Quartz: A Tool for Tuning Parallel Program Performance

Thomas E. Anderson and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle WA 98195

September 1989

Abstract

Initial implementations of parallel programs typically yield disappointing performance. Tuning to improve performance is thus a significant part of the parallel programming process. The effort required to tune a parallel program, and the level of performance that eventually is achieved, both depend heavily on the quality of the instrumentation that is available to the programmer.

This paper describes Quartz, a new tool for tuning parallel program performance on shared memory multiprocessors. The philosophy underlying Quartz was inspired by that of the sequential UNIX tool gprof: to appropriately direct the attention of the programmer by efficiently measuring just those factors that are most responsible for performance and by relating these metrics to one another and to the structure of the program. This philosophy is even more important in the parallel domain than in the sequential domain, because of the dramatically greater number of possible metrics and the dramatically increased complexity of program structures.

The principal metric of Quartz is *normalized processor time*: the total processor time spent in each section of code divided by the number of other processors that are concurrently busy when that section of code is being executed. Tied to the logical structure of the program, this metric provides a "smoking gun" pointing towards those areas of the program most responsible for poor performance. This information can be acquired efficiently by checkpointing to memory the number of busy processors and the state of each processor, and then statistically sampling these using a dedicated processor.

In addition to describing the design rationale, functionality, and implementation of Quartz, the paper examines how Quartz would be used to solve a number of performance problems that have been reported as being frequently encountered, and describes a case study in which Quartz was used to significantly improve the performance of a CAD circuit verifier.

Index Terms – multiprocessor, performance, measurement, parallel programming, tuning

1. Introduction

The primary motivation behind building multiprocessors is to cost-effectively improve system performance. Even moderately increasing a uniprocessor's power can require substantial additional design effort as well as faster, and thus more expensive, hardware components. By contrast, once a scheme for interprocessor communication has been added to a uniprocessor design, the system's peak processing power can be increased linearly simply by adding processors. The incremental cost per processor has been reported to be as little as 15% of the initial system cost for small to moderate numbers of processors [Thacker et al. 1988], and larger but still close to linear for greater numbers of processors [BBN 1985; Pfister et al. 1985].

Of course, multiprocessors lose their advantage if this processing power is not effectively utilized, and while it is relatively easy to get good performance when there are multiple independent sequential job streams, it can be difficult to achieve good performance from parallel applications. The literature describes many attempts to parallelize algorithms and applications. (Burkhart and Millen [1989] survey some of this experience.) Typically, an initial implementation results in disappointing performance, but significant improvements can be obtained with subsequent effort. Sequent, for example, tells of a major customer whose first attempt at parallelizing a "dusty deck" resulted in a program that, given 8 processors, executed only 50% as fast as the original sequential program. After considerable effort by skilled engineers, nearly perfect speedup (a factor of nearly 8 on an 8 processor machine) was achieved [Rodgers 1986].

A major factor contributing to the large amount of skilled effort typically required to achieve good parallel program performance is the shortage of good performance analysis tools. In the absence of such tools, performance problems must be identified through a combination of guesswork, folklore, and application-specific instrumentation. The subject of this paper is the design rationale, functionality, implementation, and use of a new tool for tuning parallel program performance.

The philosophy underlying our work is that an effective tool for tuning parallel program performance must be based on a clear view of the causes of poor performance, and on a specific methodology for improving that performance. By being selective about what it measures and presents, the tool can focus the programmer's attention on the information needed to tune performance, eliding details about second-order effects.

This material is based on work supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, the Washington Technology Center, Digital Equipment Corporation (the Systems Research Center and the External Research Program), and IBM (a Graduate Fellowship).

Authors' address: Department of Computer Science and Engineering FR-35, University of Washington, Seattle WA 98195; (206) 545-2675; *tom/lazowska@cs.washington.edu*. Quartz source code is available by anonymous ftp from cs.washington.edu.

Measurement efficiency also is improved by designing the tool to record just the important behavior.

Selectivity is possible because, although parallel performance in general is much more complex than sequential performance, experience (discussed in Section 3.2) suggests that poor parallel performance typically arises from a relatively small number of factors. For applications whose performance is dominated by periods of limited parallelism, the tool should identify those sections of code that account for most of this time so that these sections can either be re-structured to increase concurrency or optimized to reduce their impact on overall performance. Time spent spin- (or "busy"-) waiting must be correctly represented, since spinning processors appear to be busy even though they are not computing useful results. Finally, for applications with large amounts of real parallelism, performance can only be improved by optimizing (but not further parallelizing) the code that executes for the greatest proportion of time.

Based on these observations, we propose a new way to view parallel program performance on shared memory multiprocessors. We focus on both the total processor time devoted to each section of code and the number of other processors concurrently busy (as opposed to idle or spin-waiting) when that section of code is being executed. Routines can be compared by considering their *normalized processor time*: their processor time divided by the concurrent parallelism (a precise definition is given in Section 3). This metric usually reflects the relative importance of different sections of code to the overall elapsed time of the program: a routine that executes while no other processors are busy can be responsible for a large percentage of the runtime of a program, even though it uses only a small fraction of the total processor time. Further, by measuring both parallelism and processor time, we can determine whether performance can be improved by re-structuring to increase parallelism, or only by simple optimization.

We tie these measurements to the logical structure of the program's procedures. Good engineering practice demands that large programs, whether sequential or parallel, be structured using hierarchical abstractions [Graham et al. 1982]. We report our performance measures for each procedure and for all the work done on its behalf, either synchronously via a normal procedure call or asynchronously through parallelization. The programmer can use this to walk through the hierarchy, focusing on just those procedures that, along with their children, account for most of the poor performance. We expect that for parallel programs as for sequential ones, a relatively small proportion of the code will be responsible for most of the runtime.

These measurements can be made efficiently on a shared memory multiprocessor by checkpointing to memory the number of busy processors and the state of each processor, and then statistically sampling this information using a dedicated processor.

We have developed a tool to test these ideas, called *Quartz*. Quartz was built by modifying the application-level thread package described in [Anderson et al. 1989]. Quartz uses only the normal profiling support available on UNIX-like shared memory multiprocessors; it currently runs on the Sequent Symmetry multiprocessor [Sequent 1988].

