Parallel Functional Programming: An Introduction

Kevin Hammond*
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, U.K.
kh@dcs.glasgow.ac.uk

1 Introduction

Parallel functional programming has a relatively long history. Burge was one of the first to suggest the basic technique of evaluating function arguments in parallel, with the possibility of functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation [22]. Berkling also considered the application of functional languages to parallel processing [17].

Due to the absence of side-effects in a purely functional program, it is relatively easy to partition programs so that sub-programs can be executed in parallel: any computation which is needed to produce the result of the program may be run as a separate task. There may, however, be implicit control- and data- dependencies between parallel tasks, which will limit parallelism to a greater or lesser extent.

Higher-order functions (functions which act on functions) can also introduce program-specific control structures, which may be exploited by suitable parallel implementations, such as those for algorithmic skeletons (Section 3.2).

Here is a classic divide-and-conquer program, a variant on the naive Fibonacci program. Since the two recursive calls to `nfib` are independent, they can each be executed in parallel. If this is done naively, then the number of tasks created is the same as the result of the program. This is an exponentially large number ($O(2^n)$).

```
nfib n = if n <= 1 then 1
         else 1 + nfib(n-1) + nfib(n-2)
```

*Supported by a SOED Research Fellowship from the Royal Society of Edinburgh and the UK EPSRC Parade project.

1.1 Determinism

The semantics of a purely functional program define the result of that program for a fixed set of inputs. It follows that functional programs are deterministic in the following useful sense: any program which runs sequentially will deliver the same result when run in parallel with an identical input. Apart from possible resource starvation the parallel program will also terminate under exactly the same conditions. This level of determinacy is useful in several respects:

- Programs can be debugged to remove algorithmic bugs without needing a parallel machine. Since programming environments are generally better on sequential machines, and there are usually fewer time limitations on their use, this can be an important pragmatic concern. Performance problems must still be detected by actual (or perhaps simulated) parallel execution.

- The result of the program is independent of dynamic task scheduling, except in terms of memory exhaustion. Thus tasks can be executed in any desired order, with locking provided by the normal execution model.

- Deadlock is impossible, except in conditions where the sequential program would also fail to terminate due to cyclic dependencies [107].

There are some programs (e.g. branch-and-bound searches) where nondeterministic results are required. It is beyond the scope of this paper to consider nondeterminism in detail, but Section 5.1 contains some pointers to the literature.

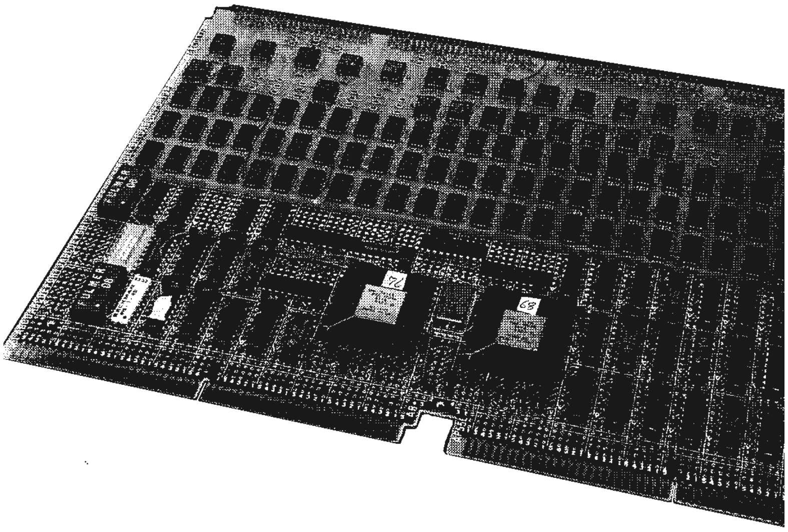

Figure 1: An ALICE Board

1.2 Strictness

There are two broad classes of functional language: strict languages, where all the arguments to a function are evaluated before the function itself is evaluated (e.g. Hope [25], OPAL [38])[1], and non-strict languages, where arguments are evaluated only if they are needed (e.g. Haskell [67], pH [92], Id [91]). There are also hybrids, where simple arguments such as integers are evaluated before the function itself, but where complex arguments such as lists and other recursive data structures are not (e.g. Hope$^+$ [100]).

Non-strict languages may be implemented using either a *demand-driven* or a *data-driven* approach. The latter is normally termed dataflow. Dataflow implementations are discussed in Section 5.2. With demand-driven languages, arguments are computed as required by the function. If the result of the argument is recomputed each time it is needed, this is *call-by-name*; conversely, if the result is shared this is *call-by-need*. Call-by-need is typically implemented using graph reduction. Non-strict languages have the advantage that only expressions which must be evaluated to give the program result actually are evaluated. It is thus possible to manipulate notionally infinite data structures. Their principal disadvantage is that, in general, it is necessary to

Strictness Analysis

Although all needed expressions can be executed in parallel, statically determining which expressions are actually needed can be difficult in a non-strict language. Since exactly one redex is chosen at each step in a call-by-need reduction, it may at first seem that there is no opportunity for parallel execution [76]. However, this ignores the fact that call-by-need is really just a means for implementing non-strict reduction.

A function `f` is strict in its argument if, according to the language semantics, `f` $\perp = \perp$, where $\perp$ is the symbol representing the undefined value. Since the result of the function will be undefined if any argument in which it is strict is also undefined, it is safe to execute the body of the function in parallel with any strict arguments. Non-terminating computations are treated as being semantically equivalent to $\perp$. So in the definition of `nfib` above, `nfib n = if n <= 1 then ...`, because `n` is used in the condition, `nfib` is clearly strict in this argument.

[1] These are "pure" functional languages. SML and Lisp are examples of "impure" but strict functional languages.

1.3 Outline of this Paper

The structure of this paper is as follows. Section 2 is a brief historical overview. Section 3 considers the main ways in which parallelism can be exploited in functional languages. Section 4 considers implementation issues, including load management, communication, speculation and memory management. Section 5 considers related research into concurrency, non-determinism, dataflow programming, functional-logic programming, term-graph rewriting and parallel execution of impure functional languages. Section 6 considers future research directions. Finally, Section 7 suggests some possibilities for further reading.

