Simulation of Multicache Parallel Graph Reduction

Andrew J Bennett
Paul H J Kelly

Department of Computing,
Imperial College of Science, Technology and Medicine,
London SW7 2BZ
email: ajb@doc.ic.ac.uk

Abstract

Parallel graph reduction is a simple model for parallel program execution which uses the shared-memory abstraction for all communication and synchronisation between processors. Shared memory is used under a strict access regime with single assignment and blocking reads. In this paper we present the design of an efficient and accurate multiprocessor simulation scheme suitable to work with a parallel graph reducer and use the simulator to study the performance and pattern of access of a suite of benchmark programs. Threads are never migrated to another processor by the parallel graph reducer, and the effect of this scheduling policy is investigated.

1 Motivation and Overview

Successful implementations of parallel graph reduction have been built on stock multiprocessors (e.g. Goldberg's Buckwheat system (Goldberg, 1988) and Augustsson and Johnsson's $<\nu, G>$ machine (Augustsson and Johnsson, 1989)), but the scalability of such systems is limited by contention for the shared snooping bus. In order to build more scalable systems, and to take advantage of modern high-speed processors, the bus must be replaced by a more complex network. Such a network inevitably has greater message latencies, but will offer a larger bandwidth, which in turn favours sending larger messages. The key questions are whether we can use a caching scheme so that only a small proportion of heap accesses require using the network, and whether we can utilise the much greater bandwidth that these networks offer.

We address these issues by means of simulation. The simulator is flexible and can be used to model a variety of memory systems. In this paper the design of the simulator is described, and results from a series of experiments using the simulator to model an ideal shared-memory are presented. The simulator allows the behaviour of the parallel graph reducer to be monitored in detail: this is used to study the sharing of heap cells used by a benchmark set of application programs. Monitoring also allows us to investigage a scheduling policy.

The simulator allows more complex memory systems to be modelled. It has been used to study the performance of a snoopy cache protocol used in commercial shared-bus systems in order to investigate the effectiveness of such systems with a variety of cache line sizes (Bennett and Kelly, 1992).

2 Shared memory

The performance of parallel programs under this regime depends critically on the provision of access to a shared heap with very high average performance. It is freedom from explicit considerations of data placement which gives the parallel graph reduction model its simplicity, and we must therefore demonstrate that it can be supported at reasonable cost.

The most straightforward way to implement shared memory is to construct a high-performance interconnection network to route requests to read and write shared memory to the appropriate memory bank. This approach has been taken in the BBN Butterfly (Crowther et al., 1985), and the IBM RP^3 prototype (Pfister et al., 1987). This has the problem that communication is incurred for all accesses to shared memory, whereas a cache-based scheme would allow programs which demonstrate some locality to benefit from much lower memory access latencies, and a much lower network traffic level.

Cache-based implementations of shared memory require some mechanism to prevent out-of-date copies of a cache line which has been updated from being used indefinitely. Essentially, it is necessary to issue an invalidation message to each remote cache which holds a copy of the line being updated. This is often done using a broadcast to all caches, but in larger systems a selective multicast is required — and the hardware must keep track of which caches hold copies of which cache line. With selective multicast of invalidations, communications between caches need occur when the program demands some interaction between the processing elements concerned; such protocols are in that sense scalable in that their effectiveness depends on the demands imposed by the program being executed.

3 Experimental design

Our objective was to study the data sharing requirements of programs under the parallel graph reduction model, in order to optimise the design of cache coherency protocols. In this section we describe and motivate how this was done. Our objective was to produce repeatable and reusable results, and so at several points we have chosen a simple approach rather than a more optimised implementation in which the issues might be obscured. We will return to these later.

3.1 The FAST compiler

The source language is a lazy, higher-order functional language in the tradition of SASL, Miranda and Haskell. The primary objective in building an optimising compiler for a lazy functional language is to reduce the frequency at which claims and references are made to the shared heap. It is therefore of great importance that the compiler used in our experiments should perform reasonably well.

