediction seems impossibly hard to achieve, so we assume a straightforward local allocation strategy only.

5.2 Task placement

Each processor holds a pool of executable tasks, and together these pools form the global task pool. The task placement strategy has two conflicting goals:

- To achieve good processor utilisation, tasks must migrate from busy processors to idle processors.
- To achieve good locality, tasks should not migrate far from the graph which they are supposed to evaluate.

Notice that to achieve optimal processor utilisation it is only necessary that all processors should be busy so long as there is any work to be done; that is, if there is any idle processor then all the task pools are empty. It is *not* necessary, for example, that all task pools are of equal size, which is an implicit assumption of many proposed schemes. Hence we use the term *load sharing* rather than *load balancing*.

Eager *et al.*[22] have obtained encouraging results from studies of adaptive strategies for task migration in a rather general context, though their results are only partially applicable to parallel graph reduction. They concern themselves exclusivly with the situation when tasks are being generated at a slower average rate than the agregate task execution rate of the whole system. If tasks are generated faster than this, all the task pools will rapidly fill up, and there is no need for task migration. They also assume that new tasks are generated uniformly across processors, and they do not include the effects of spatial locality.

In all the policies they studied, a processor attempts to export tasks when its task pool exceeds a certain fixed size. In the simplest policy, an exporting processor unilaterally sends any task it wishes to export to a random destination processor. To prevent thrashing, a task can only be exported a fixed maximum number of times, after which it is no longer eligible for export. Even this very simple strategy gave surprisingly good performance. After trying a number of more elaborate policies, they conclude that *very simple strategies give near-optimal performance, over a wide range of system parameters*.

Eager *et al.* only consider stategies in which the exporting processor plays the dominant role. Another alternative is that processors with small task pools could solicit work from other processors (starting perhaps with those nearby). A combination of the two, designed for systems in which processors can only communicate directly with a small number of neighbours, is called *diffusion scheduling*. Each processor maintains an estimate own workload and communicates this to its neighbours. If the processor believes its workload to be significantly higher than that of some neighbour, it tries to balance the load by exporting some tasks to that neighbour. In this way, work 'diffuses' across the machine from busy parts towards less busy parts.

Diffusion scheduling has two particularly attractive characteristics, both of which tend to reduce long-range communication:

- Only local system state information is required.
- Since tasks do not 'jump' right across the machine, there is some hope that tasks will tend to access data held in nearby memories.

Diffusion scheduling has been simulated in the Rediflow architecture[23] and implemented by Goldberg in the Alfalfa compiler on the Intel iPSC hypercube.[18] Goldberg experimented with a variety of variants on diffusion scheduling, and his results strongly support those of Eager *et al.*, namely that the simplest strategies often perform very nearly as well as (and occasionally better than) more complex ones.

5.3 Caching stategy

One of the most fruitful ways of improving locality in computer systems is *data caching*. In terms of graph reduction, this means that each processors caches local copies of some remote graph nodes. Typically, remote nodes will be fetched when they are first referred to, and thereafter cached until they have to be removed from the cache to make space for new nodes.

Whenever two or more separate caches hold distinct copies of the same data item, there is a danger that one processor will modify the data item in its cache, and the others will then be left holding stale data. This leads to the notorious problem of maintaining *cache coherence*, the requirement that every cache accurately reflects the true state of the memory, so that no processor can access stale data. There is a substantial literature on mechanisms for maintaining cache coherence in a multiprocessor system, but such schemes are invariably restricted to bus-based systems.

A very attractive feature of graph reduction is that *it eliminates essentially all the problems of cache coherence in multiprocessors:*

- When a graph node is unevaluated, the blocking mechanism prevents more than one task from accessing it, so that it can freely be cached by the unique processor which succeeds in accessing it.
- When evaluation of the graph node is complete, it is overwritten with its evaluated form, which causes any tasks which were blocked on the node to be resumed. The node cannot now change any further (since it is fully evaluated), so an arbitrary number of processors can cache it without fearing loss of coherence.

In other words, graph nodes can freely be cached regardless of whether they are evaluated or not. The absence of side effects in a functional language implementation has paid off with an important architectural benefit. The caches should, incidentally, implement a write-through policy, so that as soon as a node is updated with its evaluated form the original copy is also updated, thus freeing any blocked tasks.

