Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha

David Gelernter and David Kaminsky

YALEU/DCS/RR-883
December 1991

Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha

David Gelernter David Kaminsky

December 9, 1991

Abstract

In this paper we present a new system for making use the cycles routinely wasted in a local area network. In Piranha, we harness these cycles to run explicitly parallel programs. Programs written for Piranha are very similar to Linda master/worker programs[5]. We have used Piranha to run a number of production applications.

We present a description of the Piranha prototype, briefly explain the Piranha programming methodology, and explore different types of Piranha algorithms.

This work was supported by the National Science Foundation under grant number CCR-8657615 and NASA under grant number NGT-50719.

1 Introduction

As local area networks spanning large numbers of powerful workstations become commonplace, researchers have come to realize that at most sites, many nodes are idle much of the time. Ideally there would be some way to recapture some of these lost cycles, which grow increasingly formidable in the aggregate as workstations grow more powerful.

In the Piranha model, idle nodes are recycled by focusing them on explicitly-parallel programs. The user builds a parallel program, structured as a collection of worker processes sharing access to a distributed data structure in which task descriptors are stored. The *number* of worker processes (or "Piranhas") expands and contracts as the program executes. Any participating workstation has the option of joining the ongoing computation (in other words, running a worker process) when it becomes idle. When a user reclaims his workstation, it leaves the ongoing parallel computation and returns to normal duties.

The Piranha model makes no assumptions in the abstract about how parallel applications are structured. In practice, Piranha is an immediate fit to the Linda model: Linda[1][5] makes it easy to structure computations as collections of workers sharing access to distributed data structures. Thus our Piranha system is, in practice, an execution model and support system for a certain class of Linda programs.

This paper focuses on our experience in using Piranha to deliver formidable quantities of compute power on production codes. The data are preliminary but (we think) suggestive. It might in fact be possible to recapture supercomputer- or near-supercomputer-equivalents of compute power that are routinely wasted and focus them effectively on significant applications.

2 The Piranha Model

Piranha gathers idle network nodes and uses them to execute parallel programs. Restricting Piranha applications to idle nodes leaves slices of time typically ranging from a few minutes to many hours or days (see figure 1). The Piranha system is capable of making productive use of idle intervals regardless of their length (see figure 2). (It does take Piranha some time to decide that a node is not available and to start a Piranha job. This small amount of time is wasted.)

As nodes transition between being *available* and *unavailable*, the number of nodes participating in a Piranha computation varies. Piranha applications reconfigure themselves automatically when nodes are lost to owner reclamation (a transition from available to unavailable) or gained via idleness (see figure 3). Before a reclaimed node withdraws it executes a short piece of code (called `retreat`), written by the applications programmer and invoked by the system, that restores its piece of the global state of the computation. If a workstation

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

Figure attached at the end of the paper.

Figure 1: Wasted Node Time
The shaded area show the time that was wasted by nodes during a twenty-four hour period. White areas show the times a node was used by its owner. The data begin at 6:00 pm on a Tuesday. A typical "workday" is shown at the end of the graph.

Figure attached at the end of the paper.

Figure 2: Recycled Node Time
This figure shows a histogram of the time claimed by Piranha during a run of the Neutrino code (see below). The run was done at 12:30 pm on a Friday. The total run time is approximately 5.5 hours. The aggregate computation time is 132 hours. Shaded areas show the intervals that Piranha used each node. Unshaded areas indicate times where a node was claimed by its owner. The `retreat` function (see below) was called at each transition from shaded to unshaded.

becomes idle, some Piranha application incorporates it into its computational ensemble.

A standard way to design a Piranha program is to have a set of worker processes consume a collection of tasks (see figure 4). A free node spawns a process to consume tasks from the collection. The more nodes available, the faster the tasks are consumed. This type of program structure easily accommodates the dynamically changing node set intrinsic in the Piranha model. Other types of program structures will be discussed below.

In deciding how best to support these programs, the tight conceptual fit between the needs of Piranha and the resources provided by Linda become apparent immediately. Linda's tuple spaces easily support Piranha's task collections, other global state objects, and general communication among processes whose identities are mutually unknown and whose lifetimes might not overlap.

Figure attached at the end of the paper.

Figure 3: Piranha State Diagram
Starting in the *Available* state, if a Piranha application is submitted, a Piranha is spawned and the system state becomes *Exec Piranha.* Completion of the application leads to a transition back to the *Available* state. If an owner demands the node in the *Exec Piranha* state, the `retreat` function is called. After `retreat` completes, the systems transitions to *Exec User.* When the owner no longer needs the node, the system returns to the *Available* state.

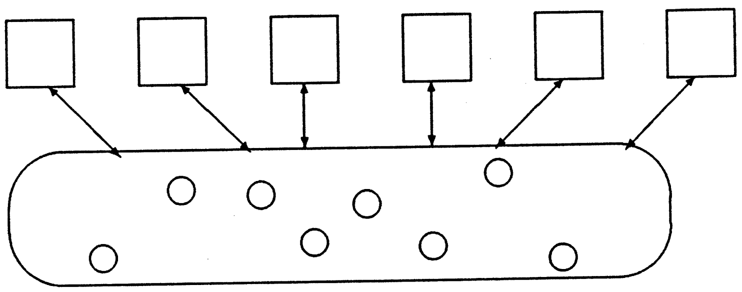


Figure 4: A dynamically changing set of Piranha share access to a global pool of tasks. In the Linda implementation, the "global pool of tasks" is a single tuple space.

