51

Experience with the Firefly Multiprocessor Workstation

by Susan Owicki

September 15, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Experience with the Firefly Multiprocessor Workstation

Susan Owicki

September 15, 1989

Author's abstract

Commercial multiprocessors are used successfully for a range of applications, including intensive numeric computations, time-sharing, and shared servers. The value of multiprocessing in a single-user workstation is not so obvious, especially in an environment where numeric problems do not dominate. The Digital Equipment Corporation Systems Research Center has had several years of experience using the five-processor Firefly workstation in such an environment. This report is an initial assessment of how much is gained from multiprocessing on the Firefly.

Reported here are measurements of speedup and utilization for a variety of programs. They illustrate four sources of concurrency: between independent tasks, within a server, between client and server, and within an application. The nature of the parallelism in each example is explored, as well as the factors, if any, that constrain multiprocessing. The examples cover a wide range of multiprocessing, with speedups on a five-processor machine varying from slightly over 1 to nearly 6. Most uses derive most of their speedup from two or three processors, but there are important applications that can effectively use five or more.

Susan Owicki

Contents

1 Introduction 1
2 Metrics 2
3 Single-user Timesharing 4
4 Concurrency within a Server 6
5 Concurrency between Client and Server 9
6 Concurrency within an Application 11
7 Conclusion 15
8 Acknowledgements 15

References 17

1 Introduction

Commercial multiprocessors are used successfully for a range of applications, including intensive numeric computations, time-sharing, and shared servers. In these uses, there is abundant scope for multiprocessing in handling simultaneous requests from separate users or in single-user computations where there is substantial concurrency in the structure of the problems.

The value of multiprocessing in a single-user workstation is not so obvious, especially in an environment where numeric problems do not dominate. Can other applications besides scientific computation exploit multiple processors? Are multiple users essential to generate a reasonable workload for the system?

For several years, the Firefly multiprocessor workstation [9] has been the primary source of computing at the Digital Equipment Corporation, Systems Research Center (SRC). Software for the Firefly spans a wide range of systems and applications programs. Most Firefly programs are written in an extension of Modula-2 [12] called Modula-2+ [7], which provides threads and synchronization primitives for concurrent programming. The Firefly workload does not include the sort of lengthy numeric calculations that have traditionally benefited from concurrency. Nevertheless, much of the software has been designed to take advantage of multiprocessing. Thus, there has been substantial experience at SRC with using multiprocessor workstations for non-numeric computation, and it seems appropriate now to assess their value. To do so, a number of instances of multiprocessing on the Firefly were examined. This report gives measures of concurrency for these examples, and describes the sources and limits of their parallelism.

The Firefly used for these measurements has five 1-MIP MicroVAX processors, so it is called a 5-MIP machine. In actual use, though, the Firefly has less computational power than a 5-MIP uniprocessor. This is partly because some of the software was originally written for a uniprocessor. But even code written for the Firefly seldom exploits all the concurrency that the processors provide. There are many reasons: some problems have limited parallelism, sometimes the overhead of concurrency is too high, sometimes another part of the machine is a bottleneck, and sometimes the implementor chose to avoid the complexity of concurrent programming.

However, the Firefly doesn't have to provide a full 5 MIPS of computing power to be cost-effective, since it is generally cheaper to build a multiprocessor than a uniprocessor with the same MIPS rating. Building a multiprocessor with, say, three to thirty processors may not cost a great deal more than building a uniprocessor with the same CPU. Thus, the multiprocessor may be economically attractive even if its processors are not always fully utilized. The benefit gained from greater computing power must be weighed against the increased cost of the multiprocessor.

This report is concerned with assessing the benefit side of the cost-benefit equation. Benefit is estimated using the standard metrics speedup and processor utilization. These metrics, which are discussed in Section 2, are less than ideal, but they do give some feeling for the degree of success in exploiting multiprocessing.

Four sources of concurrency were identified in day-to-day workstation activities:

- single-user time-sharing: concurrency between independent tasks. A user may undertake several tasks in parallel, such as editing or reading mail while a compilation is in progress.
- concurrency within a server. The window system, the file system, and other basic services are implemented with algorithms that use multiprocessing.
- concurrency between client and server. Sometimes a server can return an immediate answer to a request, then compute in parallel with its client to complete processing the request.
- concurrency within an application: some application programs are coded with multiple threads for performance.

Note that the first three sources of concurrency are available in all uses of the workstation, without requiring an application programmer to write multi-threaded code. So multiprocessors in a workstation can be useful even when running applications that do not attempt to exploit concurrency.

Sections 3 to 6 contain speedup and utilization data for a number of examples from each of the areas above. All the measurements were taken on a 5-processor, 16 Megabyte Firefly, unless otherwise noted.

2 Metrics

Parallel sorting will be used as an example to illustrate speedup and processor utilization. Figure 1 gives graphs of speedup and utilization for a Quicksort program [11] working on an array of 10000 integers. The algorithm sequentially partitions the array and then recursively applies Quicksort in parallel to the two subarrays.

The speedup reported in Figure 1a is defined in the conventional way: speedup for n processors is

$$S(n) = T(1)/T(n), \tag{1}$$

where $T(k)$ is the execution time when the program is run using k processors. Speedup is determined using an option of Taos, the Firefly operating system, that restricts the set of processors available to the scheduler. Thus, $T(k)$ is measured by disabling all but k processors and noting the elapsed time to execute the program. Since elapsed time can fluctuate due to other activities in the system, $T(k)$ was taken as the median elapsed time of three to five runs.