We selected the FAST compiler, developed by our collaborators Pieter Hartel at the University of Amsterdam, and Hugh Glaser and John Wild at the University of Southampton. Although comparing compilers is difficult, we have some confidence that their system represents the state of the art (Glaser et al., 1990). It also, conveniently, generates C, making generated code very easy to instrument and modify.

3.1.1 Example: parallel Fibonacci

To illustrate how graph reduction works, and how it is implemented in the FAST system, we present a trivial example. This function, written in our SASL-like "intermediate" language, computes the i^{th} Fibonacci number using a method best described as "multiple and surrender" (this is for illustration only; the actual code executed is somewhat more complicated):

```
DEF
fib a = IF (a = 1)
           1
           (IF (a = 2)
               1
               (fib (a - 1) + spark(fib (a - 2))))
```

The function `spark` makes a closure available for execution by another processing element, if one is free.

The FAST compiler translates this into C code, which has been modified to expose the use of `spark` (some small simplifications have been made):

```
Field _11_fib(_11_a)
Field _11_a;
{
  Field _01_subexp1, _11_subexp2;
```

```
  if(get_bool(_11_EQ(1,_11_a))) {
    result_is( 1 )
  } else {
    if(get_bool(_11_EQ(2,_11_a))) {
      result_is( 1 )
    } else {
      _01_subexp1 = spark(C_vap( 2,_01_fib_XXXX_1_1, _11_SUB(_11_a,1)));
      _11_subexp2 = _11_fib(_11_SUB(_11_a,2));
      result_is( _11_ADD(_11_demand(_01_subexp1), _11_subexp2));
    }
  }
}
Field _01_fib(_11_a)
Field _11_a;
{
  Field _01_subexp1, _11_subexp2;
  if(get_bool(_11_EQ(1,_11_a))) {
    result_is( _01_1 )  /* _01_1 is boxed representation of the constant 1 */
  } else {
    if(get_bool(_11_EQ(2,_11_a))) {
      result_is( _01_1 )                                       /* boxed 1 */
    } else {
      _01_subexp1 = spark(C_vap( 2,_01_fib_XXXX_1_1, _11_SUB(_11_a,1)));
      _11_subexp2 = _11_fib(_11_SUB(_11_a,2));
      result_is( _01_ADD(_11_demand(_01_subexp1), _11_subexp2));
    }
  }
}
```

As is apparent, the resulting code is quite complex. Much of the confusion is due to the variable names, which follow a naming convention which uses the codes 01 ("boxed") and 11 ("unboxed") to indicate whether a value is represented by a closure. There are two C versions of each of the original functions, `_01_fib` for when the result is to be returned boxed, and `_11_fib` for when an unboxed result is required. For an expression to be made available for execution on another processor, its boxed representation must be added to a pool of available work using the function `spark`.

The fib function makes two recursive calls. The first recursive call is packaged as an unevaluated application of the boxed-result function `_01_fib` (represented by a "vap", a vector application node in the shared graph). The `spark` function adds this to a global pool of available work; if another process evaluates it, it overwrites the "vap" with the boxed result. The second recursive call is then made directly, this time to `_11_fib` as an unboxed result is required. The function `_11_demand` checks whether the vap has indeed been evaluated by some other processor, and if not, it evaluates it. If a processor is evaluating the expression but has not yet finished, `_11_demand` blocks.

3.2 Process management

Parallel computation is coordinated via the heap where vaps are allocated. Work is allocated to processors via a work pool, which in the current system is a simple queue protected by a spin-lock. The heap is accessed by the processors in order to determine what needs to be evaluated, and processors synchronise using a blocking read operation.

This is implemented using a non-premptible threads package. When a processing element is idle, the scheduler refers to the work pool, and if a sparked task is available it extracts it and allocates a thread to evaluate the specified expression. Execution proceeds just as in the sequential case, except that when `_11_demand` encounters a boxed expression which is marked to indicate that it is being evaluated by some other processor, the thread blocks.

