High-performance parallel graph reduction

Simon L Peyton Jones, Chris Clack and Jon Salkild
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, United Kingdom
email: clack@uk.ac.ucl.cs, simonpj@uk.ac.ucl.cs

Abstract

Parallel graph reduction is an attractive implementation for functional programming languages because of its simplicity and inherently distributed nature. This paper outlines some of the issues raised by parallel compiled graph reduction, and presents the approach we have adopted for our parallel machine, GRIP.

We concentrate on two main areas:

- Static and dynamic techniques to control the growth of parallelism, so as to provide enough parallelism of an appropriate granularity to keep the machine busy without swamping it.
- Dynamic techniques to exploit the memory hierarchy, so that frequently-referenced data is held near to the processor that references it.

1. Introduction

Graph reduction is an attractively simple foundation for the execution of functional programs on parallel hardware[Peyt87a]. In this paper we raise some of the key issues involved in the implementation of a parallel graph reduction machine, with particular emphasis on performance. Based on our experience with the GRIP project, we also describe our current design decisions.

There are substantial communication and administration overheads to be paid for parallel execution. The main problems are

- The overheads of creating a new task. This involves not only the creation of a task descriptor, but also the insidious cost of packaging the work so that it can be communicated to an independent agent.
- The synchronisation costs involved when tasks communicate with each other.
- The danger of memory overflow due to the unrestricted growth of partly-completed tasks.
- Most serious of all, the loss of locality caused when one processor executes a task whose data is local to a different processor.

These issues raise two broad design questions, which are echoed by all parallel graph reduction systems of which we are aware:

- When should a new task be created? Enough parallelism is required to keep the machine busy, but excessive parallelism only increases the overheads without providing any new opportunities for increased performance.

"High-Performance Parallel Graph Reduction" by S.L. Peyton Jones, C. Clack, and J. Salkild from *Lecture Notes in Computer Science 365—Proc. Parallel Architectures and Languages Europe (PARLE '89)*, 1989, pp. 193-206.

- How can locality be achieved? We want to make effective use of the memory hierachy provided by the underlying architecture, which invariably provides a latency/size spectrum in which fast memory is provides only limited storage, and bulk storage is slow.

The paper comprises three parts. We begin with an overview of the GRIP systems architecture, to provide a concrete basis for the rest of the paper. This is followed by two main sections which address the design questions raised above.

The ideas we discuss represent the current state of our thinking, but we have not yet completed the compiler and system software based on these ideas. As a result, we make many conjectures, but are unable to support them with hard performance measurements, something which will be corrected in future papers.

2. Parallel compiled graph reduction on GRIP

In this section we present the mechanisms that enable parallel compiled graph reduction to be performed on GRIP. We discuss both hardware and software system architecture and then introduce the key features of the virtual machine. Further details can be found elsewhere[Peyt87b, Peyt88a]. In order to provide a specific context within which to discuss the issues of parallel graph reduction, we begin with a brief overview of the system architecture of our parallel graph reduction machine, GRIP.

2.1 Overview of GRIP system architecture

2.1.1 Hardware architecture

The GRIP hardware consists of:

- up to eighty M68020 Processing Elements (PEs), each with one megabyte of private memory.
- up to twenty microprogrammable Intelligent Memory Units (IMUs), each with 5 megabytes of globally-addressable memory (upgradable to 20 megabytes when 4Mbit RAMS are available);
- a high-bandwidth packet-switched bus which interconnects the components.

GRIP provides a shared-memory architecture in which the globally-addressable heap is held in the IMUs. The IMUs support a range of memory operations which manipulate heap-structured data, thus localising all the low-level synchronisation and heap-management operations. They may alternatively be re-programmed, for example to support the memory operations required for a parallel logic language[Reyn88a]. This shared-memory approach allows us to concentrate initially on the challenges of parallel graph reduction, whereas a distributed-memory architecture would force us first to solve the problem of achieving a high degree of locality.

The following range of operations is supported by our current IMU 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[Bake78a].
- 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 task descriptor 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.

2.1.2 System software

GRIP spends most of its resources performing graph reduction, but there are also several administrative activities such as program loading, input and output and garbage collection. In particular, there is exactly one system manager (which can reside on any PE) which is responsible for global resource-management policy decisions.

We could assign a separate PE to each of these activities, but many of them have rather a low duty-cycle so that the PE would be idle much of the time. This would be particularly serious in a small GRIP with only a few PEs. Each processor therefore runs a small special-purpose multi-tasking operating system called GLOS, which allows a single processor to be multiplexed between graph reduction and administrative activities.

2.2 Compiled graph reduction

Our first model for parallel reduction was an *interpretive* one called the **four-stroke reduction engine**[Clac86a], and we now have a running parallel implementation of this machine.