The dotted line in Figure 1a represents the theoretical "perfect" speedup. For two and three processors, Quicksort has a nearly optimal speedup. There is some additional benefit from the fourth and fifth processors, but it is less significant. The amount of parallel computation is limited by the structure of the algorithm: during the initial partitioning, for example, only one thread is active.

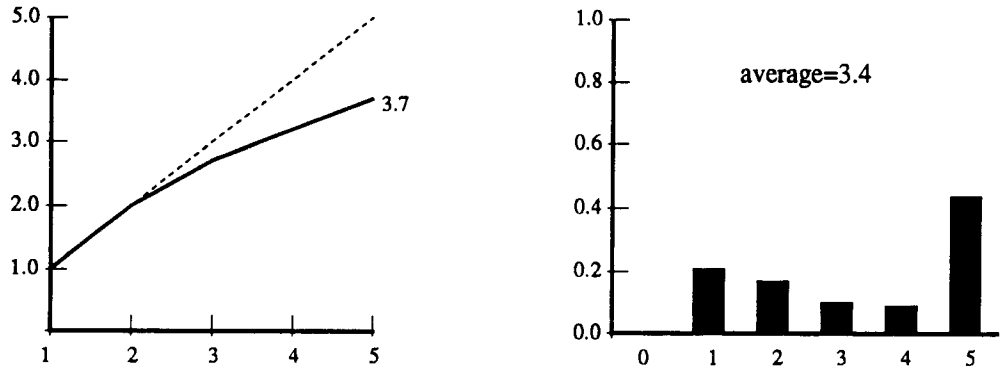

Figure 1: Quicksort of 10000 integers.

Processor utilization for a five-processor run of Quicksort is shown in Figure 1b. The height of the kth bar represents the fraction of the time when exactly k processors were busy. There was essentially no time when all processors were idle, which is to be expected in a compute-bound task like sorting an array. All five processors were busy for more than 40% of the time. This is somewhat surprising, given that the speedup figures for four to five processors were not impressive. With five processors, the highly parallel parts of the run keep all the processors busy, leaving a substantial amount of time when only one processor is busy. With fewer processors, sequential segments in one partition are more likely to overlap parallel segments of another, and each processor is busy a larger fraction of the time. The average processor utilization on a five-processor Firefly was 3.4.

Processor utilization was measured by instrumenting the operating system to record the amount of time spent with k processors busy. Once this instrumentation has been done, measuring utilization is much easier than measuring speedup, because speedup requires multiple runs for each value of n.

There is a correspondence between speedup and average processor utilization. If a program does the same amount of work when run with 1 processor or with n processors, its speedup and average utilization should be equal. When they are different, utilization is usually higher, due to overhead in the parallel version. For Quicksort, however, speedup is higher than utilization. This is because of background activity in the operating system, which averages about 20% of a processor, regardless of workload or number of available processors. Background work directly contributes to processor utilization. It also increases speedup, because the 0.2 CPU lost to background activity slows a one-processor computation more than a five-processor computation. For Quicksort, background activity increases speedup more than utilization.

Neither of the metrics discussed so far is ideal. The problem with utilization is that it measures consumption of processor cycles, but not the benefit obtained. A program that uses extra proces-

sors inefficiently may show large processor utilization but very little speedup. Even though speedup is more relevant in assessing the benefits of multiple processors, sometimes only utilization is reported here. This is because speedup is harder to measure. For some programs, interactive ones in particular, speedup cannot be measured at all because the timing of runs is not reproducible.

Unfortunately, high speedup numbers can be misleading as well. The problem is that a parallel program may perform badly when run on one processor. Introducing parallelism entails overhead: creating threads, dividing the workload, synchronization operations, and so on. When a parallel program is run on a single processor, it is likely to be slower than a good sequential algorithm. Overhead leads to a large value for $T(1)$, and this inflates the speedup figures. Ideally, speedup would be measured against a good single-processor solution to the problem. For most of the examples in this report, no such solution was available, so the speedup reported is based on the formula in equation 1. Speedup over a good sequential program is reported whenever possible; usually this is because the concurrent program was derived from an earlier sequential version.

One other speedup measure is used in this report. Some of the programs studied have a variable number of threads, and speedup can be measured by comparing the elapsed time for n threads with the time for 1 thread, with the number of processors kept constant. Computing speedup in this way measures the advantage obtained from program structure rather than from multiprocessors. Thread-based speedup may be higher or lower than processor-based speedup. It may be higher because a program can benefit from overlapping computation with some other activity, such as I/O. This requires multiple threads, but not multiple processors. It may be lower because there can be concurrent processing in the system (within the file server, for example) while even a single-threaded program is executed. Processor-based speedup is more relevant to evaluating multiprocessor workstations, but in some cases thread-based speedup is reported because it is the only data available.

3 Single-user Timesharing

A workstation often operates like a single-user time-sharing system. A user can pursue several activities at once – perhaps compiling a program, running a simulation, and reading mail at the same time. Having a multiprocessor workstation makes it possible to do this without suffering degraded performance.

