Adaptive Parallelism with Piranha

Nicholas Carriero, David Gelernter,
David Kaminsky, Jeffery Westbrook

YALEU/DCS/RR-954
February 1993

Adaptive Parallelism with Piranha

Nicholas Carriero[*] David Gelernter [†] David Kaminsky [‡]
Jeffery Westbrook [§]

Abstract

"Adaptive parallelism" refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds. Networks of fast workstations are the most important setting for adaptive parallelism at present. Workstations at most sites are typically idle for significant fractions of the day, and those idle cycles may constitute in the aggregate a powerful computing resource. For this reason and others, we believe that adaptive parallelism is assured of playing an increasingly prominent role in parallel applications development over the next decade.

The "Piranha" system now up and running on a heterogeneous network at Yale is a general-purpose adaptive parallelism environment. It has been used to run a variety of production applications, including applications in graphics, theoretical physics, electrical engineering and computational fluid dynamics. In this paper we describe the Piranha model and several archetypal Piranha algorithms. Our main goals are to show that it is an effective and workable approach, to explain how the Piranha model constrains algorithm design, and to suggest directions for theoretical and practical research.

1 Introduction

"Adaptive parallelism" refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds.

Any parallel application might in principle gain if it is built as an adaptive program. In any computing environment (including a dedicated multiprocessor), such a program is capable of taking advantage of new resources as they become available, and of gracefully accommodating diminished resources without aborting. An adaptive parallel program might grow or shrink within a single multiprocessor, might encompass processors both within a multiprocessor and in a LAN, might jump from platform to platform and so on.

Networks of fast workstations are the most important setting for adaptive parallelism at present. Workstations at most sites are typically idle for significant fractions of the day, and those idle cycles may constitute in the aggregate a powerful computing resource.

[*] Research partially supported by Air Force Grant AFOSR-91-0098

[†] Research partially supported by Air Force Grant AFOSR-91-0098

[‡] Research partially supported by NASA Training Grant NGT-50719

[§] Department of Computer Science, Yale University, New Haven, CT 06520-2158. Research partially supported by National Science Foundation Grant CCR-9009753.

Although the Ethernet and comparable interconnects weren't designed to support parallel applications, they are capable in practice of supporting 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.

Adaptive parallelism is a poor fit to the assumptions underlying most parallel algorithm and program development models. Algorithms research targeted at adaptive parallelism is a necessity if algorithms are to keep pace with ongoing practical developments in parallel systems.

Several *ad hoc* systems have been designed to solve specific computational tasks adaptively—for example testing primality or computing travelling salesman tours [8, 10].

The "Piranha" system now up and running on a heterogeneous network at Yale [7] is a general-purpose adaptive parallelism environment. It has been used to run a variety of production applications, including applications in graphics, theoretical physics, electrical engineering and computational fluid dynamics.

In this paper we describe the Piranha model and several archetypal Piranha algorithms. Our main goals are to show that it is an effective and workable approach, to explain how the Piranha model constrains algorithm design, and to suggest directions for theoretical and practical research.

2 Process vs. Task Models of Adaptive Parallelism

An adaptive parallel computation runs on a set of n processors, each of which may be *available* or *withdrawn*. An available processor can be used in the computation. When a processor withdraws, any part of the parallel computation occurring on that processor must quickly cease.

A typical model for adaptive parallelism structures an application in terms of a fixed set of *processes* that are dynamically remapped among free processors: when a processor withdraws, its processes are migrated somewhere else. Such an approach was investigated as long ago as the "MuNet" project and the early stages of Actors research (e.g. [1],[3]); a variant of this approach formed the basis of the "Amber" adaptive parallelism system [6].

This approach, while theoretically appealing, has drawbacks in practice. General process migration is expensive, and it poses obvious difficulties on heterogenous networks of processors. Holding processes constant while the node pool contracts may lead to excessive process management overhead in the shrinking pool of active nodes.

Piranha, in contrast, is an adaptive version of master-worker parallelism [5]. Algorithm designers specify in effect a single general purpose "worker function," by convention named "piranha()." They do not create processes and their applications do not rely on any particular set of named 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 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* (and not merely processors) varies dynamically.

Workers perform *tasks.* A task consists of a fixed set of input data from which a worker function produces a set of output data and a collection of new tasks, possibly consulting global program state data and communicating with other workers in the process.

Data and task descriptors are stored in distributed data structures accessible to all worker processes; the more workers, the faster the program completes, but there is no reliance on any particular number of active workers. The Piranha system uses the shared associatve object memories ("tuple spaces") provided by Linda [5] to manage coordination and the distributed data structures in which the problem state is stored.