In the last few years, much progress has been made on *compiled* graph-reduction techniques for sequential processors, so that the functional program is compiled to native machine code, with substantial performance improvements over interpretive methods[Peyt87a]. Generally speaking, these efforts have been reported in the form of the design of an abstract machine for executing functional programs; examples include the G-machine[John87a], Tim[Fair87a], the Oregon G-machine chip[Kieb87a], the Spineless G-machine[Burn88a], and the Spineless Tagless G-machine[1][Peyt89a]. The best of these implementations produce programs which run at speeds broadly competitive with compiled imperative languages[2].

The time is now ripe to extend this compiler technology to parallel machines. We are currently in the midst of extending the Spineless Tagless G-machine in this way, and the ideas in this paper have grown out of this work. The details of this abstract machine are beyond the scope of this paper, but for our present purposes the following are the key features:

- The functional program is lambda-lifted as usual, and then compiled to native machine code.
- Evaluation takes place with the aid of a stack to hold intermediate results.

2.3 Tasks and concurrency - a model for parallel graph reduction

The functional program is held as a graph in the heap. Execution proceeds by **reductions**, which transform the graph to a simpler form. Each reduction physically updates the graph node representing the reducible expression (or redex) with the result of the reduction. Reductions may take place concurrently at different sites in the graph, and this is the source of parallelism in GRIP.

A task is a sequential computation which executes a series of reductions whose purpose is to reduce some sub-graph to (weak head) normal form. Tasks are the finest schedulable unit of concurrency in GRIP, and the set of executable tasks is called the task pool. Each IMU holds part of the task pool, and idle PEs poll the IMUs to

1. Of course, graph reduction is not the only possible implementation technique for functional languages; others models include the Imperial FPM system FP field and the Categorical Abstract Machine[Cous85a]. Graph reduction lends itself particularly easily to parallel implementations.
2. We argue elsewhere that it is unreasonable to expect functional-language programs to run as fast as their imperative programs, but that this will become increasingly unimportant, provided that the performance loss is sufficiently small[Peyt89b].

request an executable task from the pool.

During its execution, a task may encounter a sub-graph whose value will be required in the future. If so, the task creates a child task, by placing a pointer to the sub-graph in the task pool: this is called **sparking a child**. 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 to evaluate E_2 (whose value will certainly be required in the future).

Subsequently the parent task will require the value of the sub-graph it sparked; in our example, after completing the evaluation of E_1 the parent will require the value of E_2. In GRIP *the parent simply tries to evaluate E_2 just as if it had never sparked the child.* There are then three possibilities:

(i) The child task has not started execution, because no PEs were free. In this case the parent evaluates E_2 itself, and the child task becomes an **orphan**.

(ii) The child task has completed evaluation of E_2, and overwritten the root node of E_2 with its value. The parent will therefore find that evaluating E_2 is rather easy, because it is already in normal form! The parent's attempt to evaluate E_2 then degenerates to a fetch of the evaluated result.

(iii) The child task has begun to evaluate E_2, but has not completed. In this case the parent is **suspended** until the child does complete its evaluation. Then the parent is **resumed**, and the situation is just like that of case (ii).

The suspension mechanism in case (iii) above also guarantees mutual exclusion between any two concurrent tasks which share a common subexpression. The second task to reach the subexpression will be suspended until the first has completed its evaluation: the second task is then resumed.

3. The efficient generation and control of parallelism

We are bound to pay some overhead penalty in exchange for parallel execution. We believe, however, that an unacceptable performance penalty will be paid if every opportunity for parallel execution is taken, irrespective of whether there is spare processing capacity to exploit it. In other words, we believe that the key to high performance is *to pay the overheads of parallel execution only when there is spare capacity available to exploit it, and otherwise to execute the program sequentially using all the short-cuts and optimisations that thereby become available.*

One way to achieve this goal is to require the programmer to take very close control over the way in which his or her program is partitioned into parallel activities. This is what is required by an array processor, for example. Unfortunately, for complex programs, with irregular structure, figuring out how best to partition the program can become extremely hard, and we believe that one of the prime merits of parallel functional programming is exactly that it frees the programmer from being forced to specify this level of detail.

Accordingly, we are interested in developing compile-time and run-time strategies for partitioning and scheduling the program. We do not expect these strategies to outperform a program explicitly scheduled by a programmer, except perhaps for very complex programs. Instead, our goal is to do a sufficiently good job for practical purposes. This section discusses some possible strategies.

3.1 Speculative and conservative parallelism

The first thing to consider is whether or not the result of a child task will actually be required. In our present implementation of GRIP we make the simplifying assumption that *a task is only sparked if it is certain that its result will be required*. This is called **conservative parallelism**, in contrast to **speculative parallelism**, where a task may be sparked if it is likely that its result will be required. Speculative parallelism is a sort of job-creation scheme to usefully exploit idle processors. For example, consider the expression

if E_1 then E_2 else E_3

While E_1 is being evaluated, we could imagine speculatively starting parallel evaluation of E_2 and E_3, so that some progress had already been made by the time the evaluation of E_1 was complete.