The primary constraint on this sort of concurrency is the user's ability to juggle multiple active tasks. Doing two things at once is common, and doing three is not unusual. While a module is compiling, for example, a user typically does something else, such as edit another module, read mail, or examine an online calendar. A long-running computation may provide a third activity. One researcher regularly commits one processor on his Firefly to a symbolic computation which can take days to complete a single problem. This CPU-intensive background program does not significantly degrade the performance of his workstation for day-to-day activities. However, he is using a Firefly with 32 Megabytes of memory. The symbolic computation requires a good deal of memory, and

running it in the background with a 16 Megabyte Firefly led to unacceptable thrashing.

The Firefly operating system also takes advantage of the UNIX[1] pipe construction to obtain command-level parallelism. A pipe is a way to direct the output of one program to the input of another. For example, the command

```
ls | wc -l
```

directs the output of ls, a list of files in the current directory, to the program wc, which counts the number of lines in its input. Thus, the result is a count of the number of files in the current directory. On a Firefly, the stages in such a pipeline are executed in parallel whenever possible.

To get a feeling for the level of multiprocessing arising from independent tasks, CPU utilization was monitored on one Firefly during several days of ordinary use. Speedup was not measured because much of the activity was interactive, and the timing of mouse motions and key strokes is too hard to reproduce. The most common parallel activity was editing while compiling. Editing is not a CPU-intensive activity, and editing while compiling used scarcely more processor-power than compiling alone. However, other parallel activity led to significant multiprocessing. In one fairly typical sequence, the user began a remote login to a time-sharing machine. Since remote login is annoyingly slow, the user filled in the time by checking for new mail. This also required communication with a remote server. While waiting, the user requested a new typescript window. Figure 2 shows the processor utilization graph for this set of operations. Average processor utilization was over 2, and at least 10% of the time there were 5 processors busy.

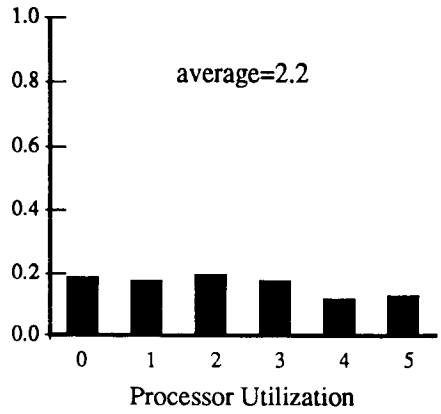

Figure 2: Set of Independent Tasks.

Another example arose very naturally when logging in to the Firefly. At login, this Firefly was configured to start two programs, the text editor Ivy and the mail system Postcard. Both require a fair

[1] UNIX is a trademark of AT&T Bell Laboratories.

amount of file access and computation at startup. Ivy recovers the state of all files that were being edited when the server was last running. Postcard checks the status of a number of mail folders and scans one of them to build a list of its current messages. Running both initializations in parallel led to an average processor utilization of 2.0.

A more systematic exploitation of single-user time-sharing is illustrated by the program parmake [6]. Parmake is a variant of the UNIX utility `make` [4], which builds a software object, such as a load module, following instructions found in a file. The difference is that, wherever possible, parmake builds components in parallel. (Parmake can enlist idle workstations in this task, as well as using multiple processors on the Firefly where it is invoked. For this report, only the single-machine behavior is important.) Parmake was measured on a large-scale recompilation consisting of 238 Modula-2+ files (65,000 source lines) drawn from various library packages at SRC. Speedup was measured by varying the number of threads, that is, the number of parallel compilations, rather than the number of processors. The maximum speedup in this benchmark was 2.2, attained by running three compilations in parallel. Running more compilations in parallel led to a severe degradation in performance, because of memory thrashing. Subsequently, the measurements were repeated on a Firefly with twice as much memory, (and faster processors, as well). The speedup was only slightly better: 2.3 for three threads, and 2.4 for four threads. It appears that the bottleneck for parmake on the larger machine is file I/O to the single local disk. A large part of the demand comes from loading the 4 Megabyte compiler once for each source file.

It should be noted that there were almost no dependencies among the modules compiled in this test. Thus, most compilations could logically proceed in parallel, and speedup was limited by the resources of the machine. Bubenik and Zwaenepoel [2] found that when `make` is used for program development, dependencies substantially constrained the available parallelism.

A final example concerns a theorem-prover called Reve. As an experiment, up to five copies of Reve were run in parallel on the same test case. (This gives an indication of how much might be gained by splitting a large proof into independent components and running them concurrently.) The test case was very compute-intensive. Two copies of Reve were able to run in nearly the same time as one, giving a speedup of 1.91. Improvement was less than linear as the number of copies increased: still, the speedup with 5 copies was 4.05.

4 Concurrency within a Server

Software at SRC makes frequent use of the client/server program structure. Typically, servers and clients run in different address spaces or even on different machines, with communication by remote procedure call. There are also some servers that operate in the same address space as their clients, with communication by ordinary procedure call.

The Taos operating system is the most important server. Taos runs in its own address space. It provides a number of operating system facilities: display management, file system, and so on. Taos

itself is multi-threaded, and its interface allows multi-threaded programs to make parallel requests to the operating system [5].

There are also other servers in the Taos address space. The Trestle window system provides a powerful set of operations for managing windows on the Firefly displays. The File Server allows files on one Firefly to be accessed from another. Other servers run in their own address space: most Fireflies run the monarch server for remote login, and the dp server, which allows an idle Firefly to be used by distributed computations. The Ivy text editor also runs as a server in its own address space.