The remainder of the paper discusses these ideas in more detail. Section 2 examines existing measurement tools for tuning program performance. Section 3 describes Quartz: its motivation, its functionality, its applicability to a number of performance problems that have been reported as being frequently encountered, and its implementation. Section 4 describes a case study in the use of Quartz to improve the performance of a specific parallel application, a CAD circuit verifier. Section 5 considers the implications of our work for the monitoring of sequential programs and non-shared-memory multiprocessors. Section 6 summarizes our results.

2. Existing Tools for Tuning Program Performance

2.1. Tools for Sequential Programs

The philosophy underlying Quartz owes much to the experience of UNIX gprof [Graham et al. 1982], a tool for tuning the performance of sequential programs running on uniprocessors.

Years of experience tuning sequential programs indicate that the major difficulty is focus: it is relatively easy for the programmer to improve the processing time of a small section of code, but lots of effort is commonly wasted in the wrong places – tweaking code that has only a small impact on overall performance.

Gprof's solution is to highlight the "hot spots" of the program, and to do so in a way that exploits the hierarchical structure of large programs. Gprof presents to the programmer the total processor time of each procedure, including time spent on its behalf if it calls other routines. With this information, the programmer can tune the program in a top-down fashion, focusing effort on those functions that have the greatest impact on performance.

Gprof is relatively efficient. It periodically interrupts the program to sample the program counter, thereby estimating the execution time of each procedure. While sampling produces only an estimate, the approach is most accurate just where it needs to be: for those routines where the program spends most of its time. Gprof also collects the call graph: who called whom how many times. This is done by using compiler support that makes each procedure execute a monitoring routine during its prologue. Gprof then computes its central metric, the processor time spent on a procedure's behalf, by making the assumption that all calls to the same procedure take the same amount of time. Processing time is propagated bottom-up from callee to caller according to the caller's proportion of the total calls.

Gprof seems so natural in retrospect that it is easy to forget the alternative approach taken by a number of other tools: to (expensively) measure everything that could conceivably be of relevance to program performance, and to report these measurements without concern for how they relate to each other or to the structure of the program.

Our goal for Quartz was to achieve a tool for tuning parallel program performance that is analogous to gprof in that it efficiently measures exactly what is important, and relates these measurements to one another and to the structure of the program. This philosophy is even more important in the parallel domain than in the sequential domain, because of the dramatically greater number of performance metrics and the dramatically increased complexity of program structures. The next two sub-sections discuss, in this context, existing approaches to tools for tuning parallel program performance.

2.2. Non-Integrated Tools for Parallel Programs

Many useful measures of parallel program performance have been proposed. Each provides a view of some important aspect of program behavior. However, in many existing tools, their lack of integration with each other and with program structure limits their usefulness.

Perhaps the simplest approach to parallel program measurement is to extend sequential UNIX gprof to run on a multiprocessor. In place of processor time on one processor, multiprocessor gprof measures the sum of the time spent on each processor [Aral & Gertner 1988]. Unfortunately, as we have noted, a procedure's total processor time is not related in a simple way to parallel runtime. Something more than a straightforward adaptation of sequential UNIX gprof clearly is necessary.

Another common tool displays the number of busy processors across time by periodically sampling the number of runnable processes. Assuming that all activity is due to the program in question, this allows the programmer to see if there are periods of time when there is too little parallelism to keep all the processors busy [Halstead 1986]. A significant shortcoming of tools like this is that it can be difficult to relate the periods of poor parallelism to specific sections of code that can be changed. Further, the fact that some processors are not doing useful work can be concealed, since spin-waiting processors appear to be busy.

The DEC SRC Firefly has a tool that measures the time spent waiting for each lock protecting a shared data structure [Thacker et al. 1988]. A lock ensures that threads manipulating the shared data structure have mutually exclusive access to it. This serial execution can limit performance. By measuring the wait time, the tool can determine which critical sections are the worst bottlenecks; these can then be re-structured to increase parallelism. This information is useful, but long waits for a lock will not affect performance if there is always other work to do during the wait, and monitoring a lock can increase the length of time that the lock is held, creating "artificial bottlenecks" when monitoring is enabled.

Quartz provides many of the same metrics as these tools, but correlates the metrics to one another and to the structure of the program. For example, Quartz measures not only how many processors are busy, but also which procedures execute during periods of low and high parallelism.

2.3. Trace-Based Tools for Parallel Programs

The issue of determining in advance exactly what information will be needed to tune the performance of a parallel program can be finessed by recording a trace of every interprocessor synchronization event with a timestamp of when the event occurred. The behavior of the program can be completely reconstructed from such a trace [Fowler et al. 1988]. † Arbitrary metrics (whether general or application-specific) can be computed using a common interface to the trace data. Finally, the metrics obtained from the trace can be integrated with each other and with the structure of the program.

One drawback to this approach is that both the collection and the post-processing of trace files is expensive. For programs that perform frequent synchronization, the trace files can be prohibitively large. Consider a program running on a hypothetical shared memory multiprocessor with 20 5-MIPS processors, each of which on average performs a monitorable event (such as acquiring or releasing a spin lock) every 500 instructions, and where each event record is 10 bytes. If the program being monitored executes for one minute, the trace file will be 100 megabytes. Similar estimates appear in [Malony et al. 1989] to justify hardware support for recording traces.

Nevertheless, tools have been developed that collect trace data and post-process it into useful measures. We argue later that the much of the information provided by these tools can be measured or approximated more efficiently by sampling.

Monit [Kerola & Schwetman 1987], for example, uses a trace file to compute the behavior of higher-level objects, such as the number of busy processors or the number of threads waiting to enter each critical section. The behavior of each object is then plotted on a timeline. After identifying those phases of execution with few busy processors, the programmer can visually correlate these phases to the behavior of other objects (discovering, for example, that parallelism is low while a specific critical section has a large number of waiting threads).

Although Monit's display is at a higher level than the raw trace data, it still can present a massive amount of data to the programmer. Only a few timelines will fit on a screen at a time, but Monit provides little help in identifying those containing information relevant to the measured lack of parallelism. As the complexity of the application increases, so does the number of objects to monitor, making focusing more important.

