Adaptive Parallelism on Multiprocessors: Preliminary Experience with Piranha on the CM-5

Nicholas Carriero, Eric Freeman, and David Gelernter

YALEU/DCS/RR-969
May 1993

Adaptive Parallelism on Multiprocessors: Preliminary Experience with Piranha on the CM-5

Nicholas Carriero, Eric Freeman, and David Gelernter *
Department of Computer Science
Yale University
New Haven, CT 06520

May 20, 1993

Abstract

Mechanisms for sharing multiprocessors among users are still in their infancy—typical approaches include simple space-sharing and inefficient, restricted forms of time-sharing. In this work we investigate a new alternative: *adaptive parallelism* [2]. Adaptively parallel programs can execute over a *dynamically changing* set of processors; many such codes can easily and dynamically share a multiprocessor by individually adapting to execute in separate groups of processors that may vary with time. Adaptive parallelism has been successfully used in *Piranha*, an execution model for Linda[1] programs that turns idle networked workstations into a significant computing resource [4]. This work explores Piranha on the Connection Machine CM-5 multiprocessor. Our preliminary results suggest that adaptive parallelism provides not only substantial computing power for parallel programs but also an attractive alternative to traditional methods for sharing multiprocessors among users.

1 Introduction

Large, scalable, multiprocessor computers are the emerging dominant species of supercomputer. How do we share such a machine among its users? Solutions are still in their infancy—space-sharing through static partitions and inefficient, restricted forms of time-sharing are currently common.

In this paper we explore the *adaptive parallelism* [2] alternative. Adaptive parallelism refers to a parallel application that executes over a set of dynamically changing processors. Adaptive parallelism has been used successfully in *Piranha*, an execution model that harnesses the computing power of idle network workstations [4].

Specifically we discuss Piranha on the CM-5 multiprocessor [3].[2] Our results suggest that adaptive parallelism can provide an alternative to traditional methods of sharing multiprocessors.

*This work is supported by Air Force Grant AFOSR-91-0098 and NASA Graduate Research Fellowship NGT-50858. CM-5 resources were provided by the Pittsburgh Supercomputing Center and the National Center for Supercomputing Applications.

[1] A registered trademark of Scientific Computing Associates, New Haven.

[2] Thinking Machines Corporation Disclaimer: these results are based upon a test version of the CM-5 software where the emphasis was on providing functionality and the tools necessary to begin testing the CM-5. This software release has not had the benefit of optimization or performance tuning and, consequently, is not necessarily representative of the performance of the full version of this software.

2 Adaptive Parallelism

A program exhibiting "adaptive parallelism" executes on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds.

In any computing environment (including a dedicated multiprocessor), an adaptively parallel program is capable of taking advantage of new resources as they become available, and of gracefully accommodating diminished resources without aborting. Our previous work has focussed on workstation networks (conventional UNIX workstations with standard LAN interconnects). Workstations at most sites tend to be idle for significant fractions of the day, and those idle cycles may constitute in the aggregate a powerful computing resource. Although the Ethernet and comparable interconnects weren't designed for parallel applications, they are able in practice to support a significant range of production parallel codes. Ongoing trends make "aggregate LAN waste" an even more attractive target for recycling: desktop machines continue to grow in power; better interconnects will expand the universe of parallel applications capable of running well in these environments. For these reasons and others, we believe that adaptive parallelism is assured of playing an increasingly prominent role in parallel applications development over the next decade.

Work on the Piranha system (discussed in the next section) has shown that a variety of coarse-grain parallel applications can indeed be run effectively as adaptive applications on LANs [4]. In some cases, supercomputer-equivalents of computing power can be achieved merely by recycling the garbage.

3 Piranha

The Piranha programming model is an adaptive version of master-worker parallelism [1]. Programmers specify in effect a single general purpose worker function, called `piranha()`. They do not explicitly create processes and their applications do not rely on any particular number of active processes. When a processor becomes available, a new process executing the `piranha()` function is created there; when a processor withdraws, a `retreat()` function is invoked, and the local piranha process is destroyed. Thus there are no "create process" operations in the user's program, and the number of participating processes varies with available processors.

