Adaptive Parallelism and Piranha

Nicholas Carriero, Eric Freeman,
David Gelernter and David Kaminsky

YALEU/DCS/RR-1016
February 1994

Adaptive Parallelism and Piranha

Nicholas Carriero* Eric Freeman † David Gelernter ‡
David Kaminsky §

Department of Computer Science
Yale University
New Haven, Connecticut 06520

February 7, 1994

Abstract. Under "adaptive parallelism," the set of processors executing a parallel program may grow or shrink as the program runs. Potential gains include the capacity to run a parallel program on the idle workstations in a conventional LAN—processors join the computation when they become idle, and withdraw when their owners need them—and to manage the nodes of a dedicated multiprocessor efficiency. Experience to date with our Piranha system for adaptive parallelism suggests that these possibilities can be achieved in practice on real applications at comparatively modest costs.

Keywords: Parallelism, networks, multiprocessors, adaptive parallelism, programming techniques, Linda, Piranha.

1 Introduction

Most work on parallelism is "static": it assumes that programs are distributed over processor sets that remain fixed throughout the computation. If a program starts out on 64 processors, it runs on exactly 64 until completion, and specifically on the same 64. "Adaptive parallelism" (AP) abolishes the requirement that processor sets remain fixed; processors may join or withdraw from the computation as it proceeds. Hence a computation that begins on 64 nodes may finish on 3, having executed at various points on 65, 79 and 15 processors in the meantime. Even if a computation beginning on 64 processors finishes on 64, they needn't be the same 64 on which it started. Because the size of the processor set may go to 0, an adaptive parallel program may be descheduled entirely during its run and remain within the bounds of the model.

*Research partially supported by Air Force Grant AFOSR-91-0098
†Research partially supported by NASA Training Grant NGT-50858
‡Research partially supported by Air Force Grant AFOSR-91-0098
§Research partially supported by NASA Training Grant NGT-50719

Adaptive parallelism raises three sets of related questions: for what sorts of problems is it appropriate? How should programs be structured? How should the underlying systems software work? We consider these questions in the framework of a particular implementation of adaptive parallelism called the "Piranha" system. After discussing the motivation for adaptive parallelism we describe the Piranha model, and then outline a series of application programming experiments.

2 What's the point?

The costs of adaptive parallelism are clear: an adaptive program is likely to be more complicated than a static one, the underlying system will be more complex, and expansion and contraction will exact runtime costs. Against these disadvantages we need to set any potential gains, and then try to establish by experimentation whether the proposition is worthwhile.

The potential gains are large. Consider first the environment that is, for now, by far the most important for production use of parallelism: conventional LANs of standard workstations. Workstations at most sites tend to be idle for significant fractions of the day, and those idle cycles may constitute a powerful computing resource in the aggregate. Ongoing trends make "aggregate LAN waste" an even more attractive target for recycling: desktop machines continue to grow in power; better interconnects will make communication cheaper, and in doing so expand the universe of parallel applications capable of running well in these environments.

One well-known approach to node recycling centers on job-level parallelism: users offload jobs from their workstations onto idle ones. Systems such as Condor [LL90], Butler [Nic90], Sprite [OCD+88], and others support this approach. Adaptive parallelism takes a different tack: instead of parceling out waiting user jobs to idle workstations, we spread a *single* user job over many workstations, thereby focusing the available power on one application.

Of course, many AP programs might contend simultaneously for the idle resources of a single LAN—in which case we apportion resources according to some appropriate scheduling discipline. In the worst case we are juggling as many AP programs as there are idle nodes, and the gains from parallel execution go to 0. There may even be net losses of efficiency, owing to the overhead introduced by the underlying AP system. But we suspect that the worst case won't crop up often. For one, there is an inherent irregularity to the demands that tend to be imposed at most sites on shared resources. Further, cheap hardware, economies of scale and the maintenance economies to be gained from uniform environments make it attractive at many sites to issue the same workstation to everyone. Inevitably, workstations end up on the desks of users who need relatively few cycles. Such environments have a built-in idle pool.

Adaptive parallelism is potentially valuable on dedicated multiprocessors as well, particularly on massively parallel processors (MPP's). It's clear that such machines are destined to become our principle supercomputing workhorses; their astronomical cost suggests that

each will have a sizable user community. How should large MPP's be shared?

Currently most such machines are space-multiplexed: user jobs are statically assigned to some partition. But there are problems with this approach. If parallel job J is sharing the machine with another job that completes, J is confined to its static 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.

Time-sharing is always a possibility in principle: many jobs might share a single partition or the entire machine, each running only during designated periods. But time-sharing involves swapping and scheduling overhead, and poses certain logical problems in the presence of parallel applications. In some cases, all processes of a parallel job will need to run simultaneously in order to avoid performance degradation or even deadlock.

