Some Early Experiments on the GRIP Parallel Reducer

Kevin Hammond and Simon Peyton Jones *

February 22, 1991

1 Introduction

GRIP is a multiprocessor designed to execute functional programs in parallel using graph reduction. We have implemented a compiler for GRIP, based on the Spineless Tagless G-machine [Pey89], and can now run parallel functional programs with substantial absolute speedup over the same program running on a uniprocessor Sun.

Parallel functional programming shifts some of the burden of resource allocation from the programmer to the system. Examples of such decisions include: when to create a new concurrent activity (or *thread*), when to execute such threads, where to execute them, and so on.

It is clearly desirable that the system should take such decisions, *provided it does a good enough job.* For example, a paged virtual memory system almost always does an adequate job, and a programmer very seldom has to intefere with it. The big question for parallel functional programming is whether good resource-allocation strategies exist, and how well they perform under a variety of conditions.

Now that we have an operational system, we are starting to carry out experiments to develop resource-allocation strategies, and measure their effectiveness. This paper reports on some very preliminary results. They mainly concern the question of when, or even whether, to create a new thread. This is an aspect which has so far received little attention — existing work (e.g. [Sar87, SW89]) has focused mainly on load sharing rather than on thread creation. Our work is aided significantly by the fact that sparks indicate only the possibility of creating a new parallel thread, not the necessity of creating this thread. It is therefore always safe to discard sparks, while affecting only the performance of the computation, not its result.

*Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, UK. Tel: +44-41-339-8855 ext {5619,4500}. Email: {kh,simonpj}`cs.glasgow.ac.uk`

We begin with an overview of the GRIP system, and how we perform compiled graph reduction. This is followed by a summary of the statistics-gathering system and the results we have obtained so far.

2 Architectural overview of the GRIP system

GRIP consists of up to 20 printed circuit boards, each of which consists of up to four processors and one Intelligent Memory Unit (IMU).

The boards are interconnected using a fast packet-switched bus [Pey86], and the whole machine is attached to a Unix host. A bus was chosen specifically to make the locality issue less pressing. GRIP still allows us to *study* locality, but does not require us to *solve* the locality problem before being able to produce a convincingly fast implementation. Communication between components on the same board is faster than between components on different boards, but otherwise no difference in communication protocol is perceived by either the sender or recipient.

Each unit which communicates with the bus therefore needs a fairly sophisticated bus interface processor (BIP) to store packets and forward them to their destination when the bus becomes available. A single BIP is shared between the IMU and the PEs.

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

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

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

- Variable-sized heap 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 [Bak78].
- Each IMU maintains a pool of executable threads. Idle processors poll the IMUs in search of these threads.
- Synchronised access to graph nodes is supported. A lock bit is associated with each unevaluated node, which is set when the node is first fetched. Any subsequent attempt to fetch it is refused, and a descriptor for the fetching thread is automatically attached to the node.

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

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

3 Graph reduction on GRIP

Our graph reduction model is based on the Spineless Tagless G-machine [Pey89]. The expression to be evaluated is represented by a graph of *closures*, each of which is held in heap node. Each closure consists of a pointer to its *code*, together with zero or more *free-variable fields*. Some closures are in *normal form*, in which case their code is usually just a return instruction, while others represent unevaluated expressions, in which case their code will perform the evaluation.

A closure is evaluated by jumping to the code it points to, leaving a pointer to the closure in a register so that the code can access the free variables; this is called *entering* the closure. When evaluation is complete, the closure is *updated* with (an indirection to) a closure representing its normal form.

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

Initially there is only one thread, whose job is to evaluate the whole program. During its execution, this thread may encounter a closure whose value will be required in the future. In this case it has the option of placing of placing a pointer to the closure in the thread pool, where it is available for execution by other processors — we call this *sparking* a child thread.

If the parent thread requires the value of the sparked closure while the child thread is computing it, the parent becomes *blocked.* When the child thread completes the evaluation of the closure, the closure is updated with its normal form, and the parent thread is *resumed.*

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

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

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

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

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

Only conservative parallelism is supported at present; that is, there are no speculative threads, whose value may or may not be required. This constraint means that we do not have to implement a priority system to ensure that progress is made on essential threads at the expense of speculative ones. In the future we plan to relax this constraint somewhat.

4 Mapping parallel graph reduction onto GRIP

We start from the belief that parallel graph reduction will only be competitive if it can take advantage of all the compiler technology that has been developed for sequential graph-reduction implementations [Pey87]. Our intention is that, provided a thread does not refer to remote graph nodes, it should be executed exactly as on a sequential system, including memory allocation and garbage collection.