There are a number of differences from conventional caches:

- The unit of caching is a graph node, whose size may not necessarily be fixed. This requires a more flexible cache structure than the usual fixed-size cache slots. Indeed, it may be most sensible to use the processor's local heap (which is already capable of allocating graph nodes) for cached data, together with an addressing mechanism to translate virtual to local addresses. This is exactly the scheme proposed for Flagship.[19]
- Conventional caches often fetch a small block of storage around the addressed word, on the grounds that the nearby store locations are relatively likely to be accessed in the near future (this can be seen as a form of prefetching). In a graph-reduction system, there may not be any relationship between graph nodes which happen to be adjacent in the virtual address space. If prefetching is implemented, it probably makes sense to fetch nodes *pointed to* by the addressed node, rather than nodes *adjacent to* it.
- The latency incurred on a cache miss may be substantial, depending on the machine architecture. If so, the latency must be hidden by running other tasks while waiting for the remote fetch to be performed. This imposes a fairly stringent requirement for rapid context-switching.[24]

6. A CASE STUDY: THE GRIP ARCHITECTURE

GRIP is a multiprocessor developed under Alvey funding to execute functional and logic languages. This section describes GRIP and its graph reduction mechanisms, to serve as a focus for the preceeding general discussion.

6.1 Architectural overview

GRIP consists of up to 20 printed circuit boards, each of which contains four processors and one Intelligent Memory Unit (IMU). The boards are interconnected using a fast packet-switched bus, and the whole machine is attached to a UNIX host. A bus was chosen specifically to make the locality issue less pressing. GRIP still allows us to *study* locality, but does not require us to *solve* the locality problem before being able to produce a convincingly fast implementation.

Each processor consists of a M68020 CPU, a floating-point coprocessor, and 1 Mbyte of private memory which is inaccessible to other processors.

The IMUs collectively constitute the global address space, and hold the graph. They each contain 1 M words of 40 bits, together with a microprogrammable data engine. The microcode interprets incoming requests from the bus, services them and dispatches a reply to the bus. In this way, The IMUs can support a variety of memory operations, rather than the simple READ and WRITE operations supported by conventional memories.

The following range of operations is supported by our current microcode:

- Variable-sized graph nodes may be allocated and initialised.
- Garbage collection is performed autonomously by the IMUs in parallel with graph reduction, using a variant of Baker's real-time collector.[25]
- Each IMU maintains a pool of executable tasks. Idle processors poll the IMUs in search of these tasks.
- Synchronised access to graph nodes is supported. A lock bit is associated with each unevaluated node, which is set when the node is first fetched. Any subsequent attempt to fetch it is refused, and a descriptor for the fetching task is automatically attached to the node.

  When the node is overwritten with its evaluated form (using another IMU operation), any task descriptors attached to the node are automatically put in the task pool by the IMU.

The IMUs are the most innovative feature of the GRIP architecture, offering a fast implementation of low-level memory operations. A separate project, called BRAVE, is writing different microcode to support Prolog on the same hardware base.

6.2 Graph reduction on GRIP

We start from the belief that parallel graph reduction will only be competitive if it can take advantage of all the compiler technology that has been developed for sequential graph-reduction implementations.[26] Our intention is that, provided a task does not refer to remote graph nodes, it should be executed exactly as on sequential system, including memory allocation and garbage collection.

We implement this idea on GRIP in the following way. Each processor uses its private memory as a local heap, in which it allocates new graph nodes. There are no pointers into this memory from outside the processor, so it can be garbage-collected independently of the rest of the system.

There are four main reasons why a processor may need to make non-local accesses, namely: fetching non-local nodes, updating non-local nodes, making new tasks available to other processors, and acquiring new tasks made available by other processors.

Fetching non-local nodes. When a processor needs a non-local node, it fetches it from the appropriate IMU, and caches a copy in its local heap. These cached copies are discarded when the local heap gets too full. (The fetch latency is such that it is not worth the overhead of context-switching while waiting for the result.)

Updating non-local nodes. When a processor completes evaluation of a global node, it updates it with the evaluated form. This is relatively expensive because, in order to maintain the invariant that there are no external pointers into a processor's private memory, the entire subgraph accessible from the updated node has to be moved into global memory (at least, those parts which are not already there).

Making tasks global. Each processor maintains a private task pool of tasks which it has sparked. If the system load is low enough, it exports part of this pool to an IMU, remembering again to move into global memory the entire subgraph thereby made accessible. Again, the cost is a necessary one: tasks which are being made public must be moved into a publicly visible place.

Nevertheless, this cost is only incurred when the system load is low. When the system fills up with tasks, each processor simply runs locally. When the local task pool is full, the processor refrains from sparking any new tasks, as discussed earlier. This is easily implemented because we use the evaluate-and-die model of task synchronisation.

