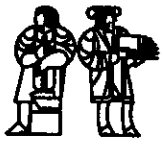
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor

Computation Structures Group Memo 303
October 27, 1989

Gregory M. Papadopoulos

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

5

Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor

Gregory M. Papadopoulos[1]

5.1 Introduction

Developers of parallel applications often feel like they are living in a Heisenbergian purgatory; instrumenting a program can radically change its run-time behavior, even causing it to produce different answers! Such experiences lead to the natural conclusion that we should fundamentally improve the way real-time data are collected on multiprocessors and expand the kinds of measurements being taken. But perhaps we should first understand *why* the need for quality multiprocessor instrumentation is so much more acute than the historical demands placed on uniprocessors. Is getting a parallel program to run well (or at all) somehow inherently more difficult than doing the same thing on a sequential processor? Can better instrumentation really abate our frustration and accelerate the effective use of parallel machines? It seems that the requirements of three development activities are confounded under the desire for better instrumentation:

[1]The author is funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

- **Debugging.** Getting the program right.
- **Program Analysis.** Understanding the inherent properties of a program (*e.g.*, its average concurrency).
- **Performance Tuning.** Making the program run well on a particular implementation.

Our investigation of dataflow architectures [1] and associated declarative languages has led us to believe that each of these activities can and should be approached independently of the others, and that much of the perceived need for highly dynamic and specific instrumentation is an artifact of the mismatch (and inadequacy) of current parallel programming languages and processor architecture.

To start, a program should be able to be developed and debugged without regard to the target architecture. There should be no "surprises"—the exposing of latent bugs—when changing the machine configuration. A declarative (or functional) programming language (*e.g.*, fp [2], SISAL [3], Id [4]) guarantees *determinate* execution, even in the presence of program bugs; re-execution of a program on the same set of data will always yield the same results and set of errors, independent of the number of processors or relative execution timings.

But determinacy does not guarantee good *performance* on a given problem; different mappings of a computation onto a machine will always yield the same answers (and errors) but may result in dramatically different execution times. And it can also be the case that *no* mapping yields acceptable performance—meaning the problem (inherently) has insufficient parallelism for the given machine configuration. But when observing a poorly running program, how are we to know if the mapping is bad or if there is simply too little parallelism for the problem size?

Ideally, it would be nice to perform an experiment that would tell us the inherent parallelism and thus *predict* the best performance that we could ever hope to get on a machine with a given number of processors. Then, we could compare our prediction with the actual measured performance and determine if further improvement is possible.

To these ends, we unabashedly propose a *scientific method* for the development of parallel applications. As shown in Figure 5.1, a debugged program can be analyzed to expose its inherent parallel characteristics and the expected performance on a given machine configuration can be predicted theoretically. Measurements from the actual program execution can then be compared against the predicted performance; a mismatch would point to either inadequate analysis or a deficiency in the machine implementation.

The utility of *instrumentation* in a multiprocessor should be gauged by its contribution to facilitating program development under this paradigm. The experimental *Monsoon dataflow multiprocessor* [5] now under construction at

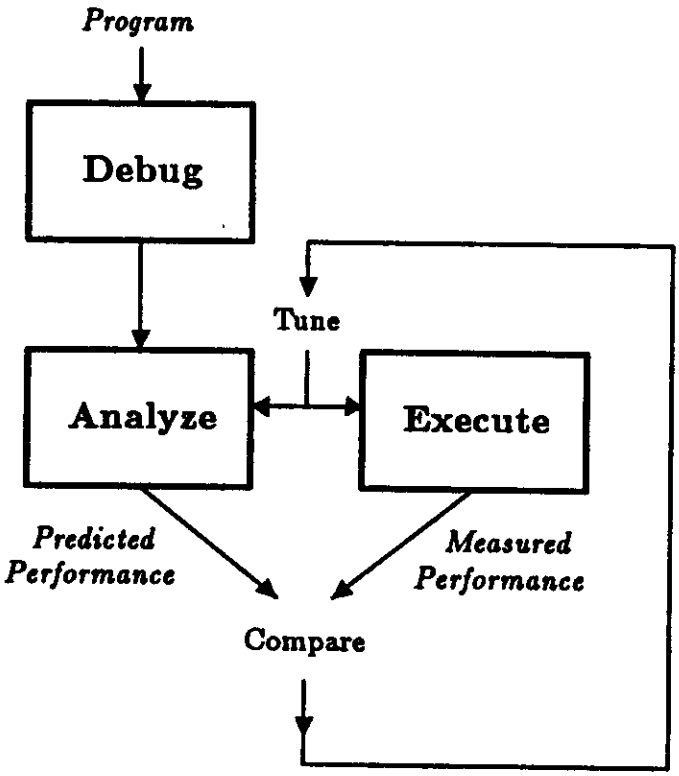


FIGURE 5.1
A scientific method for developing parallel applications.