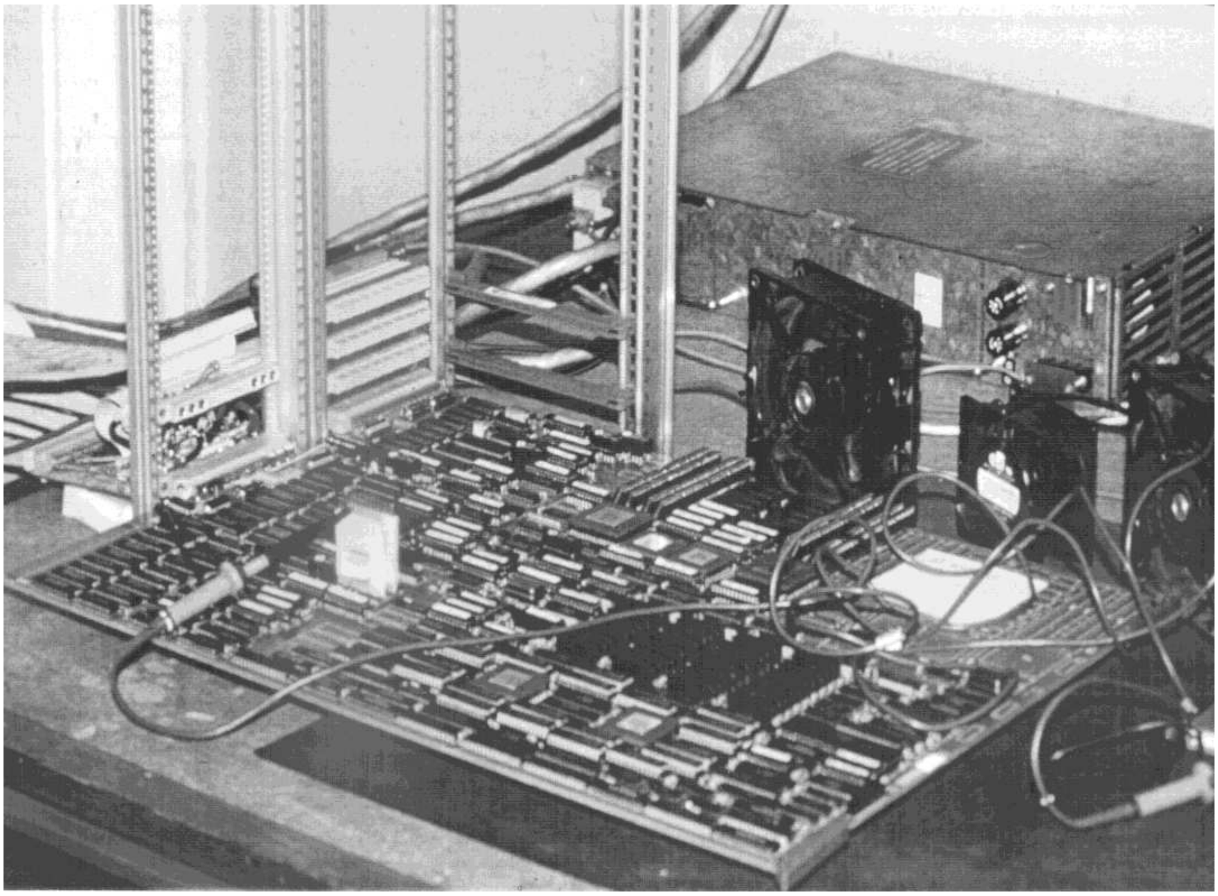

Figure 2: A GRIP board on the test rack

2 A Brief History

Not surprisingly, the history of parallel functional programming implementation is closely intertwined with developments in sequential compiler technology. Many early implementations were interpretive to a greater or lesser degree. As interest in abstract machines, such as G-machine derivatives, peaked around 1989 so special-purpose parallel abstract machines were also designed (e.g. Tuki [32], PAM [82]). In the absence of cheap and effective parallel hardware, many implementations such as ZAPP were simulated [28] long before they were realised in hardware [87].

2.1 Novel Architectures

Backus' famous paper [11] kindled much interest in functional languages as a means of breaking the von-Neumann bottleneck between processors and memory, which he perceived to be a limiting factor in the design of new computer architectures. Indeed, for some time, it was believed that novel architectures were *necessary* to achieve high performance with functional languages, and this led to a spate of designs for special-purpose machines, many of which were parallel designs. Section 5.2 describes

Reduction Machines

The first and most famous physical reduction machine was ALICE (Applicative Language Idealised Computing Engine), designed by Darlington and Reeve at Imperial College in 1981 [36] and built over a period of several years. The eventual aim was to build ALICE in VLSI, but in the event the only ALICEs actually built used stock components.

The prototype ALICE comprised 40 Transputer-based processing agents and packet pools, connected by a multi-stage switching network (Figure 1 shows an ALICE board). Overtaken by developments in sequential compiler technology such as supercombinators [69, 72], and by improvements in conventional hardware design, the absolute performance of this machine was ultimately disappointing [56]. This may be partly explained by the interpretive nature of much of the prototype software, but the use of many small packets probably also degraded performance to some extent.

Many valuable lessons were learned from the design of this machine, and these have been applied to more recent architectures such as the ICL Flagship [124, 122, 40] and EDS/Goldrush designs [121] (now emerging as a commercial, though no longer purely functional, product). These machines have been designed to address a wide variety of pragmatic usability issues, such as the provision of multi-user computing and fault tolerance, which are of less importance in a basic research setting.

The novel graph reducer GRIP (Graph Reduction in Parallel) [102, 101] is built from a network of distributed conventional processors. It has a two-level bus structure, and incorporates fast packet-switching hardware for message routing, and intelligent memory units for efficient operations on globally shared graph and spark/task pools. Because it uses microcoded CPUs for the intelligent memory units and PALs for the packet-switching hardware, the GRIP machine has proved extremely flexible. Since its inception it has undergone a transformation from a parallel abstract machine interpreter to a machine running compiled Haskell directly. Figure 2 shows a GRIP board being tested on the GRIP test rig.

Several other early machine designs were parallel combinator implementations. Examples of these are COBWEB [55], Burroughs' NORMA [106], and SKIM [119]. Most of these were built in some form, but were quickly overtaken by the widespread adoption of supercombinators before they could deliver significant results. Other interesting machine designs which have failed to come to fruition were Rediflow [74], COBWEB-2 [5], and Magó's FFP machine [83].

Most recent implementations have used conventional hardware. There have, however, been several recent proposals for novel implementations: Star:Dust suggests adding special communication and task control instructions to an otherwise unremarkable RISC processor [97]; BWM is a VLIW machine to execute functional programs [9]; and G-Line also exploits horizontal parallelism within supercombinator reductions [89].

2.2 Stack, Packet or Environment?

For a long time, there was a sharp division between packet-based and stack-based implementations. In packet-based designs, function arguments and workspace are allocated as part of a closure, whereas stack-based designs use a per-task stack to hold these values. Figure 3 shows the difference between these representations.

Packet-based implementations allow easier task distribution and migration since a packet contains all the information needed to execute it. Compared with using a stack, however, this at the cost of losing locality and perhaps increasing overall communication latency (through the need to fetch work incrementally). In contrast, in a stack-based system, if a task is exported then the entire contents of the stack must also be exported with it. Examples of packet-based systems are Augustsson and Johnsson's $\langle \nu, G \rangle$-machine [10], and Kingdon, Lester and Burn's HDG-machine [78]. Modern parallel implementations are often stack-based (e.g. the Parallel ABC Machine [94], GRIP [102]). In addition to the advantages quoted above, this has the practical benefit that it is possible to build on advances in sequential compilation technology rather than pursuing an independent development route.

The other main sequential implementation technique popular in the past, which was based on shared *environments* has some obvious problems for distributed parallel implementations. It has thus never been seriously adopted by parallel implementors.

3 Exploiting Parallelism

There are myriad possible ways to exploit the parallelism present in a functional program. Most systems have selected a set of these, and it is therefore difficult to isolate the effect of a single technique on overall performance, even when concrete performance results are available.

There are two basic strategies for deciding how to partition a program. With *implicit* partitioning, the compilation system decides which tasks should be created; with *explicit* partitioning, however, the programmer is left with the problem of determining which expressions should be created as tasks. In either case, the partition could be *static*, in which case the number of tasks which will be created at runtime is predetermined, or *dynamic*, in which case tasks are created depending on factors such as the overall runtime load, or load control annotations. Tasks may be placed on the processor creating the task, on the processor owning the data which the task requires, or on some other processor. Task placement may also be explicit or implicit, static or dynamic.

It is, of course, vitally important to choose tasks of an appropriate *granularity* (or duration) for the target machine. The best partition for a given machine will be one which maximises the available parallelism, up to the number of processors available, while minimising the parallel overhead. Coarse-grained tasks are desirable in order to minimise task creation overheads. However, excessively coarse granularity can lead to increased idle time if too few tasks are created, and can also introduce high task migration overheads, if the load is imperfectly distributed (Section 4.1).

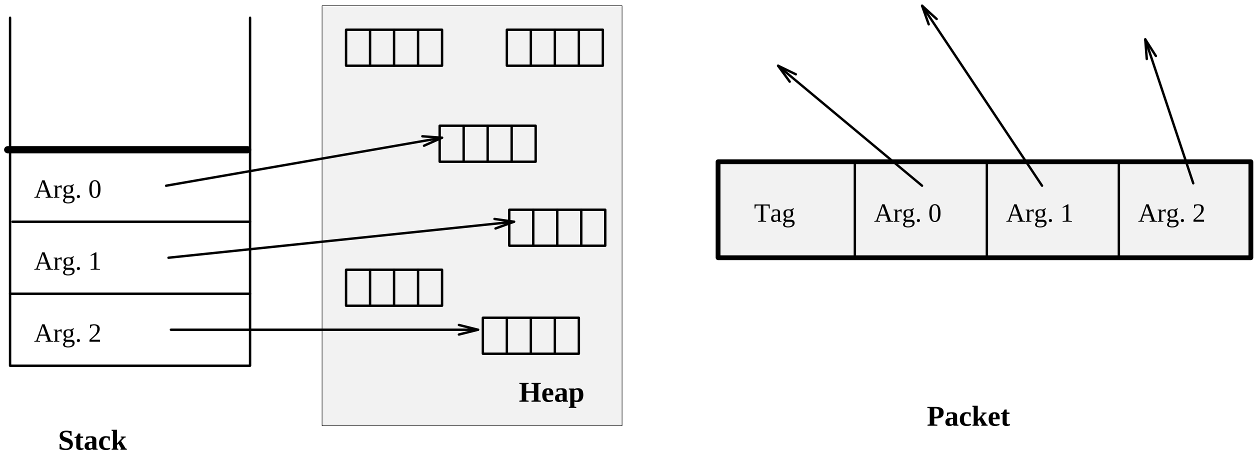

Figure 3: Argument Passing Conventions

3.1 Implicit Parallelism

Due to strictness effects, implicit parallelism takes a different form in strict and non-strict languages. It is almost embarassingly easy to partition a program written in a strict language. Unfortunately, the partition that results often yields a large number of very fine-grained tasks.