IPS [Miller & Yang 1987] attempts both to guide the programmer to performance problems and to provide useful statistics about those problems. Its central focusing metric is the time each process and procedure spends executing the critical path. The critical path is the longest path through the task graph – the series of sequential pieces of code (that cannot be done in parallel because they communicate one to the next) that takes the longest to execute. By definition, the elapsed execution time of the program can be reduced only by shortening the length of the critical path.

One drawback to critical path analysis is its expense: it requires a complete trace of all interprocessor communication. (To be fair, IPS was originally designed to measure programs running on local area networks of uniprocessors. Because of the high latency and low bandwidth communication on these systems, only programs with relatively infrequent synchronization can run efficiently. Under these conditions, the size of the trace file would be manageable.) Yet critical path analysis is still just a heuristic: there is no guarantee that reducing the critical path will actually reduce the execution time of the program. There may be another path through the task graph with almost the same length that will be unaffected by the change. Critical path analysis also does not indicate *how* to reduce a procedure's completion time. One way is to reduce its sequential execution time. Another is to parallelize it. But parallelization will only be of benefit if there are idle processors to exploit when the procedure runs.

3. Quartz: Its Functionality, Applicability, and Implementation

Our goals for a tool for tuning parallel program performance are:

- It should identify the sections of source code most responsible for poor performance.
- It should present its measurements hierarchically, to allow top-down tuning according to the logical structure of the program.
- It should measure parallelism (properly representing spin-waiting as a loss of parallelism) and it should tie this to the source code, identifying where re-structuring to increase parallelism is necessary and where code optimization is appropriate.

† Originally, Fowler et al. implemented trace collection to aid debugging the correctness of parallel programs; they then showed that the same support could be used for program tuning. An implication of our work is that the support needed for correctness debugging is not necessary for performance tuning.

- It should measure program behavior in sufficient detail to provide some insight into the type of re-structuring that will work.
- It should do all this efficiently and without significantly affecting the behavior of the measured program.

Quartz meets these criteria. Before describing it, we must define some terms. A thread (or "lightweight process") is a sequential execution stream; it is the basic unit of parallel work. A thread starts another thread by giving it a procedure to run; the initial thread continues in parallel with the created thread. Thread creation is thus essentially an asynchronous procedure call. If threads are implemented as part of an application library, they can be within an order of magnitude of the cost of a procedure call [Anderson et al. 1989]; they can thus be used for procedure-level parallelism.

Threads can synchronize with one another. One type of synchronization object is a lock, used to ensure mutually exclusive access to a shared data structure. Another is a condition or barrier used to enforce a data dependency, as when one thread reads data produced by some other thread. In both cases, synchronization may cause the thread to wait, either because the lock is busy or because the data it requires has not been produced. Since there may be more threads than processors, a waiting thread has a choice: either spin until the lock is free or the data is available, or block, relinquishing the processor to run another thread. Thus, there is a difference between a program's *effective* parallelism, the number of busy (not idle or spinning) processors, and its *nominal* parallelism, the number of runnable threads, some of which may be spinning. Our measurements refer to the activity of just the processors executing the application, and not any processors concurrently executing other applications.

The remainder of this section is divided into three sub-sections. The first describes the functionality of Quartz: the specific metrics that it reports. The second shows how these metrics can be used to detect and fix a number of performance problems that have been identified by others as commonly occurring. The third provides an overview of the implementation of Quartz. A case study in which Quartz was used to tune a specific application is described in Section 4.

3.1. The Functionality of Quartz

The principal measurement made by Quartz is normalized processor time, defined in Equation 1, where P is the number of processors. Measuring this weighting function for every procedure allows us to compare them according to their effect on overall performance.

$$\sum_{i=1}^{P} \frac{\textit{Processor time with i processors concurrently busy}}{i}$$

Equation 1: Normalized Processor Time

To understand the rationale for this metric, consider a program with two functions, one that always executes sequentially when no other processors are busy and one that computes its result completely in parallel. If each function takes the same total processor time, the sequential one requires a factor of P greater elapsed time (where P is the number of processors) and will have a much greater impact on program performance. If the two functions take the same elapsed time, then the same percentage improvements in either will have equal impact on performance.

Further, knowing the concurrent effective parallelism while a routine executes is more important than knowing the effective parallelism it generates: a serial routine that is always overlapped with other computation will have little effect on performance compared to a serial routine that always executes by itself. This is despite the fact that the *elapsed time* of the two functions is the same.

Normalized processor time reflects these observations. Quartz also measures other program behavior; the principal performance measurements in Quartz are summarized in Table 1.

For each procedure, synchronization object, and thread, and for the work done on its behalf:
Normalized processor time. Each term of the sum is measured separately.
Elapsed time spent in each state (busy, spinning, blocked, or ready), along with the average and the distribution of the number of runnable threads while it is in each state.

Table 1: Principal Performance Measurements in Quartz

To focus the programmer's attention on those areas of the program that have the greatest impact on performance, we sort procedures by their normalized processor time plus that of the work done on their behalf. This includes work done synchronously or asynchronously (via threads). The program's top-level procedure, then, has a normalized processor time equal to the elapsed time of the program; the functions it calls to do the work of the program divide that weight among them. Quartz's ordering is analogous to what gprof does with processor time, except that Quartz uses a weighting function related to parallel performance. In both Quartz and gprof, the programmer can trace performance top-down through the program.

Normalized processor time indicates *where* improvements can be made. Quartz also provides information about *what* can be done to improve performance. Part of this information comes from the measurement of concurrent parallelism needed for normalized processor time. This indicates whether performance can be improved by re-structuring to increase effective parallelism, or only by simple optimization. Certainly, procedures that always execute when all processors are busy will not benefit from further parallelism.

Given that re-structuring is necessary, an accounting of the *elapsed time* spent by a procedure can help identify what kind of re-structuring is most likely to succeed. Quartz measures each procedure's elapsed time spent busy, spinning, blocked, or ready to run, along with the average and the distribution of the number of threads that are available to run while the procedure is in each state. For example, if a procedure is busy executing and there are few other busy processors, then the nominal parallelism indicates whether the other processors are idle or spinning. If idle, then performance can be improved by parallelizing the procedure (creating threads to do its work), provided this is possible. If spinning, then there is no benefit to creating more threads. Similarly, threads that are blocked or spinning represent deferred work; if the program were re-structured to reduce or eliminate waiting, then parallelism would increase.