M.I.T. incorporates a number of features to aid in program debugging, analysis, and performance tuning. We present two novel features in this discussion:

1. Hardware support for machine-independent program analysis. Using a pair of "ping-pong" token queues we can efficiently simulate an infinite processor array on a finite collection of processors.
2. An instruction coloring technique that allows instruction mix statistics to be focused on programmer-selected subsets of procedures, allowing the rapid determination of the contribution from a given procedure to the overall program.

Our ultimate goal is to replace the *ad hoc* development of parallel applications with a rational methodology based on analysis and measurement where the processes of debugging, program analysis and performance tuning are distinct, well-defined tasks.

5.2 Getting the Program Right

Parallel machines seem to have induced a regression in programming. A programmer is often concerned with the reflection of implementation details onto the structure of the program (*e.g.*, number of processors, interconnection topology, synchronization costs). This is usually accomplished by annotating programs written in conventional sequential languages (*e.g.*, FORTRAN, C and Lisp) to indicate that certain things should be performed in parallel [6]. At a high level it may be evident that an application possesses ample parallelism, but the complexity of exposing the parallelism through annotations may introduce new errors that did not exist in the sequential version [7,8].

For example, matrix multiplication is a determinate computation, but if the the programmer accidently uses the same variable to accumulate the parallel inner products, the program is likely to show indeterminate behavior. The indeterminate behavior is often difficult to reproduce when debugging because it depends upon the relative execution *timings* and the debugging instrumentation itself alters these relationships. Moreover, the explicit parallel task model of these computations leads to execution traces that comprise collections of independent processes competing for machine resources. Pragmatically, it is hard to name and "talk about" a specific process and relate it back to the program text, as the processes are apt to move around from run to run.

For these reasons, debugging places the greatest demands on instrumentation; it must be non-invasive and it must deal with the *timing of events* rather than the statistical abstraction (the number of events) of execution performance. In fact, we believe that the instrumentation problem for debugging timing-dependent phenomena is so hard that trying to identify the sources of

nondeterminism through measurement is fundamentally impractical. While there appears to be significant progress in the development of automatic dependence analysis tools (*e.g., Ptool* [9]) to assist in detecting incorrect parallelization, we question whether these techniques can ever really uncover the truly insidious bugs in programs that employ sophisticated data structures and/or attempt to manage their own storage.

We believe the best approach to developing robust parallel codes is to provide a programming environment that guarantees determinate execution *even in the presence of program bugs.* Declarative languages can offer such a guarantee:

> Independent of the relative timing of concurrent processes and the distribution of data and processes across processors, a program will always yield the same result and the same set of errors (given the same input data, of course).

This property, known as the *Church-Rosser property* [10] is not without its costs, and the implementation efficiency of declarative languages is the subject of much research—and beyond the scope of this discussion.

It is encouraging to note, however, that scientific programs expressed in the declarative language *Id* [4,11] for our dataflow processor yield dynamic instruction mixes within a factor of two of the same applications written in FORTRAN and executed on contemporary von Neumann uniprocessors [12,13]. In addition, the FORTRAN codes generally experience a measurable increase in execution overhead when they are executed in parallel (after suitable annotations) while the number of instructions executed for the Id program is independent of the number of processors.

Thus, one option for the parallel programmer is to debug the application in the familiar development environment of a single processor. The Church-Rosser property guarantees that the program will behave identically (in terms of results produced) when executed on a multiprocessor configuration[1].

5.3 Predicting Performance

Amdahl's law for vector processing provides a simple machine-*independent* characterization of the performance limitations of vectorization. Given an application that is, say, 90% vectorizable, Amdahl's law limits the *best case* speedup to about a factor of ten—even given an infinitely fast vector unit. When we

[1] We gloss over the obvious requirements that the machines all have the same arithmetic behavior. Not to mention the fact that some applications simply exhaust the resources (like memory, or the patience of the programmer) of a single processor, in which case debugging would have to take place on a multiprocessor. We will address the problem of debugging on a parallel machine in a moment.

execute the program on an actual vector processor we are able to roughly predict its expected performance. If the experienced performance and the predicted performance agree, then we know the only way to further decrease execution time is to adjust the code to increase the amount of vectorization. If experience and analysis substantially *disagree* then we turn to performance measurements to diagnose the cause of the problem, *e.g.*, bank conflicts, short vectors, poor compilation, *etc.*

Amdahl's law applies to parallel execution as well—the maximum expected performance gain from processing things in parallel is ultimately limited by that portion of the computation that is inherently sequential. We can all understand this rule in the abstract sense and can sometimes apply it in practice. However, its utility is limited by the ability to precisely distinguish that fraction of the computation that is truly sequential.

Fundamentally, the sources of parallelism in a program are manifold[2] and trying to predict what fraction of the parallelism can be exploited by a particular machine organization can be exceedingly complex. Moreover, the language in which the application is coded may unintentionally obscure certain kinds of parallelism. It appears that there is no substitute for actually running the program on a real set of input data and then, somehow, analyzing real-time measurements in an effort to determine both the available parallelism and how much of it is effectively exploited.

Our approach is try to determine the *potential parallelism in programs* by considering only the essential data dependences involved in computing the result. Then we try to predict how the potential parallelism is attenuated during execution. Armed with these predictions, we have a better chance of understanding actual real-time measurements of the program on a particular machine configuration.