At this point, the scheduler selects another runnable thread, if any, using a round-robin (FIFO) policy. If none is present, it attempts to extract a new task from the task pool. If there are no runnable threads and no sparked tasks in the pool, the processor spins.

An important property of this scheme is that once a processor assigns a thread to a task, the task must be completed on that processor. This means that it is possible for processors to be idle while other processors have many runnable threads. We investigate some of the affects of this scheduling policy later.

3.3 Garbage collection

When storage allocated from the shared heap becomes free, it should be recycled for reuse. In a parallel system a parallel garbage collector is needed, and the area is the subject of intensive research. The behaviour of the garbage collector may interfere with normal program execution in two ways: firstly, it may change the relative timing of processes, depending on when it is activated, and whether all processors collect in synchrony. Secondly, garbage collection may substantially change the pattern in which store is allocated. This is especially apparent with copying allocators, e.g. (Appel et al., 1988), but even in non-copying collectors, it is influenced by the allocation mechanism.

We have made an important simplification here: we have no garbage collection at all. Instead, each processor is allocated a large contiguous segment of the shared address space, and it allocates space from it starting from the base.

The motivation for this decision is as follows: to introduce a garbage collector into the simulation we would have to choose one of the available algorithms, and our results would then be applicable only when a similar collector is in use. Unfortunately, there is no obvious candidate: there is no consensus on how garbage collection should be done in large shared-memory systems. However, any copying collector would finish by handing the application program a contiguous segment of free store. Thus, there is a significant period during which allocation proceeds in precisely the simple pattern we assume. We simply assume this is the case all the time.

This assumption illuminates an important point: our objective is to learn general lessons about a large class of systems. We are less concerned that the experiments predict the actual performance of some production system, as to do so we would have to introduce many factors which are orthogonal to the issues we intend to study.

3.4 Example functional programs

This Fibonacci program is hardly representative of serious applications programs. The complete suite of programs we have been using are as follows:

pfib	compute the n^{th} Fibonacci number
nqueens	compute a safe arrangement of queens on an $n \times n$ chess board
matmult	multiply two $n \times n$ matrices
quad	find the integral of a cubic function using adaptive quadrature

Less trivial example programs are available, but the programs in this suite are simple enough to offer the possibility that their behaviour might be understood, while covering a variety of parallel program structures.

3.5 Execution-driven cache simulation

Our objective is to study the behaviour of programs running on large shared-memory multiprocessors, using a variety of cache designs. To do this, simulation is required both to allow a range of experiments to be performed, and to reduce the number of variables. The design of an architectural simulator depends on the nature of the study; at the lowest possible level, a time-driven gate-level simulation may be the most practical approach, although the computational resources involved are immense. At the other end of the spectrum, cache design has been widely studied using trace-driven simulators. In a trace-driven simulator, the object program is executed just once, and the sequence of its memory references is recorded. This (or parts of it) can then be "played back", as input to a simulator of a cache memory system (see, for example, Stone chapter 2 (Stone, 1990)).

Trace-driven simulation has been widely employed in simulating shared-memory systems, e.g. (Dubois et al., 1986). However, there are problems with the validity of such studies since the relative timing of processes (and possibly their behaviour) may in reality depend on the behaviour of the memory system. Under the simulator, the traces are kept constant despite changes in the structure of the memory system being simulated. It is possible to argue the validity of simple experiments (Bitar, 1990), but without independent validation this is unconvincing.

For this reason, we have chosen execution-driven simulation, an intermediate approach. Each of the processors to be simulated is modelled by a separate process, and it runs essentially the code which the real processor would execute. All references to shared memory are intercepted, and when a processor makes such a reference, control is passed to the memory system simulator. This scheme is similar to that used in the Tango simulator (Davis and Goldschmidt, 1990) and in MiG (Muller et al., 1992).

The code executed by each processor is modified to maintain an estimate of the simulated CPU time it spends between references to shared memory. The memory system simulator uses this to maintain a global simulated time as the computation proceeds. Every time a processor refers to shared data, the simulator computes the time it would take for the reference to be satisfied. Simulation of that processor is then reactivated only when all other processors reach or exceed this point.