We implement this idea on GRIP in the following way. The IMUs together hold the *global heap*, a fixed part of the address space being supported by each IMU. Each processor uses its private memory as a *local heap*, in which it allocates new graph nodes and caches copies of global heap nodes. We ensure that there are no pointers into this local heap from outside the processor, so it can be garbage-collected completely independently of the rest of the system. When the IMUs run out of memory, the entire system is synchronised and global garbage collection is performed. Global garbage collection happens much less frequently than local garbage collections. In effect, local garbage collections recycle short-term "litter" without it ever becoming part of the global heap.

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

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

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

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

Nevertheless, this cost is only incurred when the system load is low. When the system fills up with threads, each processor simply runs locally.

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

It is clearly desirable to avoid performing updates if the node being updated is not shared. This is true in a sequential implementation but the cost of performing global updates makes it even more important in a parallel machine. Happily, recent advances in sequential compiler technology have shown how to use compile-time analysis to identify some of the nodes which are not shared, and how to avoid performing updates for such nodes [FW87, Pey89]. We plan to incorporate these ideas into our implementation.

4.1 System management

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

A single System Manager task, running on a particular processor, is responsible for polling the processors and IMUs in the system, assessing the overall system load, and making global scheduling decisions. The System Manager is also responsible for synchronising global system state-changes, such as occur during garbage collection.

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

5 Sparking strategies

In this paper we focus on one particular aspect of resource management, namely the decision about *when to spark a thread*. This is a key decision: too few sparks will fail to exploit the parallelism offered by the machine, while too many will saturate the machine with many small threads. In the latter case, not only is the granularity of each thread small, but there is also a serious communications overhead, as the sparked threads have to be flushed into global memory so that they can be made visible to other processors.

There are two times at which this decision can be made:

- At compile time. All the opportunities for sparking are planted by the compiler. When the compiler finds a point where a closure can be sparked, it plants a call to the runtime system, passing a pointer to the closure.

  A clever compiler with a strictness analyser might be able to insert sparks into an unannotated source program, but for the present we content ourselves with inserting annotations manually. So far our experience is that this is not hard to do, and not many are required (textually).

- At run time. The runtime system cannot invent new sparks, but it *is* free to ignore spark calls. This provides an opportunity to regulate sparking based on the dynamic properties of the program running on a particular machine configuration, which is very hard to predict statically.

  We have experimented with a variety of strategies for deciding when to ignore sparks.

5.1 Strategies used

So far we have experimented with four main strategies:

Unregulated. Plenty of spark annotations in the source code, and no runtime regulation at all.

Capital and income. Each processor is given some "capital" of spark credits and a regular "income" of spark credits. Whenever the runtime system is given the opportunity to spark a closure, it does so if its spark credit is positive, and decrements the credit.

This strategy gives a regular flow of work into the system, but is not capable of matching the rate of work creation to its rate of consumption. The latter is critically dependent on the size of the threads created, which is very hard to predict.

Adaptive income. This is the same as the capital-and-income scheme, except that whenever a processor grabs a new spark to execute, it increments its spark credit.

It seems that this would replace sparks as quickly as they are consumed (if any new ones become available), but actually it does not. Just before an IMU hands a spark to an idle processor, it checks whether the closure is already in normal form or being evaluated by some other thread. If so, the IMU silently discards the spark. Quite a lot of sparks disappear in this fashion.

Hand tuning. For many programs it is possible to put more work into the annotations and thereby get a much better granularity. For example, in a divide-and-conquer algorithm, the programmer can cause sparks to happen only for the first N divisions (thus creating 2^N threads), hence matching the number of threads created to the number of processors available.

It would sad if it turned out that this was always necessary, but it does give a useful benchmark against which to measure the more automatic techniques.

There are a number of other strategies we intend to try, but have not yet implemented.

Global load measurement. The System Manager polls all the IMUs to find out how many sparks they hold; and then tells the PEs what income level to use until next notified. Provided that the IMUs clear out dead sparks (that is, those pointing to closures which are now in normal form or under evaluation) from their spark pool, this should give quite a good measure of system load.

Idle-adapted income. Under this strategy, processors notify the System Manager when they become idle, and the System Manager in turn increases the income of all the PEs. This is quite easy to implement, but is really just a poor approximation to the global load-measurement strategy.

Delayed spawning. A common bad scenario is when a parent thread sparks a child, and then turns out to require the value of the child almost immediately. A typical example is the expression $E_1 + E_2$: suppose the parent sparks E_1, and then discovers that E_2 is either already in normal form, or very quick to evaluate. The parent now returns to E_1, and may well get blocked if another processor has started evaluating E_1 in parallel. Even if no other processor is evaluating it, a considerable cost has been paid to flush it into the global heap.