In non-strict languages, implicit parallelism is normally obtained from needed expressions, which are detected by strictness analysis. However, neededness is undecidable in general, and strictness analysis has so far failed to deal effectively with realistic parallel programs.

Serial Combinators

Serial combinators express fairly low-level control information as pseudo-functions in a functional language [65]. They are not intended for direct use by programmers, but are intended to be inserted by an automatically partitioning compiler.

For example, the `nfib` function can be written (in a slightly sanitised syntax) as

```
nfib n =
  (demand n
     (spawn ((n1 (nfib (n-1)))
             (n2 (nfib (n-2))))
       (wait (n1 n2)
             (n1+n2+1))))
```

The Alfalfa project implemented serial combinators for the Intel iPSC [44]. Unfortunately, the communication overheads on this machine were extremely high, and this interacted badly with programs which needed to share data such as divide-and-conquer matrix multiplication [45]. Buckwheat re-implemented this model for the Encore Multimax, a shared-memory multiprocessor. Performance results were much more encouraging for this machine, with good relative speedup being achieved for all test programs including matrix multiplication [45].

3.2 Explicit Parallelism

In the absence of good implicit techniques for distributed machines, most implementations have resorted to some degree of explicit control. These range from the GRIP approach which simply annotates potentially parallel expressions, to the detailed partitioning and placement possible in Caliban [75].

Annotations

Annotations for parallelism have been proposed by many authors. One of the first and simplest is Burton's $@_P$ to indicate parallel function application [30]. More complicated annotations to control the precise degree of evaluation through strictness annotations were proposed for Hope$^+$ on Flagship [77]. This can be seen as being akin to an explicit version of Burn's work on evaluation transformers to exploit strictness information [24, 23]. The most complete example of a annotation-based language is perhaps Concurrent Clean, which has annotations to control graph copying and sharing as well as task placement and scheduling [1].

Hudak introduced the term *para-functional programming* for functional languages with annotations that preserve the functional semantics [68]. For example, `exp $on left($self)` could execute `exp` on the processor to the left of the current one. A later paper refined these ideas by introducing *schedules* of events which include explicit demands and process creation/termination, plus sequential and parallel compositions as well as task mappings to particular processors [64].

Control Languages

Caliban expands on this idea by providing a separate functional language whose entire purpose is to specify task placement. A program comprises a process part and a wiring part. Normal functions may include a `moreover` clause which specifies how named expressions are connected. The connection is implicit, through function application and may be captured through higher-order functions (which act as wiring templates). The wiring system must be resolved entirely statically: Caliban includes heuristics to prevent excessive resource consumption by cyclic specifications. For example, the following function defines a pipeline. The `@` syntax is used to create an anonymous process which applies the function it labels (this is called □ in [75]). `arc` indicates a wiring connection between two processes. `chain` creates a chain of wiring connections between elements of a list. The result of the pipeline function for a concrete list of functions and some argument is thus the composition of all the functions in turn to the initial value. Moreover, each function application is created as a separate process.

```
pipeline fs x = result
  where    result = (foldr (.) id fs) x
  moreover (chain arc (map (@) fs))
           /\ (arc @(last fs) x)
           /\ (arc @(head fs) result)
```

There has not yet been much practical experience with Caliban. An implementation has been produced for the Meiko transputing surface [33], but no concrete performance results have yet been published. A practical study into the use of Caliban to program an implementation of the Gamma model of parallel execution (see Section 5.3) gave disappointing results under simulation [57] giving a speedup of only 30%. On the positive side, it did prove straightforward to express the required process network in Caliban.

Commutative Monads

Monads have proved popular in encapsulating state problems in sequential functional programming [120]. The idea is derived from category theory, and allows type-based control of certain kinds of state.

Because monads are generally used to program state, their implementations are usually deliberately single-threaded. This is not necessary, however. If the monad is commutative, then the operations captured within it can be computed in parallel. This has been exploited to produce a parallel type inference algorithm [50], and to formulate a general framework for parallelism [73]. In the latter system, for some monad `Par`, with operations

```
  unit :: a -> Par a
  bind :: Par a -> (a -> Par b) -> Par b
  fork :: Par a -> Par b -> Par (a,b)
```

`Par` is *commutative* if the following two definitions of `fork` are equivalent.

```
fork1 p q = p `bind` \ x -> q `bind` \ y ->
            unit (x,y)

fork2 p q = q `bind` \ y -> p `bind` \ x ->
            unit (x,y)
```

It is simple to reformulate `nfib` in terms of these operations

```
nfib n :: Par Int
nfib n = (nfib (n-1)  `fork`
          nfib (n-2)) `bind` \ (n1,n2) ->
         unit(n1+n2+1)
```

Unfortunately, the type of `nfib` now precludes its use in non-monadic functions without some special trickery.

Algorithmic Skeletons

Algorithmic skeletons were so named by Cole [31]. The idea, which pre-dates the name, is to capture patterns of parallel computation, such as divide-and-conquer or pipelining, in higher-order functions. These parallelism templates can then be instantiated by the programmer to suit a particular algorithm. The approach has the advantage of restricting parallelism to a small, easily isolated part of the program (the skeletons themselves, plus the places where they are used). Ideally, the same skeleton can be used for different architectures: it is necessary only to change the implementation of the skeleton in order to change the program's behaviour, the program itself is unchanged, both textually and semantically.

For example, a divide-and-conquer skeleton `divCon` might be defined by:

```
divCon :: (Prob   -> Bool)   ->
          (Prob   -> [Prob]) ->
          ([Soln] -> Soln)   ->
          (Prob   -> Soln)   ->
          (Prob   -> Soln)

divCon divisible split join f prob =
     if divisible prob then
        join (parmap f (split prob))
     else
        f prob
```

The `divisible` argument determines whether the problem of type `Prob` can be subdivided. If so, then the the problem `prob` is split into a list of subproblems each of type `Prob`; the worker function `f` (whose argument is a `Prob` and whose result is a `Soln`) is applied to each subproblem; and finally the list of solutions each of type `Soln` is combined into a single solution of type `Soln` using `join`. For example, `nfib` could be skeletonised as:

```
type Prob = Int; type Soln = Int

nfib n = divCon div split join nf n
  where div n = n > 1
        split n = [n-1,n-2]
        join [r1,r2] = 1 + r1 + r2
        nf n = if n > 1 then nfib n
                        else 1
```

Skeletons are not a universal panacea: one problem is that the set of skeletons required to express all possible parallel programs is large, and perhaps even infinite. Thus, no implementation based on a fixed set of skeletons can hope to be completely general. It remains to be seen whether adequate sets of skeletons can be found to suit most common parallel programming paradigms [105]. A secondary problem is that good partitioning and scheduling may require architecture specific information, and perhaps different skeletons.

The skeleton approach is seductive, however, and unlike some of the other approaches mentioned here, work has continued at several sites. Implementations are now starting to appear [35, 21], though performance has often been problematic. While skeletons are generally seen as a static technique, there is no real reason why dynamic skeletons could not be employed to obtain good partitioning, providing suitable hints to a good runtime control system. Indeed the well-known ZAPP [28] and serial combinator [65] approaches can be seen as early examples of this. ZAPP has since achieved good results for divide-and-conquer parallelism on networks of transputers [46, 87].

3.3 Data Parallelism

The techniques which have been described so far attempt to exploit parallelism in control structures. Several researchers have tried the alternative approach of exploiting *data parallelism*, which arises where common operations can be applied to all elements of a large data structure in parallel.

A simple example is a parallel `map` function, which applies some function to each element of a list in parallel to produce a result of the same length as its argument. For instance, the function `findImpacts` is part of a simple ray tracer. Given a set of rays and a scene which those rays may intersect, it returns a list representing the first point at which each ray impacts an object in that scene.

```
findImpacts :: [Ray] -> Scene -> [Impact]
findImpacts rays scene =
    parmap (earliestImpact scene) rays
```

The effect is to create one task for each ray. Since these tasks are identical they can be easily executed on a SIMD machine.