Adaptive parallelism represents an alternative scheduling strategy. The MPP is treated as a resource pool for dynamic allocation among competing jobs. Suppose a new job is dropped into the pool but all MPP nodes are currently in use; if the priority of the job warrants, currently running AP jobs may back off dynamically, freeing nodes for the new job. When a job terminates and leaves the machine, existing AP jobs 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. Clearly, some parallel jobs won't be amenable to this kind of dynamic treatment; they will require a fixed number or configuration of nodes. But we do know at present that a broad collection of useful applications *can* be refitted for effective AP execution and that AP and "fixed" jobs can co-exist peacefully.

3 The Piranha Model

Several *ad hoc* systems have been designed to solve specific computational tasks adaptively—for example testing primality or computing traveling salesman tours [LM90]. Other approaches to adaptive parallelism have tended to center on what we call the "process model," in which an application is structured as a set of *processes* that grows and shrinks without regard for the number of available processors, and is often much larger than the number of available processors. Processes are dynamically remapped among free processors as needed. When a processor withdraws, its processes are migrated somewhere else. Such an approach was discussed as long ago as the "MuNet" project and the early stages of Actors research [Agh86]; a variant of this approach formed the basis of the "Amber" adaptive parallelism system [CAL+89].

Piranha, in contrast, is an adaptive version of master-worker parallelism (see [CG90]). Programmers specify in effect a single general purpose (but application specific) "worker function" called `piranha()`. They do not 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 special `retreat()` function (which is also provided by the application programmer) is invoked, and then the local piranha process is destroyed. Thus, there are no "create process" operations in the user's program, and the number of participating *processes* (and not merely processors) varies dynamically.

The Piranha model is (not exactly by coincidence) a perfect fit to Linda's[1] tuple-space-based coordination model. A tuple space is a virtual shared, associative, object memory accessible to all nodes within a parallel computing environment—for example, to all workstations on a LAN. Tuple space supports *uncoupled* communication: producers and consumers of data need not know each other's identity or location; they always interact via the intermediary tuple space. Uncoupling is clearly desirable when supporting a system in which processes' locations (physical addresses) may be constantly changing. And because tuple space is persistent—its lifetime is not bounded by the lifetime of any particular process—this memory remains available as individual piranha are created and destroyed. The Piranha system stores task descriptors, results, intermediate problem state and other relevant information in tuple space.

Piranha's approach to adaptive parallelism has a number of advantages. Processes need never be moved around: *task descriptors* and not *processes* are the basic movable, remappable unit in the computation. This approach supports strong heterogeneity. A process image representing a task in mid-computation can't realistically be moved to a machine of a different type, but a task descriptor can be. In our implementation of Piranha, task descriptors are stored in tuple space. Since the underlying tuple space implementation is heterogeneous, nodes of different types can share access to tuple spaces and, in particular, they can all access the same pool of task descriptors. Thus a task begun by a Sun node can be picked up and completed by an IBM machine. (Industrial harmony in action.)

Our approach has the disadvantage that applications must spell out explicitly what steps to take when a process is forced to vacate a node. The process won't simply be frozen and moved; the programmer must supply a special handler (`retreat()`) to deal with the situation. The next section discusses `retreat()` and the other elements of a Piranha program.

4 Elements of a Piranha Program

A Piranha program must define three distinguished routines, called `feeder`, `piranha` and `retreat`. These three collectively establish the coordination framework—the code that holds the multiple processes of a parallel application together and coordinates their activities.

A Piranha job has a single, persistent *feeder* process and a varying number of *piranha* processes. Their collective role is to transform a distributed data structure describing a problem into one that describes its solution. If a piranha process is forced to leave a

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

computation, it executes a short *retreat* function and exits.

The primary role of the feeder is to manage the ongoing computation. It creates a distributed *task data structure* holding work to be done. The piranha processes transform the task data structure into a distributed *result data structure* holding the solution to the problem. Typically, each piranha process executes a loop in which it claims a task, computes its result and adds the result to the result data structure. When the computation is complete, the feeder gathers the results and presents them to the user.

The feeder is not restricted to creating the tasks and collecting the results; it can perform tasks too. Typically the feeder builds the task data structure and the piranha processes transform it into the result data structure, but the roles can be more symmetric: both can assist in the construction of either.

In simple cases, there is no ordering among tasks; they may all be processed simultaneously. In more complex cases, inter-task dependencies require that tasks be started in some particular order. When a process completes a task, it might add data to the result data structure, to an auxiliary distributed data structures holding dependency data or to both. To consume a task, a process removes a task descriptor from the task data structure and gathers any dependency data from other distributed data structures. All distributed data structures are stored in tuple space, where they are directly accessible to all participating processes.