Getting new tasks. When a processor runs out of local tasks, it polls the IMUs until it finds some new ones to do. (It starts this process with the IMU on its own board, of course.)

It is clearly desirable to avoid performing updates if the node being updated is not shared. This is true in a sequential implementation but the cost of performing global updates makes it even more important in a parallel machine. Happily, recent advances in sequential compiler technology have shown how to use compile-time analysis to identify some of the nodes which are not shared, and how to avoid performing updates for such nodes.[27,28] We are incorporating these ideas into our implementation.

Only conservative parallelism is supported at present.

6.3 System management

Each processor can run a number of *processes*, under a lightweight operating system. Most processors run exactly one process, which performs graph reduction, but some processors run system management processes which a relatively low duty cycle, and therefore do not merit a processor to themselves.

A single System Manager process, running on a particular processor, is responsible for polling the processors and IMUs in the system, assessing the overall system load, and informing the processors whether they should seek to export tasks or not. The System Manager is also responsible for synchronising global system state-changes, such as occur during garbage collection.

Other processes are responsible for performance monitoring, system loading, and input/output.

6.4 Current status

An interpretative implementation of graph reduction is running on this hardware, and a compiled implementation is currently in the late stages of development. Funding is now being sought to develop real applications on this system, in order to test the claims of this paper, and to provide a realistic setting in which to develop effective resource-allocation strategies.

7. OTHER ARCHITECTURES

There are very few operational parallel implementations of functional languages, and we briefly review them in this section.

7.1 MIMID architectures

The ALICE machine was probably the first parallel implementation of a functional programming language.[29] It consists of up to 40 *processing agents* and *packet pool segments* (a form of intelligent memory), connected with multistage switching network. Each agent and memory board was built from transputers, programmed in Occam. Two machines exist, one at Imperial College and one at AIAI in Edinburgh.

The Zapp machine, developed at the University of East Anglia, consists of a network of transputers connected in a *virtual tree* (not dissimilar to a hypercube).[30] It has demonstrated very high speed-ups (for example, a speed-up of 39.9 on a 40-transputer Zapp), but it is limited to a simple divide-and-conquer programming methodology. It is clearly not a general-purpose graph-reduction machine, and is programmed in Occam rather than a functional language, but serious attempts were made to generate the sort of code which could be generated by a functional-language compiler.

The Flagship project is much more ambitious follow-up to ALICE.[31] The project addresses the issues of parallel *declarative* programming in the large, including non-determinism, transaction processing, security, and a host of related problems. A parallel architecture forms part of the project, and a multiprocessor emulator exists, based on 68020 boards.

Goldberg has implemented a parallel functional language using compiled graph reduction on both the Intel iPSC hypercube, and the Encore Multimax shared-memory machine, and reports in detail on his work in his thesis, which is an extremely useful source.[18]

Based on their original work on the G-machine, Johnsson and Augustsson have developed a model for parallel evaluation which they call the ν-G-machine. They are now developing a compiler targeted for the Sequent Symmetry shared-memory multiprocessor.[32]

7.2 SIMD architectures

Rather surprisingly, it has turned out to be possible to perform graph reduction on a SIMD architecture.[33, 34] The idea is to break the program up into the composition of a small fixed set of operators. One operator is chosen by the control processor and all its instances executed simultaneously, while all the other operators are quiescent. Then the control processor selects a new operator for execution, and so on.

Another approach, on a tree-structured SIMD machine is suggested by O'Donnell *et al.*[35]

7.3 Dataflow architectures

So far this paper has focussed on implementation techniques based on graph reduction. A completely different approach to the parallel implementation of functional languages is exemplified by *dataflow architectures.*[39] The main groups active in this area are at MIT, Manchester and the Electrotechnical Laboratory in Japan; Arvind and Culler give a good survey.[39]

Dataflow is based on the principle of data items (or *tokens*) flowing along arcs between elementary computational units, which await all their input tokens before *firing* and producing an output token. In practice it has turned out that this simple idea requires quite complex implementations when non-strict data structures, recursive functions, and higher-order functions are included. The solutions to these problems, based on *tagged-token dataflow* and *I-structures*, are now quite well understood.[40]

Dataflow seems at first to be an extremely radical, fine grain form of parallel execution, inherently incapable of taking advantage of advances in conventional von Neumann computer architecture. Recently, however, ways of combining von Neumann and dataflow architectures have been proposed, which should lead to interesting new developments.[41, 42]

7.4 Implications for architecture

Functional programming languages and their implementations look at first like excellent candidates for special-purpose machine architectures. Whilst there have been some attempts to build such machines,[36, 37] the reality seems to be more prosaic: the gains that special-purpose machines can deliver can be substantially surpassed by the application of good compiler technology. For the most part, functional programs can be compiled to run efficiently on conventional hardware.