An idea which simultations suggest is promising is to delay making the spark public for some fixed period. Then, if the parent returns to it quickly it will never be flushed, but if the parent does have lots of work to do, parallelism is not lost. Clearly the onset of parallel execution is delayed somewhat, but this may well be a price worth paying.

6 Measurements

We have instrumented GRIP to monitor its runtime behaviour. At present we are collecting two main sets of data profiled against time: processor activity and spark usage.

6.1 Processor activity

We break down the distribution of processor time into the following categories:

- Reduction: this corresponds almost exactly to what a sequential Spineless Tagless machine would do.
- Communication: almost entirely due to flushing out parts of the graph into global memory and reading other parts in.
- Idle: time spent looking for work.
- Garbage collection: in all our experiments this formed less than 2% of the total time so it does not appear in our figures.
- Other overheads: time spent running other GLOS tasks. This is always negligible.

As the program runs, each processor accumulates a profile of this breakdown against time, which is collected by the host when the program completes.

Since the activities are interleaved quite finely, it is not adequate to refer to a real-time clock at the transition between activities. Instead, whenever there is a clock interrupt, we inspect a global variable which tells what the processor is currently doing, and increment an appropriate count. We maintain these counts in an array of 100 "buckets", each corresponding to a fixed time interval. When we run out of buckets, we combine the counts in adjacent buckets and double the time interval corresponding to a bucket. In this way we always have data with a time resolution of better than 2%, without having to decide any parameters in advance. Its precision within each bucket naturally depends on how many clock ticks correspond to a bucket.

The breakdown can be plotted either on a processor-by-processor basis, or aggregated into a single plot.

6.2 Spark profiles

We accumulate a time profile of how sparks are generated and used, in three categories:

Sparks generated. This count is incremented whenever a processor sparks a new thread.

Sparks used. Incremented whenever a processor gets a new thread to execute. There will often be fewer of these than sparks generated, because of the ones discarded because they turn out to be in normal form or under evaluation by the time they are selected for execution.

Threads resumed. This is incremented whenever a processor resumes execution of an old thread, which had been blocked but which has now been freed for execution.

These profiles are accumulated in the same bucket array as the processor-activity profiles, and dynamically rescaled in the same way.

7 Some Experiments

This section details the results of some experiments carried out on the GRIP multiprocessor. The programs used were parallel versions of the familiar *nfib* and *nqueens* benchmarks, as shown in figures 1-3. Parallelism is obtained exclusively from the *par* construct. *par* is a function of two parameters whose operational effect is to spark its first argument, and evaluate its second argument on the current processor. Semantically *par* is equivalent to:

```
par spark expression = expression
```

Each of the throttling strategies described above was used, including hand-placement of sparks to represent a coarse-grained version of the program. The fine-grained version is the one we would expect to be obtained using strictness information. In the long term we expect *pars* to be placed automatically. Until appropriate techniques are developed all sparks are placed manually.

We consider the *nfib* benchmark in most detail. This is not because we believe the *nfib* function is intrinsically valuable, but rather because it reflects an "ideal" parallel program with a straightforward decomposition into large threads. However, the decomposition which is naturally suggested by strictness analysis is degenerate, resulting in a large number of extremely small threads. Studying the behaviour of this program will provide us with a target for other, more realistic programs when we come to tune our throttling strategies. The strategies evolved for efficient execution of *nfib* should also apply to other divide-and-conquer algorithms.

```
let rec
     nfib n = if (n<2) then 1 else
                let n1 = nfib (n-1)
                in par n1 (n1 + nfib (n-2) +1)
in
itos (nfib 30)
```

Figure 1: Parallel nfib

```
let rec
    nfib n = if (n<2) then 1 else
                let n1 = nfib1 (n-1)
                in par n1 (n1 + nfib1 (n-2) +1)

...

and nfib4 n = if (n<2) then 1 else
                let n1 = nfibn (n-1)
                in par n1 (n1 + nfibn (n-2) +1)

and nfibn n = if (n<2) then 1 else
                let n1 = nfibn (n-1)
                in n1 + nfibn (n-2) +1
in
itos (nfib 30)
```