It is well known that speculative parallelism raises many problems, which we summarise briefly as follows[Huda83a]:

- Speculative tasks compete for resources with vital tasks. In the example, there is danger that the machine will spend all its resources evaluating E_2 and E_3, and never make progress with E_1. Hence, some sort of priority scheme is required.

- Speculative tasks may become vital. For example, if E_1 evaluates to True, the evaluation of E_2 becomes vital, and should have its priority upgraded (and so should its children, and their children...).

- Speculative tasks may become useless. If E_1 evaluates to True, the evaluation of E_3 is positively pernicious, since it is consuming machine resources to compute a result which is now known not to be required. Hence the task evaluating E_3 must be killed (and its children, and their children...).

- The situation becomes more murky still when we realise that speculative and vital tasks may share sub-expressions.

The unrestrained use of speculation therefore risks causing a lot of extra administration in return for modest increases in parallelism, so GRIP only initiates conservative parallelism at present. For the future, there are situations where speculation seems inevitable[Burt85a]:

- Indeterministic choice, where the programmer simply wants to select the result of whichever of two or more computations completes first. Some of the computations may not terminate at all, so considerations of fairness are involved here as well.

- Program aborts, where the programmer wants to abort a computation using "Control-C", without halting the entire machine.

3.2 Achieving dynamic granularity

Once we realise the costs imposed by parallelism, it becomes clear that we should strive to generate only enough parallel activities to keep the machine busy, which will be some (small) constant multiple of the number of processors in the machine. Our goal is to generate parallel tasks until the machine becomes "sufficiently" heavily loaded, and then to run each task sequentially in the local memory of its processor, communicating with the rest of the world only when necessary. When the load drops, new tasks should be generated again. Thus, as the machine becomes loaded, each task runs longer sequential threads of execution, thereby dynamically increasing the grain size.

The parent-child synchronisation mechanism in GRIP makes this particularly easy to achieve. Recall that a child task *does not notify the parent of its completion*. This is in contrast to ALICE, FLAGSHIP, and Alfalfa, all of which require the parent task to wait for notification from every child. This property has a number of important consequences:

- **The length of threads of execution are maximised.** The parent is only suspended if the child has begun evaluation but not completed. In all other cases, the parent continues uninterrupted. In other words, *the costs of synchronisation are only paid if a collision actually takes place.* In effect, the grain of execution is increased dynamically.

- **A task should never spark an expression it is about to evaluate.** In the expression (E_1 + E_2), it would be possible for the parent to spark two children, for E_1 and E_2. However, the next action of the parent is to

evaluate E_1 itself anyhow! Hence it is better for the parent to spark E_2 and to evaluate E_1 itself.

- **Orphan tasks can be discarded.** A task in the task pool is held in the IMU that contains the root node of the task. When an IMU fetches a newly-sparked task from the task pool, in response to a request from an idle PE, it may first check to see if the graph specified by the task is already being, or has been, evaluated. If so, the task is an orphan, and can simply be discarded.

- **The system is free to discard sparks at will.** This is possible because of the absence of explicit notification. It may be desirable if the machine is heavily loaded. We think of sparks as advisory messages to the system, giving advance warning that a sub-graph will later be evaluated, and thus giving the opportunity to evaluate it in parallel. Discarding sparks has two beneficial effects for a heavily-loaded system:

  (i) locality cannot be lost by the task migrating to another processor;
  (ii) a parent-child collision cannot take place, thereby reducing synchronisation overheads.

We regard this policy, of discarding sparks when the machine is sufficiently loaded, as extremely important. It allows us to run dynamically-sized tasks with efficiency approaching that of high-quality compiled sequential implementations.

Of course the notion of "sufficiently loaded" is hard to quantify, but fortunately great precision is not required. In the GRIP system, the System Manager computes a slowly-varying load average and distributes it regularly to each processor. (In a more distributed system, the load average could be computed using only local information.) Each processor then decides whether or not to spark new tasks, by comparing this load average with some threshold value. The value of this threshold, and the frequency of load average collection and distribution, is a matter for experiment.

There is a risk of losing concurrency by this method. Consider a function which sparks several large arguments, and then evaluates them sequentially, and suppose that these sparks were discarded because the machine was busy at the time. Now, if machine becomes un-loaded midway through the execution of the function, it would be committed to sequential evaluation of the remaining arguments, despite the abundance of idle processors!

We believe that the risk of loss of concurrency is far outweighed by the benefits of dynamic granularity, and but this remains to be demonstrated.

3.3 The effect of scheduling policy

The scheduling policy governs which processor executes which executable task. There are two main considerations: *when* a task should be scheduled, which affects memory usage; and *where* it should be executed, which affects locality. We discuss the former immediately, and postpone discussion of the latter until a later section.