When a node becomes unavailable, any piranha currently running there must exit. The user supplied `retreat()` function is responsible for making this exit an orderly one. If the process modified any distributed data structures, retreat allows it to restore their consistency; the computation then continues without the process. In a simple case, a retreating process need only replace its current task in the task data structure. In more complicated cases, retreat may store intermediate results in some appropriate data structure, may adjust the task data structure to indicate that the interrupted task must be picked up and resumed with high priority (because other tasks depend on it), or may store other information about the current status of the computation.

To insure consistency in the presence of retreat, Piranha routines define a critical section around the operations in which they update the result data structure and withdraw a new task from the task data structure; during this interval they may not be interrupted by retreat. Modifications to distributed data structures outside of this critical section must be carried out using special tuple space operations that insure consistency of shared data in the event of a retreat-induced interrupt.

A computation begins when the feeder begins executing. A user activates the feeder by typing the name of the executable, along with any arguments, at the command-line. The feeder, which we assume belongs to the user who submitted the piranha job, is required in our current implementation to stick with the computation from beginning to end; the feeder never retreats. (This requirement may be relaxed in later versions of the system.)

On every node that becomes idle during the course of the computation (other than the one on which the feeder is executing), a process is automatically created to execute the

`piranha()` function. No "create process" operations appear explicitly in a user's piranha program. Each piranha process remains active until the entire computation is finished, or until the owner of the node on which it is executing indicates (explicitly or implicitly by, e.g., touching a key or moving the mouse) that he wants his node back.

When a node's owner needs his node back (or, in the case of a dedicated MPP, the system scheduler decides that a node should be reassigned to a different job), the piranha process is interrupted and the retreat routine is automatically invoked by the system. Retreat procedures execute under a system timer. If they exceed the allotted time they are terminated, and the entire piranha computation aborts. Only piranha processes are subject to interruption and retreat.

The *Linda Program Builder*[AG92] supports parallel programming with a series of built-in templates and high-level operations (in addition to other services). The three-part piranha program framework is a template supported by the Program Builder. Our goal is to integrate into the Program Builder a large collection of sample `retreat` routines as well, in hopes that most programmers will find an appropriate or almost-appropriate routine already in stock when they need one.

5 Application Experiments

Like all parallel applications, a Piranha program must be developed with both correctness and efficiency in mind. Piranha applications face another requirement as well: they must continue to execute correctly and efficiently in the presence of arbitrary additions to and deletions from the set of hosting nodes.

We discuss several examples below. The first, *Atearth*, is about as simple as a Piranha program can be; its tasks are completely independent. The second, DNA sequence similarity assessment, demonstrates the more elaborate program structures that are required when there are inter-task dependencies—relatively complex ones in this case. Finally we briefly discuss some experiments with Piranha on Thinking Machine's CM-5 multiprocessor.

5.1 Atearth

Atearth, written by Martin White of the Yale Physics department, simulates the flight of neutrinos from the sun towards earth. The simulation consists of a number of trials, where each trial simulates the flight of a neutrino with given characteristics (e.g., energy and direction of flight). The trials are independent, making this code an ideal candidate for master/worker parallelism. In the Network Linda implementation, a task consists of one trial. The master generates tasks, the workers consume them, carry out the simulations described and return the results.

The corresponding Piranha program is a straightforward transformation of the master/worker program: the `feeder()` generates task descriptors and deposits them in tuple space. Each `piranha` repeatedly grabs a task, computes its result and places the answer in

tuple space. The `feeder` collects the results. Pseudocode for this type of program is given in figure 1.

Table 5.1 shows the performance of Atearth. We present data for sequential, Piranha and Network Linda versions of the program. "Sequential" time is for unmodified Atearth, a Fortran code. "Network Linda" time is for a static[2] Linda version running on the same workstation network on which the "Piranha" times were obtained.

Figure 2 shows how each value in the efficiency table is calculated. For the Piranha version we measure run time and worker time. Worker time is the aggregate compute time used by the piranha processes. In the examples presented, feeders do not consume tasks. Since feeders do not speed the computation, we chose to omit feeder time from worker time. Strictly speaking, since feeders consume node cycles, feeder time should be included in worker time. However, omitting feeder time promotes a direct comparison with sequential time: in both cases we measure only process time spent working on the problem. In addition, as we examine an application's scaling properties, omitting the feeder from a plot of efficiency versus number of nodes gives us a truer picture of the change in efficiency with an increasing number of nodes (if included, the graph would initially show a tendency for efficiency to *increase* as nodes are added). We treat the master process of master/worker codes in a similar way.