When a server is multi-threaded, as many are, its client programs can get improved performance on a multiprocessor, even if the clients are single-threaded. This effect can be observed in copying a set of files in the Firefly's local file system. The concurrency of copy was measured by invoking the operation from the command line, on a directory containing 9.5 megabytes in 89 files. This led to one call on the Taos Copy command for each file in the directory. Figure 3 shows the processor utilization and speedup for this operation. Even though copying files is an I/O intensive activity, there is about 50% speedup in going from one to two processors, and a slight increase in going from two to three. The utilization graph shows that two processors are in use about 30% of the time. The dip in speedup at 4 processors is a measurement artifact.

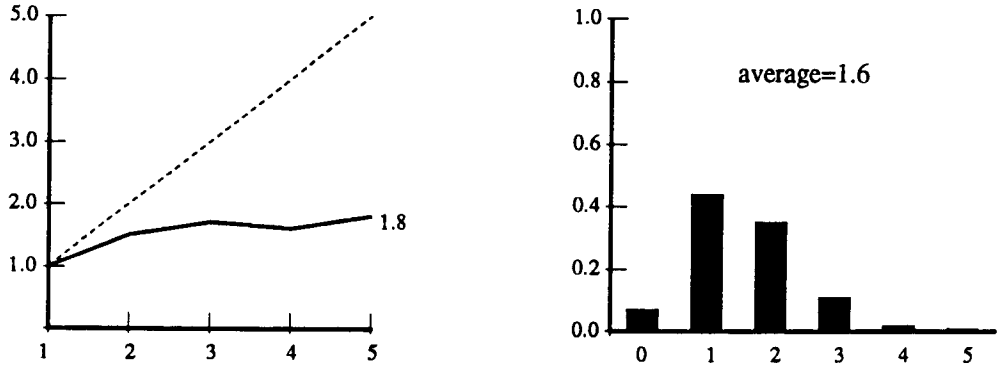

Figure 3: Concurrency within a server: copying files.

The concurrency in the Copy operation comes from the activity of five threads. Two are created by the Copy procedure: one to read the input file to a buffer and the other to write the output file from the buffer. At a lower level, the file system maintains a read cache filled by two read-ahead threads, and a write buffer emptied by one writer thread. In spite of these five threads, however, average processor utilization is only 1.6. The low-level threads spend most of their time waiting for the disk: multiple threads are used to keep the disk busy, not the processors.

Trestle, the primary Firefly window system, is more able to exploit multi-processing. Trestle uses parallelism in two ways. The first is a 4-stage pipeline that handles paint requests. The first stage combines client paint requests into batches; it operates in the client address space. The second stage is the remote procedure call to send a batch of requests to the Trestle server, usually on the same Firefly. The third stage's main task is clipping, and the fourth stage translates the batch into display-controller commands. A benchmark program that blackens a large number of rectangles on the screen showed that the pipeline gave a speedup of 4.0, according to the naive speedup definition of equation 1. To obtain a more realistic measure, Trestle was reconfigured for a uniprocessor by removing all pipeline code. This resulted in a speedup measure of 3.1.

The second source of concurrency in Trestle can be seen when a window is re-configured. A window may contain subwindows, which may have subwindows of their own, and so on. When a window is reconfigured, its subwindows are processed in parallel. A benchmark program called the Tiling Monster builds and then reconfigures a window with many nested subwindows. Measurements of the Tiling Monster showed a speedup of 4.7, or nearly the entire processing power of the Firefly. Once again, it was possible to measure the performance of a good sequential algorithm; using the sequential algorithm as a basis, the speedup was 3.7.

Of course, both of the Trestle benchmarks were designed to take full advantage of the potential concurrency in Trestle's algorithms. A workstation user typically does not drive the window system so hard. But the power is available to graphics-oriented applications.

Another server, Facades, is a windowing package that incorporates a desktop paradigm for display management. It provides some of the functionality of Trestle, but has better performance when bound into the same address space as the client. The goal of Facades is to make it easier to write applications that use windows for an interactive user-interface. Like Trestle, Facades uses a pipeline to handle paint requests. It has three stages. The first stage accepts a request and checks whether the window to which it applies is on-screen. If it is, the request is passed to the second stage. The second stage relocates the request to the screen and clips it to the visible portions of the window. The third stage transforms the relocated and clipped paint actions into commands to the screen controller. For a long sequence of easy requests, these pipeline stages took 100, 100, and 60 microseconds respectively. This suggests a CPU utilization of about 2.6 by the pipeline. To evaluate speedup, Facades can be compared to a good sequential program, which would not contain code to buffer requests between pipeline stages. Based on experimentation and instruction counting, it was estimated that the speedup over such an implementation would be less than two.

Zeus, a system for algorithm animation, provides yet another example of concurrency within a server. Zeus allows a user to observe multiple views of a program's execution. All views are updated in parallel as the program runs. It is also possible to have multiple threads in the program being animated. For a single-threaded algorithm, the concurrency in a Zeus run comes from Zeus's updating multiple views in parallel, and from the concurrency provided by the Trestle server.

In an animation of insertion sort for 500 random integers, CPU utilization was 2.4 when only one view was displayed. The sort algorithm itself is sequential, so the concurrency here must be due

to Trestle. With two views, CPU utilization increased to 3.3, while with four views it rose to 4.1. Further increasing the number of views led to slower execution rather than greater CPU utilization.