We believe that fine-grained dataflow graphs offer one of the cleanest formulations of parallel computation. Dataflow graphs *avoid overspecifying* the instruction execution sequence by dictating only the essential data dependences. An actual execution of the program may impose additional ordering, but all such orders obey the data dependences. Thus these graphs capture all sorts of parallelism—vector, inner loop, outer loop, producer-consumer, *etc.*

The first step is to interpret the dataflow graph in an *ideal* environment that imposes no additional ordering and thus reveals an amalgam of the potential program parallelism. Then, we refine our estimations by selectively accounting for finite processor resources and non-zero communication latency. At the center of this analysis is a time-independent formulation of computational progress.

[2]Indeed, vector processing is a restricted form of parallel processing.

5.3.1 Idealized Execution

The dataflow graphs produced by the Id compiler are based on a fixed set of schema and rules for composition that ensure deterministic behavior on all execution orders [14]. One such execution order of interest is the so-called *infinite-processor* or *ideal execution order* that follows the simple execution rule:

> At step j execute all instructions that have their operands, then proceed to step $j+1$.

We term each step a *parallel computation step*. While we do not specify the order in which instructions should be executed within a step, the set of instructions executed by step j is completely determinate. The *parallelism profile* of the computation is defined as a plot of the number of instructions executed in each parallel computation step versus the step number. By definition, all of the instructions in a step are completely independent of each other (they all have their required operands and don't need to intercommunicate). So the parallelism profile graphically reveals inherent parallelism in the program.

Figure 5.2 shows the parallelism profile of the SIMPLE code, a hydrodynamics and heat code that has been studied extensively both analytically and by experimentation[3]. This profile shows three iterations on a 20×20 mesh, while a typical production run performs 100,000 iterations on a 100×100 mesh. We can interpret the parallelism profile as the trace obtained from measuring an *ideal machine* with the following properties:

- Unit time per operation,
- Unbounded number of processors,
- Zero communication delay,

In the SIMPLE example, almost one and a half million instructions were executed; this took our ideal machine only 1,976 "clocks"—where every processor can complete an instruction every clock cycle. We term the number of steps required to complete the computation the *critical path*: there exists a sequence of 1,976 operations such that operation k requires the result of operation $k-1$ for all k. That is, there is at least one inherently sequential thread in the computation comprising the critical path number of instructions.

[3]See [15] and [16] for a detailed discussion.

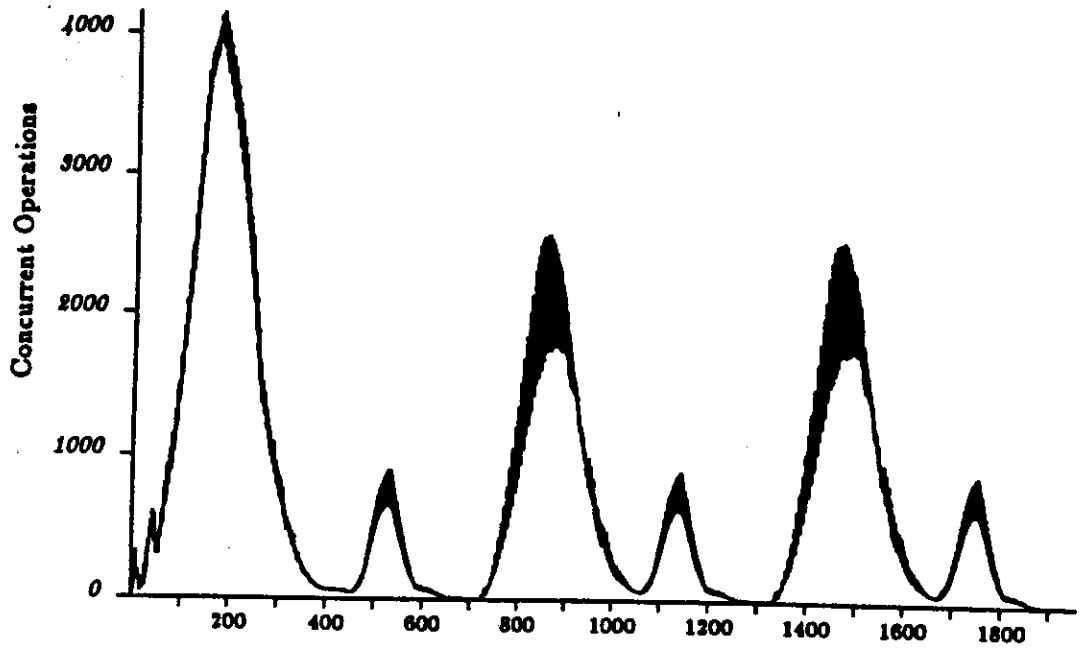


FIGURE 5.2
Parallelism profile for SIMPLE (3 iterations, 20×20).