In our benchmarks, we attempted to balance the average *compute intensity*—the compute power available per unit time—available to Piranha and to Network Linda. We first ran Piranha five times and computed the average number of nodes used. We then ran Network Linda using a fixed number of nodes close to the average number used by Piranha. When we could not equalize the compute intensity, we used a slightly lower Network Linda intensity. This favors Network Linda, because using fewer nodes tends to improve efficiency.

Even when we balance compute intensities, a direct comparison of Network Linda and Piranha understates Piranha's value. Piranha not only *uses* idle resources, it also *finds* them. We can run Piranha at any time, knowing that it will ferret out available idle cycles. If a Network Linda job is to avoid interfering with interactive work we must run it only when we can find a sufficient number of free nodes, which must be available not only when the job starts but throughout the run. Guaranteeing that a node will remain idle for the duration of a computation is difficult, especially when running during the day. If we do not know that a node will be idle (or if it is unexpectedly so), that node's power is wasted. In many cases, these requirements can only be met by running Network Linda at night on reserved nodes.

The efficiency of Piranha Atearth was 94% with respect to the sequential code and 98% efficient with respect to the Network Linda code. Since Atearth is very coarse grained (an average of 1.6 seconds per task) and its task descriptor and result tuples are small (less than 50 bytes each), it has a high computation:communication ratio and excellent efficiency.

We also benchmarked Atearth on a large collection of Sparcstation 1's. Data are pre-

[2] "Static" as opposed to adaptive, i.e. not changing during the course of a *single* run—but here, as with virtually all Linda master/worker codes, the number of workers can be adjusted from one run to the next.

```
feeder ()
{
  int i;
  while (get_task (&data)) {
    out ("task", i, data);
    task_count++;
  }

  for (i=0; i < task_count; i++) {
    in ("result", i, ? result_data);
    store_result (&result_data);
  }
}

int current_task_id;
piranha ()
{
  /* retreat blocked until enabled */
  while (1) {
    in ("task", ? current_task_id , ? data);
    task_start (); /* explicitly enable retreat */
    compute_result (current_task_id, & data, & result_data);
    task_done (); /* block retreat */
    out ("result", current_task_id, result_data);
  }
}

retreat ()
{
  out ("task", current_task_id, data);
}
```

Figure 1: Unordered graph code skeleton

- Run times (R_s, R_p and R_{nl}) are measured wall clock times
- Worker time:
  - Piranha worker time (W_p) is the sum of the time used by each Piranha process (excluding the feeder): $W_p = \sum_i W_i$
  - Network Linda worker time (N_{nl}) is the product of the number of workers and the run time: $W_{nl} = N_{nl} \times R_{nl}$
- Average number of nodes:
  - The average number of Piranha workers (N_p) is calculated by dividing the worker time by the run time: $N_p = W_p/R_p$
  - The number of Network Linda workers (N_{nl}) is a run-time parameter
- Speedup is calculated by dividing the run time by the sequential time:
  - Piranha: $S_p = R_s/R_p$
  - Network Linda: $S_{nl} = R_s/R_{nl}$
- Efficiency is calculated by dividing the sequential time by the worker time:
  - Piranha: $E_p = R_s/W_p = R_s/\sum_i W_i$
  - Network Linda: $E_{nl} = R_s/W_{nl} = R_s/(N_{nl} \times R_{nl})$

Figure 2: Piranha evaluation measures

	Average Nodes	Run Time	Worker Time	Speed Up	Eff. (%)
Sequential		25.0			100
Piranha	8.5	3.13	26.7	8.0	94
Net Linda	8.5	3.07	26.1	8.1	96

Table 1: Atearth performance. Time is given in hours. The Piranha pool consisted of 10 IBM RS/6000 model 340's. The aggregate time wasted to retreat was .04 hours. Fractional values for the number of nodes used in a Network Linda run are obtained by averaging runs with different numbers of nodes.

	Average Nodes	Run Time	Worker Time	Speed Up	Eff. (%)
Sequential		3.6			100
Piranha	29	.13	3.8	28	94

Table 2: Performance of the Atearth on a network of Sparcstation 1's. Times are given in hours. A total of 33 nodes participated in the computation.

sented in table 5.1 for Atearth on a pool of 33 nodes. Even using a relative large set of nodes, Piranha performed 94% efficiently.[3] Note, though, that this is "wind assisted"—the computation to communication ratio is much higher for the Sparcstation 1's than for the RS/6000s.

5.2 DNA Sequence Similarity Assessment

[CG90] discusses DNA sequence similarity assessment using a technique developed by Gotoh [4]. The algorithm requires a program structure that accommodates inter-task dependencies. The program computes the elements of a matrix starting at the top, left-hand corner. The value of each matrix element depends on the initial sequences and on the values of the elements to its west, northwest and north. The parallel version divides the matrix into blocks. The subdivision includes some overlap, which eliminates the dependency of each block on the block to its northwest. A portion of the graph showing intertask dependencies is shown in figure 5.2.