Figure 2: Parallel nfib – Coarse Grained Parallelism

```
let rec
   power 0 f = id
|| power n f = f o (power (n-1) f)

and diff x []    = x
||  diff x (b.y) =
        ( diff (remove b x) y
          where rec
             remove b []    = []
          || remove b (a.x) = if a=(b:Int) then x else a.remove b x )

and queens n = ( power n g [[]]
           where rec
           g = (foldl gg []
                where rec
                  gg a b = (par b (x @ b) where x = f a)

                and f l  = ( (foldl ff [])  (diff (range 1 n) l)
                         where rec
                         ff a b  = (par b (x @ b) where x = h l a)
                        )
           and delta y e =
                (exists o parlist id o map (check' (1,e)))
                (zip ((range 2 n),y))
           and h y e =  if delta y e
                        then []
                        else [e.y])

and check (i,j) (m,n) = (i=(m:Int)) | (j=(n:Int)) | (i+j=m+n) | (i-j=m-n)

and check' (i,j) (m,n) = (j=(n:Int)) | (i+j=m+n) | (i-j=m-n)

and all    = foldl (&) true

and exists = foldl (|) false

and range a b   = head (b-a+1) (from a)
in
itos (sum (map sum (queens 8)))
```

Figure 3: Parallel nqueens

No. of PEs	Speedup
1	1.0
2	1.9
4	3.7
10	9.5
20	17.0

Figure 4: nfib – Relative Timings

Configuration	Time(s)	Speedup
Sun 3/50	76.3	1.00
Sun 3/60	59.6	1.28
Sun 3/260	47.9	1.59
GRIP – 1	75.0	1.02
GRIP – 3	27.3	2.79
GRIP – 6	14.3	5.34

Figure 5: nfib 30 – Absolute Timings

Redn	36.58%
Idle	15.26%
Flush	45.43%
GC	2.70%
RUNTIME	2.79s

Figure 6: Activity graph for 8 PEs (nfib 25 – Unrestricted Sparking)

Figure 7: Sparking graph for 8 PEs (nfib 25 – Unrestricted Sparking)

Figure 4 shows the relative speedup obtained with the hand-tuned *nfib* for varying numbers of GRIP processors up to 20. These figures, which are the first real performance results obtained for GRIP, show a near-linear speedup for a completely untuned system.

Figure 5 shows the absolute timings for *nfib 30* running on various Sun-3s, and on GRIP with up to 6 processors. These figures clearly show that a 3-processor GRIP or better outperforms a high-end Sun-3. For the purpose of comparison, *nfib 30* is 2692537, so the 6-processor GRIP performs approximately 188000 function calls/second[1].

The comparison with Sun-3s is especially interesting since GRIP uses the same processor as the Sun 3/50 (MC68020 running at 16MHz). Since we use the Spineless Tagless compiler for both machines, GRIP is executing essentially the same machine-code program as the Suns. The slight performance advantage for a single processor GRIP may be attributed to the performance advantage of our lightweight operating system (GLOS) compared with Unix. The absolute performance of the compiler approximates that of the Unix Pascal compiler.

We now consider some detailed results, which motivate the use of a throttling strategy

Figures 6 and 7 show the effect of unrestricted sparking in an 8-processor GRIP running the fine-grained version of *nfib 25*. From the activity graph, it is obvious that communication completely swamps reduction, even though the processors display relatively little idle time. This communication is mostly caused by writing closures to global memory as

[1]Although this figure is not high by comparison with e.g. the Sun implementation of Clean, this is for a system which is not yet optimised for numeric computation. Optimisations which improve performance in the sequential case will carry over directly to the parallel implementation.