We can also identify some important structural aspects of the parallelism. For example, as can be seen from iterations 2 and 3, there is no significant parallelism *between* the outer loop iterations of SIMPLE. The tremendous variability of the potential parallelism is also noteworthy, and typical of our experience with even the most highly parallel programs. This strongly suggests that reducing the profile to a single "average parallelism" number (analogous to the percentage vectorization) is apt to mask important dynamics, like the sections of inherently low parallelism. Yet even this highly idealized execution provides a hard upper bound:

No machine can complete the computation faster than the critical path number of instruction times.

Every instruction on the critical path depends directly on the result of its predecessor, thus these instructions *cannot* be rearranged to run in parallel nor can they be pipelined (pipelining requires independence of adjacent instructions). We can also conclude that a processor with an instruction pipeline depth of d cannot execute the program in fewer instruction times than the product of the critical path length and d.

5.3.2 Bounded Processor Profiles

Another clear consequence of the ideal parallelism profile is that a *single* processor machine will take at least as many instruction times to execute the program as the area under the curve—the total number of instructions executed. So now we know two extrema: the best time on an infinite collection of processors and the best time on a single processor. What about the space in between? Deriving the best performance achievable on n processors would establish an upper bound for evaluating real-time measurements. A "linear" speedup curve is often cited as the metric for performance comparisons for multiprocessors, but this is as naive as assuming that a linear improvement in the performance of a vector unit will cause a linear improvement in the execution of any application!

The intermediate points along the speedup curve have a fairly unremarkable relationship to the parallelism profile. Figure 5.3, shows the parallelism profile of the SIMPLE example where the number of processors has been limited to 1000, slightly greater than the average parallelism. As expected, the peaks of the profile have been truncated to 1000 parallel activities. The area under the curve is the same (the same number of instructions are executed in both cases) and the length of the critical path has increased from 1,976 to 2,763.

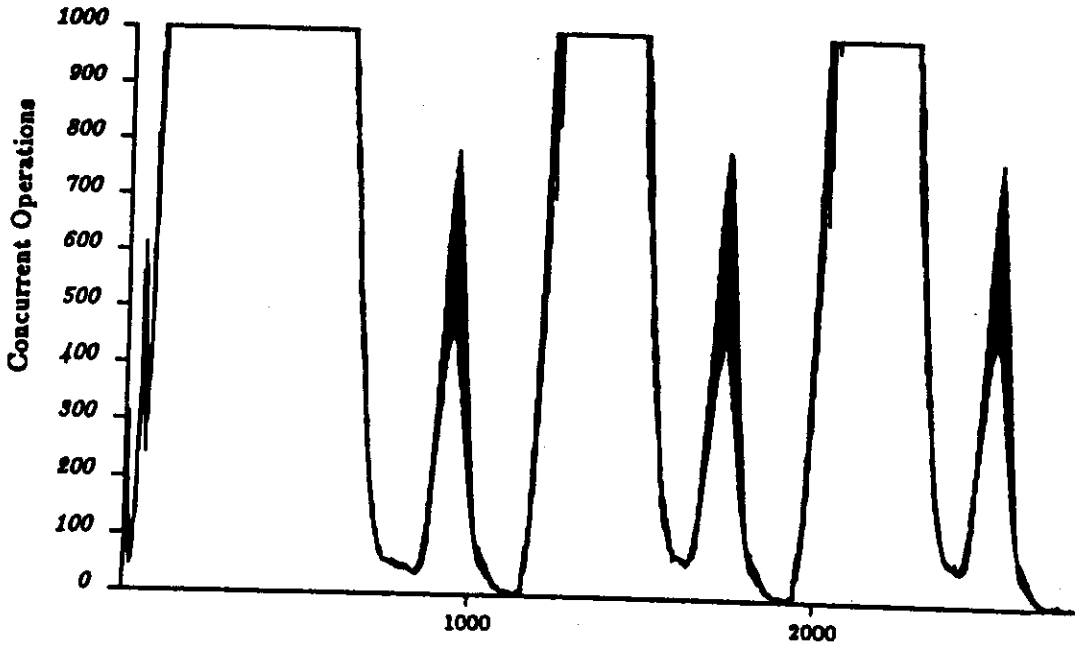


FIGURE 5.3
Parallelism profile for `SIMPLE` bounded to 1000 processors.

5.3.3 Accounting for Latency

The "processors" assumed in the bounded simulations really represent concurrent computations. For example, a pipelined processor of depth d would account for d "processors", as the pipeline computes d instructions in parallel. An alternative view is that the pipelined processor indeed represents a single "processor" but it introduces a *latency* between dependent computations equal to d time units. A model that incorporates latency could also account for the processor-memory and interprocessor communication times. Figure 5.4 shows the parallelism profile for the SIMPLE example where the number of processors has been limited to 100. The top profile has latency of zero while the bottom has latency of 10—it takes 10 time steps for the result produced by an instruction to be distributed to any dependent instructions.

Surprisingly, the critical path is increased by a factor of two, not a factor of 10. This suggests that excess parallelism can be invested in tolerating latency, and that the program has enough parallelism to perform well on a realistic machine comprising 100 processors.