In a conservative parallel regime, the order in which tasks are scheduled is immaterial, since all tasks are doing useful work. Nevertheless, scheduling policy can have a substantial effect on the rate of growth of parallelism and storage usage, as the Manchester dataflow group have discovered[Rugg87a]. Far from lacking parallelism, they encountered serious problems because their machine's memory rapidly filled up with partly-executed tasks. This led them to suggest two possible policies for scheduling sparked tasks: LIFO (last-sparked-first-scheduled) and FIFO (first-sparked-first-scheduled). This decision has a dramatic effect on the rate of growth of parallelism in divide-and-conquer algorithms, and hence on the storage used to hold partly-executed tasks.

In effect, LIFO scheduling explores the process tree in a depth-first manner, and FIFO scheduling in a breadth-first manner. The former minimises storage usage while the latter maximises parallelism. Switching dynamically between LIFO and FIFO scheduling allows this tradeoff to be adjusted at run time.

Similar effects have been observed in the Concert Multilisp system[Hals86a] in which depth-first execution is pursued

by each individual processor, with breadth-first execution arising when processors steal tasks from each other's task stacks.

The IMUs can easily implement such scheduling policies, provided the System Manager informs them regularly of which policy to pursue. We also have the additional freedom to schedule resumed tasks before or after sparked tasks, which should have a similar effect.

If we omit considerations of the amount of intermediate storage used, how much effect can scheduling policy have on the overall speedup? At first it might appear that poor scheduling policies may give very bad processor utilisation and seriously limit speedup. However, Eager, Zahorjan and Lazowska [Eage86a] show that for any work-conserving[3] scheduling policy the speedup for N processors must be larger than NA/(N+A+1), where A is the average level of parallelism which would be attained by executing the same program with an unbounded number of processors. If A>>N, then this lower bound approaches N, which as good as we can hope for. This reassures us that scheduling policy is comparatively unimportant to overall speedup for highly-parallel problems, a result which is strongly supported by Goldberg's experiments[Gold88a].

3.4 Sequential and parallel versions

The loss of the efficiency of sequential code, coupled with the overhead of building closures, represents another source of inefficiency. Suppose we have to compile code for the function g, where

$$g\ x_1 \ldots x_m = f\ E_1$$

and E_1 is some arbitrary expression. A sequential G-machine would build a closure (or graph) for E_1 and pass this to f (this is call-by-need). Now suppose that f is known to evaluate its argument; one of the most important optimisations of the G-machine is now to evaluate E_1 in-line before calling f (this is call-by-value, which is substantially more efficient). In the G-machine jargon, we can use the E compilation scheme for E_1, instead of the C scheme, because f is strict[4].

To generate maximum parallelism in a *parallel* machine the code for g should build a closure for E_1 in the heap, spark it, and then enter the code for f (eager evaluation). However, if the machine is busy then call-by-value would have been better, because the opportunity for parallel evaluation of E_1 cannot be exploited.

In general, the opportunities for in-line evaluation are *exactly those* where parallelism demands eager evaluation (namely, where the function is known to evaluate its argument). The tension between these two evaluation mechanisms is vitally important to the performance of the machine - by building closures we risk losing the benefits of one of the most important G-machine optimisations, and yet in-line evaluation may lose opportunities for parallelism.

This hidden cost, of building closures rather than evaluating them in-line, suggests that there should be two versions of g, one of which has in-line evaluation of E_1, and one of which generates a closure for E_1 and sparks it (the sequential version and parallel version respectively). The version of g which is used at run-time can be chosen dynamically, depending as before on the load on the machine. While there are idle processors, we choose

3. A work-conserving scheduling discipline is one that never leaves idle a task that is eligible for execution when there is a processor available.

4. This approach to parallelism differs from our earlier work[Clac86a], in which the application of g to its arguments was rewritten to the graph of (f E_1), where the application node had a special run-time annotation to indicate a strict application. Then, evaluation of (f E_1) was begun, and the evaluation mechanism (the four-stroke engine) sparked E_1 when it found the annotation on the application node.

   If a closure must be built and sparked, it is far better for the code for g to do this sparking in-line. In effect, the annotation is now embedded in the code for g.

function versions which generate lots of parallelism; then when all the processors are "busy enough", we switch to the (more efficient) sequential versions of the functions.

Generating multiple versions of the code for a function clearly involves a significant expansion in the size of the code for the program. How much of a problem this will be depends very much on the particular architecture, but we observe that the size of the heaps required to run typical functional programs substantially exceeds the size of the code so, to first order, code size is not a problem.

4. How to achieve locality

The memory of any scalable parallel machine is arranged in a hierarchy in which increasing size carries the cost of increasing latency. For high performance it is essential that a large fraction of memory references are to local memory; that is, a high degree of locality is essential.

There are two main techniques that can be used to increase locality in a graph-reduction machine, where the graph is spread through the machine's memory:

- Wherever possible, ensure that a processor is working on local data.
- Use caching techniques to keep local copies of remote graph nodes.