Our starting point is a master/worker code that assigns one of the initial rows of matrix blocks to each worker. Each worker starts with a block along the left-hand edge of the matrix and computes all sub-blocks to the right of it. Before computing a block, a worker reads the bottom edge from the block above. When a worker completes a block, it releases the the block's lower edge. When a worker completes a row, it obtains another row assignment. For this non-adaptive code, a worker's forward progress depends on a steady stream of edges coming from the worker cruising the row of blocks above. A worker that stalls or exits will eventually cause workers beneath it to hang.

The Piranha version defines a task to be the computation of a single block of the matrix. We label each task with the number of incomplete tasks on which it immediately depends. Figure 5.2 shows the initial labeling of each task. Each process claims and completes one enabled task (or blocks if none exists). After completing the task, it decrements the

[3] At the time these data were gathered, we did not collect data on Network Linda. Since heterogeneous collection of newer machines subsequently replaced our Sparcstation 1's, we omit a comparison to Network Linda.

[4] Faster algorithms for DNA sequence comparison have been developed. We use this algorithm to illustrate an important class of Piranha programs.

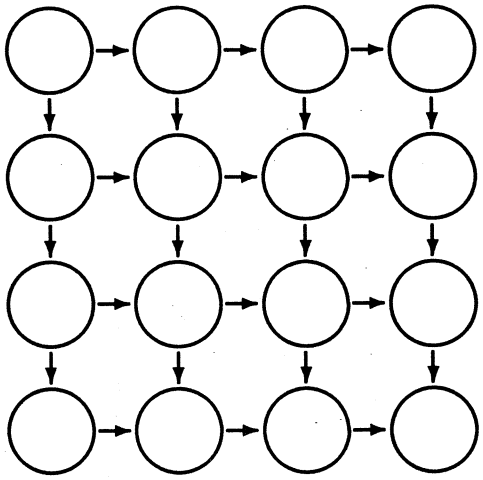

Figure 3: The intertask dependency graph for a 4×4 Piranha block sequence comparison.

dependency count of the tasks to its right and below it. If the task to its right is now enabled, the piranha process publishes the lower edge of its current task, decrements the count of tasks on which the task below depends, and claims the task to the right.

If the task to the right is not enabled, but the task below is enabled, the piranha process publishes the right edge of its current task, and claims the task below. If neither the task to the right nor the task below is enabled, the process publishes both the right-hand and lower edges of its current task.

Note that if either adjacent task is enabled, we avoid passing one set of edges. In any case, we are guaranteed to make progress since processes work only on enabled tasks (i.e. those depending only on published edges). When a Piranha retreats, it releases its current task and the edges on which it depends. Performance data is given in Table 5.2.

5.3 Experiments with Multiprocessor Piranha

In addition to our Network Piranha results, we have obtained results for our multiprocessor version of Piranha[CFG94] from runs of Piranha codes on a 64 node partition of a Connection Machine CM-5.[5] Our CM-5 Piranha implementation presents an essentially complete user-level Piranha system, but it does not offer fully adaptive run-time support. In particular, a node's "availability" is not a function of its actual load but of whether or not the

[5]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.

0	1	1	1
1	2	2	2
1	2	2	2
1	2	2	2

Figure 4: Each square of the matrix is labeled with the number of tasks on which it immediately depends. When a dependency is satisfied, the number in the square is decremented.

	Average Nodes	Run Time	Worker Time	Speed Up	Eff. (%)
Sequential		359			100
Piranha	3.25	134	432	2.7	83
Net Linda	3	131	393	2.7	91

Table 3: Sequence comparison performance on two 17,000 base pair sequences. The comparison matrix was divided into 10 rows and 50 columns. Time is given in seconds. The Piranha pool consisted of 4 IBM RS/6000 Model 340's.

Piranha scheduler has assigned it a piranha.

All nodes in our dedicated Piranha partition start out unassigned. When a Piranha job is submitted all unassigned nodes start executing its `piranha()` function—whether or not the job can use that many nodes. As a result some nodes that are really idle (running a "starved" piranha) appear busy. In the network case, such a node still looks idle because the load average will be unaffected by the starving piranha, but here the scheduler assumes the node is now busy because the node has been assigned to a job and that it will remain busy until the job is finished. So, for example, a job is submitted that has only ten tasks—it is still given 64 piranha. Ten piranha get something to eat, the rest go hungry. Another job is submitted that has 54 tasks. The scheduler will take away half of the first job's nodes (not necessarily just "starved" nodes either—the system as it stands simply doesn't distinguish between truly and apparently busy, so retreats may be directed to either sort of node) and give them to the second, even though the second could use more and the first needs fewer. So, effectively, the current Piranha system dynamically partitions the machine among Piranha jobs—beneficial in itself as we shall see, but much more is possible. To achieve what is possible, more work on both the Piranha system and the CM-5's OS is needed. One goal of our current effort is to characterize the necessary modifications.