3 Piranha in Context

It has long been understood that computations can be run profitably on idle network nodes [15]. Over the last few years, many projects have focused on offloading jobs from heavily loaded nodes to idle ones. Examples include Amoeba [17], Butler [12], Condor [11], Sprite [13], and V [6]. Such systems deliver *job level parallelism*: a collection of (essentially) independent and unrelated sequential programs can be run simultaneously.

Some studies suggest that job level parallelism may not be a very effective way to soak up idle cycles. (The Sprite developers report that over two-thirds of their workstations are idle during the middle of the day, yet less than five percent are used by migrated processes [8].) At any rate, Piranha's goals are different: it attempts to deliver maximum performance on a *single* job, assuming that this job can be expressed as a parallel program of the appropriate type.

Piranha, in short, provides a mechanism for solving large problems via parallel programs on idle workstations. Systems such as Condor or Sprite provide a more pleasant working environment by keeping per node workload down.

NeXT's Godzilla system [7] is probably Piranha's nearest neighbor among existing systems. Godzilla allows a user to fork worker processes on idle network nodes. Each process computes a result and passes it back to a master process. Interprocess communication is severely restricted: Parameters are passed from the master to the worker when the process is started, and results flow back on task completion. No communication takes place among workers. Piranha, on the other hand, supports (via Linda) full interprocess communication. This capability is important: many Linda applications rely on mutable state objects stored in tuple space, and Piranha itself relies on tuple space in ways to be discussed below.

4 Implementation

The Piranha kernels manage the system's behavior. They run as low priority daemons, one per participating node. Each kernel monitors the environment on its node and determines whether Piranha applications should be hosted locally. Using a set of user defined criteria (with system-supplied defaults), the node is declared either ***available*** to host Piranha applications or ***unavailable***. If a node is available, its kernel consults a global table of pending Piranha applications stored in a system tuple space (***not*** the tuple space through which applications communicate). In our prototype, the kernel chooses an application from the list at random and **`forks`** a process to execute it. If a node running a Piranha application switches from being available to unavailable, the `retreat` function (see below) is called and the computation on that node suspends.

Piranha programs consist of three basic functions: **`feeder`**, **`piranha`**, and **`retreat`**. The **`feeder`** runs on the so-called "home node", the node belonging to the user who submitted the job. Our current implementation requires that a user who submits an application donate his node for the duration of the computation, so that the feeder is never suspended. Its functions often include distributing tasks (creating the task collection) and gathering the results. It may also join in consuming tasks.

The **`piranha`** function is automatically executed on all nodes that join a computation, except for the home node. Typically, a piranha function executes a loop that reads tasks and consumes them until none remain. The code to consume a task is application-dependent.

If a node aborts a computation in the middle of a task, the `retreat` function restores the global state of the system. In the simplest case, `retreat` returns the uncompleted task to tuple space. More complicated `retreat` functions pass accumulated state from a retreating process to some other Piranha process.

5 Evaluating Piranha

Evaluation of Piranha performance is difficult because the resource that drives the system varies drastically from run to run. The standard speedup measurements are not helpful, because we cannot control the number of usable Piranha nodes (or rather we can, but the moment we do, we are no longer measuring a Piranha system as we have defined it). Furthermore, because the production codes presented in this paper require very large amounts of computing time, their sequential runtimes are hard to measure directly.

Another measurement omitted from this paper is the marginal value of a node to a Piranha computation. One would expect that adding a node to a computation would result in a reduction of computing time. However, we have noticed large variations in execution times. These variations are caused by differing numbers of node **`retreats`** and by different network loads. Furthermore,

Table 1: Piranha vs. Network Linda

Run	Network Linda	Piranha
1	322	311
2	354	330
3	339	342
4	319	336
5	324	338
Avg	332	331
Min	319	311

Piranha and network Linda run times (in seconds) for five runs of an 18 city traveling salesman problem run on 16 Sparcstations.

the characteristics of a nighttime run differ drastically from those during the daytime.

Our first goal is to minimize the overhead imposed by Piranha. The Piranha prototype is an enhanced version of Network Linda and runs on Unix workstations. We tested Piranha performance against standard network Linda on an 18 city traveling salesman problem run on 16 Sparcstations. We chose this problem because it is compute intensive (communication-intensive jobs will not show speedup over the Ethernet, with its low throughput and large latencies, whatever the communications model) but includes inter-worker as well as master-worker communication. The results are listed in Table 1. We ran the tests at night to avoid network interference and Piranha suspensions. (No artificial restrictions were imposed on Piranha. Nodes happened to be and remained idle). The differences in times easily fall within experimental error. These data suggest that Piranha imposes very little overhead on the underlying network Linda system for this application. (The overhead imposed by Linda *itself* has also been found to be small relative to non-Linda sequential codes on a broad range of production applications.[4] But evaluating the cost of Linda *per se*, as opposed to the overhead of Piranha versus plain Linda, is outside the scope of this paper.)

Delivering all available machines is the responsibility of the Piranha system. The effect of adding nodes to a Piranha computation depends on the design of the application. Well designed Piranha programs (in fact all Piranha programs that we have observed) complete more quickly if more nodes are available. Figure 5 shows the effect on idle nodes of submitting a Piranha application.

In a third test, we measured the system's impact on node owners. In deciding whether a node is available, we consider one and ten minute UNIX load averages and keyboard and mouse idle time. Watching for keyboard and mouse activity insures that Piranha will not run when the owner is using his node interactively. The load average measure stops Piranha from interfering with compute intensive

Figure attached at the end of the paper.

Figure 5: Free Nodes During a Piranha Run
Before a Piranha application was submitted, over 20 nodes were available. Once the application became active, any idle nodes immediately joined the computation. After the calculation was completed, nodes once again became idle.

processes that the owner may have left running. We chose highly conservative criteria to promote a low profile Piranha system. In principle, many more criteria could be used in the availability decision (e.g. virtual memory use and free disk space).