There is one trivial way to ensure that each processor is always working on local data, namely to run the whole program sequentially on a single processor; this observation shows up the fundamental tension between parallelism and locality. It follows that the granularity-increasing techniques of the previous section, which suppress unnecessary parallelism, will have the effect of increasing locality of reference as well; indeed, this is the main motivation for increasing granularity. The section begins with a brief discussion of scheduling techniques to improve locality.

The rest of the section focuses on the second technique for increasing locality, namely caching. We discuss a number of ideas we are implementing in the GRIP machine, which have quite general applicability.

4.1 The effect of scheduling policy on locality

In a parallel machine with distributed memory, the question of where a task is executed is of crucial importance to locality[5].

There are some simple scheduling expedients which may increase locality. For example:

- We can ensure that tasks allocate new graph in their local memory, and flush it to their local IMU, so a task's locus of activity should gradually become more and more local.
- We can attempt to resume a blocked task on the same PE which was executing it before it became blocked, because that PE probably has much of the task's data in its cache (see the discussion below on caches).

In a more general setting, experiments by Goldberg[Gold88a] and Eager *et al*[Eage86b] provide a good starting point for further study. Happily, both conclude that simple scheduling policies work nearly as well as more complex ones.

5. It is not as important on GRIP because of the bus-based architecture. We deliberately chose this organisation precisely because it allowed us to achieve good performance without solving this difficult problem!

All these ideas concern run-time scheduling heuristics. Unfortunately, efficient algorithms for distributed-memory multiprocessors often rely on distributing the *key data structures* for the problem across the memory of the machine, which gives rise to a natural distribution of the *tasks*. This requires compile-time pre-planning of data allocation policies, which is certainly a hard problem.

Currently, we make no attempt to perform this sort of planning. Instead, we rely on the caching strategies outlined below to migrate the relevant data into the processor(s) which are accessing it, regardless of where the data was first allocated. It would be interesting to study whether the benefits of more sophisticated compile-time planning justify the costs of performing it.

4.2 Caching in a graph-reduction machine

Caching is a well-known technique for increasing locality by keeping copies of recently-referenced data, so that the copy is rapidly available if the same data is referenced again. In a multiprocessor system with multiple caches, it is essential to maintain *coherence* between the caches, so that all caches contain up-to-date copies of their data, which is awkward if unrestricted writes are allowed. This problem has been effectively solved for bus-based multiprocessors, using "bus-watching" techniques, but appears much more intractable for non-bus-based multiprocessors.

However, it turns out that *cache coherence is not a problem in a graph-reduction system*, because arbitrary writes are not permitted. The natural unit of caching is a single graph node, which may or may not be in normal form:

- When a graph node is not in normal form, only one task will be allowed to access it, so the processor running that task can cache it freely. Other tasks attempting to access the node will be blocked until it has been updated with its normal form.

- When a graph node is in normal form, it can never change any further, so it can be freely cached by any processor that wishes to do so.

In other words, the same mechanism that deals with synchronisation between tasks also ensures that all accessible nodes are cacheable with no loss of coherence. This is a significant benefit: to our knowledge, no viable automatic cache-coherence scheme for an arbitrary multiprocessor has even been proposed, let alone implemented.

Functional languages are often praised for their clean semantics, deriving from the absence of side effects. It is rather pleasant to discover the same property leading to a significant architectural benefit, with important consequences for performance.

4.3 Exploiting a two-level store

The GRIP hardware provides a two-level address space: the fast, private memory in the PEs (the local address space) providing a simple read/write interface and the slower, larger, shared memory in the IMUs (the global address space) providing a more sophisticated interface. The former is an order of magnitude faster than the latter, and the two are addressed in different ways. This is unlike the flat address-space provided by commercially-available bus-based multiprocessors, and by the FLAGSHIP machine. In these machines, each PE's local memory forms a part of a single global address space.

The IMU address space also has two levels. Each GRIP board contains four PEs and one IMU, so that access to the on-board IMU has a somewhat lower latency than access to an off-board IMU, though the protocol is identical.

A flat address space is certainly an easier model to use from a programming point of view, but we believe that a two-level address space has important advantages which make it worth careful consideration. We make some preliminary observations.

- There is a clear analogy with the registers of a conventional CPU, which form a separate address space from the main memory, and whose effective exploitation is crucial to high performance. Indeed, the recent trend is to increase the size of the register set, rather than to eliminate it in favour of a flat address space.

- The use of a flat one-level address space requires an effective caching system *implemented in hardware*, since every memory access is made to this address space. Unfortunately, the usual cache-management hardware structures are probably inappropriate, because of the lack of spatial locality, the need to set lock bits when fetching non-local graph nodes, and the variable size of graph nodes. It is particularly inappropriate for the GRIP system, because of our use of intelligent memories: conventional cache technology depends on a simple read/write interaction of the processor with memory, whereas we use a more sophisticated processor-memory interaction protocol.