Typically, in real data parallel programs, more complex functions such as the families of *fold* and *scan* functions are applied to large data structures. These functions then control how arguments are communicated to the tasks, and how results are collected. For example, a rightmost scan operation for lists could be defined by,

```
scanl :: (a->b->a) -> a -> [b] -> [a]
scanl f a xs =
    [foldl f a (take i xs)
              | i <- [0..length xs-1]]
```

These ideas also arise in imperative languages, of course, such as Connection-Machine Lisp [118]. However, functional languages have the advantage of richer data structures and the ability for the programmer to create new data parallel operators using higher-order functions. Clearly there is a connection between data-parallelism and the skeleton approach: the right skeleton can be used to introduce data parallelism. Some recent approaches to exploiting data parallelism in functional language are POD comprehensions, which aim to combine data parallelism with lazy evaluation [60], bidirectional fold and scan [96], and the data parallel language NESL, which provides a mechanism for nesting parallelism [19, 20].

4 Implementation

This section surveys issues which arise in the implementation of parallel functional languages: load management, communication, memory management and speculation.

Many of these issues arise in all parallel implementations, but there are often more options for automatically controlling the parallel execution of functional languages. This is, in fact, one of their perceived advantages over imperative techniques.

4.1 Dynamic Load Management

It is useful to distinguish between *load balancing*, which aims to maintain an even workload across the machine by suitable task distribution, and more general *load management* which also aims to control which tasks are created. Good load management often includes some load balancing, but this can be a secondary consideration. Because it is often expensive to migrate tasks between processors in a distributed memory machine, it may in fact be preferable to accept an uneven load in order to minimise overall execution time. Load balancing techniques are most effective on shared-memory machines.

In spite of Eager et al's result, which states that for an ideal shared-memory architecture no schedule is more than a factor of 2 worse than the best schedule for a program [39], it is known that scheduling can be extremely poor in the worst-case [27], and this is borne out in practice [54]. Techniques such as local task pool scheduling can have a significant effect on granularity and overall performance [52]. Hofman's thesis [61] is a good source for further reading on scheduling and granularity issues.

The term *spark* is used here to denote a node which has been marked for potential parallel execution. This is distinct from the task which actually evaluates the node to produce its result, not least because the spark has only a minimal state attached and thus imposes much lower memory overheads than a task. The term *future* is often used in parallel Lisp implementations with much the same meaning.

Throttling

The Manchester dataflow machine [47] introduced the idea of *throttling*: controlling the rate at which tasks are created by changing the scheduling strategy [108]. In this machine, tasks are symmetric and fine-grained. There is a single task pool, the *token store*, and no real distinction between sparks and tasks.

Given a definition of `nfib` similar to the one defined earlier (and noting that the dataflow machine is data-driven and strict),

```
nfib n = if n <= 1 then 1
         else 1 + nfib(n-1) + nfib(n-2)
```

each time this function is executed, two new tasks are created. The parent task is suspended until both children complete execution and notify it with their values. It then sums their results and adds one before notifying its own parent.

If the task pool is represented by a FIFO queue, then the effect is to create an explosion of suspended tasks ($O(2^n)$). If however, a LIFO stack is used, then the effect is more like sequential evaluation, and only $O(n)$ tasks will be created. This avoids flooding the token store with suspended tasks but, on the dataflow machine, is not effective for programs involving loops. Ruggiero and Sargeant therefore implemented a mixed strategy, which changes between FIFO and LIFO scheduling on the basis of system load [108]. In low load situations, FIFO scheduling is used to stimulate task creation. In high load situations, LIFO scheduling is used to prevent the creation of excess tasks. In a (demand-driven) functional setting, this strategy works well for divide-and-conquer parallelism [111].

4.1.1 Hysteresis

In an attempt to prevent rapid changes between the two scheduling strategies, which can prove disruptive a form of *hysteresis* was implemented. Rather than using a single load-based threshold, two thresholds are provided: high H and low L. When the workload rises above the high threshold, LIFO scheduling is used. Conversely, when the workload drops below the low threshold, FIFO scheduling is used. If $H > L$ then there is a range of settings s, $L <= s <= H$, where the strategy used depends on the one that was previously being used. So if LIFO scheduling is in force, and the workload drops below H, but not below L, LIFO scheduling will remain in force. If however, FIFO scheduling is in force, and the workload rises above L, but not H then FIFO scheduling will remain in force.

Unfortunately, hysteresis was not found to be particularly effective for the programs which were studied. In practice, it proved quite tricky to set the hysteresis parameters and to find settings which were universally good [54].

Task Creation

Careful task creation can also have a highly beneficial effect on scheduling, since superfluous tasks can be absorbed into their parents. This has the effect of increasing task granularity depending on system workload.

The simplest technique is *load-based inlining*. A decision as to whether to create a task is made immediately a node is sparked, based on the processor's understanding of the current system load. If the load is below some threshold, then the task is created, otherwise the spark is discarded. Several researchers have studied this technique ([90, 97]) and it has proved reasonably effective on simple programs. Its principal disadvantages arise from the fact that decisions concerning task creation must be taken at the time a node is sparked. This has three negative effects:

- A significant amount of memory may be needed to maintain the workspace for the newly created task (for stacks etc.); consequently only a few tasks can be maintained.
- A significant overhead is incurred if the tasks created are fine-grained, whether they are exported or retained locally.
- There is a strong probability of discarding significant parallelism.

Two similar approaches which attempt to overcome these problems are *evaluate-and-die* [101] and *lazy task creation* [90].

With lazy task creation, sparks are maintained in a local pool. Sparks are exported on demand to idle processors. If a spark is not exported, then it is absorbed into its parent when the parent task returns to the spark point.

The evaluate-and-die technique improves on this by taking advantage of graph reduction. If a sparked node is needed (not speculative), then it must be attached to the parent computation. When the parent task needs the child's value, one of three situations may apply.

1. The spark has been exported and evaluated by another task: the parent obtains the child's value as normal.

2. The spark has not been exported: the parent evaluates the node as if it had not been sparked and discards the spark.

3. The child node has been exported and is still being evaluated in another task: the parent suspends until the child has been evaluated and its value updated.

It is only necessary to synchronise tasks in the third case.

A refinement on this is to defer updates for remote nodes until the value of the node is demanded. This helps reduce communication, especially if (as is often the case) the node is not shared. As a further refinement, because any non-speculative spark must be contained within the parent computation, it is safe to discard any spark if desired, at the cost of reduced parallelism.

The effect of these schemes is to combine LIFO and FIFO scheduling, as with the Flagship model [111]. If the least recently created sparks are exported, then exported tasks are effectively generated in LIFO order. These tasks are the ones which are most likely to generate further parallelism in a divide-and-conquer setting, and are also likely to have the greatest granularity if the process tree is well balanced. The most recently created sparks

An alternative scheme for increasing granularity is based on the rate at which tasks have previously been created [3].

Task Export

There are two basic techniques for exporting tasks when they are created: they can be distributed to remote processors on the basis of load information, or as a result of annotations; or they can be stolen by idle processors as needed. The former runs the risk of distributing work where it is not needed. The latter may increase the latency before tasks are created remotely (because they are exported only on demand rather than being exported eagerly).

Because tasks are relatively large entities, it is normally best to export sparks, which can then be turned into tasks by the recipient processor. This both optimises the use of the communications system and reduces the overhead on the exporting processor.

4.2 Simulated Annealing

The above are all dynamic techniques. Sargeant has studied the granularity which can be obtained with strict, but still purely functional languages using feedback from sample executions to drive the compilation process [112]. Sarkar's work on automatic partitioning for Sisal took a similar approach [113].

4.3 Communication Issues

One of the advantages of using pure functional languages is that communication can be handled implicitly, through demands on shared data structures. This basic mechanism can allow data to be exported with a newly created task, and results to be communicated to the parent task. Implicit communication helps avoid deadlock, and considerably simplifies the programming task.

Blocking

When a task needs the value of a graph node which is being evaluated by another processor, it has two basic choices:

- It can re-evaluate the node locally.

- It can *block* until the node has been evaluated remotely.

Re-evaluation has the disadvantage of increasing the overall workload (which thus reduces potential parallelism), and perhaps of losing sharing (and so introducing a space leak). In the absence of information concerning the cost of evaluation and the size of the result, this is probably undesirable. If, however, the cost of re-evaluating the node is low, and the cost of communicating the result is relatively high it may be sensible to do this. As a corollary, this implies that a naive implementation of *full laziness* [69], where nodes are shared whenever possible regardless of evaluation cost, can be positively harmful in a parallel environment.