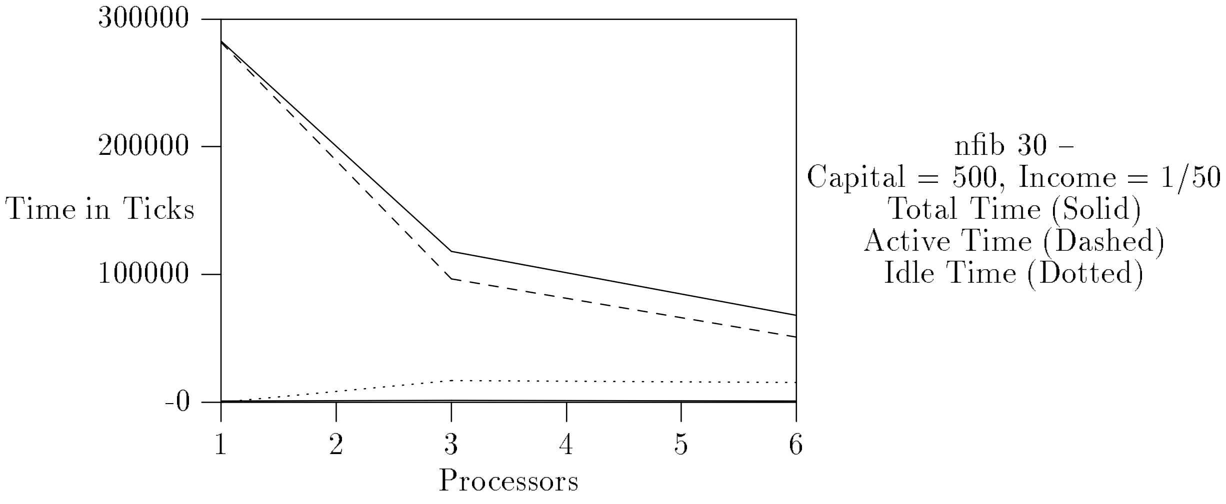

Figure 8: nfib 30 – Fine Grained Version – 500/50

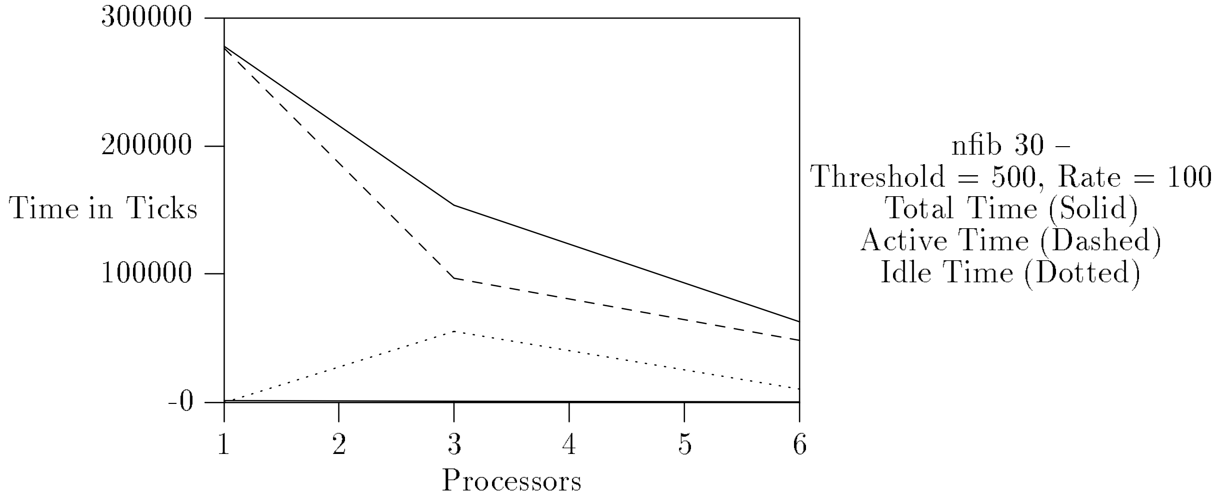

Figure 9: nfib 30 – Fine Grained Version – 500/100

a result of sparking, and reading these closures from global memory as a result of thread creation. This is obvious by comparison with later graphs where throttling is applied. The sparking profile show a continuously high rate of sparking with a comparatively low rate usage (apart from a few bursts which probably represent the evaluation of several small threads).

Such results led us to introduce a throttling strategy to control excess sparking, as detailed above. We have also considered *sparking* strategies that may avoid communication, for example retaining sparks locally unless the overall system load reaches some critically low level. These strategies have not yet been implemented, but the figures obtained so far lead us to expect a great improvement in performance even for unrestricted sparking.

Figures 8-9 show the relative overall performance of 1, 3 and 6 PEs on *nfib 30* for the fine-grained version with the same capital (500) but two different income settings (1/50 and 1/100). The higher income (1 new spark per 50 ticks) is clearly preferable. These results and others suggest an optimal income of 1 extra spark per 25-50 ticks for this problem.