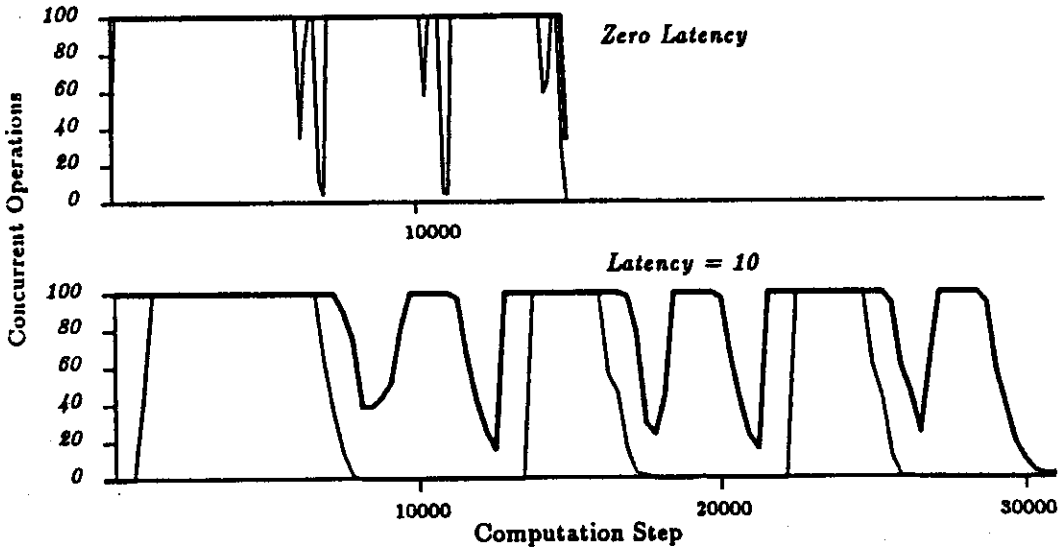


FIGURE 5.4
Parallelism profile for SIMPLE bounded to 100 processors.

5.3.4 Estimating Speedup from Ideal Profiles

Arvind, Culler and Maa [17] have shown that good estimates of the speedup expected on a finite number of processors with a given latency can be obtained from simple analysis of the *ideal parallelism profile*! Thus, we can avoid the costly computation of numerous scenarios of bounded processors and fixed latency. The derivation of the estimates is beyond the scope of this discussion, but we have shown the results on the SIMPLE example (this time, one iteration on a 32 × 32 mesh) in Figure 5.5. The curves (parameterized by latency) show the estimated performance derived from analysis of the ideal execution profile. The discrete points are the results of actually simulating the machine configuration on our graph interpreter.

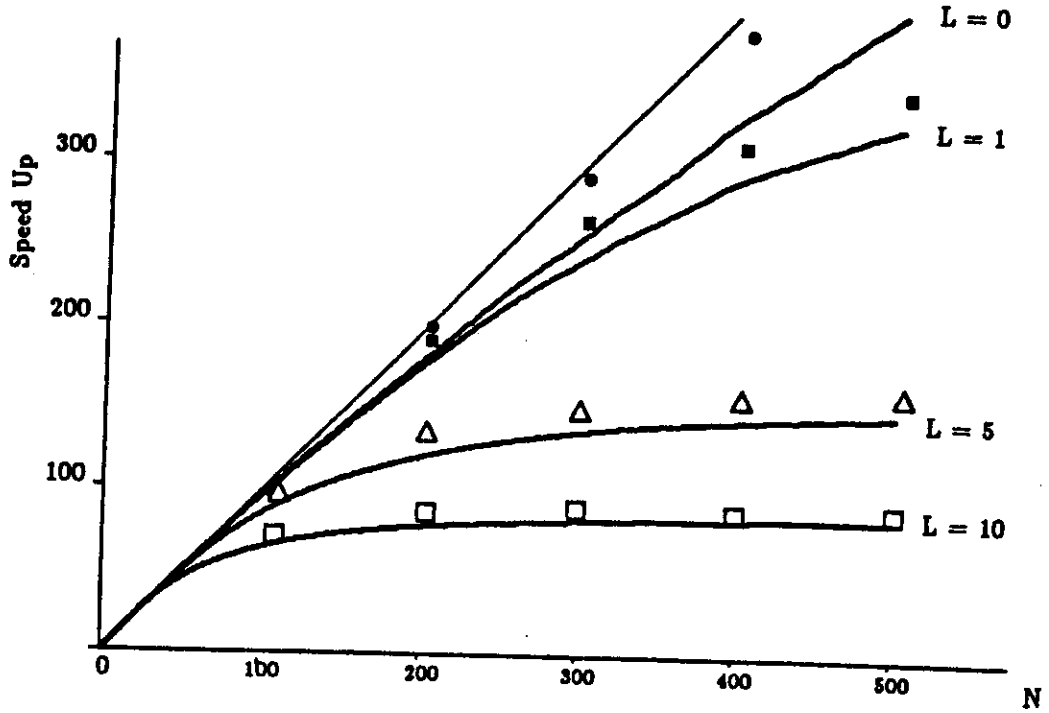


FIGURE 5.5
Speedup estimates for SIMPLE (1 iteration, 32 × 32) from the ideal parallelism profile.