Workers perform *tasks*. A task consists of a set of input data from which a worker function produces a set of output data and possibly a collection of new tasks. Tasks may be ordered or unordered, depending on the logical requirements of the application.

Piranha applications may use Linda's shared, associative object memories ("tuple spaces") to store descriptors and other data in distributed data structures accessible to all worker processes. The underlying Piranha system itself uses another tuple space to coordinate user and system activity.

The Piranha system is discussed in greater detail and compared to alternative models in [2].

4 Adaptive Parallelism on Multiprocessors

How should large distributed-memory multiprocessors be shared?

Most current approaches to the sharing of such machines center on space-sharing. The machine's nodes are typically paritioned into disjoint sets. A user grabs one set to run his job. Jobs "share" the machine by running in separate sets. But there are problems with this approach. If the machine is shared among several jobs running in fixed partitions, left-over nodes may be wasted if they don't correspond to the static partition requirements of any waiting application. If parallel

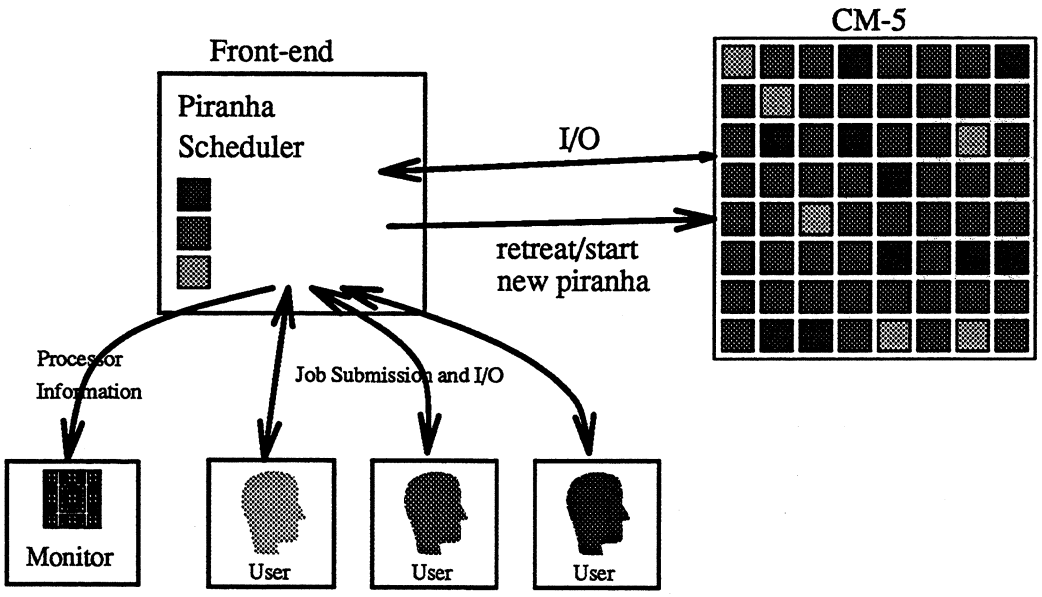

Figure 1: CM-5 Piranha Design.

job J is sharing the machine with another job that completes, J is confined to its initial partition despite the existence of idle nodes freed up by the terminated job. Even if more jobs are queued, they may not need or want as large a partition as the just-completed job occupied; or, J might be a higher-priority job, more deserving of newly-available resources. But current programming approaches don't allow for dynamic expansion and contraction of parallel programs.

Time-sharing is always a possibility in principle: many jobs might share a single partition or the entire machine, with each running only during designated periods. But time-sharing as implemented (in a preliminary way) on the CM-5 is an inflexible and inefficient strategy; unlike uniprocessor time-sharing (where I/O and execution are overlapped), the current CM-5 version never preempts waiting or blocked processors. Context switches are only performed when the quantum expires, and as a result, processor time is wasted when a job is blocked on I/O. As implemented, CM-5 time-sharing reduces machine utilization rather than improving it. A more flexible time-sharing strategy would pose hard problems involving co-scheduling of processes with inter-dependencies. Timesharing in any case does not address the problem of making effective use of partitions of the wrong size or shape.