Quartz makes the above measurements separately for each procedure, synchronization object, and thread. Measurement based purely on procedures would ignore the fact that parallel performance can depend more on the data object passed to a procedure than on the implementation of the procedure itself. It is only mildly interesting to know the total time spent waiting at all barriers; it is much more useful to know that some specific barrier accounted for most of that time. As a special case, the time spent executing in a critical section is attributed to the lock on that critical section. (Quartz also measures queue length distributions for synchronization objects.) Per-thread information allows us to determine if different threads executing the same procedure (on different data objects) have different performance.

By measuring synchronization objects and threads in the same way as procedures, we can present the programmer a uniform focusing metric. All are ranked in the same way to simplify tracing performance through the program. For example, if contention for a lock determines performance, the lock object will have a high normalized processor time since its critical section is always executing while few other processors are. (Spin time is factored out in computing normalized processor time.)

An important difference from gprof is that we explicitly measure the work done on behalf of a procedure or lock object. Gprof explicitly measures only the work done within each procedure, making the assumption that the processor time of its children is independent of who called them. Gprof uses the call graph (who called whom, and how many times) to propagate processor times from callee to caller according to the caller's percentage of the total calls to the callee. We cannot make a similar assumption. The effective and nominal parallelism while a procedure executes depend not only on what that procedure does, but also on the parallelism when it is called. Different calls to a low-level formatting routine might have vastly different concurrent parallelism. Still, even though it is not useful for propagating our measurements, Quartz does record the call graph (including calls to/from synchronization objects) to help the programmer in tracing performance top-down through the program.

3.2. Detecting Frequently Encountered Performance Problems Using Quartz

In this section, we argue by example that Quartz is useful for detecting and fixing a number of common parallel program performance problems. We asserted in Section 1 that although parallel performance in general is much more complex than sequential performance, experience suggests that poor parallel performance typically arises from a relatively small number of factors. One piece of evidence for this is Table 2, which lists the performance problems most frequently encountered by three "vendors" of parallel computing systems who participated in a working group concerned with "Sources of Performance Degradation" at the *NSF/CMU Workshop on Performance-Efficient Parallel Programming* in 1986. The key observation is that none of the problems involve subtle timing issues that might require a complete trace of synchronization activity.

The first issue in tuning any parallel program is to identify which segments are responsible for the poor performance. As with sequential programs, we would expect that of the large number of functions in a parallel program, a relative few will be responsible for most of the program's runtime. By computing a weight based on both processor time and parallelism, and by accounting for all of the activity done on behalf of a function, Quartz allows the programmer to walk through the program hierarchy to find those few functions. Once the general area of difficulty has been located, the approach to tuning depends on the situation:

3.2.1. Functional Decomposition

Some computations have several functionally distinct parts, each assigned to a distinct processor. An example of this is a pipelined compiler: separate threads (and processors) execute the scanner, parser, optimizer, and code generator, streaming results one to the next (Sequent #1 in Table 2). Performance difficulties usually relate to load imbalance. If one phase has more work to do than the others, the others must sit idle waiting for it. If the optimizer is the bottleneck, the scanner and parser will have to wait for buffer space to forward their partial results, while the code generator must also periodically wait for results to be completed.

Quartz would identify this problem: the thread executing the optimizer would have a longer execution time, spend more time executing when few other processors are busy, and thus have a larger normalized processor time than the threads executing the other phases. Other tools would also handle this case. For example, a display of processor activity across time would show that the processor executing the optimizer was always busy, while the other processors sometimes wait. (Of course, many tools that display processor activity fail to relate processors to procedures.) Similarly, a critical path analysis would show that the execution of the optimizer constituted most of the critical path.

It is also easy with Quartz to identify the phase that is the secondary bottleneck – in the compiler example, the one that would limit performance if the optimizer were improved. If the code generator ran for almost as long as the optimizer, then it would have slightly less normalized processor time, indicating that attention should be focused on improving both phases. It is difficult to extract this information from a timeline, since all phases but the optimizer periodically block. Critical path analysis only identifies the primary bottleneck, so iteration would be required.

Another performance problem with pipelines is starvation (Harris #3). This occurs if the problem size is small relative to the time for each phase to start streaming results. In this case, the

Sequent

1. A problem decomposition that puts most of the work in one thread (e.g., the optimizing phase of a concurrent compiler or a "busy" region in a ray-tracing algorithm), so that little real concurrency can be realized.
2. Memory thrashing due to a poor choice of operating system parameters.
3. Excessive I/O that is not overlapped with computation.
4. A synchronous software structure, such as might arise from a very large granularity, a producer-consumer relationship with a small number of buffers, or the use of an unnecessarily restrictive synchronization construct (e.g., barriers where critical sections would suffice).

Harris

1. Synchronization overhead.
2. Contention for shared variables, including counting semaphores, task queues, and the "problem heap".
3. Starvation due to a small problem size.

Warp

1. Excessive I/O that is not overlapped with computation.
2. Data dependencies in loops.

Table 2: Frequently Encountered Performance Problems
(NSF/CMU Workshop on Performance-Efficient Parallel Programming)

later phases spend much of the total time waiting to start; the earlier phases *finish well before the program completes*. Quartz would show that each phase spends much of its *elapsed time* idle. (Normalized processor time would highlight the first and last stages, since their work is the least overlapped with other stages.) A solution is to reduce the time before each phase starts streaming its first results.

3.2.2. Data Decomposition

Some programs compute the same function on many pieces of data. These programs can be parallelized by assigning different pieces of data to different processors. Unlike functional decomposition, each processor executes the same function at the same time. Again, a frequent issue is load balancing: the required computation may vary widely for different pieces of data. An example of this is ray-tracing where part of the picture has the majority of the activity (Sequent #1); another is a fluid dynamics computation where turbulence is concentrated in certain regions. In such situations, performance is limited by the processor assigned to the data regions with the longest execution times.

Whether a different thread is used to run the function on each object, or on collections of objects, Quartz will show if the execution times are balanced. If they are not, one of the threads will execute while other processors are idle, and there will be long average queue lengths at the barrier that checks that all threads have completed before continuing.