This approach is much faster than time-driven simulation of the processors at every step, since when no shared-memory reference is made, the code runs at near-native speed (slowed only by the need to count instructions executed). It is less accurate, in that it fails to account fully for performance variations due to processor pipeline structure, for example, but this level of detail is not of interest here. It is slower than trace-driven simulation, since the object code is reexecuted for each experiment, but it eliminates the validity problems of trace-driven simulation since the simulated processors read the data as it is at the simulated time at which the reference is made. The simulator is flexible enough to model a wide range of shared-memory systems

3.6 Execution time model

Between consecutive global events generated by a particular application process, an arbitrary amount of time is spent accessing local memory (code, stack and private data), performing arithmetic operations, etc. The virtual timer associated with each application process must be altered to account for this time.

A simple and moderately accurate approach to this would be to count the number of machine instructions executed, incrementing a count register at the end of every block. The disadvantage of this is that the counts would be influenced by the CPU architecture, the C compiler, and the FAST compiler. To eliminate these variables, we instead count "significant" events in the computation, and assign fairly arbitrary execution times to each:

1. primitive arithmetic: 2 cycles.
2. stack push or pop: 2 cycles.
3. procedure call: 8 cycles.
4. add or take from a task or thread queue: 2 cycles.

This again gains simplicity and repeatability at the expense of predictive accuracy. The weights are chosen somewhat arbitrarily, being reflective of some conceivable implementation, although not of any actual one. To monitor the influence of this we keep five separate counts of time spent "computing", idle, spinning on locks, manipulating queues and accessing the shared heap. The total estimated time determines the relative timing of simulated processors.

4 Results

This paper reports preliminary results from modelling an ideal shared memory system, together with some analysis of the traces which result. Simulation of a variety of more realistic systems will be reported in (Bennett, 1992).

4.1 Modelling an Ideal Shared-Memory System

We begin by simulating PGR under an ideal shared-memory implementation, in which the latency of all shared-memory operations is always a single cycle. This control experiment serves to validate the behaviour of the PGR system, and provides a baseline against which to compare more practical shared-memory systems' performance.

4.2 Simulated Execution Time

Figure 1 shows the relative performance of each of the benchmark programs for upto 40 processors. The pfib example shows the most impressive results with a relative performance of 36 for forty processors. Linear speedup is not possible due to the lack of parallelism available at the start and end of the computation.

4.3 General Application Characteristics

Table 1 shows details of the different types of heap access for 16 processor simulations. For each program the number of each type of access is recorded, along with the corresponding percentage. Write percentage 1 indicates the proportion of heap reads and writes that are writes, whereas write percentage 2 indicates the proportion of all heap accesses that are writes.

The percentages vary greatly between the benchmarks, but some conclusions can be drawn. Fib only generates heap cells in order to make tasks available for execution by other processors, and this causes the write-percentages to be particularly large. Matmult has significant data structures that are read by multiple processors, producing a lower write-percentage.