This approach has a number of advantages. Processes need never be moved around. Heterogeneity is easily accomodated—including the kind of "strong heterogeneity" in which a task begun on one type of machine is continued on another. The application collapses cleanly with little overhead into a degenerate "non-parallel" case, in which a single active process runs on a single participating node. Handling the degenerate case is important. Research should aim (we believe) at making adaptive parallelism a default programming mode for *all* compute-intensive applications—which implies that the system must impose little or no overhead in the single-processor case.

3 The Structure of Piranha Programs

3.1 Processes

Piranha programs consist of two types of processes: one `feeder` and multiple `piranha`. The `feeder` runs on the *home node*. The home node is available at all times. The feeder has many functions, including 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 a task and corresponding input data, performs the task, and creates zero or more new tasks and output data. The code to consume a task (and the definition of a "task" itself) is application-dependent.

If the piranha function creates a new task, it may choose not to write the task out to distributed memory, but instead reserve the task for itself and begin processing it. This optimization can reduce global memory access time when the output data of one task is the input data to another. Thus a piranha process may have a local collection of task descriptors and task input data.

Let $\mathcal{T}(t, n)$ denote the set of tasks held at processor n at time t. This set includes the task the piranha function is currently executing. A task disappears from $\mathcal{T}(t, n)$ only once it is completely processed.

Let $\mathcal{O}(t, n)$ be the output data held at processor t at time n. This is data that has been produced by a task but not yet written to distributed memory.

Figure at back.

Figure 1: Time to access non-local memory
Time to perform 500 non-local put/get pairs on a 10 MBit/second Ethernet.

3.2 Tuple Space

Piranha processes communicate through a global, associative, object memory called tuple space.[1] Tuple space (and the "tuples" stored in it) persist irrespective of which nodes are currently active. Processes can communicate even if they are never active concurrently.

Since tuple space is global, tasks and results stored in tuple space are available to all processes. The cost of accessing tuple space is linear in the number of bytes transferred. Figure 1 shows transfer time as a function of tuple size.

Processes also have local memory that can be accessed cost-free. We impose no limits on local memory.

3.3 Retreat

When an available node withdraws the Piranha process on that node must cease operation. This is the job of the `retreat` function.

Upon retreat at time t, the Piranha process n takes the following actions.

1. Immediately terminate the local piranha computation.
2. Transfer into distributed memory the set $\mathcal{T}(t, n)$.
3. Transfer into distributed memory the output data from any completed tasks.
4. Relinquish all local memory. Thus any intermediate results of the task computation are lost.

A retreat returns a task being processed to distributed memory as if it had never been started.[2] Thus if a processor is becomes available for just slightly less than the time required to complete a task, all the work is wasted and we are not making good use of available cycles. One can define a task to be a single instruction of the workstation microprocessor. In that case the input and output data of a task are the executable images. This guarantees finest granularity of the parallel computation and ensures that progress is made on all but the very shortest segments of availability, but leads to prohibitively high cost on retreat, since the entire image must be written. On the other hand, one can choose a task to be the entire computation, which leads to no parallelism but minimizes retreat time.

In general, larger granularity reduces distributed communication time at the risk of less efficient use of available time and of more costly retreats. On the other hand, algorithms that maintain the entire program state in tuple space use more communication time, but have very low cost retreat functions. Good Piranha algorithms strike a balance between these concerns.

[1] A description of tuple space access functions is found in [5].

[2] A clever programmer can save intermediate state at the cost of a more complex retreat function.

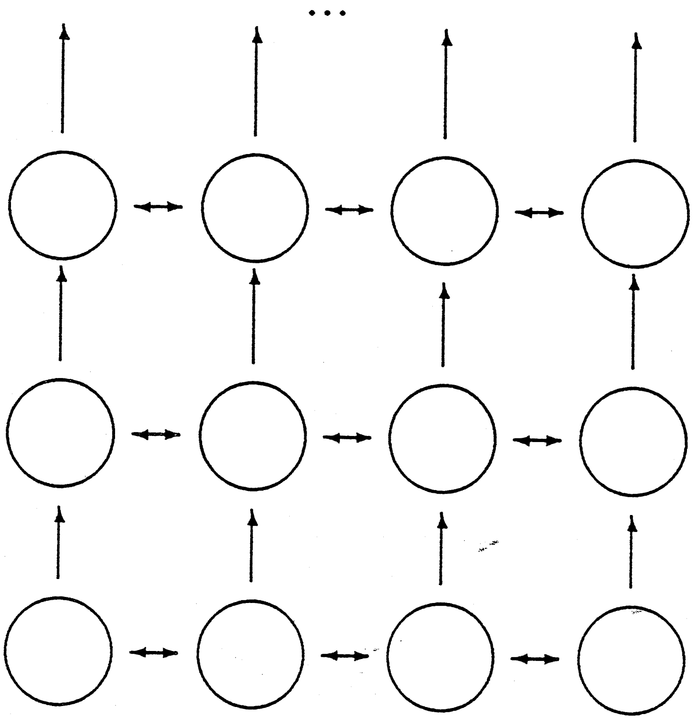


Figure 2: Heat Diffusion task graph

4 Task Graphs

Our starting point in the design of efficient Piranha programs is an algorithm's acyclic "task graph": tasks are nodes and directed arcs are dependencies. A task is "enabled" iff all of the tasks on which it depends are complete.[3]