5.4 Hardware Support for Idealized Execution

The ideal parallelism profile provides a solid foundation for predicting the performance of an application on a given multiprocessor configuration. However, even computing a single profile can be very expensive as it is a true simulation of the problem on given set of input data. Indeed, the idealized simulation produces the same numeric result as any real-time execution (Church-Rosser, again). This can be interpreted as a flaw—if the parallelism profile is highly dependent on the input data (it certainly is a function of problem size) then many ideal simulation runs would be required to characterize the program.

We believe that the parallel computation step abstraction can play such an important role in application development that hardware support for idealized execution should be developed. We have arrived at a simple hardware extension for the Monsoon dataflow that efficiently implements a parallel computation step, and appears to have general applicability among various dataflow architectures.

Monsoon, like most dataflow machines, is a collection of pipelined processing elements. The processing element pipeline independently transforms computation state descriptors called *tokens*. Tokens are the means by which data are communicated from instruction to instruction. A Monsoon processing element can consume a token and produce zero, one or two result tokens. Thus, a large buffer or *token queue* is required to store the transient of excess tokens that are generated during periods of high parallelism. Please refer to Figure 5.6. The token queue is analogous to a work queue. A processing element keeps consuming tokens from the queue and adding new ones. The only time it idles is when the queue is empty.

Our hardware trick is simple. Rather than have a single token queue we equip each processor with *two* queues. A parallel computation step works as follows (see Figure 5.7):

1. Let Q_0 contain the tokens that were produced by computation step j.
2. All processing elements consume the tokens from Q_0 but insert any result tokens into Q_1. This continues until all processors have emptied their respective Q_0 and all idle. Each processor counts the number of instructions executed.
3. The roles of Q_0 and Q_1 are reversed, and the next computation step is performed: dequeuing from Q_1 and enqueuing into Q_0.

Interprocessor communication is also accounted for: a token, which is produced by a processor and destined for another processing element, is automatically (by the normal hardware rules) transmitted over the network to the

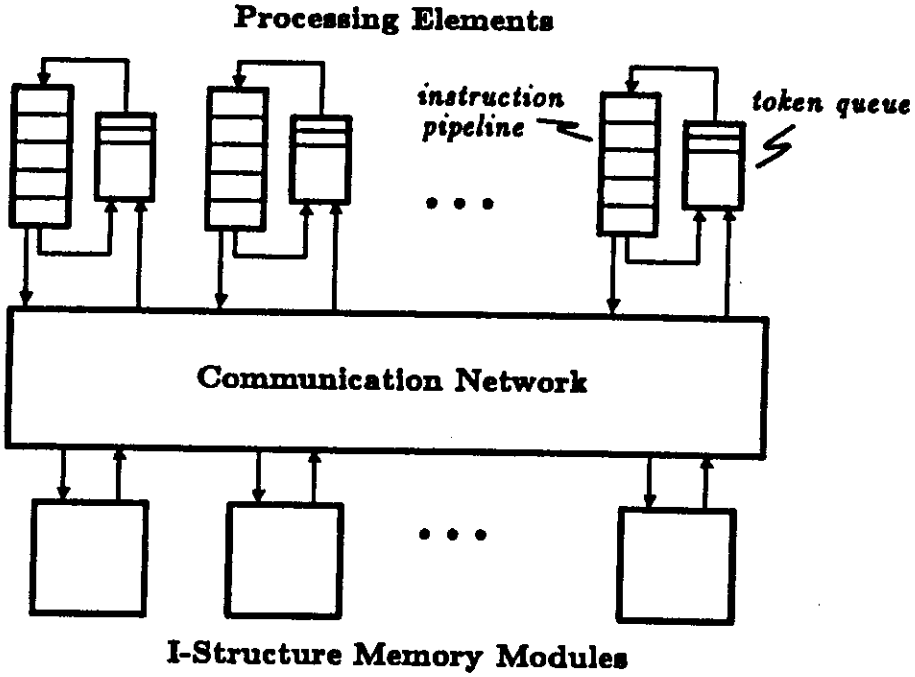


FIGURE 5.6
A collection of pipelined monsoon processing elements.