Before we can install Piranha on a node, we must be given explicit permission by the owner. If Piranha disrupts his work, he may request that it be removed. In order to form an initial impression of user satisfaction with the system, we performed a blind test to see if users could tell when their nodes were hosting Piranha applications. We submitted a Piranha application at arbitrary times and measured whether node owners could detect the resulting effect on their nodes. We polled three users whose nodes were running Piranha and two whose nodes were running a similarly named program that did nothing (it executed a UNIX `sleep`). We ran the Piranha program ten times and polled users five times. Node sluggishness was reported twice. Neither incident proved to be attributable to Piranha. These small tests agree with our general experience to date: Over 60 node owners in the Yale Computer Science Department have contributed their nodes (often with much reluctance after heavy persuasion); none have withdrawn their nodes, although their right to do so was clear from the start. Large production applications have used the system intermittently during this period.

6 Experience with Piranha

The prototype has been used to run a number of scientific and industrial codes. In this section we will discuss different types of Piranha algorithms characterized by the type of `retreat` function they require.

6.1 Solar Neutrino

A maximally simple Piranha application consists of processes consuming an unordered bag of tasks. When a Piranha is active, it removes a task, consumes it, and returns a result. When a process retreats, it returns the task to tuple space. A neutrino simulation code [9] run under Piranha takes this form.

The problem is described as follows:

> The program assumes a set of parameters (mass differences, mixing angles and magnetic moments) for the neutrinos and solves nu-

Table 2: Time delivered to the Solar Neutrino code

Elapsed time	Compute time	Avg Nodes
13.0	558.5	43.0
8.0	320.2	40.1
6.1	252.9	41.4
5.7	132.4	23.4

Piranha time (in hours) over four runs of the Solar Neutrino code. The elapsed time is wall clock time. Compute time is the sum of the times delivered by each participating Piranha; that is, it is aggregate delivered compute time.

> merically the quantum mechanical equations for the propagation of the neutrinos from their point of origin, through the Sun, to the Earth. The properties of the Sun are taken from the standard solar model. Comparison of the experimentally observed event rates with our predictions for various parameter sets allows us to constrain, statistically, the possible neutrino parameters, providing information about possible extensions to the standard model of particle physics. [19]

The sequential version of the Solar Neutrino code, written in Fortran, consists of four nested DO-loops. Each iteration is independent of the others. We parallelized the code by creating a task (consisting of four parameters) for each loop iteration.

The Piranha version consists of a Piranha-Linda front end, a Fortran back end, and a `retreat` function. The front end performs task management using tuple space. The back end receives the parameters from the front end and does the computation. The Fortran code for the back end was taken directly from the sequential version. The `retreat` function simply returns the active task (a set of parameters) to tuple space. Since each task depends only on the four parameters, nothing more is required of `retreat`. As shown in Table 2, we have delivered over 1,250 cumulative hours of recycled computation time to the neutrino code in less than 33 elapsed hours.

6.2 Dipole Localization

A second example of the bag-of-tasks paradigm is a dipole localization code.[2] The Piranha program has two phases, each solved using a bag-of-tasks. The first

[2]The biomagnetic imaging problem can be stated as follows: given a set of magnetic field measurements from a discrete array of sensors with a known geometry, find the positions and magnitudes of N test dipoles such that the squared error between the magnetic field produced by these test dipoles and the given magnetic field measurements is minimized. This amounts to minimizing an objective function (the squared error) in a 3-N dimensional search space.

phase locates the minima in subspaces, and the second phase further localizes them. Within each phase a task defines a search region described by a set of coordinates. Each search is independent, so no global state is retained. The `retreat` function returns the task (again a set of coordinates) to tuple space. In two runs of the Dipole code, Piranha yielded in total over 800 computation hours in a total of less than 20 clock hours.

6.3 Freewake

In a slightly more complicated class of Piranha program, tasks must be completed in some order. Rather than an unordered bag-of-tasks, we use an ordered bag (i.e. a task queue). Processes continue to remove tasks from the head of the queue until all have been consumed. When a process retreats, it places its active task at the head of the queue.

It should be noted that we are not working with a strict task queue. If the tasks must be finished in strict order, then the program is inherently sequential at the task level (i.e. we cannot introduce parallelism at the task level). Instead, we are concerned with problems that allow parallel execution of the tasks within a sliding window. In other words, given a window of size k, if tasks 1 through $i-1$ have been completed, tasks i through $i+k-1$ may be executed in parallel. Task $i+k$ cannot be started until task i is completed. The window of tasks that may be consumed slides as earlier tasks are completed. Retreated tasks are placed at the head of the queue so the window can continue to slide.

An example is a computational fluid dynamics code used to simulate a helicopter rotor. The code, called Freewake,[3] is an n-body problem run for a succession of timesteps [3]. The steps within an iteration are run using sliding window parallelism. It is possible to rewrite this code using a bag-of-tasks, but we chose sliding window to preserve the program structure.

Table 6 presents the times to run a Linda version of Freewake on Piranha, a network of Sparcstations running TSnet[4], and the Intel iPSC/860.[5] Forty-six nodes were in the Piranha pool (but only a fraction were idle). The data show that Piranha on an average of 36 nodes was slower than standard Network Linda on 32 nodes. However, the Network Linda runs were done when the

It was found that a number of search techniques would yield reasonable results for $N = 1..4$ even when applied "blindly". But for $N > 4$, all of the search algorithms converged incorrectly to local minima for all but a few carefully contrived examples.