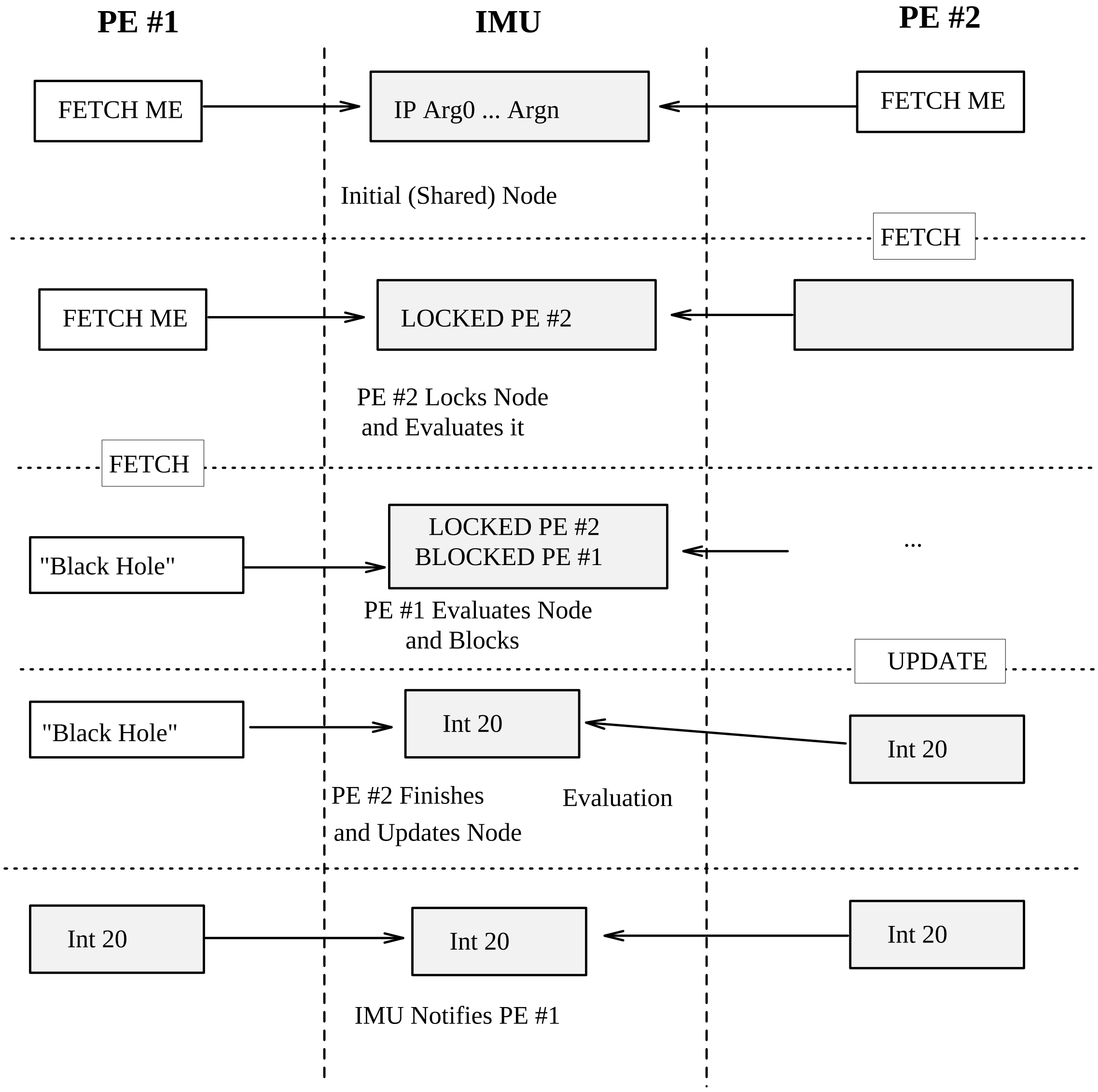

Figure 4: Blocking in GRIP

If a task blocks, then it is suspended until the shared node has been evaluated, and the fact that it is blocked is recorded in the processor which is evaluating the node. When the node is updated with its result, the updating processor *notifies* any blocked tasks. These may then be resumed. In some systems, it is possible to set up a notification on a shared node without becoming blocked. This allows the task to perform further work if possible, such as evaluating the remaining strict arguments to a function, and so helps reduce idle time at the cost of complicating the blocking mechanism and code generation.

Figure 4 shows the sequence of blocking on GRIP, where communication occurs through independent Intelligent Memory Units (IMUs), which collectively hold the globally shared program graph.

4.4 Speculation

When the system workload is low, it is tempting to be able to create speculative tasks which may later become useful, and so contribute to the overall execution. Several major issues must be addressed before a speculative implementation can be produced.

- Either a priority or a fair scheduling algorithm must be implemented in order to ensure that program termination is not affected due to executing speculative tasks at the expense of mandatory ones.

- Speculative evaluation runs the risk of memory exhaustion, both directly through stack and other memory usage, and through creating results which are connected to mandatory tasks, but which are not yet required.

- It must be possible to upgrade a speculative task if a mandatory task needs its result, and to kill a speculative task if its result can no longer be reached.

An early scheme to manage speculative evaluation in a distributed system was suggested by Hudak [63]. Unfortunately, this scheme seems both complicated and costly, and has apparently never been implemented. Partridge has simulated a similar scheme for shared-memory machines. His results are promising for some applications, but remain to be verified on real machines [99].

Mattson has implemented a speculative scheme for the STG-machine on a 64-processor shared-memory Butterfly [85]. Speculative tasks create revertible "grey holes" (in analogy to the "black holes" created by mandatory tasks, which indicate that the node is under evaluation). If a speculative task demands the value of a grey hole node, it suspends until that value is computed. If a mandatory task demands the value of a grey hole, this becomes a black hole, and the speculative task is temporarily upgraded to mandatory status. Initial results were somewhat disappointing, but perhaps reflected problems with the implementation rather than the idea. Mattson and Griswold have also investigated local speculative evaluation on GRIP [86].

Several authors have proposed fairly simple systems, where speculative tasks are restricted by being given a limited amount of "fuel", but are otherwise treated as mandatory tasks [58, 126]. The advantages are that special schedulers are not required, and that it is relatively easy to kill such tasks. However, memory reclamation is still difficult. It remains to be seen whether, and how well, this strategy performs in practice.

Speculative evaluation is applicable to both strict and non-strict languages, since even strict languages have some non-strict constructs (function bodies or conditionals, for example). Because more parallel tasks are likely to be created in a strict environment, there may be fewer opportunities to exploit speculation in a strict language, however.

4.5 Memory Management

Initially, much effort was expended in devising complicated schemes to ensure that garbage collection could proceed in parallel, perhaps using some variant of weighted reference counting [18]. One of the most elaborate of these schemes, which is capable of detecting cycles in distributed graphs was devised by Lester [81]. There is some doubt that such schemes are practical for distributed memory architectures, however, communication costs seem generally too high for practical implementation.

The current practice in GRIP is to use two levels of garbage collection [102]. This allows frequent garbage collections locally to each processor, with infrequent collections of the entire program. More sophisticated schemes do not seem worthwhile.

Other garbage collection schemes are those devised by Watson and Watson [123], Augusteijn [8], Hudak and Keller [66] and Hughes [70].

5 Related Areas

5.1 Concurrency/Nondeterminism

Concurrent programs differ in kind from those mentioned earlier. The purpose of concurrency is to allow the construction of programs comprising multiple communicating processes, without particular regard to execution on parallel hardware. Communication is generally explicit, either through channels or shared variables, and deadlock is entirely possible. In order to eliminate the possibility of deadlock in some circumstances, a fair scheduler is needed. Applications where concurrency arises naturally include operating systems, user interfaces, and distributed systems such as bank auto-tellers. In these settings, concurrent processes are often written independently, and linked through predefined communication protocols or interfaces.

Generally, concurrent programs are nondeterministic in their execution order, and may also have nondeterministic results (exceptions are [62, 2] but note that these approaches can be used nondeterministically, and are only deterministic if systems are carefully designed). Many authors have considered how nondeterminism can be provided in functional languages without destroying their basic properties. One approach is based on the use of *oracles* which can be consulted to give a definitive value of an expression for one execution of a program [26]. Other approaches have been based on data structures such as sets which can be executed in parallel with one result chosen to represent the set [71], or bags with nondeterministic access functions [84]. ?[29].

5.2 Dataflow

Dataflow or single-assignment languages may have either strict or non-strict semantics. They are implemented by evaluating all the arguments to functions before evaluating the application. A non-strict semantics is obtained if all redexes are evaluated in parallel and no redex has its evaluation delayed indefinitely. For example, in Id [91] any redex not in the body of a conditional or lambda-abstraction will be reduced.