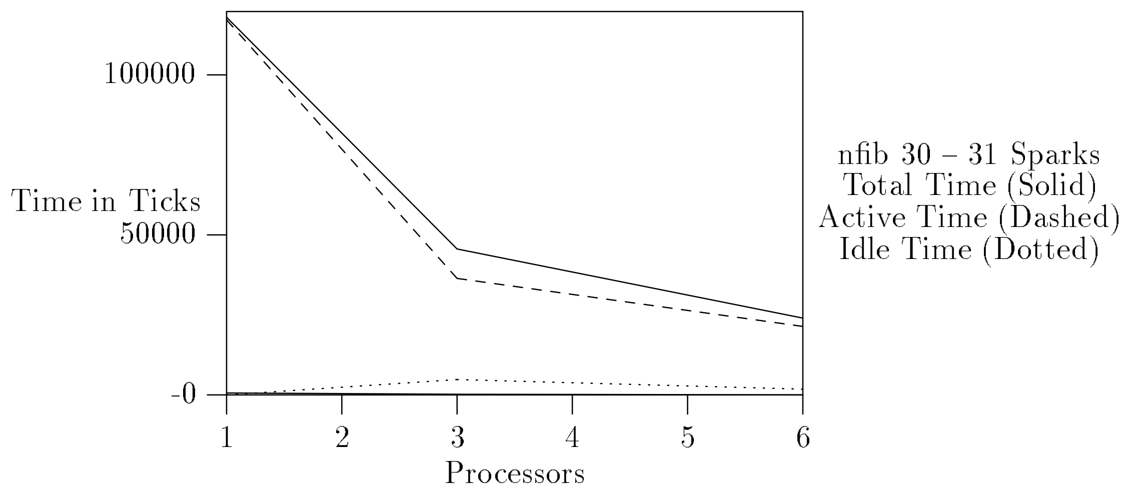

Figure 10: nfib 30 – Coarse Grained Version (31 Sparks)

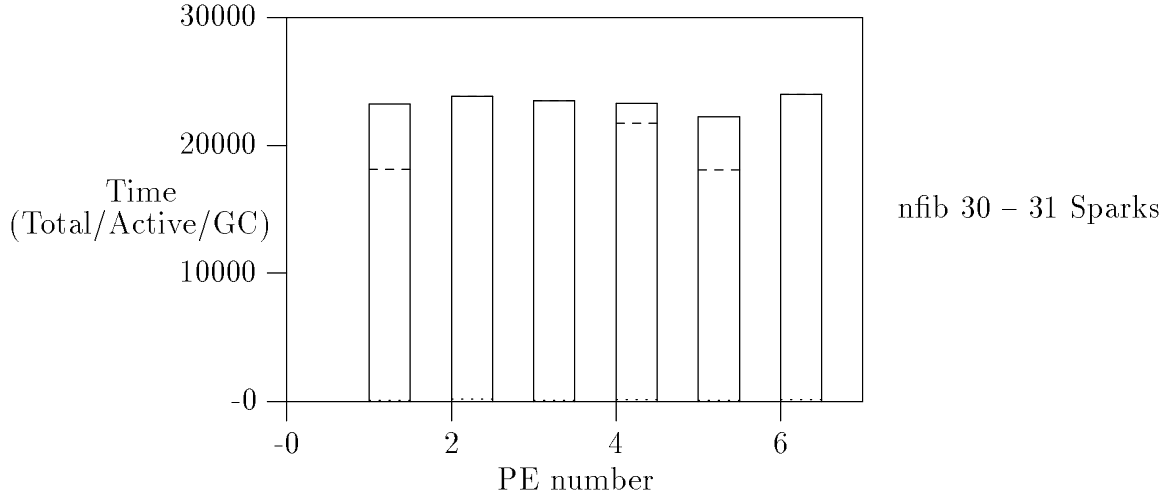

Figure 11: nfib 30 – Coarse Grained Version (31 Sparks), Idle Time

Figure 10 shows the corresponding performance results for the hand-tuned version. The coarse-grained version is a factor of 2 better than the fine-grained version, even with throttling.

Figure 11 shows the relative division between idle time (above the dashed line), active time (above the dotted line) and GC time (from 0) for a 6-processor GRIP running a hand-tuned *nfib*. Each PE is active for the majority ($\approx$ 90%) of the time. This reinforces our statement that *nfib* represents almost an ideal parallel program. The active time quoted here includes time spent both reducting and communicating with other PEs/IMUs. Later analyses will show that it is important to separate these figures if an accurate estimate of GRIP's performance is to be obtained.

Figures 12 and 13 show the overall activity and sparking graphs for one execution of *nfib 25* on a 6-processor GRIP. The throttling strategy allows a "capital" of 500 sparks, with an income of 1 spark per 50 ticks thereafter. It is easy to see that the initial burst of parallel activity rapidly dies away and thereafter GRIP runs essentially sequentially –

Redn	24.55%
Idle	72.10%
Flush	3.11%
GC	0.22%
RUNTIME	1.46s

Figure 12: Activity graph for 6 PEs (nfib 25 – FG 500/50)

Figure 13: Sparking graph for 6 PEs (nfib 25 – FG 500/50)

the throttling strategy is too successful at killing tasks. This can be seen in detail if the individual activity graphs are considered (figure 14) – PE 16.1 remains active long after the other PEs have terminated.