Because of the current run-time limitation, our interpretation of the CM-5 results will differ somewhat from the interpretation of the data from network Piranha experiments. We will focus here on how Piranha can help a busy machine run more efficiently. Consider some applications that "don't fit" on our 64-node partition: they keep every node busy only part of the time. Piranha should make it possible to put the nodes that are unneeded by one job to work on other jobs—effectively giving each job less of an opportunity to waste nodes.

Our tests used the Atearth code described above and an electrical engineering application that performs dipole localization in biomagnetic imaging, also developed at Yale. The dipole code is a two-phase bag-of-tasks program: the first phase locates minima within a coarse grid, the second phase further refines the positioning[GK91].

We present the sequential and Piranha execution times of these programs in figure 5. 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. For these runs, Atearth generates 96 tasks of equal size and thus uses the partition fully to execute the first 64 tasks (one task per node), then uses only half the machine to execute the remaining 32 tasks. Over an entire run, Atearth is able to use the 64 nodes with 68% efficiency. The dipole code fully uses the machine in the first pass, but in the second pass it generates only 4 (more complex) tasks, which use only 1/16th of the machine; over the entire run it uses the 64-node partition with 24% efficiency.

It is, of course, possible to select problem sizes here that will better match the resources available—but that is not really the point. We are interested in creating systems that are generally accommodating of mismatches, both because we want to make use of whatever resources happen to be available (i.e. they are not fixed) and because for some codes (the dipole code is one example) it is not easy to categorize resource requirements *a priori*. As a first step, we chose to investigate the performance of the current CM-5 Piranha system with

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

Figure 5: CM-5 Piranha Test suite for Single Runs.

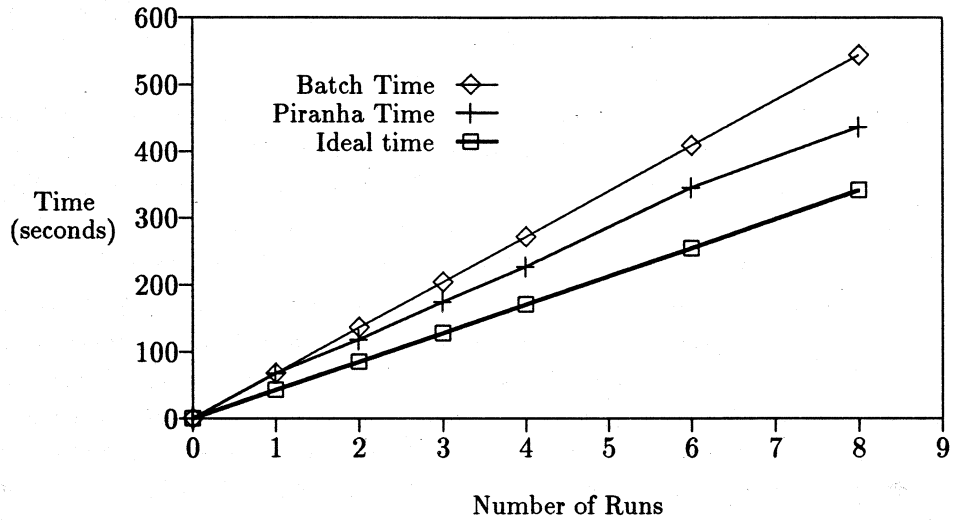

Figure 6: Atearth Performance in Piranha (64 nodes).

test cases that are likely to make its potential clear in this context (and that accommodated the limited machine time at our disposal).

Figure 6 shows data for 1 to 8 *simultaneous* runs of Atearth in Piranha and 1 to 8 *successive* runs of Atearth in the batch system. Figure 7 shows the same information for dipole. Batch time in both cases is the amount of time required to execute n runs of the code in the CM-5 batch system (as currently implemented, 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 8 shows the detailed results for the case of eight simultaneous runs of each code. In the case of Atearth we raise the efficiency from 68% (batch) to 78% with Piranha. Dipole does even better, because a single run uses the machine less efficiently. With dipole efficiency rises from 24% to 54%.