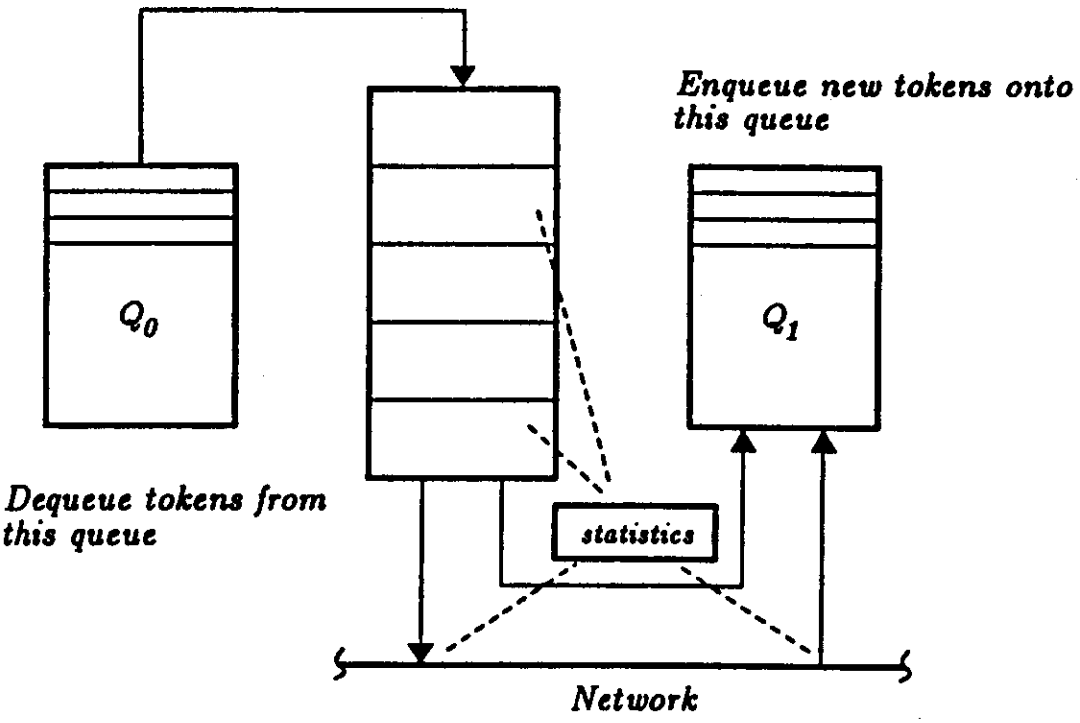


FIGURE 5.7
A modified monsoon processor for performing parallel computation steps.

destination processing element, that simply enqueues the token until the next computation step.

Notice that the actual processing of instructions is performed at full speed by each processor. The overhead from this technique arises from the implied barrier synchronization—no processor can flip its queues until all other processors have come to an idle. This overhead is incurred only as many times as there are steps on the critical path, so programs that exhibit even modest amounts of parallelism ought to run fairly well. We would view executing within an order of magnitude of real-time acceptable, and expect a typical degradation of about a factor of three. In contrast, even our fastest graph interpreters execute in no better than 0.1% real-time.

The parallel computation step can also provide an unexpected benefit for debugging programs. Church-Rosser only guarantees that the same *set* of errors will be encountered from run to run, but the *order* in which the errors occur is not determinate. In fact, the strategy of "stopping" the machine upon detecting an error will seldom yield a reproducible state. The only certain way of consistently ending up in the same error state is to let the program come to a natural halt (*i.e.*, wait for the effects of the error to propagate through the computation). Suppose, however, that we debug using parallel computational steps with the following error rule:

When an error is detected stop computation after completion of the current step.

This strategy has two desirable qualities:

1. **It is guaranteed determinate.** Execution will always halt on the same computation step with the same errors.

2. **The error does not propagate.** By definition, all instructions within a computation step are independent. Dependent instructions can only appear in the *next* step, which we never execute.

The technique applies equally well to programmer-inserted breakpoints and traces. In many ways, the parallel computation step is analogous to a *single-step* on sequential uniprocessors.

We think there are analogs of parallel computation step for course-grain models as well. In these settings, all tasks in the active task queue would execute until some blocking event, and then enqueue into the task queue for the next step. Statistics and error information could be gathered at the end of each step and *task parallelism profiles* could be generated.

5.5 Instruction Coloring

Between each computation step various processor and network statistics can be gathered from each processor for later analysis and display. They include:

- instruction mixes, *e.g.*, fixed and floating point operations, fetch, store,
- processor performance, *e.g.*, idle cycles, cache misses, pipeline bubbles, and
- network performance, *e.g.*, message counts, blocking statistics.

This same set of event counters can be sampled periodically during real-time operation to provide a trace of runtime performance data—with the usual concern over sufficient sample frequency. However, the sampling *between* computation steps during idealized execution is sufficiently frequent as the instructions processed during a step are, by construction, completely independent. So nothing "interesting" would be revealed by sampling the statistics counters in the middle of a parallel computation step.

We note that a programmer might also be interested in displaying statistics other than total instructions executed in the style of parallelism profiles. For example, the user might want to display only the floating-point operations per step, or the aggregate network traffic to identify potential network hotspots that might be exposed during real-time execution.

One often desires to filter the data to display the contribution of a *subset* of procedures (maybe just a single function) to the overall execution trace. This is accomplished on Monsoon by replicating the instruction mix counters in each processing element and then adding a field to every instruction (its *color*) that specifies which set of counters should be updated. All instructions belonging to the subset of procedures to be monitored would be assigned the same color.