Piranha represents a different approach. Under Piranha, a large multiprocessor is treated as a pool of resources to be dynamically allocated among competing jobs. Suppose a new job is dropped into the pool but all nodes are currently in use. If the priority of the job warrants, currently running jobs may be forced to back off dynamically, freeing nodes for the new job. When a job terminates and leaves the machine, existing jobs may dynamically expand to fill the available space. A long running job might thus (for example) maintain a toe hold on a small number of nodes during particularly busy periods, and expand to fill a much greater number of nodes during relatively less busy times. Piranha doesn't in itself dictate the scheduling policy under which a node decides which available job it should be executing (research on such scheduling policies is ongoing): Piranha does provide a framework under which such schedulers become plausible and useful.

Clearly, some parallel jobs won't be amenable to this kind of dynamic treatment; they will

require a fixed number or configuration of nodes. We do know at present that a broad collection of useful applications *can* be Piranhafied. Our intention is to allow conventional fixed-partition jobs to be embedded in a dynamic Piranha environment.

5 CM-5 Piranha: design and implementation

Ideally we would like the Piranha runtime system to be a scheduler for all programs, allowing Piranha programs and native fixed-topology CM-5 programs to co-exist. Native CM-5 jobs would request specific resources from the Piranha scheduler. Piranha jobs would contract and expand around the native jobs to recycle previously unused machine time into Piranha computations. In addition, multiple Piranha jobs could expand and contract relative to each other to use available resources more efficiently.

In reality we currently only manage adaptively parallel codes with our Piranha system (because of the current CM-5 OS design). Adding native CM-5 jobs is only a small logical step beyond—they are in concept Piranha jobs that never expand or retreat. Our current system manages multiple Piranha jobs within one of the "traditional" partitions of the CM-5—although if all we (and the system managers) cared about were Piranha codes, that partition could be the whole machine.

Figure 1 is an overview of the runtime system. The runtime system consists of a job submission and scheduling process on the CM-5 front-end, which we call the "scheduler." The scheduler accepts requests to start user jobs, and assigns those jobs to subsets of nodes on the partition. The scheduler then directs that each reassigned node reboot and reload itself with the new job's executable, executing the old job's retreat function in the process. All I/O from the processors passes back through the partition manager. Monitoring facilities can be connected to the scheduler at any time.

Scheduling is handled in a simple way by allocating a "fair" portion of processors to each job. Figure 2 shows the scheduling behavior we have in mind. Initially there are no jobs in the system and all processors are idle. Job 1 enters the system and consumes all processors that are available. At timestep 20 job 2 enters, with job 1 still present. Job 1 is asked to give up some nodes, and both jobs proceed with half the processors available in the system each. At time step 60, job 2 leaves and job 1 reclaims those processors. Job 3 then enters the system and we have the same situation as before, each job computing on half the processors. Finally job 4 enters the system, and both jobs 1 and 3 are asked to give up processors, resulting in each process computing with roughly one-third of total processors.

This describes the current scheduler. One can envision (and we will soon be implementing) a more general system. For instance, when a job no longer needs some of its processors, they can be allocated to another process. Under this scheme jobs do not always acquire $1/n$ processors, where n is the number of jobs in the system; they acquire as many processors as are available without exceeding what is necessary.

5.1 CM-5 Piranha Scheduler: implementation

The Piranha scheduler controls the execution of multiple Piranha jobs on the CM-5 in a global manner. The scheduler accepts requests from users and from Piranha processes to perform various tasks, and makes scheduling decisions based on a processor table that maps CM-5 processors to Piranha jobs.