- Graph nodes in the PE's private address space need less administrative information attached to them, because they cannot be accessed by other tasks concurrently. Hence, manipulation of local data is likely to be faster than manipulation global data, even discounting the effects of latency.

- As we discuss below, it is possible for a PE to garbage-collect its local store independently of the rest of the system. This is an important benefit which is not easily available in a flat address space.

Just as compilers strive to use the registers effectively in a CPU, so we use the local PE memory as a large, explicitly-managed cache. Local memory is used, of course, for system software residing in the processor, and we devote all the remaining local memory to implement a **local heap**, which acts as a cache for the global heap.

A task running on a particular processor allocates new graph nodes in the processor's local heap. The new nodes are not immediately written out to the global heap. Indeed, they may never be written out, because each local heap has the important property that *it can be garbage-collected independently of the rest of the system*. Thus a node may be allocated, used, and garbage-collected locally without ever migrating into the global heap. (This can be thought of as a sophisticated form of write-back cache: data is not written back into main memory if it is known to be garbage.)

When a task needs to access a non-local graph node, it fetches it and creates a local copy in its heap, including with the local copy a pointer back to the original non-local node.

Whenever a task updates a local node with its normal form, it must also check whether it is a local copy of a global node; if so, it must also write the normal form out to the global node, in case there are tasks blocked on it. Of course, if the local node has no global counterpart, no further action need be taken.

When the local heap becomes too full, local copies of global nodes can be discarded freely; they will be reloaded again if they are required. (This corresponds to flushing a datum out of a cache.) If it is still too full, local nodes without global counterparts can be flushed out of the cache by allocating them as new nodes in the global heap.

All of this results in rather a fine grain of non-local access. For example, if a task is iterating down a non-local list, each list node would be fetched individually. This problem affects all caches, and the standard solution is to fetch rather more data than is actually required, in the hope that the extra data will subsequently prove useful. Conventional caches normally prefetch data which is *physically adjacent* to the data actually required, but in our situation it would clearly be better to prefetch data *pointed to* by the required node. In this way, larger units of data can be fetched from non-local memory. (How much extra data should be prefetched will certainly be specific to the particular architecture.)

There is an important complication: it is dangerous to prefetch unevaluated objects. When a node is fetched into a processor's local memory, the global copy must be locked to prevent other tasks from attempting to evauate it. The danger is that if the node is prefetched, it will thereby be locked even though the prefetching processor may never evaluate it. Other tasks may then block indefinitely on the node.

4.4 Local heap management

In order for the local heap to be independently garbage-collectable, it is essential for the processor to have knowledge of all the pointers into the local heap. This raises an interesting design issue: should it be possible for pointers into a particular processor's local heap to exist elsewhere in the system? For example, should it be possible for a global node in an IMU to point to a local node in some processor's local heap?

If this is allowed, then the processor has to maintain an "entry table", which contains an entry for each such inbound pointer. Then these entries can be used as starting-points for garbage collection, and the indirection they provide allows the garbage collector to move nodes around within the local heap. Furthermore, the processor has to be prepared to service remote requests from other processors, and to implement the complete task synchronisation mechanism on local nodes.

We have elected to take the alternative view: *no pointers can exist from outside a processor into its local heap*. This obviates the need for an entry table, and allows the processor to concentrate on graph reduction without concern about being unavailable to service remote fetch requests. Furthermore, a local node can be accessed without fear that a remote fetch is simultaneously accessing it. Nevertheless, our approach carries its own costs.

For example, a task sparks a child task by placing a pointer to the subgraph representing the child in the task pool held in an IMU. If there are to be no pointers into the processor's local memory, the entire subgraph representing the task must be flushed into global memory. (Here is another strong reason to avoid unnecessary sparks!)

4.5 Global heap management

The IMUs support an instruction set which includes allocation of variable-sized graph nodes, and synchronised access to these (cf Section 2.1.1). Garbage collection is performed using a variant of Baker's copying collector. There are three phases:

- First there is a global synchronisation, in which all PEs and IMUs agree to start garbage collection.
- All the processors perform a local garbage collection and tell the IMUs about each pointer into the global heap which they hold. The IMUs move the indicated node from From-space into To-space, and respond with new location of the node. This corresponds to "copying the roots" in a normal Baker collector.
- Now the processors can revert to graph reduction, while the IMUs concurrently scavenge To-space in the usual manner, communicating directly with each other when they encounter inter-IMU references. The usual Baker real-time method can be used to ensure that processors only have To-space pointers.

Thus, the vast bulk of global garbage collection is performed autonomously by the IMUs, which are highly optimised for just this kind of pointer-manipulation.

4.6 Stacks, blocking and resumption

Two sorts of graph node deserve special attention, namely stack segments, and function code blocks, which we discuss in this section and the next.

As mentioned earlier, each task uses an evaluation stack in a similar way to conventional compiled programs. When a task becomes blocked, this stack forms part of its state which must be preserved so that it can be resumed later.