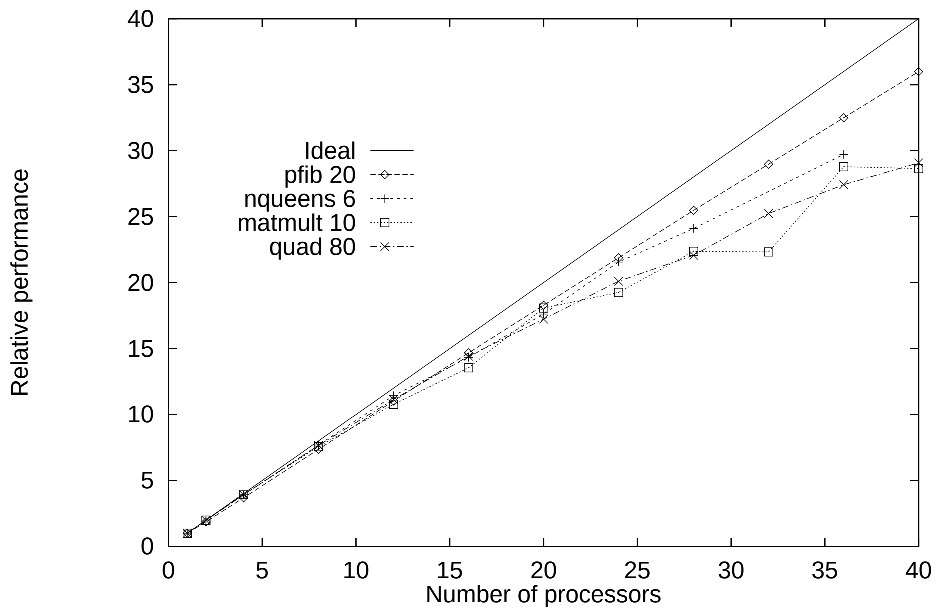

Figure 1: Relative Performance of the Benchmark Set.

Application program	Cells used	Heap writes	Heap reads	Heap locks	Heap unlocks	Write percentage 1	Write percentage 2
pfib 20	9375	43079 27%	88098 56%	13595 9%	13585 9%	33%	44%
nqueens 6	11148	20548 14%	115600 80%	4566 3%	4502 3%	15%	20%
quad 80	8146	23930 21%	71813 64%	7853 7%	7840 7%	25%	36%
matmult 10	11844	12052 9%	115558 90%	489 0%	483 0%	9%	10%

Table 1: General Application Characteristics.

4.4 Sharing of Heap Cells

Active sharing is important since only data that is actively shared can become incoherent. Sharing information is collected at runtime by the simulator and was used to produce table 2. A count of the total number of processors that access each heap cell was made. For nqueens and matmult, approximately 90% of heap cells are not shared, and thus only 10% may become incoherent. Of the shared cells, most are shared by a small number of processors, and thus multicast invalidations appear appropriate.

Processors	pfib 10	nqueens 6	quad 80	matmult 10
1	32.12%	88.58%	94.03%	91.06%
2	67.87	8.62	5.39	2.87
3	0.00	1.09	0.26	0.67
4	0.00	0.59	0.15	0.02
5	0.00	0.55	0.04	0.27
6	0.00	0.31	0.04	0.78
7	0.00	0.11	0.04	1.03
8	0.00	0.03	0.01	0.78
9	0.00	0.02	0.00	1.53
10	0.00	0.01	0.00	0.82
11	0.00	0.02	0.00	0.00
12	0.00	0.00	0.00	0.00
13	0.00	0.00	0.00	0.02
14	0.00	0.00	0.00	0.02
15	0.00	0.00	0.00	0.00
16	0.01	0.07	0.04	0.13

Table 2: Heap Cell Sharing Statistics.

4.5 The Effect of Non-Migratable Threads

The quad benchmark decomposes well for parallel execution. When it's execution is simulated under ideal shared-memory, the good speedup curve is expected. However, the processor activity graph of figure 2 shows erratic behaviour after about 21000 cycles. In this graph, the number of active processors is plotted over the simulated execution time of the quad program when simulated with 12 processors.

The graph indicates that all processors are active for most of the execution time, and therefore nealy linear speedup is produced. There is a period starting at approx. 21000 cycles where the number of active processors drops rapidly to 7, and gradually returns to 12 about 450 cycles later. It is unclear from the graphs presented so far what is causing this effect. Further instrumentation of the run-time system and simulator reveals that the number of active processors falls from 12 when the task queue becomes empty. That is, by this time the evaluation of all tasks that have been sparked has begun.

However, the dip in the graph could be eliminated if resumed threads were allowed to migrate, since the simulator shows that some resumed threads are available in queues for the duration of the dip. Allowing threads to migrate on demand would produce a rather more uniform activity graph. However, migration would have an impact on the memory access characteristics which in turn affect memory access latencies in a more realistic shared-memory implementation.