It can be difficult to relate the performance of a thread to the symbolic names of the data objects it works on, particularly in conventional (non-object-oriented) languages. For instance, the procedure a thread is to execute can be passed an index that only implicitly refers to the object it is to work on. As a result, we rely on the programmer to make this connection by providing a symbolic name when each thread is created. In an object-oriented language such as C++ we could extend Quartz not only to keep track of the symbolic names of data objects passed to threads, but also to explicitly take measurements for each procedure-data object pair, to allow both an object-oriented and a procedure-oriented view of performance. We intend to port Quartz to Presto [Bershad et al. 1988], a C++ based parallel programming system, to further explore this topic.

3.2.3. Synchronization

The need to synchronize the work of different processors can cause another class of performance problems. For instance, execution time is increased by the overhead of parallelizing the job: distributing work to various processors, serializing access to shared data structures, and enforcing data dependencies (Harris #1). Even if the program is perfectly parallel, this time can dominate. Fortunately, it is easy to measure. If there is a sequential version of the program, many of its functions will correspond to equivalent functions in the parallel version, and the execution time of each function can be directly compared to determine the effect of overhead. (The execution time added by monitoring must of course be factored out.) Alternately, given measurements of the performance of the thread package, the number of calls to each thread function, such as to create a thread or to acquire a lock, can be used to compute overhead.

Performance can also be affected by waiting for data dependencies to be satisfied (Warp #2; Sequent #4) or for access to a busy critical section (Harris #2). Waiting threads represent deferred parallelism; Quartz identifies this by measuring queue lengths and the average wait time (the total elapsed time spent waiting divided by the number of accesses to that synchronization object). For example, if a loop data dependency limits parallelism, there will be a long queue length at the point where the data dependency is enforced. Note that two of the examples of contention cited in Table 2 are for locks within the thread package; we measure contention for these locks in the same way that we measure locks in the application code.

Even if there are many threads waiting on a synchronization object, the question of whether it makes sense to re-structure the program to release that parallelism depends on whether the time is spent spinning or blocked, and on the nominal parallelism. When there are at least as many runnable threads as processors, blocked threads have no impact on performance beyond the initial context switch. Re-structuring to increase the number of ready threads does not help in this case. By contrast, spin-waiting always wastes processing cycles, regardless of the number of runnable threads, but if there are excess runnable threads then performance could be improved by blocking instead of spinning.

If re-structuring is necessary, the number of threads waiting at a lock can be decreased by any of: reducing the number of accesses (from the call graph), thereby reducing contention; decreasing the size of the critical section (its busy time); distributing accesses more evenly across time (if the queue length is sometimes zero and sometimes very long); or modifying the protected data structure to allow parallel accesses (for example, by giving each processor a separate copy).

Waiting for data dependencies can be reduced by computing the data earlier, or, if an overly restrictive synchronization construct was used, by allowing the thread to continue temporarily without it. Fuzzy barriers are a special case of the latter [Gupta 1989].

3.2.4. Input/Output

The time spent doing I/O was mentioned twice in Table 2 (Sequent #2, Warp #1). If a program reads a significant amount of data from an I/O device, then the reads should be overlapped with the computation; in other words, the reads should be started early so that they complete before the data is needed. The natural style, however, is synchronous: when the data is referenced, start a disk read and wait until it returns.

As a result of the operating system interface on the Sequent, the current implementation of Quartz measures time spent doing I/O as processor time, attributed to the procedure that performs the I/O. If the I/O is not overlapped, the relative importance of the time spent waiting in the kernel will be increased because processors will be idle waiting for the I/O to finish. Given kernel support for threads, Quartz could monitor the kernel disk queue as a normal synchronization object.

If a program spends a lot of time doing disk accesses, it may benefit from exploiting parallelism in the disk sub-system. Tuning a program's use of parallel disks is in many ways similar to tuning its use of parallel processors, although initial file placement is an issue as well. We expect that some of the techniques we have described in this paper could be applied to this problem.

3.2.5. Limitations

We have designed Quartz to measure only those aspects of program behavior that are needed to detect and fix frequently occurring parallel performance problems. The tradeoff is that Quartz therefore does not help with every performance problem that can occur in parallel applications.

When threads execute at the same time, Quartz weights each equally even though only one is on the critical path. As an example, consider a program with a critical section that restricts parallelism. The processor time spent executing outside of the critical section can appear important, because there are few other processors concurrently executing, even though reducing or parallelizing it will have no effect on program's performance. Although this can seem anomalous, Quartz's metric can help in this case by identifying code that may be a secondary bottleneck. We are currently investigating ways of augmenting Quartz's measurements to address this limitation. For instance, each time a processor goes idle, we could measure how long it stays idle, and then add that time to processor time of the code that causes the processor to become busy. This metric correctly handles busy critical sections (the time spent waiting for the lock would be attributed to the critical section), but it does worse than normalized processor time in other situations.

Quartz also does not measure thread scheduling decisions (although problems can sometimes be identified, for instance, if a thread spends a long time waiting for a processor and then executes serially) or contention for the bus or memory, even though these can affect performance.

3.3. The Implementation of Quartz

We have implemented Quartz on a Sequent Symmetry shared memory multiprocessor [Sequent 1988]. The Sequent runs DYNIX, a multiprocessor adaptation of UNIX. Since DYNIX processes are too expensive to use directly as threads, we built our system by adding monitoring code to the thread package described in [Anderson et al. 1989]. That thread package works by creating a DYNIX process for each processor, and then multiplexing threads onto the DYNIX processes. Our implementation did not modify DYNIX or the C compiler; it used only the support they provide for gprof.

Our implementation addresses the twin concerns of efficiency and accuracy. Because program tuning is iterative and interactive, a tool's usability depends on the elapsed time from program compilation to report production. Accuracy is trickier. Unlike sequential programs where the execution overhead due to monitoring is easily factored out, a change to a parallel program can alter its behavior in subtle ways. For instance, monitoring code that increases the time that a lock is held may increase the contention for the lock. Analogously, instrumentation added outside of a critical section will cause a net decrease in the contention for that critical section.

Our approach is to use statistical sampling by a dedicated processor. A set of processors executes the program normally, maintaining their state in shared memory by special code executed during thread operations and at procedure entry and exit. This state is then sampled by a dedicated processor that does not participate in executing the program. We impose no synchronization beyond hardware interlocks between the sampling processor and the other processors; rarely-accessed locks are used by the normally executing processors in building the call graph.