We implement the stack by allocating a fixed-size local graph node, called a stack segment, to use as stack space for the task. If the stack overflows this node, we allocate a new stack segment, copy up the top section of the old stack into the new one, and place a link in the new stack segment back to the old one. When the stack shrinks again, we discard the new segment and revert to the old one.

Stacks are thereby regarded as perfectly ordinary graph nodes, and can be flushed out into global memory like any other node. (Of course, we don't actually flush the unused words in a stack segment.) When a task is blocked, its stack is tidied up, flushed into global memory, and a pointer to the topmost segment is attached to the blocking node. When the blocking node is finally updated with its final value, the IMU places the pointer to the stack object into the task pool, where any processor may pick it up, load in the stack, and resume the task.

We implement the following optimisation to this scheme. When a task is blocked, a pointer to its current *processor* is attached to the blocking node. Meanwhile, the processor maintains a task table containing pointers to the suspended tasks which it is holding. When the node is updated, the IMU sends notification of this fact to the processor concerned, which can then resume the task when it next has an opportunity. In this way, the task's stack never gets flushed. On the other hand, if the processor needs to clear some local heap space, it may flush the suspended stack, and inform the IMU of its (now global) location.

4.7 Function code blocks

Each closure in the heap contains a code-pointer and some arguments. But to what does the code-pointer point?

One method is to place the code for the closure in another graph node, and point to that. When a PE needs to execute the code, it dynamically loads the graph node in the usual way from the IMU which holds it, unless it already has a copy of that node. Since the PE's local heap will normally be significantly larger than the code for the program being executed, the local heap will eventually contain all the code for the currently-executing program.

Regarding code as graph nodes is elegant, because it provides a uniform way to garbage-collect code that is no longer in use, which in turn allows "eternal" programs to be written, such as operating systems.

Unfortunately, it also prevents the code being executed as efficiently as a sequential machine. For example, a return address pushed on the stack now has to be a pointer to a proper graph node, rather than a simple code address, because by the time it is activated the code to which it points may have been flushed or moved. In general, *a layer of address translation is added to almost every control transfer in the implementation.*

A better alternative is to use a conventional paging system in each processor to handle executable code. Code exhibits substantial spatial locality (in contrast to arbitrary graph nodes), so the communication system would be used more efficiently by moving pages at a time rather than individual function code segments. Paged-out pages can be represented in the global address space as graph nodes, but when they are paged in they become part of the processors local address space, with conventional hardware address-translation support.

We regard this as the right way to go, but GRIP does not support paged virtual memory, so for the present we load each processor with the code for the whole program.

5. Related work

A number of groups are working on parallel graph reduction machines, but so far only a few designs have been implemented.

The ALICE multiprocessor[Darl81a], built at Imperial College, is probably the first genuine parallel graph reduction machine. The graph is held (only) in a global memory connected to the processors by a switching network, and scheduling is on the basis of individual reduction steps. The processors and memory units are built from transputers running Occam, which imposes a layer of interpretation on many operations. As a result of these factors the machine is quite slow, but quite a lot has been learned from it[Harr86a].

The Alfalfa and Buckwheat systems, built by Ben Goldberg[Gold88a], are impressive implementations of compiled

parallel graph reduction on an Intel Hypercube multicomputer and an Encore Multimax bus-based multiprocessor respectively. His systems are broadly similar to ours, but differ in many important details (for example, children use explicit notification to reawaken their parents; and tasks do not use an evaluation stack). Each graph node carries quite large amounts of administration information, so it seems likely that his systems pay a fairly heavy overhead for parallelism whether or not the machine has capacity to exploit it.

The Flagship project[Wats87a, Wats87b] is far more wide-ranging than ours, but part of it concerns the architecture and organisation of the parallel graph reduction machine itself, and a prototype has been built. The architecture provides a flat address space, and this is the main source of differences between their approach and ours. They are clearly targetted at a scalable architecture, and so the issues of scheduling and locality are of crucial importance to them. One aspect of this is that Flagship has a rather sophisticated (perhaps oversophisticated?) mechanism for distributing work thorough the machine.

Other projects with well-advanced designs include the Dutch Parallel Reduction Machine project[Brus87a], the Mars project[Cast86a] and the PAM project[Loog88a].

6. Summary

Our overall goal is to execute functional programs on a parallel machine, where each processor runs with efficiency broadly comparable with a sequential implementation, except when communication and synchronisation are unavoidable. We believe this goal is achievable, and expect to have a working implementation of the ideas we have discussed by the middle of 1989.

References.

Bake78a. Henry Baker, "List processing in real time on a serial computer", *CACM* 21(4) pp. 280-294 (Apr 1978).

Brus87a. TH Brus, MCJD van Eckelen, MO van Leer, and MJ Plasmeijer, "Clean - a language for functional graph rewriting", pp. 364-384 in *Functional programming languages and computer architecture, Portland*, ed. G Kahn, LNCS 274, Springer Verlag (Sept 1987).