The corresponding adaptive strategy, on the other hand, is obviously superior, both in parallel activity and in execution time. The spark profile clearly shows sparks being created in response to usage. However reduction still only occupies 58% of the PEs' time – communication takes 23% of the time available and 17% is spent in the idle loop. The relatively high communication overhead probably represents traffic caused by sparked closures being flushed to/read from global memory. These figures suggest that we should easily be able to obtain a further 15% improvement in time performance by using a better throttling strategy, and may improve performance by 40% if we can also reduce the communication overheads (for example by lazy flushing). We are already within 42% of the best possible performance for this configuration: once this figure is attained, further improvements can only be obtained by applying better compiler technology to the sequential implementation.

The activity graphs for individual PEs (figure 17) show that no PE is significantly idle until the termination phase.

Figures 18 and 19 show some similar results obtained for *nqueens 8*. Relative timings and sparking graphs are shown for 1, 3 and 6 processors. These graphs show a speedup with greater numbers of processors, but not the linear speedup we desire. Further work is necessary to discover the reasons for this, including analysis of detailed performance graphs as for *nfib*.

Figure 14: Individual activity graphs for 6 PEs (nfib 25 – FG 500/50)

Redn	58.45%
Idle	16.87%
Flush	22.71%
GC	1.96%
RUNTIME	0.72s

Figure 15: Activity graph for 6 PEs (nfib 25 – FG 500/50 Adaptive)

Figure 16: Sparking graph for 6 PEs (nfib 25 – FG 500/50 Adaptive)

Finally, figures 20 and 21 show that even *nfib* can hold surprises. These graphs are for a hand-tuned version with a total of only 61 sparks. The activity graph fluctuates between near-total reduction (which we expect) and near-total inactivity (which we find surprising). Further investigation is needed to isolate the cause of this fluctuation, which is not apparent with fewer sparks. We suspect that the behaviour may be caused by the retention of blocked threads even when other PEs are idle.

8 Conclusion

In conclusion, we have performed several experiments on the GRIP parallel processor. Our results are mostly encouraging, and demonstrate that GRIP can outperform the best sequential functional programming systems. However it is clear that some kind of dynamic thread-control strategy is necessary to control excess parallelism in the fine-grained case. Figures given here demonstrate that a spark-discarding throttling strategy can lead to time performance within a factor of 2 of that of a hand-tuned program. We are optimistic that this can be improved. Much work remains to refine these results in order to tune GRIP for general problems or for a specific algorithm. It will also be necessary to test more rigorously the implication that fine-grained parallel execution of functional programs need not be significantly less efficient than coarse-grained parallel execution.

Certain of our results have highlighted the need for improvements to the "simple" implementation we currently have available. Such findings are unsurprising at this stage in the development of the GRIP prototype, and are a useful benefit of our detailed statistical approach. As we tackle the problems discovered by our statistics so the performance of

Figure 17: Individual activity graphs for 6 PEs (nfib 25 – FG 500/50 Adaptive)