If anything, a strongly-typed functional language has less need for hardware support than a Lisp system, because the compile-time type-checking obviates the need for the run-time checks which Lisp has to perform. There is, nevertheless, scope for carefully-targeted hardware support in the areas of memory management, garbage collection, and task synchronisation. For example, Kieburtz has made extensive measurements of the run-time behaviour of G-machine-style implementations, as a basis for the design of a processor intended specifically for functional programming.[38]

8. CONCLUSION

Parallel functional programming is a technology that is in its adolescence. Its potential is enormous, but it needs to be tried out in practice on real applications before it can command wide credibility. Fortunately implementations are now becomming available which are able to support real applications, and we can expect to see a substantial increase in the use of functional programming in the next few years.

As this paper has demonstrated, the very factors which make functional programs attractive are exactly those which place heavy demands on the compiler and run-time resource management system. However, as implementations improve, the costs of functional programming should become increasingly neglible; and as software becomes more complex, the benefits should become increasingly valuable.

Acknowledgements

I gratefully acknowledge the help of Phil Wadler and Paul Hudak in commenting on the text of this paper. Work on the GRIP project has been carried out in partnership with ICL and High Level Hardware Ltd, and has been supported by the UK Alvey Directorate. Many fruitful technical meetings have taken place as part of this project, which have contributed to the view I have presented above. Particular contributors to these meetings have been Geoff Burn, Chris Clack, Nic Holt, John Hughes, David Lester, John Robson and Jon Salkild.