There has been much successful work on data-flow programming. Results obtained from the Sisal language [88], in particular, rival those for Parallel Fortran on numeric problems. Dataflow programs typically produce much fine-grained parallelism, which must be managed carefully to avoid memory exhaustion [34]. There have been attempts to increase granularity by combining instructions into larger basic blocks, but the result is still relatively fine-grained [114].

Dataflow Machines

The first successful dataflow machine was built at Manchester [48, 47]. This demonstrated good relative speedup for several small applications, and provided much basic data on task scheduling. The prototype machine with 12 Function Units ran dataflow programs at about 25% of the speed of a VAX 11/780 running C [47]. The substantially similar tagged token dataflow architecture (TTDA) designed at MIT [6] was never realised in hardware, but a much revised design was eventually built in the shape of Monsoon [98]. Another early dataflow machine was the Japanese SIGMA-1 [127], designed for scientific computations. Work on dataflow architectures has now ceased at Manchester, but is still being pursued at MIT in the P-RISC [7] and *T [93] designs, which are both derived from conventional RISC machines.

5.3 Term-Graph Rewriting

Term-graph rewriting [13] is a practical technique derived from theoretical term rewriting. Where term rewriting assumes (computationally expensive) syntactic equality, term-graph rewriting relies instead on identity through shared graph nodes. This is theoretically equivalent to working with labelled terms. Parallel computation is easily achieved by choosing a reduction strategy whose effect is to create a pool of possible redexes for parallel execution (this strategy is often predetermined, but is sometimes explicitly programmed). Term-graph rewrite systems are exemplified by approaches such as the DACTL parallel intermediate language [43, 51] or Lean [14]. The Clean language, which has roots in graph-rewriting but which is orthogonal and uses a fixed reduction strategy, is essentially a functional language.

The books by van Eekelen, Plasmeijer and Sleep [103, 117] are good sources for further reading on graph rewriting.

The Gamma model, whose intuition involves creating "chemical reactions" by nondeterministically matching "reagents" from a bag (or multiset) of computations [12] seems to have some aspects in common with term rewrite languages, as do functional-logic languages (e.g. []), which attempt to combine the higher-order nature of functional languages with the non-deterministic, first-order predicate logic nature of logic languages.

5.4 Impure Features

There is insufficient space here to rehearse the arguments over the precise nature of purely functional languages. However, it is clear that features such as assignment, general exceptions (as opposed to error values), and side-effecting I/O will have a detrimental effect on parallel execution. The techniques adopted for languages such as Lisp, Scheme and ML are thus much more closely related to those for imperative languages such as Fortran or C, than those discussed here. In particular, most implementations of these languages are concurrent rather than parallel, in the senses these terms are used here.

6 Future Directions

Parallel functional programming has come a long way since its inception. Absolute performance is still a major issue, but modern sequential implementations are starting to eliminate the gap with imperative languages and this can be exploited by derivative parallel implementations. For example, for the Pseudoknot benchmark, several functional language compilers can produce code whose performance is close to that obtained with the GNU C compiler for sequential machines [41].

This section discusses some other current trends and presents some challenges for future parallel functional language implementations which have not been raised in previous sections.

6.1 Architecture

Despite the relative success of novel machines such as GRIP, it seems likely that most future parallel functional implementations will exploit conventional hardware. It is easier (and cheaper) to buy an up-to-date general-purpose machine than to design and maintain a one-off prototype. It is also easier to share and compare results. The only recent exceptions to this trend are projects such as the *T dataflow machine which is supported by Motorola [93]. It is still worth researching the architectural implications of parallel functional programming, however, since this may influence mainstream architectural research, especially where the designs are not limited to functional languages.

At first sight, shared-memory implementations seem to present fewer implementation problems than distributed memory implementations, since locality is less of an issue. It is, however, increasingly necessary to exploit cache or local memory locality. Compared with their imperative cousins, functional languages have the advantage of significantly reducing cache coherency problems. It should be possible to exploit this to minimise the costs of using coherency hardware in a shared memory (or virtual shared-memory) machine. Bennett has performed some simulations which may prove useful here [16].

6.2 Implicit Parallelism

The early goal of cheap implicit parallelism still remains elusive for distributed memory machines. Experience suggests that if a program is written completely without thought of parallelism then it will almost certainly be difficult or even impossible to produce a parallel version without substantial algorithmic changes.

In spite of the best efforts of many good researchers, strictness analysis is still some way from being usable as a practical technique for parallelism detection in general non-strict languages. The problems are manifold: a really effective analyser needs to handle higher-order functions, polymorphism and arbitrary data structures. In addition, in order to cope with real programs, it must be fast enough that it can be used repeatedly, and it must cope with separate compilation. This is a tall order, though there are indications that progress is being made [116].

Even so, strictness analysis is just the start. Perfect strictness information is not sufficient by itself to achieve a perfect parallel partition. Since most strict expressions will be small and cheap to compute, information is also needed about both the size and cost of individual expressions (a *granularity* analysis [125]) if even a good dynamic system is to be able to make good scheduling decisions. A perfect cost analysis, as proposed by some authors, is not needed; a fast, reasonably accurate analysis is likely to be as effective. A good analysis will also need sharing analysis in order to correctly apportion costs to sub-expressions in a lazy environment. Granularity information can also be used to control the extent to which data structures are communicated or recomputed.

6.3 Distributed Systems

Throughout the world there are many loosely-coupled networks of relatively unused, relatively high-performance workstations. It is not a new idea to attempt to exploit these resources, but a functional language could be the ideal "glue" to knit together distributed machines. Concurrent Clean [95] and the Glasgow GRAPH for PVM [49] are examples of how this could develop.

6.4 Data Structures

Although some work has been done on distributing data structures such as arrays [79], this is still a far from solved problem. One big issue is that efficient sequential access to large data structures, for example using linear types or a state monad, usually involves imposing single-threaded access – a disaster in a parallel environment. Good data placement is also important for applications such as parallel databases [4].

6.5 Performance Prediction

Predicting the performance of even sequential functional programs is still something of a black art. Detailed performance results are the best way of seeing how a program has behaved in practice, but do not always allow future performance to be predicted.

Runciman and Wakeling's quasi-parallel performance "profiler" [109] simulates an idealised parallel execution on the basis of a sequential run. The information produced can be informative, but needs to be regarded with some circumspection. In its present form, the profiler fails to consider communication and task creation overheads or any sophisticated form of scheduling, and is incapable of dealing with speculative evaluation. It is also somewhat unrealistic in assuming that each supercombinator reduction is equally expensive, but this is a common problem with simulation.

6.6 Applications

Functional languages are surprisingly general. The primary applications base is, of course, symbolic, and good examples of such applications are the Lolita natural language recogniser, designed at Durham, or many of the parallel demonstrators produced as part of the FLARE project [110]. Dataflow languages such as Sisal show, however, that properly-designed languages can also succeed at fast parallel scientific computations.

One application area that is especially likely to repay further study is that of parallel functional databases. For some time, functional languages have been known to be good for constructing parallel read-only queries, as shown by, for example, the AGNA system [59], but the presence of implicit control dependencies managed by the underlying implementation rather than an explicit lock manager also makes them good for general transactions involving updates [4]. The PARADE project at Glasgow is actively working on these and other related issues.

7 Further Reading

Parallel functional programming is a broad area, which this paper has barely begun to cover. Roe's thesis contains a useful introduction up to 1991 with examples of different styles of parallelism [107]. Schreiner's annotated bibliography is highly useful, and relatively complete, with over 400 entries [115]. Ben-Dyke has edited a timeline for parallel functional programming, which was consulted heavily when writing this paper [15].

8 Acknowledgements

Many people have helped with this paper. I would particularly like to thank John O'Donnell, Andrew Partridge, Jim Mattson, Sigbjorn Finne, Hans Loidl and Phil Trinder for their careful comments on earlier versions of this paper. The photograph of the ALICE board was provided by Andrew Bennett of Imperial College.

References

[1] P. Achten. Annotations for Load Distribution. In *[42]*, pages 247–264, 1991.

[2] P. Achten and M. J. Plasmeijer. A Framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language Clean. Technical report, University of Nijmegen, The Netherlands, 1994.

[3] G. Aharoni, D. G. Feitelson, and A. Barak. A Run-time Algorithm for Managing the Granularity of Parallel Functional Programs. *Journal of Functional Programming*, 2(4):387–405, October 1992.