Thus, we decided to do a thorough investigation of the energy surface, focusing specifically on the local minima: where they are located, and their width and depth (relative to the global minima). The program breaks the search space into a number of subspaces, and a coarse-grid local-minima search is performed on these subspaces. The local-minima are then localized using a factorial-pattern search. [14]

[3]Freewake was written by Alan Egolf of United Technology Research Center, Hartford, CT.

[4]TSnet, a registered trademark of Scientific Computing Associates, New Haven, CT, is a network version of Linda.

[5]Data from TSnet and the iPSC/860 was provided by Robert Bjornson[3].

Figure attached at the end of the paper.

Figure 6: Freewake Data
Bjornson [3] ran an 8k Freewake problem for five time steps on a network of Decstations and the Intel iPSC/860. The run times (in seconds) are given for runs on four to 128 nodes. A scatterplot of Piranha runs is also shown. The data show that the Piranha runs were competitive on a per node basis both within the Decstations and the iPSC/860.

machines and network were idle, while the Piranha data were collected midday on a weekday. We attribute the performance differences to network traffic and to work lost by Piranha retreats. Of course, a significant advantage of Piranha is the freedom it gives users to run parallel codes on networks during times of high node use without interfering with owner activity.

6.4 More Complex Cases

The following is a preliminary discussion of more complex Piranha program structures. Runtime data are not yet available, but algorithm design is complete.

Thus far we have focused our discussion on bag-of-task type computations. There is an important class of Piranha algorithms that does not fit this model. Consider a problem in which data is repeatedly transformed until an answer is derived. For example, in an LU decomposition program the columns of a matrix are scaled once per row. We could code such a problem by **in**ing a column, scaling it, and **out**ing it back to tuple space. However, if the dimension of the matrix is large the communications overhead of the tuple space operations quickly becomes unacceptable.

A more efficient solution statically assigns each worker an equal number of columns. A worker is responsible for repeatedly updating its columns. However, a process may be forced to suspend and transfer its columns to another worker. A column imbalance will result. Furthermore, new nodes are not assimilated in this static partitioning scheme. This defeats the goal of making maximum use of idle resources.

We have experimented with a dynamic rebalancing approach that has shown promising results.[6] The key to the algorithm is redistributing work among the active Piranha when the load is out of balance. When a process P is ready to balance, it chooses another Piranha Q. If P has a greater load than Q, P passes half the difference to Q. P and Q are now in balance. If Q had been more loaded (or the loads were equal), no columns would have been transferred. Column transfer is initiated only by the more loaded nodes. This algorithm is attractive insofar as data is moved only when necessary to restore balance.

[6]Joint work with Sandeep Bhatt and Jeffery Westbrook of the Yale Computer Science Department.

Table 3: Column Balancing over Five Piranha

Column range	50-150	75-125	95-105	tolerance
Balances	3.2	5.3	10.7	14.8

Table 3 shows the number of balances needed to bring all of the workers within the range given at the top of the column. Tolerance means that neighbors are within a given tolerance of each other (in this case, neighbors are less than 2 columns apart). Initially, one worker was assigned all 500 columns. Four workers were then allowed to join.

This approach can be extended to allow for entering and exiting nodes. When a node P balances, it first looks for a node Q with no work. If it finds one, P transfers half of its work to Q. Any imbalance between P or Q and other processes is removed using normal balancing. Node suspension is also handled using the balancing routine. When a node suspends, it passes its load to some other process. Normal balancing is used to equalize the load.

We measured the rate at which this algorithm balanced a load among workers. We started with 500 columns assigned to one worker. Four other workers then joined the computation. Table 3 shows the results. 5.3 balances were necessary, on average, to bring the allocations to within 25% of the ideal. 10.7 balances brought them to with 5%. The algorithm required four additional iterations to achieve optimality. These are small numbers of balances when compared to the large number of iterations usually required by Piranha applications.

This algorithm was fairly complicated to design and code. However, the Linda environment allows the user to slot it into place automatically using the Linda Program Builder[1].

6.5 Rayshade

Rayshade, a ray tracing program, is another example of a program that requires partitioning and dynamic rebalancing. Rayshade renders an image row by row starting at the bottom of the frame. The value of a pixel is dependent on the values of some of the pixels below it.

The Piranha implementation of Rayshade divides the columns of the image among the active Piranha workers. Each Piranha shades its columns up to a row barrier. When every Piranha completes the row, a balancing routine is invoked. Since transfer of columns between processes is costly (a worker must do a precomputation for each new column), we move columns among workers only to correct an imbalance. Rayshade requires that each Piranha have a contiguous block of columns, so balancing occurs only between neighbors. As described above, if a Piranha has more columns than a neighbor, half of the difference is passed to that neighbor. After the balancing is completed, the next row is updated.

When a Piranha suspends, it first checks for Piranha that do not have any columns (i.e. those that have recently restarted). If one exists, all of the columns are passed from the suspending process to the new Piranha. If no clean Piranha exist, the suspending process passes its columns to one if its neighbors. Any resulting imbalance is corrected using standard balancing.

Similarly, a balancing Piranha P looks first for a clean one Q. If one exists, P passes half of its columns to Q. This leaves P and Q in balance and insures that all available Piranha will be used in the computation. If P and Q then have fewer columns than their neighbors, normal balancing will restore the global balance. P will only balance with a neighbor if no clean Piranha exist.

7 Future Work

While evaluating the Piranha prototype, we identified a number of issues for future study. The current system runs on a homogeneous network of Sun Sparcstations. (We ported an earlier version to a network of Decstations.) To introduce heterogeneity to Piranha, we must address the issues of data representation and binary incompatibility of executables. Heterogeneity will require network Linda to convert to a standard network byte ordering. Binary incompatibility problems can be overcome by maintaining a separate executable for each architecture.