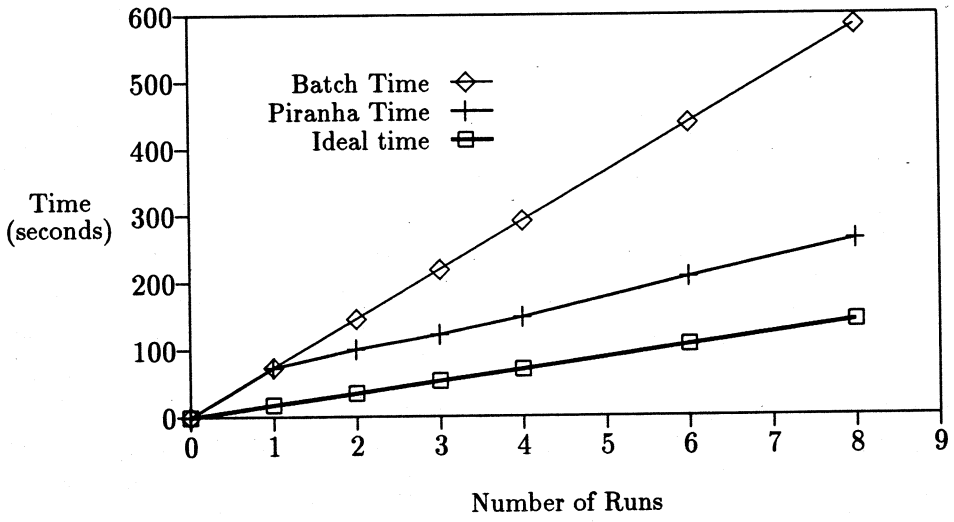

Figure 7: Dipole Performance in Piranha (64 nodes).

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

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

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

Figure 9: Comparison of Piranha and batch overhead for 8 runs on 64 nodes.

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 5.3 we show the overhead for Atearth and dipole over eight runs. In the case of Atearth, 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.

Figure 5.3 shows data for intermixed job streams of the dipole and Atearth codes. Each data point represents a job stream in which dipole and Atearth are each run five times in an intermixed and random fashion. Each job stream was run on a clean CM-5 partition (there were no other processes running); there was always at least one Piranha job executing. Note that this does not imply that all processors were always busy; there might, for example, be only one Piranha job in the system, and that job might not use all 64 processors. The horizontal axis represents the job stream's "load average". This load average is computed by averaging over the number of jobs running in the Piranha system over the course of the run. The horizontal line at approximately 655 seconds[6] is the time it would take to execute five runs of dipole and Atearth in the batch system (this time is independent of job order).

At a load average of 1 we have the equivalent of the batch system—there is only one job in the Piranha system at a time—and as a result the piranha and batch execution times are the same. As we move in the direction of increasing load average, the machine is used more efficiently until we reach a load average of approximately three, after which is no additional benefit to packing jobs more tightly. Note that in the neighborhood of a 1.1 load average we actually do worse than the native batch system on account of Piranha overhead (most likely in the form of computation lost to piranha retreats).

If we again consider the ratio of the runtime overheads, the overhead of the batch system is roughly twice that of the Piranha system. In this calculation the perfect speedup time for the job streams would be 302 seconds and the batch overhead is $655 - 302 = 353$ seconds. We use the roughly asymptotic runtime value of 484 for Piranha (in the case of a 4.0 load average) to compute an overhead of $484 - 302 = 182$ seconds and a ratio of $353/182 = 1.9$.

[6]Based on the data in figure 5 one might expect the batch execution time to be 695 seconds; however, the job stream tests were conducted after additional improvements to the operating system and message passing libraries were made by Thinking Machines.

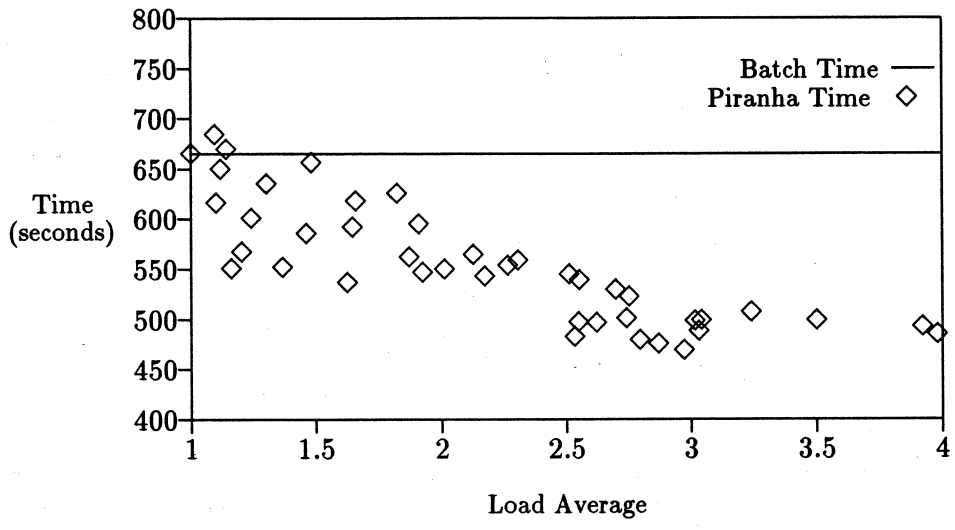

Figure 10: Random streams of jobs consisting of five runs of dipole and five runs of atearth intermixed.