[4] G. Akerholt, K. Hammond, S. L. Peyton Jones, and P. W. Trinder. Processing Transactions on GRIP: a Parallel Graph Reducer. In *PARLE '93*, pages 634–647. Springer-Verlag LNCS 694, 1993.

[5] P. Anderson, C. L. Hankin, P. H. J. Kelly, P. E. Osmon, and M. J. Shute. COBWEB-2: Structured Specification of a Wafer Scale Supercomputer. In *PARLE '87*, pages 51–67. Springer-Verlag LNCS 258, 1987.

[6] Arvind, V. Kathail, and K. K. Pingali. A Dataflow Architecture with Tagged Tokens. Technical Report LCS Memo TM-174, MIT, 1980.

[7] Arvind and R. S. Nikhil. Can Dataflow Subsume von Neumann Computing? Technical Report CSG Memo 292, MIT, November 1988.

[8] L. Augusteijn. Garbage Collection in a Distributed Environment. In *PARLE '87*, pages 75–93. Springer Verlag LNCS 259, 1987.

[9] L. Augustsson. BWM: a Concrete Machine for Graph Reduction. In *Glasgow Workshop on Functional Programming*, pages 25–35. Springer-Verlag WICS, 1991.

[10] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$-Machine. In *FPCA '89*, pages 202–213, 1989.

[11] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Comm. ACM*, 21(8):613–641, August 1978.

[12] Jean-Pierre Banâtre and Daniel Le Métayer. Chemical Reaction as a Computational Model. In *[37]*, pages 103–117, 1989.

[13] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *PARLE '87*, pages 141–158. Springer Verlag LNCS 259, 1987.

[14] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Towards an Intermediate Language based on Graph Rewriting. In *PARLE '87*, pages 159–175. Springer-Verlag LNCS 259, 1987.

[15] A. D. Ben-Dyke. The History of Parallel Functional Programming. FTPable from ftp.cs.bham.ac.uk, August 1994.

[16] A. J. Bennett and P. H. J. Kelly. Locality and False Sharing in Coherent-Cache Parallel Graph Reduction. In *PARLE '93*, pages 329–340. Springer-Verlag LNCS 694, 1993.

[17] K. J. Berkling. Reduction Languages for Reduction Machines. In *2nd. Annual ACM Symp. on Comp. Arch.*, pages 133–140. ACM/IEEE 75CH0916-7C, 1975.

[18] D. I. Bevan. Distributed Garbage Collection using Reference Counting. In *PARLE '87*, pages 176–187. Springer Verlag LNCS 259, 1987.

[19] G. E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[20] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation

of a Portable Nested Data-Parallel Language. In *Principles and Practices of Parallel Programming*, pages 102–111, 1993.

[21] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *[104]*, pages 23–34.

[22] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[23] G. L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[24] Geoffrey L. Burn. Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (Extended Abstract). In *FPCA '87*, pages 446–470. Springer-Verlag LNCS 274, 1987.

[25] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope. Technical Report CSR-62-80, Edinburgh University, 1980.

[26] F. W. Burton. Nondeterminism with referential transparency in functional programming. In *First Intl. Lisp Conference*, 1980.

[27] F. W. Burton and V. J. Rayward-Smith. Worst Case Scheduling for Parallel Functional Programming. *To appear in J. of Functional Programming*, 1994.

[28] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, 1981.

[29] F.W. Burton. Encapsulating Nondeterminacy in an Abstract Data Type with Determinate Semantics. *J. of Functional Programming*, 1(1):3–20, January 1991.

[30] Warren Burton. Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs. *ACM TOPLAS*, 6(2), 1984.

[31] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[32] S. Cox, H. Glaser, and M. J. Reeve. Implementing Functional Languages on the Transputer. In *[37]*, pages 287–295, 1989.

[33] S. Cox, S.-Y. Huang, P. H. J. Kelly, J. Liu, and F. Taylor. Program Transformation for Static Process Networks. In *PARLE '92*, pages 497–512. Springer-Verlag LNCS 605, 1992.

[34] D. Culler and Arvind. Resource Requirements of Dataflow Programs. In *15th. Annual ACM Symp. on Comp. Arch.*, 1988.

[35] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming using Skeleton Functions. In *PARLE '93*, pages 146–160. Springer-Verlag LNCS 694, 1993.

[36] J. Darlington and M. J. Reeve. ALICE: A Multiple-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In *FPCA '81*, pages 65–76, 1981.

[37] M. K. Davis and R. J. M. Hughes, editors. *Glasgow Workshop on Functional Programming*. Springer-Verlag WICS, 1989.

[38] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures, Zurich, Switzerland*, pages 228–244. Springer-Verlag LNCS 782, March 1994.

[39] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus Efficiency in Parallel Systems. Technical report, Dept. of Computational Science, University of Sasketchewan, 1986.

[40] J. Darlington et al. An Introduction to the FLAGSHIP Programming Environment. In *CONPAR '88, Manchester*. Cambridge University Press, 1988.

[41] P. Hartel et al. Pseudoknot: a Float-Intensive Benchmark for Functional Compilers. *Submitted to J. of Functional Programming*, 1994.

[42] H. Glaser and P. Hartel, editors. *Proc. 3rd. Intl. Workshop on Parallel Impl. of Funct. Langs.* Technical Report CSTR 91-07, University of Southampton, 1991.

[43] J. R. W. Glauert, N. P. Holt, J. R. Kennaway, M. J. Reeve, M. R. Sleep, and I. Watson. Specification of Core DACTL1. Technical Report SYS-C87-09, UEA, 1987.

[44] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In *Workshop on Graph Reduction, Santa Fé, New Mexico*, pages 94–113. Springer-Verlag LNCS 279, September 1986.

[45] B. F. Goldberg. Multiprocessor Execution of Functional Programs. *Intl. Journal of Parallel Programming*, 17(5):425–473, 1988.

[46] R. G. Goldsmith, D. L. McBurney, and M. R. Sleep. Parallel Execution of Concurrent Clean on ZAPP. In *[117]*, chapter 21.

[47] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Comm. ACM*, 28(1):34–52, January 1985.

[48] J. R. Gurd, C.C. Kirkham, and J. R. W. Glauert. A Multilayered Data Flow Computer Architecture. Technical report, Manchester University, 1978.

[49] K. Hammond. Getting a GRIP. In *[104]*.

[50] K. Hammond. Efficient Type Inference Using Monads. In *Draft Proceedings, Glasgow FP Workshop, Ullapool, Scotland*, August 1990.

[51] K. Hammond. *Parallel SML: a Functional Language and its Implementation in DACTL*. Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[52] K. Hammond, J. S. Mattson Jr., and Peyton Jones S. L. Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer. In *CONPAR '94*. Springer-Verlag LNCS, September 1994.

[53] K. Hammond and J. T. O'Donnell, editors. *Glasgow Workshop on Functional Programming*. Springer-Verlag WICS, 1993.

[54] K. Hammond and S. L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *[80]*, 1992.

[55] Chris L. Hankin, P. E. Osmon, and M. J. Shute. COBWEB – a combinator reduction architecture. In *FPCA '85*, pages 99–112, September 1985.

[56] P. G. Harrison and M. J. Reeve. The Parallel Graph Reduction Machine, Alice. In *Workshop on Graph Reduction, Santa Fé, New Mexico*, pages 181–202. Springer-Verlag LNCS 279, September 1986.

[57] R. Harrison. Parallel Programming with Pure Functional Languages. In *Research Directions in High-Level Parallel Programming Languages*. Springer-Verlag LNCS 574, June 1991.

[58] C. T. Haynes and D. P. Friedman. Engines Build Process Abstractions. In *ACM Conf. on Lisp and Functional Programming*, 1984.

[59] M. L. Heytens and R. S. Nikhil. List Comprehensions in AGNA, a Parallel Persistent Object System. In *FPCA '91*. Springer-Verlag LNCS, 1991.

[60] J. M. D. Hill. The AIM is Laziness in a Data-Parallel Language. In *[53]*, pages 83–99.

[61] R. Hofman. *Scheduling and Grain Size Control*. PhD thesis, Universiteit van Amsterdam, 1994.

[62] I. Holyer and D. Carter. Deterministic Concurrency. In *[53]*, pages 113–126, 1993.

[63] P. Hudak. Distributed Task and Memory Management. In *ACM Symp. on Principles of Distributed Computing*, pages 277–289, August 1983.

[64] P. Hudak. Para-Functional Programming in Haskell. In Boleslaw K. B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196. ACM Press, 1991.