Also important to Piranha is reliability. We have enhanced network Linda to include an open tuple space called Interp that provides reliability for certain types of tuples. The Piranha kernels use Interp to survive node failure. However, since Interp is slow and does not provide general reliability, it is insufficient for Piranha applications programs. Work has been done on reliable Linda systems (e.g. [2][10]), but none have demonstrated adequate performance. We consider this an important area for future work.

An interesting application of Piranha technology is in scheduling multiprocessor machines. When two programs are run on a multiprocessor (e.g. an Intel iPSC/860), each process is statically assigned some portion of the nodes. If one of the programs completes before the other, some of the processors sit idle. It would be beneficial if a program could use these extra nodes. We believe that the concepts developed in Piranha might be helpful here.

References

[1] Ahmed, S., Carriero, N., Gelernter, D., "The Linda Program Builder", Proc. Third Workshop on Languages and Compilers for Parallelism (Irvine, 1990), MIT Press 1991.

[2] Anderson, B. and Shasha, D., "Persistent Linda: Linda + Transactions + Query Processing", Proc. Research Directions in High-Level Parallel Programming Languages, June 17-19, 1991, Mont Saint-Michel, France, pp. 129-142.

[3] Bjornson, R., "Linda on Distributed Memory Multiprocessors", Ph.D. Thesis, Yale University, 1991.

[4] Bjornson, R., Carriero, N., Gelernter, D., Kaminsky, D., Mattson, T., and Sherman, A. "Experience with Linda", YALEU/DCS/TR-866, 8/91.

[5] Carriero, N. and Gelernter, D. *How to write parallel programs: A first course.* (Cambridge: MIT Press, 1990).

[6] Cheriton, D. "The V Distributed System", CACM, pp 314-333, 3/88.

[7] Crandall, R., "Tales of godzilla: Adventures in Distributed Computing", NeXTon Campus, Summer, 1990.

[8] Douglis, F. and Ousterhout, J., "Transparent Process Migration: Design Alternatives and the Sprite Implementation", Software–Practice and Experience, Vol 21(8), pp 757-787, 8/91.

[9] Gates, E., Krauss, L., White, M., "Solar Neutrino Data and Its Implications", YCTP-P26-91, Yale University, 8/91.

[10] Kambhatla, S. and Walpole, J. "Recovery with Limited Replay: Fault-Tolerant Processes in Linda", Oregon Graduate Institute, Department of Computer Science and Engineering Tech Report CS/E 90-019, 9/90.

[11] Litzkow, M., Livny, M., and Matka, M.W. "Condor - A Hunter of Idle Workstations", Presented at the 8th Intl Conf on Distributed Computing Systems, San Jose, CA, 6/88.

[12] Nichols, D.A. "Multiprocessing in a Network of Workstations", PhD Thesis, CMU, CMU-CS-90-107, 2/90.

[13] Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., and Welch, B.B. "The Sprite Network Operating System", IEEE Computer Vol 21 No 6, pp 23-36, 2/88.

[14] Rao, S. personal communication.

[15] Shoch, J.F. and Hupp, J.A. "The Worm Programs — Early Experience with a Distributed Computation", CACM, pp 172-180, 3/82.

[16] Silverman, R., "Massively Distributed Computing and Factoring Large Integers", CACM Vol 34 No 2, pp 95-103, 11/91.

[17] Tanenbaum, A."Amoeba: A Distributed Operating System for the 1990's", IEEE Computer, pp 44-53, 5/90.

[18] Waldspurger, C., Hogg, T., Huberman, B., Kephart, J., and Stornetta, S. "SPAWN: A Distributed Computational Economy", XEROX-P89-00025, 1989.

[19] White, M., personal communication.