Figure 3a shows a state diagram for the scheduler. This scheduler sits in a dispatch loop waiting for requests. Three types of requests are possible: user requests, piranha requests, and

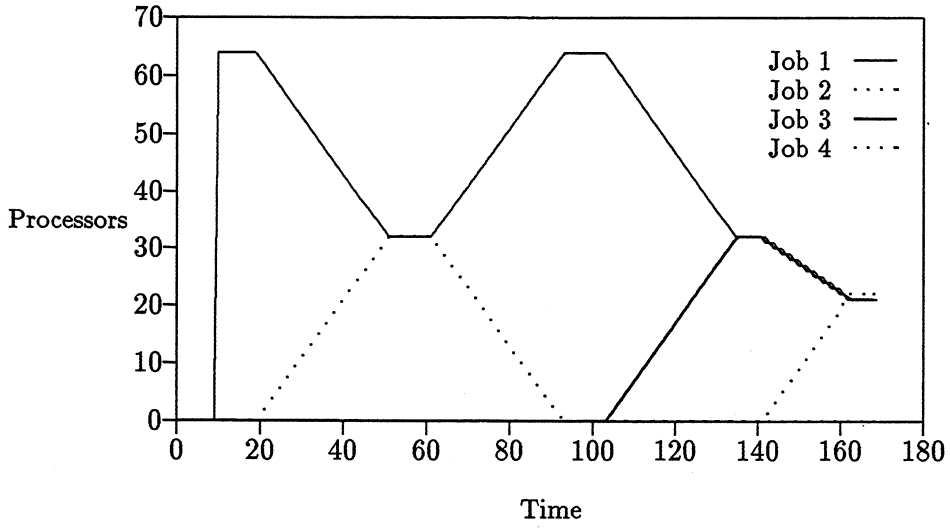

Figure 2: Runtime Characteristics of Piranha on a 64 node system.

master requests. The only request a user can make is to run a Piranha job. When this occurs the scheduler reassigns processors to the new job by, first, recording the new system state in the processor table. The scheduler then goes through the table and interrupts all processors allocated to the new job. The selected processors are then "rebooted" (a process described in detail later) and sent the new executable. The scheduler then returns to the dispatch loop.

Masters send a request to the scheduler when they have completed their computations, which notifies the scheduler that the master and all its associated piranha can be recycled to other jobs. The scheduler goes through the same process above, reassigning processors to the remaining jobs and then sending executables.

The Piranha request is sent from an individual piranha that has finished computing tasks. This situation occurs, for instance, when a Piranha application computes in two phases and the second phase requires less piranha (workers) than the first. By notifying the scheduler, the existing piranha's processor can be reassigned to another job.

5.2 CM-5 Piranha Loader: implementation

Because the CM-5 supports only the single program, multiple data (SPMD) programming model, a small bootstrapping loader is needed to provide a multiple program, multiple data (MPMD) environment for Piranha. The Piranha loader resides on each node and loads executables broadcast from the scheduler. After loading the executables into memory, the loader starts the new program.

In the event that the scheduler reassigns a node to a new job, the node is rebooted by interrupting the node and returning control to the loader. The loader then executes a code segment that calls the retreat function, cleans up the stack, and waits for the next executable.

Figure 3b shows the state diagram of the loader. The loader initially receives an executable from the scheduler and then jumps to the entry address of the executable. At this point the user's Piranha code takes control of the processor. If the Piranha code finishes normally, control returns