[65] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In *FPCA '85*, pages 382–399, September 1985.

[66] P. Hudak and R. M. Keller. Garbage Collection and Task Deletion in Distributed Applicative Systems. In *ACM Symp. on Lisp and Functional Programming*, pages 168–178, 1982.

[67] P. Hudak, S. L. Peyton Jones, and P. L. Wadler. Report on the Programming Language Haskell: a Non-Strict, Purely Functional Language. *Special Issue of SIGPLAN Notices*, 16(5), May 1992.

[68] P. Hudak and L. Smith. Para-functional Programming: A Paradigm for Programming Multiprocessor Systems. In *ACM POPL*, pages 243–254, January 1986.

[69] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford University, September 1984.

[70] R. J. M. Hughes. A Distributed Garbage Collection Algorithm. In *FPCA '85*, pages 256–272, September 1985.

[71] R. J. M. Hughes and J. T. O'Donnell. Expressing and Reasoning about Nondeterministic Functional Programs. In *[37]*, pages 308–328, 1989.

[72] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 55–69, Montreal, 1984.

[73] M.P. Jones and P. Hudak. Implicit and Explicit Programming in Haskell. Technical Report YALEU/DCS/RR-982, Dept. of Computer Science, Yale University, August 1993.

[74] R. M. Keller, F.C.H. Lin, and J. Tanaka. Rediflow multiprocessing. In *IEEE Compcon*, pages 410–417, February 1984.

[75] P. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[76] J. R. Kennaway. A Conflict Between Call-by-Need Computation and Parallelism. In *4th. Intl. Workshop on Conditional and Typed Term Rewriting Systems*, Jerusalem, July 1994. Springer-Verlag LNCS.

[77] J. M. Kewley and K. Glynn. Evaluation Annotations for Hope$^+$. In *[37]*, pages 329–337.

[78] H. Kingdon, D. R. Lester, and G. L. Burn. The HDG-Machine: a Highly Distributed Graph-Reducer for a Transputer Network. *The Computer Journal*, 34(4), 1991.

[79] H. Küchen. Applikative Datenstrukturen in der parallelen abstrakten Maschine PAM. In W. Dosch, editor, *Funktionale und Logische Programmierung — Sprachen, Methoden, Implementationen.* Report 214, University Augsburg, Germany, December 1989.

[80] H. Küchen and R. Loogen, editors. *Proc. 4th. Intl. Workshop on Parallel Impl. of Funct. Langs.* Technical Report 92-19, RWTH Aachen, 1992.

[81] D. R. Lester. Distributed Garbage Collection of Cyclic Structures. In *[53]*, pages 156–169.

[82] R. Loogen, H. Küchen, K. Indermark, and W. Damm. Distributed Implementation of Programmed Graph Reduction. In *PARLE '89*, pages 136–157, 1989.

[83] G. A. Magó and D. F. Stanat. The FFP Machine. In *High-Level Language Computer Architectures*, pages 430–468, 1989.

[84] G. Marino and G. Succi. Data Structures for Parallel Execution of Functional Languages. In *PARLE '89*, pages 346–356, 1989.

[85] J. S. Mattson Jr. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, University of California, San Diego, 1993.

[86] J. S. Mattson Jr. and W. G. Griswold. Local Speculative Evaluation for Distributed Graph Reduction. In *[53]*, pages 156–169.

[87] D. L. McBurney and M. R. Sleep. Transputer-Based Experiments with the ZAPP Architecture. In *PARLE '87*, pages 242–259. Springer-Verlag LNCS 258, 1987.

[88] J. McGraw. *SISAL: Streams and Iterations in a Single-Assignment Language: Reference Manual version 1.2.* Lawrence Livermore Natl. Lab., 1985. Manual M-146, Revision 1.

[89] R. Milikowski and W. G. Vree. The G-line: A Distributed Processor for Graph Reduction. In *PARLE '91*, pages 119–136. Springer-Verlag LNCS 505, 1991.

[90] R. Mohr, D. A. Kranz, and R. H. Halstead. Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[91] R. S. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, Lab. for Computer Science, MIT, July 1991.

[92] R. S. Nikhil, Arvind, and J. Hicks. pH Language Proposal (Preliminary), 1st. September 1993. Electronic communication on comp.lang.functional.

[93] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th. ACM Annual Symp. on Comp. Arch.*, pages 156–167, 1992.

[94] E. G. J. M. H. Nöcker, M. J. Plasmeijer, and S. Smetsers. The Parallel ABC-machine. In *[42]*, pages 351–382, 1991.

[95] E. G. J. M. H. Nöcker, S. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent clean. In *PARLE '91*, pages 202–219. Springer-Verlag LNCS 505.

[96] J. T. O'Donnell. Bidirectional Fold and Scan. In *[53]*, pages 193–200.

[97] G. A. Ostheimer. *Parallel Functional Programming for Message-Passing Multiprocessors.* PhD thesis, University of St. Andrews, 1993.

[98] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Laboratory for Computer Science, MIT, August 1988.

[99] A. S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages.* PhD thesis, University of Tasmania, 1991.

[100] N. Perry. Hope$^+$. Technical Report IC/FPR/LANG/2.5.1/7 Issue 5, Imperial College, London, February 1988.

[101] S. L. Peyton Jones, C. Clack, and J. Salkid. High-Performance Parallel Graph Reduction. In *PARLE '89*, pages 193–206, Eindhoven, The Netherlands, June 12–16, 1989. Springer-Verlag LNCS 365.

[102] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – a High-Performance Architecture for Parallel Graph Reduction. In *FPCA '87*, pages 98–112. Springer-Verlag LNCS 274, 1987.

[103] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[104] M. J. Plasmeijer and M. C. J. D. van Eekelen, editors. *Proc. 5th. Intl. Workshop on Parallel Impl. of Funct. Langs.* Nijmegen, 1993.

[105] F. A. Rabhi. Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented" Approach. In T. Lake and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press, 1993.

[106] H. Richards. An Overview of Burroughs NORMA. Technical report, Austin Research Centre, Burroughs Corp., January 1985.

[107] P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Glasgow University, April 1991.

[108] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Machine. In *FPCA '87*, pages 1–15. Springer-Verlag LNCS 274, 1987.

[109] C. Runciman and D. Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *[53]*, pages 235–248.

[110] C. Runciman and D. Wakeling, editors. *Functional Languages Applied to Realistic Examplars: the FLARE Project*. UCL Press, 1994.

[111] J. Sargeant. Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines. Technical Report UMCS-86-11-5, Manchester University, 1987.

[112] J. Sargeant and I. Watson. Some Experiments in Controlling the Dynamic Behaviour of Parallel functional programs. In *[42]*, pages 103–121, 1991.

[113] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[114] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *ACM Symp. on Lisp and Functional Programming*, pages 202–211, 1986.

[115] W. Schreiner. Parallel Functional Programming — an Annotated Bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993.

[116] J. Seward. *Abstract Interpretation: a Quantitative Assesment*. PhD thesis, Manchester University, 1994.

[117] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice*. Wiley, 1993.

[118] G. L. Steel Jr. and W. D. Hillis. Connection-Machine Lisp. In *ACM Symp. on Lisp and Functional Programming*, pages 279–297, 1986.

[119] W. R. Stoye. *The Implementation of Functional Languages using Custom Hardware*. PhD thesis, University of Cambridge, 1985.

[120] P. L. Wadler. The Essence of Functional Programming. In *ACM POPL '92, Santa Fé, New Mexico*, January 1992.

[121] I. Watson. Simulation of a Physical EDS Machine Architecture. Technical report, Department of Computer Science, University of Manchester, UK, September 1989.

[122] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: a Parallel Architecture for Declarative Programming. In *15th. Annual ACM Symp. on Comp. Arch.*, page 124, 1988.

[123] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *PARLE '87*, pages 432–443. Springer Verlag LNCS 259, 1987.

[124] P. Watson and I. Watson. Evaluating Functional Programs on the FLAGSHIP Machine. In *FPCA '87*, pages 80–97. Springer-Verlag LNCS 274, September 1987.

[125] K. G. Waugh. Parallel Imperative Programs from Functional Prototypes. In *[42]*, pages 75–88, 1991.

[126] W. F. Wong and C. K. Yuen. A Model of Speculative Parallelism. *Parallel Processing Letters*, 2(3):265–272, 1992.

[127] T. Yuba, T. Shimada, K. Hiraki, and H. Kashiwagi. SIGMA-1: A Dataflow Computer for Scientific Computations. *Computer Physics Communications*, pages 141–148, 1985.