Burn88a. Geoff Burn, Simon L Peyton Jones, and John Robson, "The Spineless G-machine", pp. 244-258 in *Proc ACM Conference on Lisp and Functional Programming, Snowbird* (July 1988).

Burt85a. F Warren Burton, "Speculative computation, parallelism and functional programming", *IEEE Trans Computers* C-34(12) pp. 1190-1193 (Dec 1985).

Cast86a. M Castan and et al, "MARS - a multiprocessor machine for parallel graph reduction", in *Proc 19th Hawaii Intl Conf on System Sciences* (1986).

Clac86a. Chris Clack and Simon L Peyton Jones, "The four-stroke reduction engine", *Proc ACM Conference on Lisp and Functional Programming*, pp. 220-232 (Aug 1986).

Cous85a. G Cousineau, PL Curien, and M Mauny, "The Categorical Abstract Machine", pp. 50-64 in *Functional Programming Languages and Computer Architecture, Nancy*, ed. JP Jouannaud, LNCS 201, Springer Verlag (Sept 1985).

Darl81a. John Darlington and Mike Reeve, "ALICE - a multiprocessor reduction machine for the parallel evaluation of applicative languages", pp. 66-76 in *Proc Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, ACM (Oct 1981).

Eage86a. DL Eager, J Zahorjan, and ED Lazowska, "Speedup versus efficiency in parallel systems", Tech Report 86-08-01, University of Sasketchewan (Aug 1986).

Eage86b. DL Eager, ED Lazowska, and J Zahorjan, "Adaptive load sharing in homogeneous distributed systems", *IEEE Trans Software Engineering* SE-12(5) pp. 662-675 (May 1986).

Fair87a. Jon Fairbairn and Stuart Wray, "TIM - a simple lazy abstract machine to execute supercombinators", pp. 34-45 in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, ed. G Kahn, Springer Verlag LNCS 274 (Sept 1987).

Gold88a. Benjamin F Goldberg, "Multiprocessor execution of functional programs", YALEU/DCS/RR-618, Dept of Computer Science, Yale University (April 1988).

Hals86a. RH Halstead, "An assessment of Multilisp - lessons from experience", *International Journal of Parallel Programming* 15(6) (Dec 1986).

Harr86a. PG Harrison and M Reeve, "The parallel graph reduction machine ALICE", pp. 181-202 in *Graph reduction: proceedings of a workshop, Santa Fe*, ed. RM Keller, LNCS 279, Springer Verlag (Oct 1986).

Huda83a. Paul Hudak, "Distributed task and memory management", pp. 277-289 in *Symposium on Principles of Distributed Computing*, ed. NA Lynch et al, ACM (Aug 1983).

John87a. Thomas Johnsson, "Compiling lazy functional languages", PhD thesis, PMG, Chalmers University, Goteborg, Sweden (1987).

Kieb87a. RB Kieburtz, "A RISC architecture for symbolic computation", in *Proc ASPLOS II* (Oct 1987).

Loog88a. R Loogen, H Kuchen, K Indermark, and W Damm, "Distributed implementation of programmed graph reduction", in *Proc workshop on implementation of lazy functional languages, Aspenas* (Sept 1988).

Peyt89b. SL Peyton Jones, "Parallel implementations of functional programming languages", *Computer Journal*, (April 1989).

Peyt87a. Simon L Peyton Jones, *The implementation of functional programming languages*, Prentice Hall (1987).

Peyt87b. Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie., "GRIP - a high-performance architecture for parallel graph reduction", pp. 98-112 in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, ed. G Kahn, Springer Verlag LNCS 274 (Sept 1987).

Peyt88a. Simon L Peyton Jones, Chris Clack , Jon Salkild , and Mark Hardie, "Functional programming on the GRIP multiprocessor", in *Proc IEE Seminar on Digital Parallel Processors, Lisbon, Portugal*, IEE (1988).

Peyt89a. Simon L Peyton Jones and Jon Salkild, "The Spineless Tagless G-machine", RN/89/21, Dept of Computer Science, University College London (March 1989).

Reyn88a. TJ Reynolds, SA Delgado-Rannauro, ASK Cheng, and AJ Beaumont, "BRAVE on GRIP", Department of Computer Science, University of Essex (1988).

Rugg87a. Carlos A Ruggiero and John Sargeant, "Control of parallelism in the Manchester dataflow machine", pp. 1-15 in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, ed. G Kahn, Springer Verlag LNCS 274 (Sept 1987).

Wats87b. I Watson, J Sargeant, P Watson, and V Woods, "Flagship computational models and machine architecture", *ICL Technical Journal* 5(3) pp. 555-574 (May 1987).

Wats87a. Paul Watson and Ian Watson, "Evaluating functional programs on the FLAGSHIP machine", pp. 80-97 in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, ed. G Kahn, Springer Verlag LNCS 274 (Sept 1987).