We sample by means of a dedicated processor rather than interrupts because interrupts cannot provide accurate correlations between processor state and overall program state. On the Sequent, as with most multiprocessors, interrupts are fielded by each processor asynchronously; by the time the program state is sampled, it may have changed in a way affected by the fact that there was an interrupt. For example, if the interrupted processor is holding a lock, the queue length at the lock will be greater than a purely random sample would indicate. Similarly, measuring a procedure's execution time directly with timestamps does not allow us to correlate that time to the number of busy processors. Of course, although it reduces the effect of monitoring on the measured program, sampling by a dedicated processor does not eliminate distortion. Updating state adds time to the computation, and even the recording of samples by the dedicated processor can increase bus and cache coherence traffic, thereby slowing other processors.

The nominal and effective parallelism are maintained in centralized counters, updated when a thread is added to the ready queue and when a processor becomes idle or starts or stops spinning. The counters are maintained with atomic increment and decrement instructions, to avoid making access to them a bottleneck. Most multiprocessors, including the Sequent, support such instructions.

In addition to the execution stack, we maintain a profile stack of monitored procedures for each thread. This allows us to record both the time spent in a procedure and the time spent on behalf of the procedure. The dedicated processor copies the number of busy and ready threads, copies the profile stack, and then bumps the appropriate measurement record for each different procedure on the stack. (Recursive procedures are counted only once.) While the stack may have changed between recording the number of busy threads and copying the stack, reducing consistency, sampling itself is only an approximation. In particular, we do not lock the profile stack to prevent changes from occurring; this would unnecessarily perturb the execution of the program. Note that locking would be harder to avoid if we were to sample directly from the execution stack since that would require tracing the chain of frame pointers.

The profile stacks are of fixed size, established at compile time. Overflows are caught, prevented, and later reported to the programmer. We expect that overflows will occur only rarely, since we push a procedure onto the stack only if the previous entry is different, eliminating immediately recursive calls, the most common cause of arbitrarily deep stacks. This also has the effect of reducing the work of the sampling processor.

We use only the normal compiler support provided for gprof. A monitoring routine is called in the prologue of each profiled procedure. Exactly as gprof does, we use this routine to update the count of calls to the procedure from its caller; we also push the procedure onto the profile stack. Because the compiler inserts only a prologue call, we manipulate the execution stack so that when the procedure returns, it returns first to our code that pops the profile stack, and then to the caller procedure. This is a bit inefficient, but easy to implement.

To simplify mapping from the entries on the profile stack to the measurement data for each procedure, we assign each procedure a unique ID. The gprof monitoring routine is passed a pointer to a procedure-specific location; this was originally used to count the number of calls to the procedure. After the program has been linked, we modify the object file so that each procedure's location holds a unique ID; this ID is what is pushed onto the stack and used to index the procedure's data record. Gprof, by contrast, uses the address in the program counter to index the data record for a procedure; this requires space proportional to the size of the code segment. By using procedure IDs, Quartz requires space proportional to the number of procedures times the number of processors.

The synchronization routines in our thread library are specially modified. Each object has a data record containing the call graph and execution time information. When the object is accessed, the call graph is updated and a pointer to the synchronization object is pushed; the pointer is popped when the thread no longer must wait. Locks are handled as a special case. Normally, the procedure that acquires a lock is the one that releases it, in which case we are safe to push the lock before it is accessed and pop it after it is released. This attributes the time spent waiting for and holding the lock to the lock object, and only adds two instructions to the inside of the critical section: setting the state of the thread to no longer spinning, and incrementing the number of busy processors.

When a thread is created, we copy the profile stack from the creating thread to the new thread. This allows the sampling processor to attribute execution time across asynchronous procedure calls.

Quartz has two other useful features. More than one processor can be dedicated to sampling to improve measurement accuracy and resolution. Quartz automatically removes from its measurements most of the time spent by the normally executing processors in updating their monitored state.

Our system does not currently provide for interactive control of which routines are to be profiled. This would be easy to add, but in truth, we doubt that it is the right approach. Aral and Gertner [1988] argue that gprof's overhead is too high to allow only compile-time control. They use this to motivate Parasight, a system for execution-time code modification and re-linking. But the overhead of gprof, and of our system, could be dramatically reduced with a small amount of compiler support. For example, most of the time in gprof is spent building the call graph; it crawls up the execution stack to find the caller address, hashes on it, checks the callee address, etc. A simpler method is to determine caller-callee pairs at compile time and to simply bump a statically allocated counter before each call. Calls made via function pointers, a rarer case, could use the current, slower approach.

4. A Case Study: Using Quartz to Tune a CAD Circuit Verifier

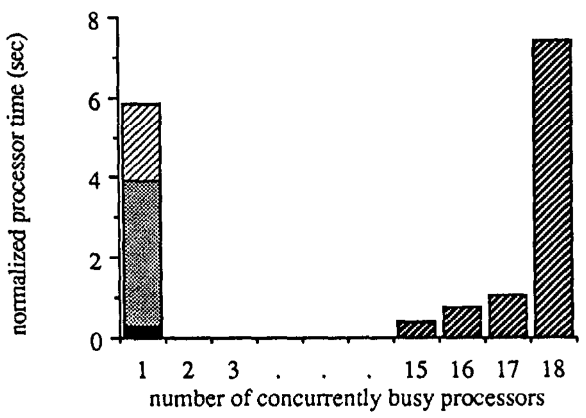
We argued "abstractly" in Section 3.2 that Quartz is well-suited to detecting and fixing a spectrum of parallel program performance problems that have been identified by others as commonly occurring. Of course, the crucial question is whether Quartz is an effective tool in practice. In this section, we describe our experience in using Quartz to tune an existing parallel application.

The application we tuned, called Verify [Ma et al. 1987], compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. It was written for a dissertation to demonstrate that an existing production CAD program could be parallelized with good speedup. The program has 2900 lines of C code, and was written for a Sequent Balance with twelve processors. The circuits we used as inputs in our tests were combinational benchmarks for evaluating test generation algorithms.