6 Implementation Overview

We briefly summarize the architecture of the Network and CM-5 Piranha systems. More details on these systems can be found in [Kam94, Fre94].

6.1 Network Piranha

An executing Network Piranha system comprises the run-time system of Piranha daemons (one per participating node), and a (possibly empty) set of Piranha user applications. Each Piranha daemon controls the behavior of its node with respect to Piranha applications. It determines whether the node is idle and manages Piranha applications on its node. Daemons communicate with their peers through tuple space.

The feeder process runs on the submitting user's node. Piranha processes are started on idle nodes. By default, Piranha uses two sets of criteria: the keyboard, mouse and remote logins must be idle for five minutes and the one, five and ten minute load averages must be below 0.4, 0.3 and 0.1, respectively. A node's owner can modify these settings if he chooses.

6.2 Multiprocessor Piranha

In theory, a multiprocessor Piranha system could act as a scheduler for all programs, allowing Piranha programs and native fixed-topology programs to co-exist. Native multiprocessor jobs request specific resources from the Piranha scheduler. Piranha jobs contract and expand around the native jobs to recycle previously unused machine time. In addition, like Network Piranha, multiple Piranha jobs expand and contract relative to each other to use available resources more efficiently.

Our CM-5 Piranha runtime system consists of a job submission and scheduling process that runs on the CM-5 front-end, which we call the "scheduler." The Piranha scheduler accepts job requests from users. It assigns each job to a subset of nodes in the partition, directs those nodes to perform a retreat if necessary, and then start the new job.

CM-5 Piranha currently manages only Piranha codes (because of the current CM-5 OS design), but adding support for native CM-5 jobs is only a small logical step beyond—they are in essence just Piranha jobs that never expand or contract. The CM-5 Piranha Scheduler schedules adaptive jobs in a "fair" manner. Any time there are available CM-5 nodes, each active piranha job gets a roughly equal portion of them (by forcing retreats, if necessary). Other scheduling strategies are possible and are being investigated.

7 Conclusions

Adaptive parallelism has the potential to exploit the considerable but usually wasted aggregate power of idle workstations. It has the potential to integrate heterogeneous platforms seamlessly into a unified computing resource. And it has the potential to permit more efficient sharing of "traditional" parallel processors than current systems allow

Piranha has allowed us to realize this potential for a number of applications and hosting platforms. It has addressed, in the process, the questions and concerns raised earlier. AP is certainly appropriate for pure master/worker codes, but it is also applicable to codes involving low to moderate intertask dependencies. [Kam94] presents a methodology for structuring codes based on the nature of their intertask dependencies. [Kam94] also presents a detailed description of the underlying implementation.

The coding effort required for AP above and beyond what a "static" parallel code requires is a strong function, not surprisingly, of the complexity of the task interdependencies. It also depends, to a lesser extent, on the importance of capturing work-in-progress when a task retreats. What is surprising is that the simplest cases can be handled with virtually no additional coding effort.

Performance results indicate that AP's runtime costs are often modest—allowing for significant performance improvements via recycled idle resources in the LAN setting, and substantial reductions in sharing inefficiencies in the MPP setting.

References

[AG92] Shakil Ahmed and David Gelernter. A CASE environment for parallel programming. In *Proc. Fifth International Workshop on Computer-Aided Software Engineering*, July 1992.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[CAL+89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[CFG94] Nicholas Carriero, Eric Freeman, and David Gelernter. Adaptive parallelism on multiprocessors: Preliminary experience with Piranha on the CM-5. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, 1994.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A first course.* MIT Press, Cambridge, 1990.

[Fre94] Eric Freeman. Piranha on the Connection Machine CM-5. Technical Report YALE/DCS/RR-1011, Yale University, February 1994.

[GK91] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Sixth ACM International Conference on Supercomputing*, July 1991.

[Kam94] David L. Kaminsky. *Adaptive Parallelism with Piranha.* PhD thesis, Yale University, 1994.

[LL90] M. Litzkow and M. Livny. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October 1990.

[LM90] A. K. Lenstra and M. Manasse. Factoring by electronic mail. In *Proceedings of Eurocrypt '89*, number 173 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[Nic90] D.A. Nichols. *Multiprocessing in a Network of Workstations.* PhD thesis, Carnegie-Mellon University, 1990.

[OCD+88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(6):23–36, February 1988.