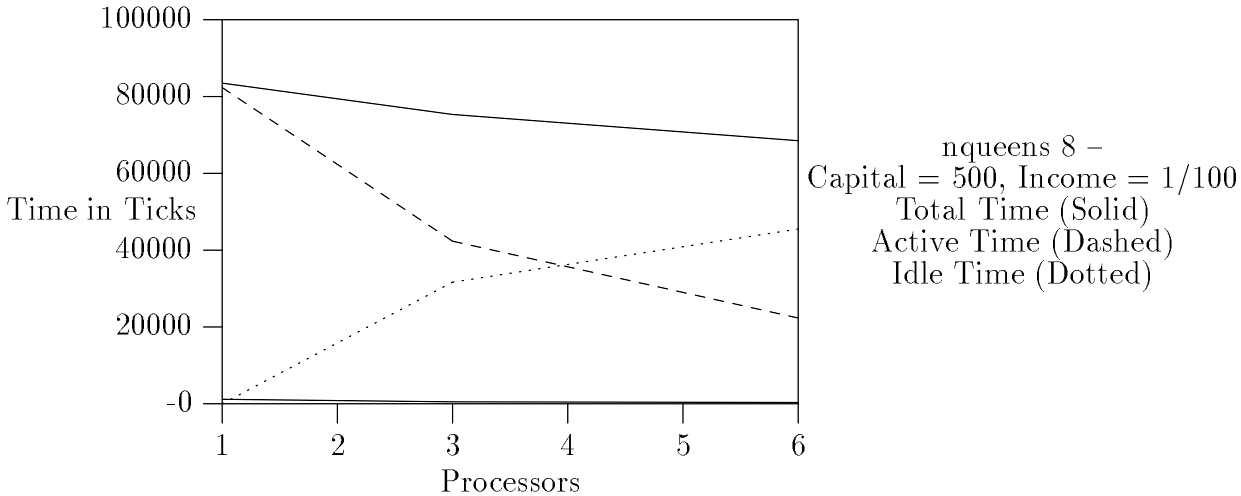

Figure 18: nqueens 8 – Fine Grained Version – 500/100

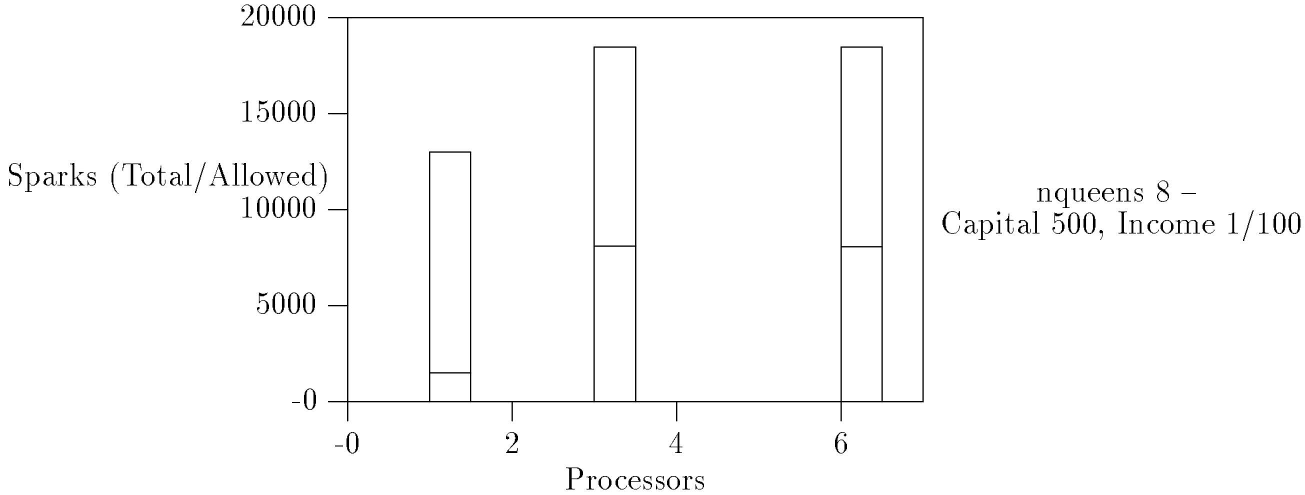

Figure 19: nqueens 8 – Fine Grained version, Sparking – 500/100

Redn	46.60%
Idle	53.30%
Flush	0.02%
GC	0.06%
RUNTIME	1.76s

Figure 20: Activity graph for 12 PEs (nfib 25 – 61 Sparks)

Figure 21: Sparking graph for 12 PEs (nfib 25 – 61 Sparks)

normal functional programs will be improved.

References

[Bak78] Baker HG, "List processing in real time on a serial computer", *Comm ACM* 21(4), April 1978, pp. 280-294.

[DFG85] Dijkstra EW, Feijen WHJ and van Gasteren AJM, "Derivation of a termination detection algorithm of distributed computations", *in Control Flow and Data Flow - Concepts of Distributed Programming*, ed. Broy, LNCS Springer Verlag, 1985.

[FW87] Fairbairn J, and Wray S, "TIM - a simple lazy abstract machine to execute supercombinators", *Proc FPCA 87*, Portland, Oregon, ed Kahn G, LNCS Springer-Verlag, 1987.

[JCSH87] Peyton Jones SL, Clack, C, Salkild, J and Hardie, M "GRIP – a high-performance architecture for parallel graph reduction", *Proc FPCA 87*, Portland, Oregon, ed Kahn G, Springer-Verlag LNCS, 1987.

[Pey86] Peyton Jones SL, "Using Futurebus in a fifth generation computer", *Microprocessors and Microsystems*, March 1986.

[Pey87] Peyton Jones SL, *The implementation of functional programming languages*, Prentice Hall, 1987.

[Pey89] Peyton Jones SL, and Salkild J, "The Spineless Tagless G-machine", *Proc FPCA 89*, London, ed MacQueen, Addison Wesley, 1989.

[Sar87] Sargeant J, "Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines", *Internal Report UMCS-86-11-5*, Department of Computer Science, University of Manchester, January 1987.

[SW89] Siedl H, and Wilhelm R, "Probabilistic LOad Balancing for Parallel Graph Reduction", *Proc IEEE TENCON 89*, Bombay, India, Novermber 1989, pp. 879-884.