Although the effect of the loss of parallelism is not significant in this case, it inevitably varies between applications and the number of processors simulated. Figure 3 shows the

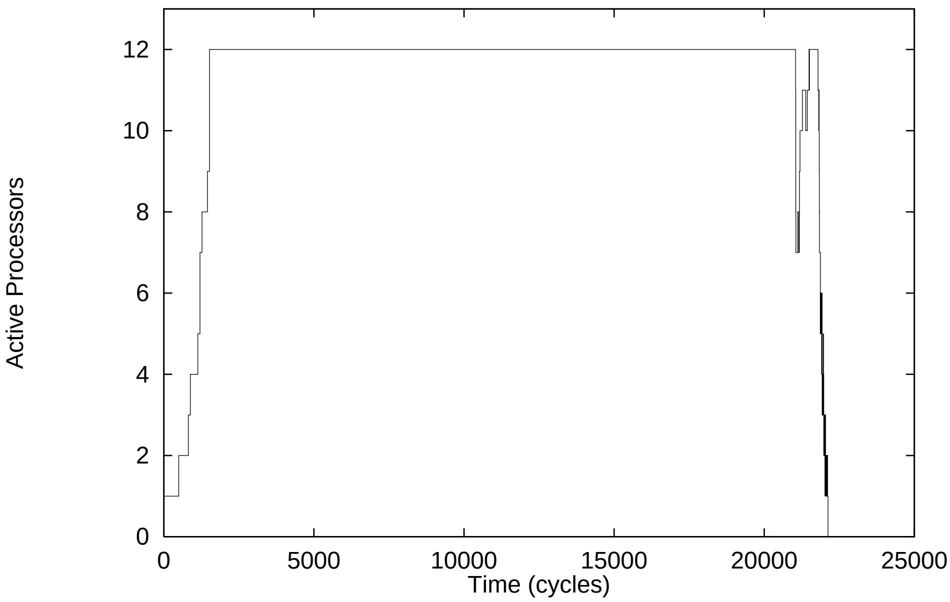

Figure 2: Activity Graph for quad 80 with 12 Processors.

activity graph for a 40 processor simulation of quad 80 under ideal shared-memory. The effect is more pronounced, but again it is unclear whether allowing threads to migrate would significantly lower simulated execution time.

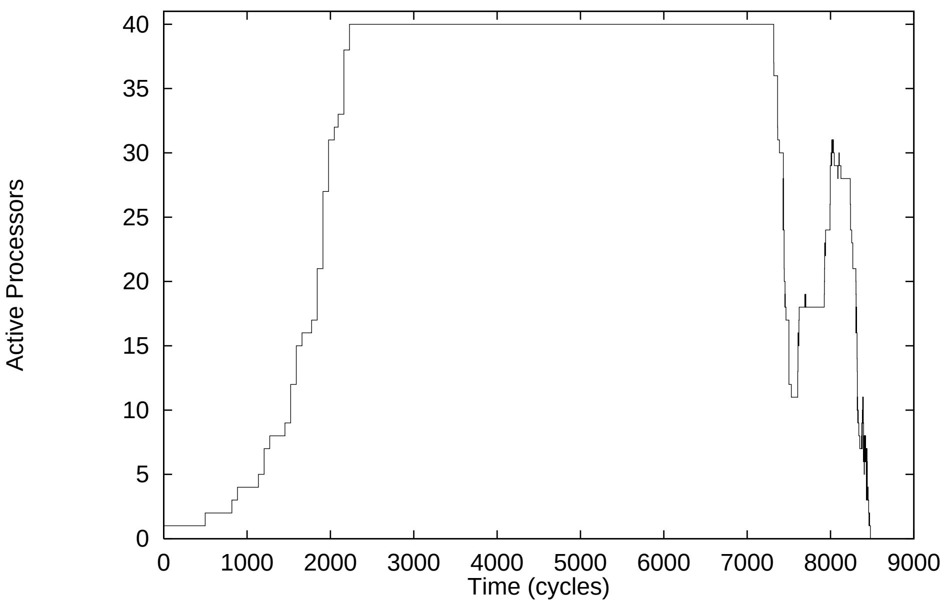

Figure 3: Activity Graph for quad 80 with 40 Processors.