Figure 2

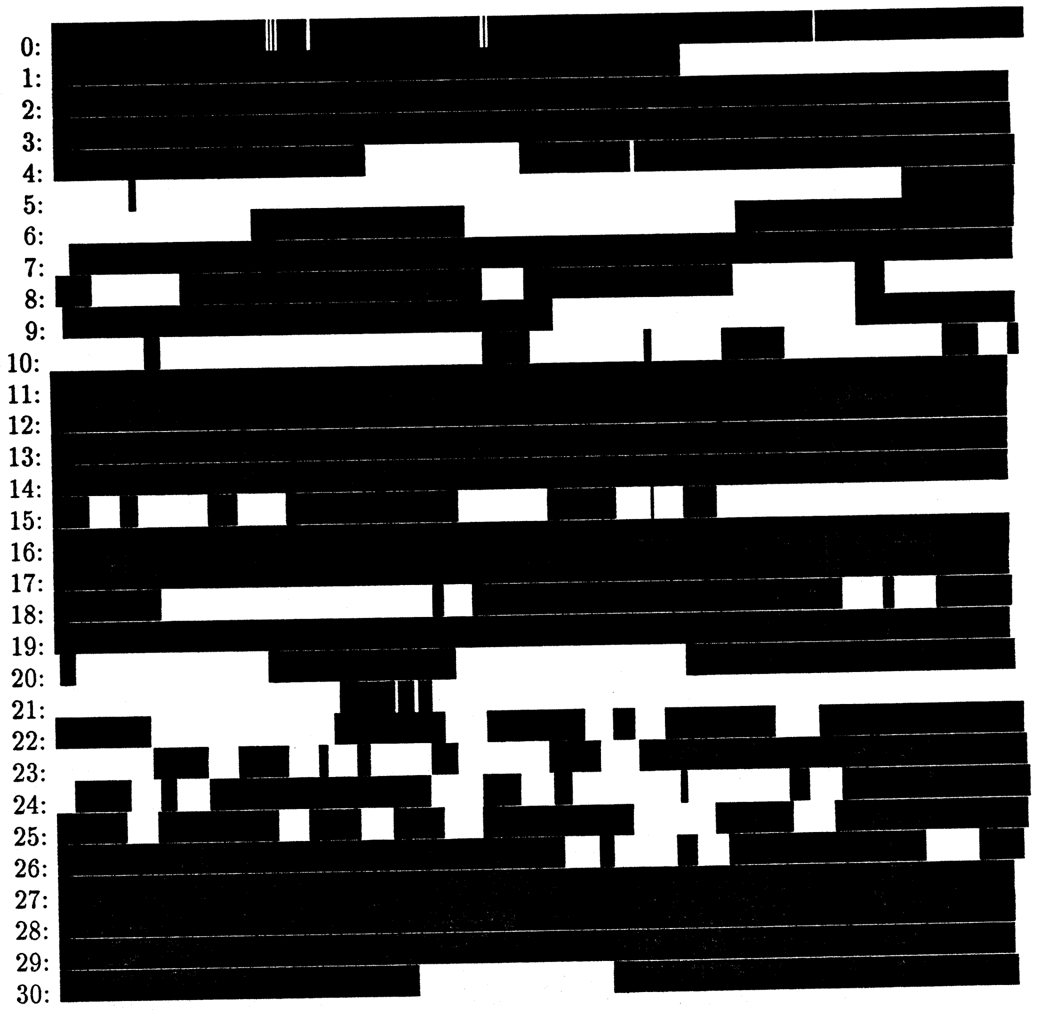

0:
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:

Figure 1

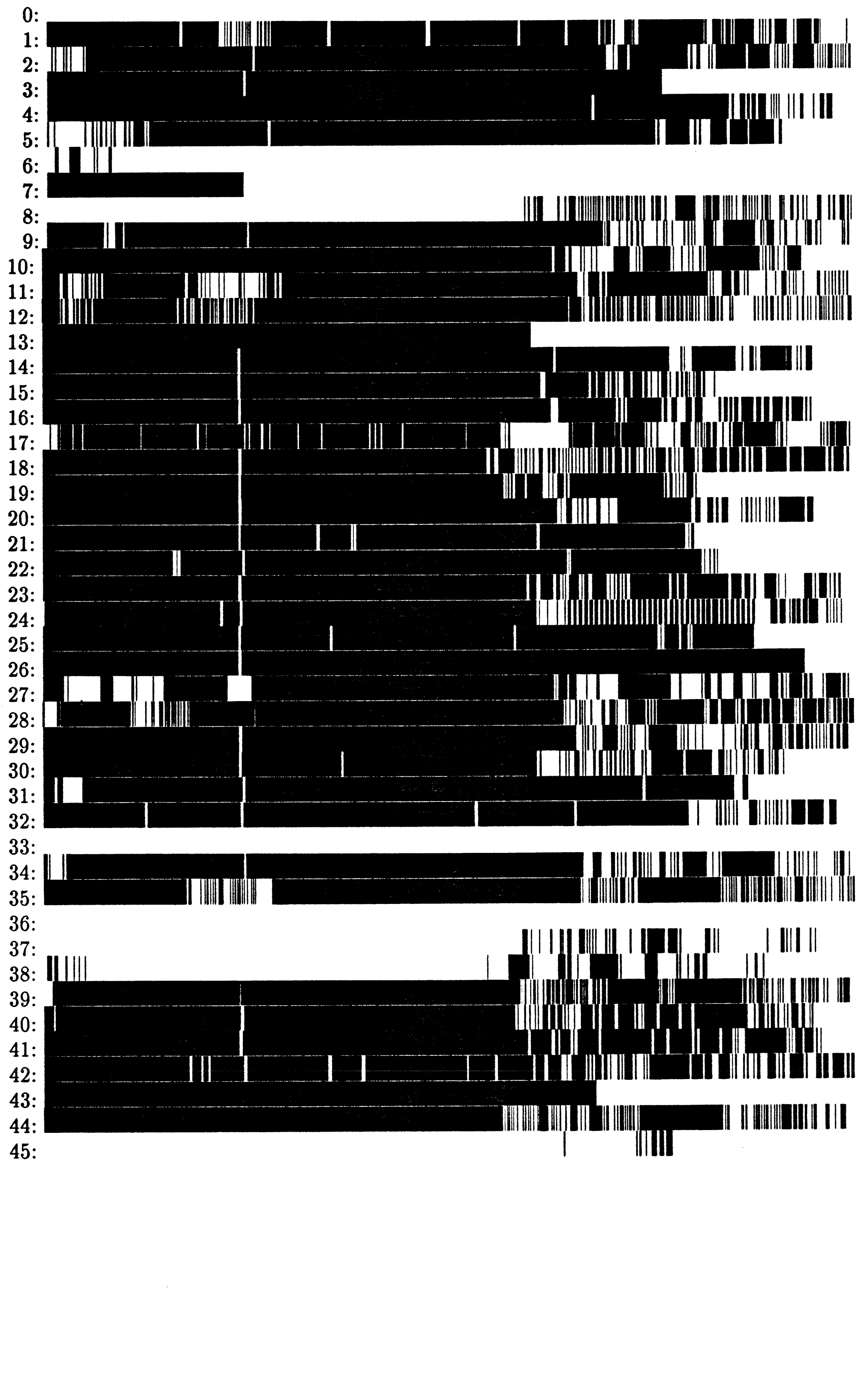

0:
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:

Figure 5

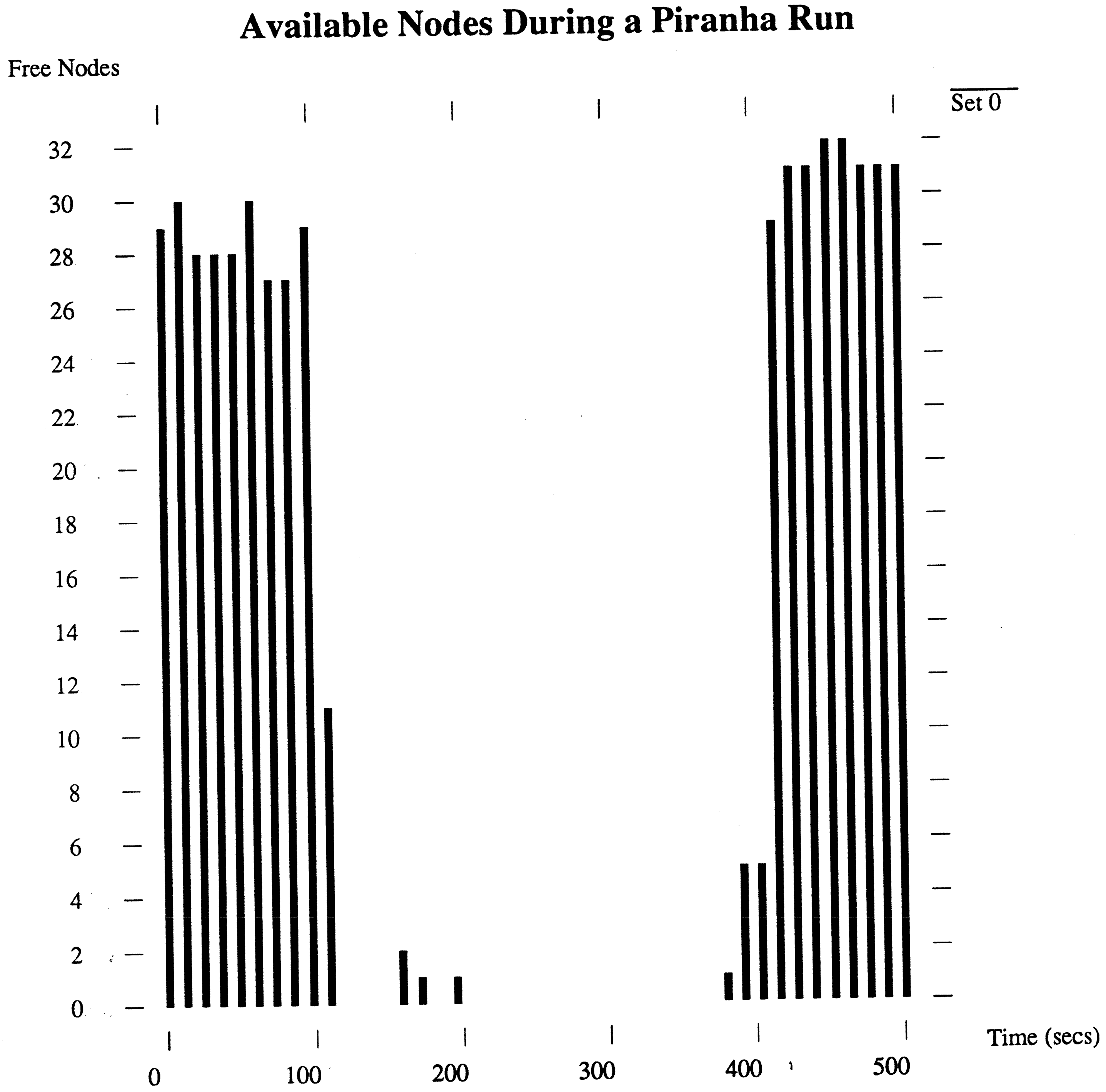

Available Nodes During a Piranha Run
Free Nodes
Set 0
32
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
0
100
200
300
400
500
Time (secs)