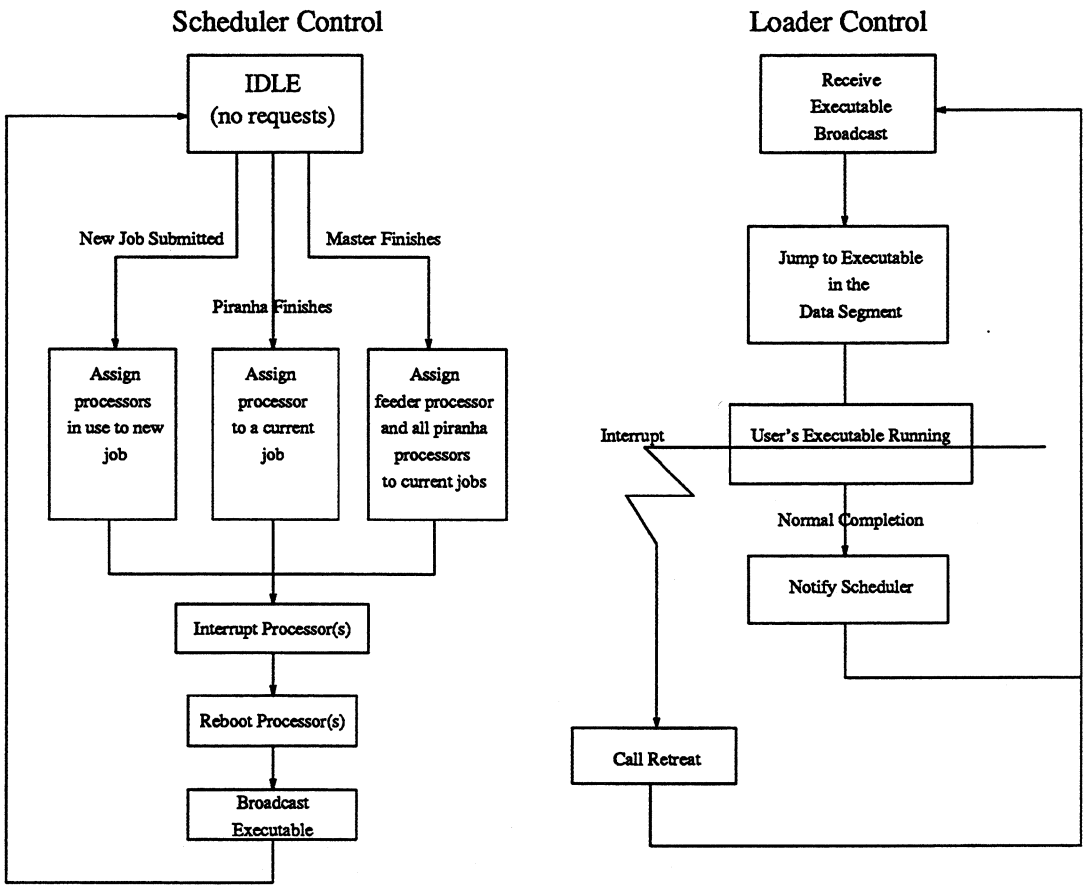

Figure 3: Scheduler and Loader Control.

Program	Sequential	CM-5 Piranha	Speedup	Efficiency
Neutrino	2729 secs	64 secs	43	68%
Dipole	1144 secs	75 secs	15	24%

Figure 4: CM-5 Piranha Test suite.

to the loader and notification is sent to the scheduler. If the processor is interrupted to run another Piranha executable, the retreat handler executes and control returns to the loader at the top level.

6 Evaluation

Is Piranha effective at sopping up idle resources and gracefully making room for new jobs without introducing excessive overhead? We used the following as a preliminary test. Consider some applications that "don't fit" on our experimental 64-node partition: they keep every node busy only part of the time. If Piranha is effective, running many such jobs simultaneously should be more efficient than running many sequentially in batch mode: Piranha should make it possible to put the nodes that are wasted in batch mode to good use.

6.1 Preliminary Results

Two Piranha programs developed in collaboration with other departments at Yale are used in our tests: a physics code for computing propagation of neutrinos and an electrical engineering code for finding dipole localization in biomagnetic imaging; both are described in [4]. The neutrino code is an "unordered bag of tasks" program. A number of tasks are generated and computed by the piranhas. The dipole code is also a bag of tasks program, but with two phases. The first phase locates minima and the second phase further localizes them [4].