5 Conclusion

This paper has presented the design and implementation of a parallel graph reduction system, based on a state-of-the-art optimising compiler, running under an event-driven simulation of a multiprocessor computer.

The simulator is flexible and can be used to model complex memory sytems (such as multicache and weak-coherency systems). Results from ideal shared-memory experiments indicate that the parallel graph reduction system is well behaved and near-linear speedups can be obtained. Active sharing statistics have been produced that indicate that relatively few heap cells are actively shared, and most of those that are shared are accessed by relatively few processors.

A high proportion of heap accesses are locks, and a more efficient locking scheme could be adopted to reduce the invalidation traffic these would cause in a typical multicache shared memory since there is little contention for these locks.

Requiring that threads are only executed on the processor that they started on causes some serialisation towards the end of the computation that becomes more significant as more processors are used. Further experiments are required to quantify the severity of this problem and its effect on network traffic and access latencies in more realistic shared-memory designs.

Acknowledgements

Enormous thanks are due to our collaborators on the FAST project, John Wild and Hugh Glaser at Southampton University, Pieter Hartel at the University of Amsterdam, and J Liu and Frank Taylor at Imperial. We gratefully acknowledge Erik Hagersten of the Swedish Institute for Computer Science, whose helpful comments led us to the feasibly efficient simulation technique described. This work was supported by an SERC research studentship (for AJB), and partially by SERC project number GR/F/35081.

References

Appel, A. W., Ellis, J. R., and Li, K. (1988). Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia*, pages 11–20.

Augustsson, L. and Johnsson, T. (1989). Parallel graph reduction with the $<\nu, G>$-machine. In *Functional Programming Languages and Computer Architecture*, pages 202–213. ACM. Imperial College, London.

Bennett, A. J. (1992). *Shared Memory in Parallel Graph Reduction (working title)*. PhD thesis, Imperial College, University of London. In preparation.

Bennett, A. J. and Kelly, P. H. J. (1992). Sharing and contention in coherent-cache parallel graph reduction. In *UK Performance Engineering Workshop, London, Sept.*

Bitar, P. (1990). A critique of trace-driven simulation for shared-memory multiprocessors. In Dubois, M. and Thakkar, S. S., editors, *Workshop on Cache and Interconnect Architectures in Multiprocessors, Eilat, May*, pages 37–52, Boston. Kluwer Academic Publishers.

Crowther, W. et al. (1985). Performance measurements on a 128-node Butterfly parallel processor. In *Proceedings of the International Conference on Parallel Processing*, pages 531–540.

Davis, H. and Goldschmidt, S. (1990). Tango: a multiprocessor simulation and tracing system. Technical report, Stanford University.

Dubois, M., Briggs, F., Patil, I., and Balakrishnan, M. (1986). Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 909–916.

Glaser, H., Hartel, P., and Wild, J. (1990). A pragmatic approach to the analysis and compilation of lazy functional languages. Technical report CSTR 90-10, Department of Electronics and Computer Science, University of Southampton. (In Proc. of the Workshop on Parallel and Distributed Processing, Sofia, 1990. North-Holland 1991).

Goldberg, B. F. (1988). *Multiprocessor Execution of Functional Programs.* Research report, Yale University Department of Computer Science.

Muller, H. L., Langendoen, K. G., and Hertzberger, L. O. (1992). MiG: Simulating parallel functional programs on hierarchal cache architectures. Technical Report CS-92-04, Department of Computer Systems, University of Amsterdam.

Pfister, G. et al. (1987). *Experimental parallel computing architectures*, chapter An introduction to the IBM Research Parallel Processor Prototype (RP3), pages 123–140. Elsevier Science Publishers.

Stone, H. (1990). *High performance computer architecture.* Addison Wesley.