The final example in this section shows that multiprocessing can sometimes speed up a computation by reducing overhead. Benchmark measurements of a single thread making inter-machine remote procedure calls show a speedup of 1.33 when the calling machine uses 2 processors [8]. (Adding more processors gives a very small additional speedup.) This is surprising, given that the calls are made sequentially and there is no concurrency in the RPC implementation. The explanation involves the background activity mentioned in section 2. Part of the speedup comes from processing RPC's in parallel with background tasks. But that is not enough to account for the whole change. It appears that additional speedup comes from reducing the number of context switches. The Firefly scheduler allows a waiting thread to remain in an idle loop as long as its processor is not needed by another thread. In RPC, the calling thread must wait to receive a response from the server machine. With one processor, the odds are good that the waiting thread will lose its processor to a background thread. Then, when the response arrives, the RPC thread will have to be rescheduled. Having two processors makes it more likely that the RPC thread will still be scheduled on a processor when the response is received.

5 Concurrency between Client and Server

The procedure-call interface between client and server is synchronous, that is, the client is delayed until the call returns. But sometimes the semantics of an operation allow a server to return control to the client before the work is completed. The client can continue while one or more threads in the server complete the work.

Trestle provides an example of this sort of concurrency. The Trestle pipeline operates in such a way that requests to paint the screen return control to the client early on, while the paint request is moving its way through the pipeline. This effect can be seen in the execution of the program Forest. Forest recursively draws a picture of an attractive tree whose shape is determined randomly. It is typical of graphics applications that perform some computation and call on Trestle to display the results on the screen. Forest is completely compute-bound, and its single-threaded computation is the limiting factor in the performance of the program.

Figure 4 shows the speedup and utilization measurements for a program called Forest. Forest itself keeps one processor busy at all times, so utilization never falls below 1. When two or more processors are busy, it is because of concurrency between Forest and Trestle. When three or more processors are busy, it is because of additional concurrency within Trestle. The speedup graph shows that using more than three processors does not improve performance, and most of the benefit is achieved with two.

Another example of client/server concurrency occurs in Zeus, when the algorithm to be animated continues to run one or more threads while Zeus is updating the display. This possibility is exploited

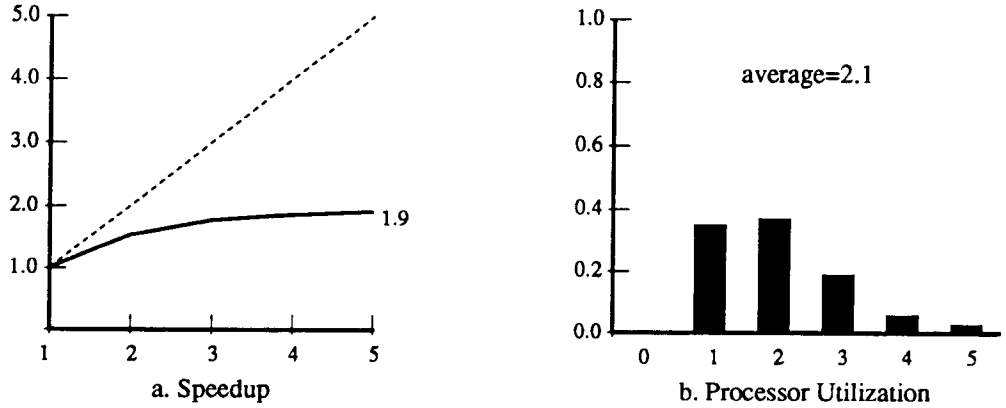

Figure 4: Concurrency between client and server: Forest.

in the animation of a multi-level hashing scheme [1]. The animation shows the filling of hash tables as a series of keys are entered. The algorithm's behavior is most interesting when there are many collisions. In order to illustrate this case, keys are selected in a way that gives a high probability of collision. Finding such keys requires more computation than the hashing algorithm itself. From one to three producer threads find the keys, while a single consumer thread uses them to drive the animation. CPU utilization averages about 3.8 when presenting a single view of the algorithm, compared with an average utilization of 2.4 for animating a single view of insertion sort.

A final example of concurrency between client and server is concurrent garbage collection. A garbage collector is not a typical server, because it is part of the runtime system, and because its actions are not explicitly invoked by the program. Nevertheless, it provides a service to the application program which can be a source of concurrency, so it is included in this section.

The Modula-2+ garbage collector is a reference-counting collector which runs in a thread of its own, in parallel with the application program, or mutator. Each mutator assignment that changes a reference count causes an entry to be enqueued for the collector. When a collection is initiated, the collector processes the queue and updates reference counts. This updating, plus the detection and reclaiming of garbage, take place in parallel with mutator operations. Measurements on several single-threaded applications showed that the collector thread ran about 40-70% of the time. The mutator thread ran most of the time, though it incurred a few percent overhead in synchronization operations. If these figures are typical for Modula-2+ programs, the collector thread must use about 55% of a CPU to keep up with one compute-bound mutator thread. This suggests that, on a five-processor Firefly, a multi-threaded mutator could have an average CPU utilization of over 3 and still leave adequate CPU available to the collector.

An experimental copying collector for the programming language ML has also been imple-

mented on the Firefly [3]. On one benchmark, a program using the concurrent version of this collector ran 18% faster than one using a sequential version. The collector thread was active about 40% of the time.

6 Concurrency within an Application

Quite a number of the applications written for the Firefly are multi-threaded. In fact, about 40% of the software packages at SRC fork one or more threads. (Packages vary in size, but typically contain code for a single application program or a library of related procedures.) Often the threads are created for program structure rather than performance; a rough estimate is that 20% of the packages use threads to gain performance through multiprocessing.