For an experimental processor like Monsoon, the added hardware of this approach is tolerable. Replication of the statistics counters is relatively inexpensive. Because all instruction mix counters are mutually exclusive (only one counter from one color set will updated during a given cycle), we can store the current counts in a high speed memory and have only a single incrementer. Adding more colors only involves increasing the size of the memory. A more practical limit on the number of colors is dictated by the amount of time required to read all of the sets of counters during each sample period. Of course, we must also pay for a few extra bits on *every* instruction memory location. We have found four bits (sixteen colors) to be a reasonable compromise.

5.6 Performance Monitoring

At the other end of the development cycle is the performance tuning of a working program. In the purest of worlds, the ideal execution profiles would accurately predict the real-time performance of the application. At least, the ideal execution profiles provide reasonably tight bounds on the *best* we can hope to do on a given machine configuration. So when we observe performance that is poor, but matches prediction, we know not to indict the machine. If this performance is inadequate then the only choice is to adjust the application—for example by reformulating the problem with algorithms that exhibit greater parallelism.

What happens when the real-time performance is significantly *worse* than predicted? The first step is to try to relate the real-time statistics back to the predicted execution profiles. Instruction coloring can help identify a computation phase by indicating when certain procedures are active. Attention can then be focused on those phases where the measurements and predictions diverge—the places where aggregate machine utilization is significantly lower than projections. There is apt to be a small number of explanations for the divergence, as we have already factored out the inherent limitations of computation:

- **Poor workload distribution.** Some processors have a disproportionate share of procedure activations, inducing other processors to idle.
- **Excessive network contention.** The network traffic is imbalanced leading to restrictions in bandwidth or inordinate increases in latency.
- **Memory contention.** Memory requests are unevenly distributed causing certain modules to become bottlenecks.

We believe that of the divergence better analysis of the ideal execution statistics can also help reveal the cause of the divergence. After each computation step we can query the memory modules and discover any excessive loading. Similarly, the network elements can be examined to determine relative traffic and contention. It should then be possible to determine at each step the rate limiting term: be it the processors, network or memory.

All of this may be interesting to the programmer but not very helpful unless there is some way to *fix* the problem. Unfortunately, we will have to wait until we have gained more experience with large configurations before we know what mixture of programmer control directives are truly useful.

5.7 Conclusion

If we are to leave the reader with one thought, it is that it is possible to *systematically* develop a reliable, well-performing application for a parallel processor.

But this nice world of determinate debugging, theoretical performance prediction and relaxed execution instrumentation is only realizable if we make significant changes to our programming languages and constituent processor architectures. Declarative languages provide a natural basis for a robust parallel programming environment. Dataflow machines like Monsoon that provide especially efficient interprocessor synchronization and context switching can insulate the programmer from the vagaries of the underlying implementation.

References

1. Arvind and D. E. Culler, Dataflow Architectures, in *Annual Reviews in Computer Science*, Annual Reviews Inc., Palo Alto, CA, Vol. 1, pp. 225–253, 1986.
2. J. Backus, Can Programming be Liberated from the von Neumann Style?, *Communications of the ACM*, 21(8), August 1978.
3. J. R. McGraw, SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2, Technical Report M–146, Lawrence Livermore National Laboratory, March 1985.
4. R. S. Nikhil, Id Nouveau Reference Manual, Part I: Syntax, Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.
5. G. M. Papadopoulos, Implementation of a General Purpose Dataflow Multiprocessor, Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.
6. A. H. Karp, Programming for Parallelism, *Computer*, 20(5), May 1987.
7. J. R. McGraw and T. S. Axelrod, Exploiting Multiprocessors: Issues and Options, in *Programming Parallel Processors*, Addison-Wesley, Reading MA, 1988.
8. D. A. Mandell and H. E. Trease, Parallel Processing a Real Code — A Case History, Technical Report LA-UR 88-1836, Los Alamos National Laboratory, May 1988.
9. L. A. Henderson, R. E. Hiromoto, O. M. Lubeck and M. L. Simmons, On the Use of Diagnostic Dependency-Analysis Tools in Parallel Programming: Experience Using Ptool, Technical Report LA-UR 88-1968, Los Alamos National Laboratory, Los Alamos, NM 87545, June 1988.
10. G. Huet, Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *Journal of the Association for Computing Machinery*, 27(4):797–821, October 1980.

11. Arvind, R. S. Nikhil and K. K. Pingali, Id Nouveau Reference Manual, Part II: Semantics, Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

12. Arvind, K. Ekanadham and D. E. Culler, The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures, in *CONPAR 88, Manchester, England*, 1988.

13. K. Ekanadham, Arvind and D. E. Culler, The Price of Parallelism, in *Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30–June 2 1988.

14. K. R. Traub, A Compiler for the MIT Tagged-Token Dataflow Architecture, Technical Report LCS TR–370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

15. Arvind and K. Ekanadham, Future Scientific Programming on Parallel Machines, in *Proceedings of the International Conference on Supercomputing, Athens, Greece*, June 1987.

16. W. P. Crowley, C. P. Hendrickson and T. E. Rudy, The SIMPLE Code, Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

17. Arvind, D. E. Culler and G. K. Maa, Programming for Parallelism, in *Computer*, 20(5), May 1987.