The initial speedup of Verify on our Sequent Symmetry was already good: 9.2 using 18 processors. (Because no sequential version of the program was available, speedup was measured as the time to run the program, including process creation and I/O, on one processor divided by the time to run it on 18 processors.) Even though neither of us was familiar with the program or with CAD algorithms in general, over the course of several hours we were able to improve its performance by 40%. Its initial runtime was 114.4 seconds on one processor and 12.4 seconds on 18; with our changes, the runtime dropped to 7.7 seconds on 18 processors. Most of this improvement came within the first few minutes of using Quartz, demonstrating the utility of using normalized processor time as a weighting function.

Figure 1 shows a portion of the Quartz output when run on the initial version of Verify. After the data has been collected, Quartz can interactively draw a graph (on an X Window display) of the total normalized processor time of any monitored procedure or synchronization object as a function of the number of concurrently busy processors. Normalized processor time is based on each routine's processor time divided by its concurrent parallelism, whether the parallelism is due to that routine or to other activity in the program. The top-level routine "main", however, is responsible for all of the activity in the program; its graph of normalized processor time is equivalent to the elapsed time the program spent with each number of busy processors. Different shadings highlight the role of the routine itself, and its children, in the routine's total normalized processor time.

```
                Procedure: main
Normalized processor time: 16.36 sec. (100%)
```



Name	Normalized processor time		Calls
▨ work	10.43	(64%)	18
▦ create_cone	3.62	(22%)	2
▧ input	1.93	(12%)	2
■ main self	.31	(2%)	

Figure 1: Initial Quartz Output for Verify "main"

The Quartz output for Verify shows that although most of the program is indeed highly parallel, a significant portion of its runtime is spent executing serial code. Further, Quartz identifies "create_cone" as being responsible for most of this serial execution time. By examining the graph for "create_cone", and in turn the graph for its child with the largest normalized processor time, we found that the program was spending a quarter of its time executing print routines in order to log a trace of shared data structures as they were created.

While Quartz easily identified this performance problem, gprof would not have. The logging routines accounted for less than 2% of the program's sequential processor time, but because they occurred during the serial initialization phase of the program, they accounted for a much larger share of the parallel performance. Before Quartz pointed it out, we did not know that the program was even doing logging.

Quartz allowed us to make an informed choice about a performance tradeoff: we could substantially improve performance by removing logging, or if this functionality was central to the program, at least we would know by how much it reduced performance. Hypothesizing that it was not important, we made logging a command line option. With logging turned off, the program's parallel performance improved by close to the amount predicted by Quartz, while its serial performance improved only slightly. As a result, the program's speedup improved from 9.2 to 12.2.

When we re-profiled the modified program, Quartz showed that a significant fraction of the program's runtime was still spent in the sequential initialization phase. To reduce this, we read in and allocated data structures for the two input circuits in parallel with each other and with the operating system process creation needed to start the program running on all 18 processors. This improved performance somewhat to a speedup of 12.9.

At this point, we stopped trying to further parallelize the program. Once a program's speedup is high, further improvements become much more difficult. The routines responsible for the difference from ideal speedup account for only a small fraction of the total program runtime; thus even radical improvements to these routines can reduce overall runtime only slightly.

In our case, Quartz showed that virtually all of the program's runtime was now being spent executing entirely in parallel. The remaining time was split between processor starvation during initialization and termination. During initialization, Quartz showed that performance was limited by the fact that the input files were not balanced (one circuit was larger and therefore took longer to read in than the other). During termination, the problem was that some processors finished early and had to wait for the rest to finish; dividing the problem into smaller size sub-problems might help this problem. Fixing these problems seemed to be more effort than it was worth.

Instead, we noted that small changes in the routines that account for most of the parallel execution time would have a large relative effect on runtime. According to Quartz, two routines accounted for over half the execution time of the modified program, and we were able to improve performance somewhat with a few quick tweaks to these routines. (These routines were already highly tuned from the original sequential program.)

Table 3 summarizes the changes that we made to Verify and their effects on sequential and parallel performance. Note that our last improvement in fact decreased the program's speedup. By reducing the execution time of the parallel portion of the program, we increase the relative importance of the program's sequential component.

Version	Elapsed Time (sec.)		Speedup
	Serial	Parallel	
Original	114.4	12.4	9.2
Without logging	109.7	9.0	12.2
Parallel input	109.7	8.5	12.9
Serial optimization	92.3	7.6	12.2

Table 3: Performance Effect of Changes to Verify

A major motivation behind Quartz is efficiency. We measured the overhead Quartz added to this application. Quartz increased the elapsed runtime of Verify by roughly the same amount as gprof: about 70%. (While Quartz is able to remove most of this overhead from its measurements, some overhead does appear in the graph in Figure 1.) Quartz is as fast as gprof because even though Quartz does more work on each procedure call and synchronization event, it need not periodically interrupt (and thereby slow) the execution of the program, because a dedicated processor is used for sampling instead. Verify makes stringent demands on Quartz: it generates roughly 9 million procedural and synchronization events (roughly 1 million per second when running on 18 processors). Even with 18 processors running, a single dedicated sampling processor was able to sample each processor's activity every 6 milliseconds, faster than gprof's sampling rate on DYNIX.

Something that we did not expect was demonstrated by using Quartz on a real application: there is less "performance locality" in parallel programs than in sequential ones. The top eight procedures account for 95% of the elapsed time of Verify on one processor. With 18 processors, though, it takes over 20 procedures to reach the same 95% level. In retrospect, the reason is obvious. The routines that account for most of the time on one processor are parallelized and therefore account for much less of the program's runtime on multiple processors; at the same time, routines that take only a small amount of processor time can become important if they run sequentially.

5. Implications for Other Systems

While we have implemented Quartz on a shared memory multiprocessor, our work has implications for other systems.

On multiprocessors with distributed memory, such as the Intel Hypercube, a dedicated sampling processor would not have efficient access to the state of other processors. Explicit messages would have to be used to update the counts of effective and nominal parallelism, as well as the procedures each processor was executing. A further problem is that programs on these systems are often explicitly written to use a specific number of processors because of the need to explicitly control the communication pattern; removing one for sampling might require re-writing the program.

Alternative approaches also have significant drawbacks on such systems. In particular, recording and post-processing a complete trace may already be impractical for some programs, and will become more so as distributed memory multiprocessors support faster rates of interprocessor communication.