We present the sequential and parallel execution times of these programs in figure 4. The sequential version ran on a Sparc processor with the same performance as a CM-5 node. The piranhafied version ran on a 64-node CM-5 partition. Neutrino generates 96 tasks of equal size and thus uses the machine fully to execute the first 64 tasks (one task per node), then uses only half the machine to execute the remaining 32 tasks. Over the entire run, neutrino is able to use the 64 nodes with 68% efficiency. With dipole it is harder to use the CM-5 efficiently: it fully uses the machine in the first pass, but in the second pass it generates only 4 tasks, using 1/16th of the machine; over the entire run it uses the 64-node partition with 24% efficiency. Both codes are good candidates for our Piranha system insofar as they both waste resources: the neutrino code because it is impossible to load-balance its tasks evenly across the nodes of the partition (resulting in some nodes idling while others work), and the dipole code because the second phase requires fewer resources than the partition provides (again a mismatch of available work to available nodes, so some go idle). Note that, while we could adjust problem size to correct for this in the case of the neutrino code, such adjustment is not straightforward for the dipole code. For the current problem size, both present a challenge.

6.2 Comparison of Piranha and Batch Execution

Figure 5 shows data for 1 to 8 *simultaneous* runs of neutrino in Piranha and 1 to 8 *successive* runs of neutrino in the batch system. Figure 6 shows the same information for dipole. Batch time

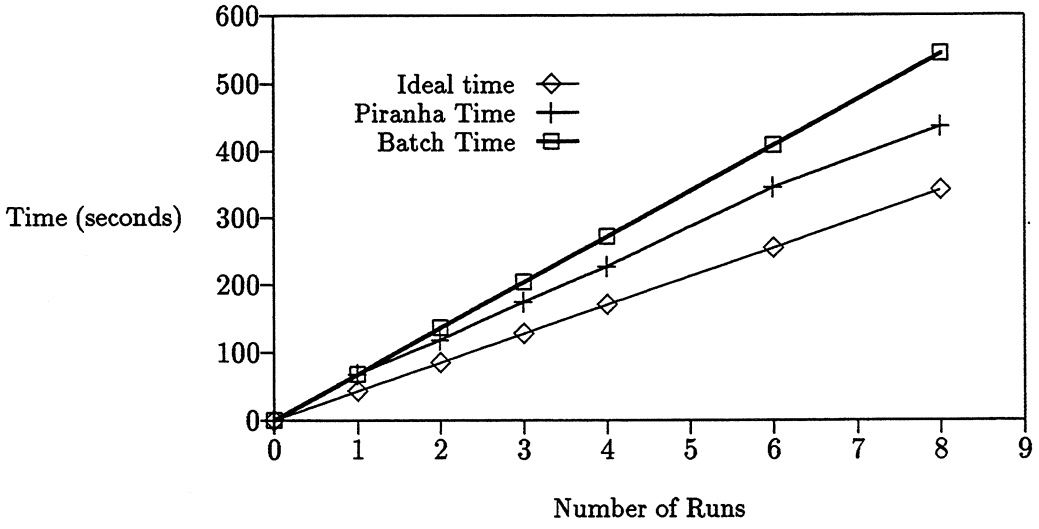

Figure 5: Neutrino Performance in Piranha (64 nodes).