Simulation is an application area that seems especially suited to concurrency, although sometimes speedup may be harder to achieve than it first appears. Regsim, a register-level simulator for synchronous digital logic, is able to successfully exploit the parallelism in its input. The circuits input to Regsim are composed of connected objects. In each simulation phase, some fraction of the objects are evaluated. Certain objects must be evaluated every phase; others are evaluated on demand. When more than one thread is used for evaluation, the synchronization is such that some objects may be evaluated more than once.

Figure 5 shows the processor utilization and speedup for the simulation phase of Regsim on a circuit consisting of about 5000 objects. (The simulation phase, which took 1693 seconds with five processors, was preceded by a largely sequential initialization phase that took 99 seconds.) In this case, the number of threads was a parameter. For the speedup measurements it was set to give the best performance; this meant n threads when running on n processors, except that with five processors, four threads was optimal. In the utilization graph, note that 90% of the time there are four or five processors busy. Clearly Regsim contains enough parallelism to keep the Firefly processors busy. Moreover, they are being used productively, as illustrated by the speedup diagram. In fact, the speedup is better than linear.

Of course, no theoretical limit is really being exceeded. The anomaly occurs because of background activity on the Firefly, as mentioned in Section 2. Even when no user task is being executed, about 20% of a processor, on average, is consumed with ethernet activity, operating system timers, and so on. The level of background activity is essentially independent of the amount of work being done on the Firefly. Thus, when one processor is available for computation, only about 0.8 processor can be used by an application program. When 5 processors are available, 4.8 can be used by the application. This allows the application to speed up by a factor of 6 rather than 5.

Prf is another design automation tool that gains considerably from multiprocessing. Prf is a post-processor for routed printed-circuit boards. Its input is produced by a router with an abstract model of routing; prf adjusts the routing to conform to a more physically accurate model. For example, it introduces jogs to route traces around holes and rounds off sharp corners. Prf can process traces

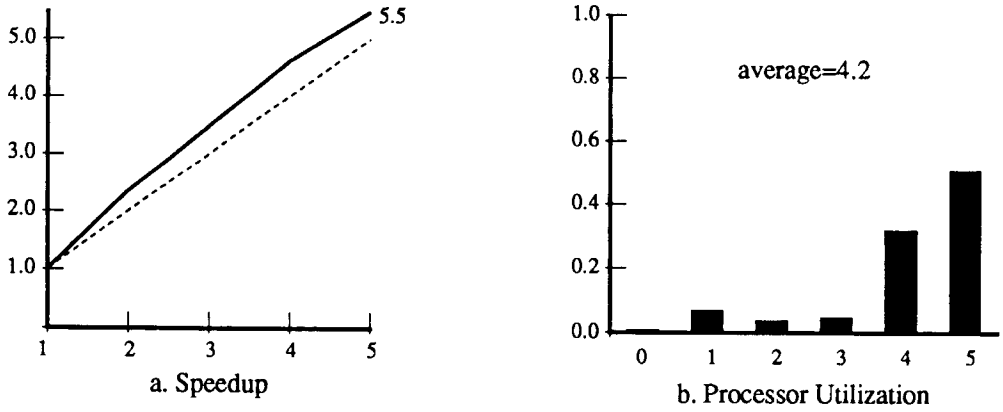

Figure 5: Concurrency within an application: Regsim.

in parallel because the modifications it makes are strictly local to a trace. A typical run may have about 900 traces, so there is ample scope for concurrency. In fact, the average utilization for the processing phases of prf is 4.6, so nearly the full processing power of the Firefly is exploited. The final output phase is also parallel; here the parallelism comes from independent processing of the ten layers of the circuit board. In this phase, processor utilization averages 4.4. Unfortunately, the overall performance of prf is not as good as these figures suggest. First, about 40% of execution time is consumed by parsing the input file sequentially. In this phase, utilization is 1.6 because of concurrency in the file system. It would probably be possible to parse the input file in parallel, but this optimization has not been pursued. Second, some of the processing phases do not achieve as much speedup as the utilization figures suggest, for reasons that are not yet known. For example, the longest processing phase has a utilization of 4.7 and a speedup of 3.2. The result is that average utilization for the entire program is reduced to 3.1, and speedup is 2.3.

Several multi-threaded compilers have been developed for the Firefly. The current Modula-2+ compiler exploits a limited amount of concurrency by running the lexical analyzer as a separate thread. The lexical analyzer thread is able to produce tokens faster than they can be used, so the main compiler thread never has to wait for a token. Thus, lexical analysis is essentially free so far as elapsed time is concerned.

Michael Junkin and David Wortman, at the University of Toronto, have made considerable progress in parallelizing the Modula-2+ compiler. The Toronto compiler gets parallelism from two sources. First, scopes (procedures, main modules, and definition modules) are compiled in parallel. The primary limits on this easy parallelism are fast recognition of scopes, and data-dependencies between scopes. Second, some parallelism is exploited in compiling a single scope. This involves pipelining lexical analysis and analyzing declarations in parallel with parsing statements. Prelim-

inary measurements comparing the Toronto compiler with a traditional single-threaded compiler show that, with one processor, the Toronto compiler is 1.3 times faster. The speedup is 3.1 for five processors, and 3.3 for a specially configured Firefly with seven processors.