REFERENCES

1. R. E. Genner, J. L. Gustafson and R. E. Montry, Development and analysis of scientific applications programs on a 1204-processor hypercube, SAND 88-0317, Sandia National Laboratories, Albuquerque (Feb. 1988).
2. Paul Hudak, Para-functional programming. *IEEE Computer* **19**, 60–71 (1986).
3. A. W. Appel, J. R. Ellis and Kai Li, Real-time concurrent collection on stock multiprocessors. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation, Atlanta*, pp. 11–20. ACM (June, 1988).
4. A. Deb, Parallel garbage collection for graph machines. In *Graph Reduction* (ed. J. H. Fasel and R. M. Keller), pp. 252–243. Lecture Notes in Computer Studies [LNCS] 279, Springer Verlag, (1986).
5. M. Rudalics, Distributed copying garbage collection. *Proc ACM Conference on Lisp and Functional Programming* (Aug., 1986).
6. J. Cohen, Garbage collection of linked data structures. *Computing Surveys* **13**, 343–367 (1981).
7. Carlos, A. Ruggiero and John Sargeant, Control of parallelism in the Manchester dataflow meachine. In *Proc. IFIP Conference on Functional Programming Languages and Computer Architecture, Portland* (ed. G. Khan), 1–15. LNCS 274, Springer Verlag (1987).
8. Alan Mycroft, The theory and practice of transforming call-by-need into call-by-value pp. 269–281. In LNCS 83. Springer Verlag (1981).
9. G. L. Burn, C. L. Hankin and S. Abramsky, Strictness analysis for higher order functions, *Science of Computer Programming* **7**, 249–278, (1986).
10. Paul Hudak and Jonathan Young, Higher-order strictness analysis for untyped lambda calculus. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 97–109. ACM (Jan., 1986).
11. Philip Wadler and John Hughes, Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture* (ed. G. Kahn). LNCS 274, Springer Verlag (1987).
12. D. H. Grit and R. L. Page, Deleting irrelevant tasks in an expression-oriented multiprocessor. *ACM TOPLAS* **3**, 49–59 (1981).
13. Paul Hudak and Robert M. Keller, Garbage collection and task deletion in distributed applicative processing systems. In *Proc. ACM Symposium on Lisp and Functional Programming*, pp. 168–178 (1982).
14. Philip Wadler, Fixing a space leak with a garbage collector. *Software–Practice and Experience* **17**, 595–608 (1987).
15. Paul Hudak, Distributed task and memory management. In *Symposium Principles of Distributed Computing* (ed. N. A. Lynch *et al.*), pp. 277–289. ACM (Aug., 1983).
16. Simon B. Jones, Abstract machine support for purely functional operating systems, PRG-34. Programming Research Group, Oxford (Aug., 1983).
17. Arvind and R. S. Nikhil, Executing a program on the MIT tagged-token dataflow architecture. In *Proc. PARLE (Parallel Languages and Architectures, Europe) Conference, Eindhoven*. LNCS, Springer Verlag (1987).
18. Benjamin F. Goldberg, Multiprocessor execution of functional programs. YALEU/DCS/RR-618. Dept of Computer Science, Yale University (April, 1988).
19. Paul Townsend, Flagship hardware and implementation. *ICL Technical Journal* **5** 575–594 (1987).
20. D. L. Eager, J. Zahorjan and E. D. Lazowska, Speedup versus efficiency in parallel systems. Tech. Report 86-08-01, University of Sasketchewan (Aug., 1986).
21. F. Warren Burton and M. R. Sleep, Executing functional programs on a virtual tree of processors. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire*, pp. 187–194. (Oct. 1981).
22. D. L. Eager, E. D. Lazowska and J. Zahorjan, Adaptive load sharing in homogeneous distributed systems. *IEEE Trans Software Engineering* **SE-12**, 662–675 (1986).
23. R. M. Keller and F. C. H. Lin, Simulated performance of a reduction based multiprocessor. *IEEE Computer* **17**, 70–82 (1984).
24. Arvind and R. A. Iannucci, Two fundamental issues in multiprocessing. *Proc. DFVLR Conference on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg* (June, 1987).
25. Henry Baker, List processing in real time on a serial computer. *CACM* **21**, 280–294 (1978).
26. Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall (1987).
27. Jon Fairbairn and Stuart Wray, TIM–a simple lazy abstract machine to execute supercombinators. In *Proc. IFIP Conference on Functional Programming Languages and Computer Architecture, Portland*, (ed. G. Kahn) pp. 34–45. LNCS 274, Springer Verlag (1987).
28. S. L. Peyton Jones, The Spineless Tagless G-machine. In *Proc. Workshop on Graph Reduction, Aspenas Sweden*. Programming Methodology Group, Chalmers University, Sweden (Sept., 1988).
29. P. G. Harrison and M. Reeve, The parallel graph reduction machine ALICE. In *Graph Reduction: Proceedings of a Workshop, Santa Fe* (ed. J. H. Fasel and R. M. Keller), pp. 181–202. LNCS 279, Springer Verlag (1986).
30. D. McBurney and M. R. Sleep, Transputer-based experiments with the Zapp architecture. In *Proc. PARLE Conference I*, pp. 247–259. LNCS 258, Springer Verlag (1987).
31. I. Watson, J. Sargeant, P. Watson and V. Woods, Flagship computational models and machine architecture. *ICL Technical Journal* **5**, 555–574 (1987).
32. Thomas Johnsson, The ν-G-machine – an abstract machine for parallel graph reduction. Dept of Computer Science, Chalmers University, Goteborg, Sweden (Sept, 1988).
33. W. D. Hillis and G. L. Steele, Data parallel algorithms. *CACM* **29**, 1170–1183 (1986).
34. P. Hudak and E. Mohr, Graphinators and the duality of SIMD and MIMD. In *Proc. ACM Conference on Lisp and Functional Programming*, pp. 224–234. ACM (Aug., 1988).
35. J. T. O'Donnell, T. Bridges and S. W. Kitchell, A VLSI implementation of an architecture for applicative programming. In *International Conference on Frontiers in Computing, Amsterdam*, (Dec., 1987).
36. William Stoye, Thomas Clarke and Arthur Norman, Some practical methods for rapid combinator reduction. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pp. 159–166 (Aug. 1984).
37. Mark Scheevel, NORMA – a graph reduction processor. *Proc. ACM Conference on Lisp and Functional Programming* (Aug. 1986).
38. Richard B. Kieburtz, Performance measurement of a G-machine implementation. In *Graph Reduction – Proceedings of a Workshop, Santa Fe* (ed. J. H. Fasel and R. M. Keller), pp. 275–296. LNCS 279, Springer Verlag (1986).
39. Arvind and D. E. Culler, Dataflow architectures, *Annual Reviews in Computer Science* (1986).
40. Arvind and R. S. Nikhil, Executing a program on the MIT tagged-token dataflow architecture, *Proc Parallel Architectures and Languages Europe, Eindhoven, June* (1987).
41. R. A. Iannucci, A dataflow/von Neumann hybrid architecture, *TR 418, Lab for Computer Science, MIT, Nov* (1988).
42. R. S. Nikhil and Arvind, Can dataflow subsome von Neumann computing, *CSG Memo 292, Lab for Computer Science, MIT, Nov* (1988).