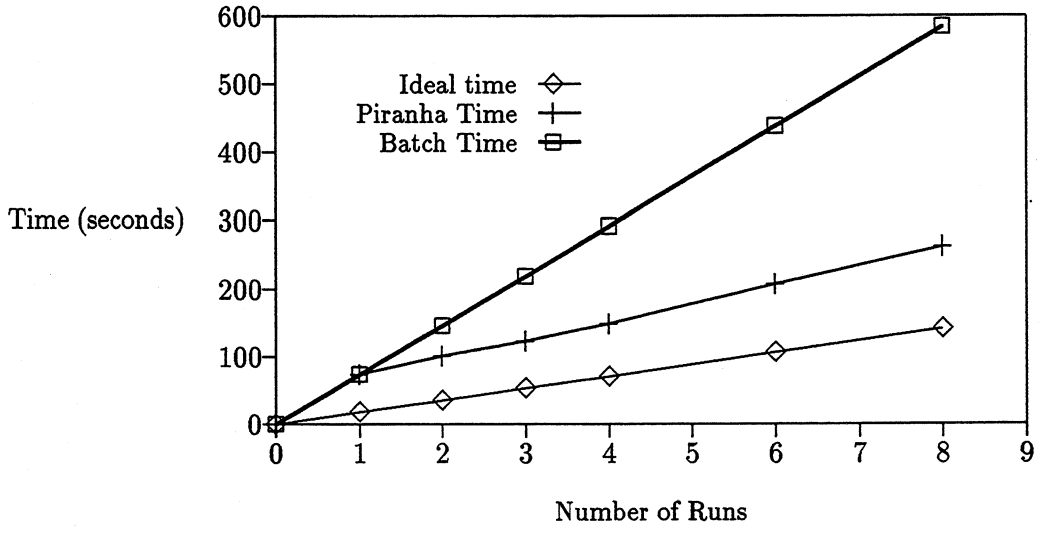

Figure 6: Dipole Performance in Piranha (64 nodes).

Program	Batch	Piranha	Ideal	Batch Efficiency	Piranha Efficiency
Neutrino	544 secs	436 secs	341 secs	68%	78%
Dipole	584 secs	263 secs	142 secs	24%	54%

Figure 7: CM-5 Piranha Efficiency for 8 runs on 64 nodes.

Program	Batch Overhead	Piranha Overhead	Ratio
Neutrino	203 secs	95 secs	2.14
Dipole	442 secs	121 secs	3.6

Figure 8: Comparison of Piranha and batch overhead for 8 runs on 64 nodes.

in both cases is the amount of time required to execute n runs of the code in the CM-5 batch system (time-sharing would require more time than batch). The Piranha time is the time required to execute n runs of the code in the Piranha system, where multiple copies of the program run simultaneously. The ideal time is the amount of time required to execute n runs of the program with perfect speedup.

In both cases Piranha does better than the native batch system. Figure 7 shows the detailed results of running 8 versions of each code. In the case of neutrino we raise the efficiency from 68% (in the case of batch) to 78% with Piranha. Dipole does even better, because a single run uses the machine more inefficiently. With dipole efficiency rises from 24% to 54%.

Another way to compare Piranha and the batch system is to examine overhead (where overhead is the difference between execution time and perfect speedup). In figure 8 we show the overhead for neutrino and dipole over eight runs. In the case of neutrino, the batch system's overhead is roughly twice that of the Piranha system. With dipole, the overhead of the batch system is more than three times that of Piranha.

7 Conclusions

Adaptive parallelism provides substantial computing power to a piranha application and it also provides a novel method for sharing multiprocessors; it has the further advantage that multiple programs can be run concurrently, as in a time-sharing system, without the disadvantages of current multiprocessor time-sharing systems. Our early results suggest that this form of sharing may well be more efficient than the CM-5 batch system. Much work needs to be done to study this form of computation further. Future work will include thorough testing of a more diverse set of Piranha programs and dynamic scheduling methods. Our goal is to integrate Piranha fully into the CM-5 environment, so that Piranha applications can co-exist with native CM-5 applications.

8 Acknowledgments

We wish to thank David Kaminsky and Rob Bjornson for their invaluable help during this project. We also wish to thank the fine technical support of Thinking Machines Corporation—in particular Mike Flanigan and Adam Greenberg. Alan Klietz of the Minnesota Supercomputer Center was also of great help.

References

[1] Nicholas Carriero and David Gelernter, *How to write parallel programs: A first course.* (Cambridge: MIT Press, 1990).

[2] Nicholas Carriero, David Gelernter, David Kaminsky, and Jeffrey Westbrook. "Adaptive Parallelism with Piranha", YALEU/DCS/RR-954, February 1993.

[3] The Connection Machine CM-5 Technical Summary, Thinking Machines Corporation, Cambridge, MA, 1992.

[4] David Gelernter and David Kaminsky. "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha", Proceedings of the ACM, International Conference on Supercomputing, July 19-23, 1992.