Figure 6 is a graphical illustration of the performance of the Toronto compiler. The figure is a second-by-second display of the number of active processors during a sequence of compilations on a seven-processor Firefly. The graph was obtained by repeatedly compiling the same module, first with one processor, then with two, and so on. A 10 second idle period separates compilations. The image illustrates both the use of multiple processors (the height of the bars) and the elapsed time for each compilation (the width of the active periods).

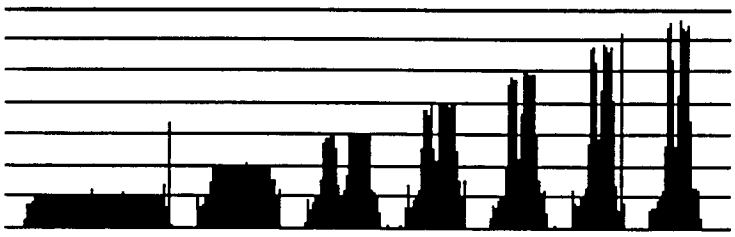

Figure 6: Parallelism in the Toronto Modula-2+ Compiler.

There is also a parallel C compiler for the Firefly [10]. This compiler consists of a two-stage pipeline. The first stage performs extended lexical analysis for the second stage, much like the current SRC Modula-2+ compiler. The second stage does the parsing and assembly code generation. It processes units of the source program concurrently; the unit granularity can vary from a single statement to a procedure.

Experiments compared the concurrent compiler, with various granularities of concurrency, to a sequential compiler on nine large source files (> 1200 lines). Pipelined scanning, with a sequential second phase, gave a speedup of 1.07 to 1.18. With a concurrent second phase and the finest unit of granularity, the compiler achieved a speedup of 2.5 to 3.3. Compiling with the finest granularity never hurt performance. For six of the source files, most of the concurrency came from processing procedures in parallel. For the remaining three, a finer granularity, at the statement level, gave a significant increase over the procedure level.

An important tool for insuring that Modula-2+ programs do not contain inconsistent versions of component modules is the inter-module checker, or imc. It takes a list of separately compiled object modules and libraries and checks that all the compilations used consistent versions of definitions files. It is organized as a 4-stage pipeline. The first stage opens the files listed on the command line. The second stage separates libraries into individual object modules. The third stage builds a list of module version numbers, and the fourth stage merges the lists and checks them for consistency.

After all stages have completed, a summary report is printed. Running on a set of 20 modules and 10 libraries, the concurrent version of the program exhibited a speedup of 1.3 over an earlier, sequential version.

The program Proof allows a user to preview a typeset document on the Firefly display. A substantial amount of processing is required to transform the typesetter instructions into commands to the Trestle window system. Proof is composed of three parallel tasks: building a display list of characters and boxes (one thread), painting the display list (up to three threads, one for each page visible on the screen), and processing font descriptions (six threads). In addition, a background thread converts the representation of frequently used fonts to speed up Trestle's processing of characters. Because Proof is used interactively, it is important to display the first page as soon as possible. To achieve this, a painter thread can inform a font-loading thread that a certain character is needed immediately. The font-loader will interrupt its current activity in order to load the requested character. Processor utilization for Proof averages 3.9 while preparing to display the first page of a document.

The program treeupdate is used to update a Firefly's local file system so that it has current copies of the files stored in a shared file system. Treeupdate recursively descends the tree of directories and files in the shared system, comparing with the local tree and doing any necessary updates. The parallel version maintains a work queue of tree nodes to be updated, with a crew of N threads serving the queue. Treeupdate is an example of "embarrassing parallelism" – except for the queue, there is no synchronization necessary. (Here "embarrassing" means "overabundant," as in "an embarrassment of riches.") Comparing this parallel version with the previous sequential one, a speedup of 3.7 was observed when running over an already consistent file system. Presumably the speedup is due to the overlapping of the overheads associated with getting the status of files in the local and remote file systems, and with sorting of directory entries.

One further parallel application must be mentioned even though it derives much of its parallelism from using multiple workstations. Factor is a program for factoring large numbers that runs continuously at SRC. It is controlled by a single program that watches for Fireflies that have been idle for some threshold period. When an idle machine is found, the control program adds it to the pool of factoring machines and starts one factor program for each of its processors. Factor is able to use virtually all of the available CPU cycles on a machine, because there is almost no synchronization between the separate programs, and the CPU is almost the only resource required. The number of Fireflies in the Factor pool fluctuates over the course of a day, with an average of about 40 machines at any time. On a typical day in January 1989, Factor executed approximately 10^{13} VAX instructions at SRC. Compared to tuned code for factoring on Cray X-MP's and NEC SX-2's, the "idle" Fireflies are factoring numbers slightly faster than one dedicated supercomputer.

7 Conclusion

The examples presented in the preceding sections should give a feel for the variety of ways in which concurrency is exploited on the Firefly. A substantial part of SRC's computing now is faster because of multiprocessing.

Even when running a single sequential application, a Firefly user benefits from having two processors, because of concurrency associated with servers. Submitting independent tasks in parallel is a common activity; here two to three processors are of value. Thus, the benefits of having at least three processors are very real, even for a user who runs no concurrent applications. A number of applications exist that effectively use four to five processors, and could probably take advantage of more. For those who frequently use such applications, having five or more processors gives a substantial increase in productivity. Other users would probably find that, most of the time, their work was adequately supported by three processors.