Efficient sampling could be implemented given hardware support for stopping all processors at close to the same time (i.e., by allowing a host computer to send parallel interrupts each processor). One of the reasons for using a dedicated processor is that interrupting any single processor to do sampling can distort the behavior being measured. If all were stopped together, the sample could be taken from that snapshot without measurement error. The sampling could be implemented efficiently by using the processing power of the stopped processors.

Absent hardware support, it may be possible to exploit the characteristics of parallel programs on distributed memory multiprocessors. Because of the requirement that interprocessor communication be explicitly programmed using messages, these systems are most commonly used for highly data-parallel applications with regular communication patterns. For these types of programs, at any point during the computation, each processor executes roughly the same section of code, although one may finish before another. As a result, sampling the behavior of each individual processor, and not the global state, may yield a sufficiently detailed picture of program performance.

The techniques used in Quartz also solve some problems with traditional approaches to tuning sequential program performance. One limitation to gprof is that it cannot be used to tune the implementation of operating system kernel-level routines, since interrupt-driven sampling cannot measure code that runs with interrupts disabled. By using a separate processor to do sampling, however, we would be able to accurately measure kernel-level execution. Note that by avoiding synchronization between the executing processors, or between them and the sampling processor, the processor in the kernel can execute the profiling code even if it holds the low level scheduler lock.

Similarly, by using a profile stack for sampling, we are able to account correctly for execution time spent out of the monitored address space. Because of the way gprof propagates execution time spent on behalf of a procedure, it cannot accurately attribute time spent executing in non-profiled code, or in the operating system on behalf of the program. However, a trend in the design of operating systems and large applications is to decompose them into separate modules in different address spaces, so that hardware protection mechanisms can be used for failure isolation. Much of the execution time of an editor, for instance, might be spent in another address space responsible for updating the display. The performance of the entire decomposed system could be measured by sampling profile stacks shared among all address spaces.

Data-oriented measurement can be helpful for tuning sequential programs as well as parallel programs. For some programs, knowing that a particular procedure accounts for a large proportion of the processing time may not be as useful as knowing that a particular object is expensive. For example, there is better graphics resolution in a more finely tiled sphere, but it also costs more to draw. Gprof introduces a systematic bias in measuring object-oriented program behavior because it assumes that a procedure call always takes the same amount of time to execute (i.e., that the time does not depend on the data that is passed). Quartz avoids this bias by propagating execution times explicitly. While we currently make only limited measurements of data-oriented performance behavior, our design is extensible to a more thorough object-oriented implementation.

6. Conclusions

Achieving good performance from parallel applications is both crucial and challenging. We have discussed the design rationale, functionality, implementation, and use of Quartz, a tool for tuning parallel program performance on shared memory multiprocessors.

The philosophy underlying our work is that an effective tool for tuning parallel program performance must be based on a clear view of the causes of poor performance, and on a specific methodology for improving that performance. By being selective about what it measures and presents, Quartz can focus the programmer's attention on the information needed to tune performance. Measurement efficiency also results from designing the tool to record just the important behavior.

By relating the execution of sections of code and the use of certain data objects with the concurrent behavior of other processors, Quartz assists in identifying areas of the program where re-structuring is necessary to improve performance, and in gaining insight into the types of re-structuring that will work. Because Quartz organizes performance information according to the logical structure of the program, the programmer can tune performance in a top-down fashion.

Acknowledgments

We would like to thank Hi-Keung Tony Ma for donating the application program we used for our case study, and Brian Bershad, Henry Levy, and John Zahorjan for their extensive comments.

References

[Anderson et al. 1989]
Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers 38*,12 (December 1989), pp. 1631-1644.

[Aral & Gertner 1988]
Ziya Aral and Ilya Gertner. Non-Intrusive and Interactive Profiling in Parasight. *Proc. ACM/SIGPLAN PPEALS 1988*, pp. 21-30.

[BBN 1985]
BBN Laboratories. Butterfly Parallel Processor Overview. 1985.

[Bershad et al. 1988]
Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience 18*,8 (Aug. 1988), pp. 713-732.

[Burkhart & Millen 1989]
H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers 38*,5 (May 1989), pp. 725-737.

[Carpenter 1987]
R.J. Carpenter. Performance Measurement Instrumentation for Multiprocessor Systems. In *High Performance Computer Systems*, ed. E. Gelenbe, North-Holland, pp. 81-92, 1987.

[Fowler et al. 1988]
Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Graham et al. 1982]
S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A Call Graph Execution Profiler. *Proc. ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[Gupta 1989]
R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 54-63, April 1989.

[Halstead 1986]
R. Halstead, Jr. An Assessment of Multilisp: Lessons from Experience. *International Journal of Parallel Programming 15*,6 (Dec. 1986).

[Kerola & Schwetman 1987]
Teemu Kerola and Herb Schwetman. Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. *Proc. 1987 ACM SIGMETRICS Conference*, May 1987.

[Ma et al. 1987]
H.T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic Verification Algorithms and Their Parallel Implementation. *Proc. 24th Design Automation Conference*, July 1987, pp. 283-290.

[Malony et al. 1989]
Allen Malony, Daniel Reed, James Arendt, Ruth Aydt, Dominique Grabas, and Brian Totty. An Integrated Performance Data Collection, Analysis, and Visualization System. *Proc. 4th Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[Miller & Yang 1987]
Barton P. Miller and C.-Q. Yang. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. *Proc. 7th International Conference on Distributed Computing Systems*, September 1987.

[Moeller-Nielsen & Staunstrup 1987]
P. Moeller-Nielsen and J. Staunstrup. Problem-Heap: A Paradigm for Multiprocessor Algorithms. *Parallel Computing 4*, North-Holland, 1987, pp. 63-74.

[Pfister et al. 1985]
G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weise. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proc. 1985 International Conference on Parallel Processing*, August 1985.

[Rodgers 1986]
David P. Rodgers. Personal communication.

[Segall & Rudolph 1985]
Zary Segall and Larry Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software 2*,6 (November 1985).

[Sequent 1988]
Sequent Computer Systems, Inc. Symmetry Technical Summary.

[Thacker et al. 1988]
Charles Thacker, Lawrence Stewart, and Edward Satterthwaite Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers 37*,8 (Aug. 1988), pp. 909-920.

[Yang & Miller 1988]
Cui-Qing Yang and Barton Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. *Proc. 9th International Conference on Distributed Computing Systems*, pp. 366-373, June 1988.