Figure 3

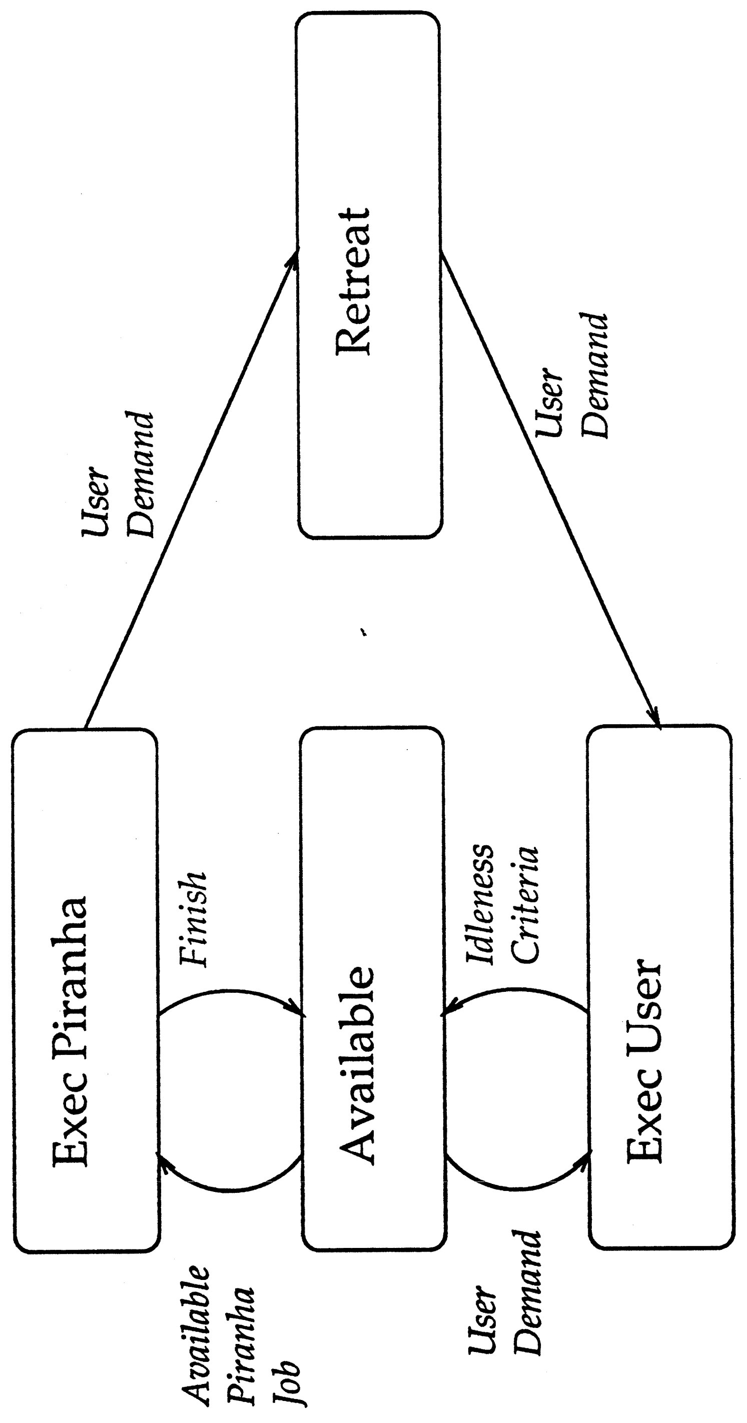
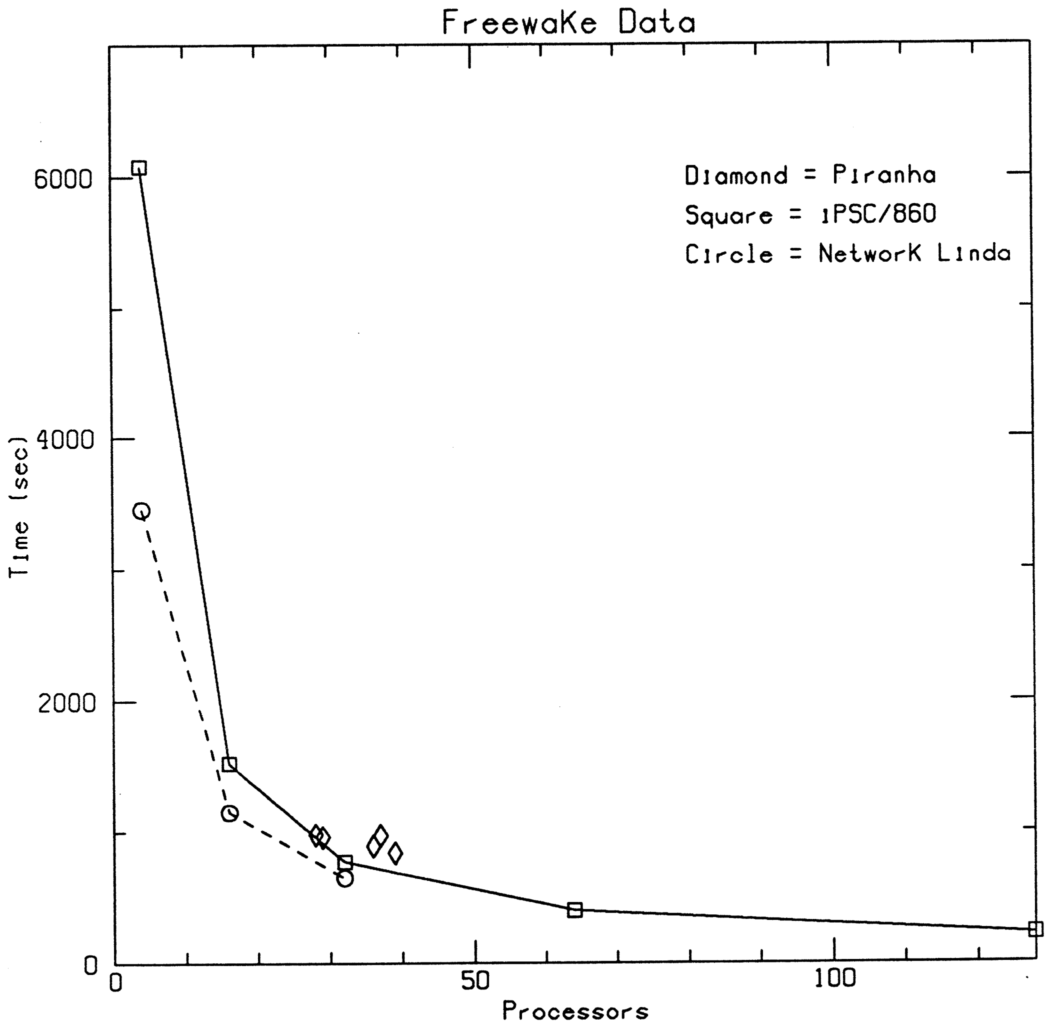

Retreat
User Demand
User Demand
Exec Piranha
Available
Exec User
Finish
Idleness Criteria
Available Piranha Job
User Demand

Figure 6



FreewaKe Data
Diamond = Piranha
Square = iPSC/860
Circle = NetworK Linda
6000
4000
2000
0
0
50
100
Time (sec)
Processors