In the simplest case, a problem with no dependencies (e.g. a Monte Carlo simulation), the graph contains no arcs. A more complicated example, the task graph for a heat diffusion algorithm, is shown in figure 2. A two dimensional plane is divided into a number of gridpoints. The algorithm iteratively computes the temperature of each gridpoint based on its previous value and the values of its four neighbors. We define a task to be the updating of one column of the grid at one iteration. Each task depends on the values in its neighboring columns and on its own values of the previous iteration.

4.1 Ensuring Deadlock Freedom

Consider the case where k Piranha are active. $k-1$ Piranha and the feeder are blocked waiting for a result to be produced by the last Piranha P. If P retreats, the computation is deadlocked. To avoid this situation, correct Piranha algorithms ensure that *some* process

[3] In some cases a Piranha can begin consuming a task before it becomes enabled, but the task can't be completed until all dependencies are satisfied.

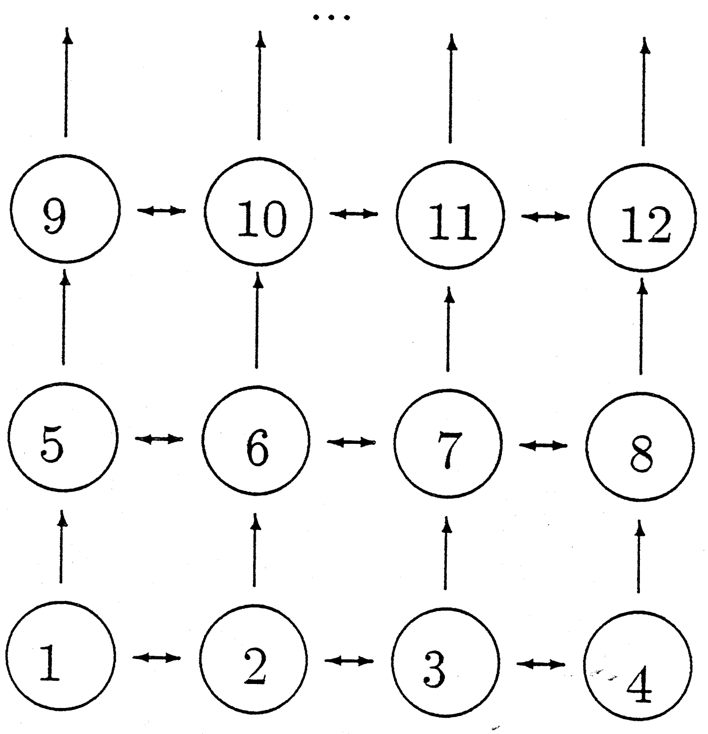


Figure 3: Labelled Heat Diffusion Task Graph

will never block waiting for data. Since any Piranha can retreat at any time, the only reasonable choice to be the non-blocking process is the feeder.

Consider figure 2. To prevent deadlock, the feeder traverses the graph executing — or being willing in principle to execute — tasks. It never starts a task unless all of that task's dependencies are satisfied (i.e. the task is enabled). If the task has already been done by a Piranha, it skips to the next. A topological traversal is analogous to a sequential execution by the feeder. Parallel efficiency is not sacrificed, since the feeder need not compute the values that have already been computed. Rather than duplicating work done by the Piranha, the feeder simply ensures that the result of each task is computed before it moves on. The feeder can "walk behind" the piranha collecting results. Should some piranha retreat while computing the next result in the task graph, the feeder gathers this task and computes the result itself.

5 Computational Efficiency

In general, the goal of a parallel computation is to reduce the actual clock time to job completion. (In the context of a set of tasks related by an acyclic priority graph, this is called the *makespan.*) In an adaptive parallel environment, one cannot make guarantees about actual clock time to completion. Over a large span of clock time very few computation cycles may be available. Hence we define time complexity in terms of *total uptime.*

The *up function* of a processor p, $a(p)$, maps any non-negative real time t to $\{0,1\}$. The function has value 1 at time t if processor p is available at time t, otherwise 0. Given a time t_0, the uptime of processor p to t_0 is the integral of $a(p)$ from 0 to t_0. The total uptime to t_0 is the sum over processors of the uptime per processor.