Many of the application programs run on the Firefly are nearly sequential. Some were developed for a uniprocessor and later ported to the Firefly. But even among programs designed for the Firefly, there are a number that take little advantage of multiprocessors. Some tasks are simply not well suited to concurrency. In other cases, designers are primarily interested in exploring new functionality, and do not want to spend their time on performance.

Still, as more applications are programmed for the Firefly, more of SRC's computing will benefit from concurrency. For example, Fireflies are often used for program development, so compiling is a common activity. The current compiler has limited concurrency, but a more parallel compiler will soon be available. In addition, a project to facilitate the development of large software systems will provide facilities that resemble parmake but avoid some of the problems that currently limit its use.

Experience at SRC makes it clear that multiple processors can be used effectively in workstations. Although current measurements suggest that most users derive most of their benefit from the first three processors, this should be considered a preliminary estimate. The use of concurrency for speedup is increasing at SRC, and it will be some time before its limits can be determined.

8 Acknowledgements

The programs described in this report are the work of a number of people at SRC. In addition to those cited in the references, they include Bob Ayers, Andrew Birrell, Marc H. Brown, Mark R. Brown, Patrick Chan, Anthony Discolo, Hans Eberle, John Ellis, Jim Horning, Bill Kalsow, Mark Manasse, Greg Nelson, Lyle Ramshaw, and Jorge Stolfi. John DeTreville worked jointly with the author to survey concurrency in SRC software; some conclusions from that survey are included here. Marc H. Brown, Mark R. Brown, Sheng-Yang Chiu, Jim Horning, Bill Kalsow, Ed Lazowska, Roy Levin, Michael Schroeder, Jorge Stolfi, and David Wortman made useful comments that improved the presentation of the paper.

References

[1] Andrei Broder and Anna Karlin. Multilevel adaptive hashing. In *First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 1990. To appear.

[2] Rick Bubenik and Willy Zwaenepoel. Performance of optimistic make. In *1989 ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems*, May 1989.

[3] John Ellis, Kai Li, and Andrew Appel. *Real-time Concurrent Collection on Stock Multiprocessors*. Research Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[4] S. Feldman. Make–a computer program for maintaining computer programs. *Software Practice and Experience*, 9(4), April 1979.

[5] Paul McJones and Garret Swart. *Evolving the UNIX System Interface to Support Multithreaded Programs*. Research Report 21, Digital Equipment Corporation Systems Research Center, September 1987.

[6] Eric Roberts and John Ellis. Parmake and dp: experience with a distributed, parallel implementation of make. In *Second Workshop on Large-Grained Parallelism*, Software Engineering Institute, Carnegie-Mellon University, November 1987. Report CMU/SEI-87-SR-5.

[7] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6), November 1986.

[8] Michael Schroeder and Michael Burrows. *Performance of Firefly RPC*. Research Report 43, Digital Equipment Corporation Systems Research Center, April 1989. To appear in 12th ACM Symposium on Operating Systems Principles, December, 1989.

[9] Chuck Thacker and Lawrence Stewart. *Firefly: a Multiprocessor Workstation*. Research Report 23, Digital Equipment Corporation Systems Research Center, December 1987.

[10] Mark Vandevoorde. *Parallel Compilation on a Tightly Coupled Multiprocessor*. Research Report 26, Digital Equipment Corporation Systems Research Center, March 1988.

[11] Mark Vandevoorde and Eric Roberts. *WorkCrews: An Abstraction for Controlling Parallelism*. Research Report 42, Digital Equipment Corporation Systems Research Center, April 1989.

[12] Niklaus Wirth. *Programming in MODULA-2*. Springer-Verlag, third edition, 1985.

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985. Revised October 31, 1986.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987. Revised September 16, 1988.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987. Temporarily withdrawn to be rewritten.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

"Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms."
Leonidas J. Guibas and Jorge Stolfi.
Research Report 37, February 14, 1989.

"Can fair choice be added to Dijkstra's calculus?"
Manfred Broy and Greg Nelson.
Research Report 38, February 16, 1989.

"A Logic of Authentication."
Michael Burrows, Martín Abadi, and Roger Needham.
Research Report 39, February 28, 1989.

"Implementing Exceptions in C."
Eric S. Roberts.
Research Report 40, March 21, 1989.

"Evaluating the Performance of Software Cache Coherence."
Susan Owicki and Anant Agarwal.
Research Report 41, March 31, 1989.

"WorkCrews: An Abstraction for Controlling Parallelism."
Eric S. Roberts and Mark T. Vandevoorde.
Research Report 42, April 2, 1989.

"Performance of Firefly RPC."
Michael D. Schroeder and Michael Burrows.
Research Report 43, April 15, 1989.

"Pretending Atomicity."
Leslie Lamport and Fred B. Schneider.
Research Report 44, May 1, 1989.

"Typeful Programming."
Luca Cardelli.
Research Report 45, May 24, 1989.

"An Algorithm for Data Replication."
Timothy Mann, Andy Hisgen, and Garret Swart.
Research Report 46, June 1, 1989.

"Dynamic Typing in a Statically Typed Language."
Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin.
Research Report 47, June 10, 1989.

"Operations on Records."
Luca Cardelli and John C. Mitchell.
Research Report 48, August 25, 1989.

"The Part-Time Parliament."
Leslie Lamport.
Research Report 49, September 1, 1989.

"An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon."
David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink.
Research Report 50, September 10, 1989.

51 Experience with the Firefly Multiprocessor Workstation
by Susan Owicki

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301