A *run* is an execution of a Piranha algorithm on a given set of input data and n up functions for the n processors. The *wall time* of a run is the real time t_0 at which the computation terminates, *i.e.*, the time the clock on the wall shows. The *work time* of the run is the total uptime to t_0. The *efficiency* of the run is the ratio of the sequential running time to work time.

Our goal is to keep the efficiency high. Then the Piranha algorithm is making effective use of the available cycles. Our definition of Piranha work time is the natural analog to the PRAM work function, (# processors) × (#time).

5.1 Efficiency in the worst-case

Consider the simplest parallel computation, a collection of m independent tasks each of which takes time 1 to complete. (Here "1" is a unit time that includes the time to read a task and process it.)

A piranha strike is successful if an up piranha gains access to a task. The fairness of the underlying communication system is crucial to determining the running time of a program. A process *starves* if it can strike an infinite number of times without gaining access to a task.

Theorem 5.1 *Suppose the communication system allows for starvation. Then the worst-case running time is infinite.*

Proof: Consider the case of 3 processors (2 piranha p_1 and p_2 and one feeder f) attempting to complete one task. (Without loss of generality; it is easy to arrange that even if there are n tasks initially, eventually only one task remains.) Let $\delta = 1 - \epsilon$ and $t_i = i\delta$ for $i = 0, 1, 2, \ldots$. We give availability functions $a(p_1)$ and $a(p_2)$ such that the task is never completed. Let $a(p_1)$ be 1 during the intervals $[t_i, t_{i+1}], i = 0, 2, 4, \ldots$ and zero otherwise. Let $a(p_2)$ be 1 during the intervals $[t_i, t_{i+1}], i = 1, 3, 5, \ldots$ and zero otherwise. The feeder is always available.

Initally, p_1 and f are active and request the task. Assume p_1 wins. It reads the task, starts to compute, but goes down before completing and writes the task back out. Simultaneously, p_2 becomes active and requests the task. p_2 starts reading the task, but fails before finishing the computation. By that time, p_1 is active and gets the task, only to fail before completing. This continues indefinitely. The feeder starves. Since there is always an available processor, the total available time is infinite. ∎

The above theorem assumes that an adversary is able to choose the scheduling of tasks to processors. Next we consider the case when we add a scheduler to the Piranha system that decides if and when an available processor will be given a task.

Theorem 5.2 *With any randomized scheduler, the worst-case running time is $\Omega(nT)$, where n is the number of processors and T is the time to process all tasks on a single processor. Hence the worst-case efficiency is $O(\frac{1}{n})$.*

Proof: The adversary simply makes each processor run in bursts of time $1 - \epsilon$ where ϵ is an arbitrarily small constant. No task can be completed on any processor other than the feeder. Per task completed the available time is $\Omega(n)$, for $\Omega(nT)$ total available time. ▮

Thus in the worst-case one might as well simply perform all tasks on the feeder. This certainly guarantees efficiency $\Omega(1/n)$. Empirical evidence suggests, however, that in real-life applications the availability functions are much more constrained than we have allowed, and worst-case scenarios do not often arise. In the next subsection we look at a simple probabilistic model for the up functions.

5.2 Markov chain up functions

We assume that the up function is determined by a simple Markov chain on two states, UP and DOWN. The probability of leaving either state for the other is p and the probability of staying in the same state is $q = 1-p$. It is easily verified that in steady state the probability of UP equal the probability of DOWN equals 1/2. We discretize time, and assume that a transition occurs each time step.

Consider the problem of processing a collection of independent tasks, each requiring c units of uptime to complete. What is the expected time?

We extend the Markov chain by splitting the UP state into c substates $u_1, u_2, \ldots, u_c$. The probability of leaving u_i for DOWN is p. The probability of going from u_i to u_{i+1} for $1 \leq i < c$ is q and the probability of going from u_c to u_1 is q.

Let ρ_i be the steady state probability of state i, where state 0 is DOWN and state i for $i > 0$ is u_i. One may verify that the steady state probabilities are:

$$\begin{aligned} \rho_0 &= \frac{1}{2} \\ \rho_i &= \frac{pq^{i-1}}{2(1-q^c)}, \quad 1 \leq i \leq c \end{aligned}$$

Every time the chain enters state u_c a task is completed. The probability of being in state ρ_c is $\frac{pq^{c-1}}{2(1-q^c)}$. Given n tasks, consider the value t_n such that $n = t_n\rho_c$. Then t_n is the time at which the expected number of tasks completed is n.

Suppose we have only a single piranha processor with uptime determined by the above Markov chain. We wish to complete a job with sequential running time s. To finish the work on the piranha, we divide the job up into n tasks. The time to process each task will be $c = \alpha + d$, where $d = s/n$ and α is a positive value that accounts for extra cost associated with reading tasks, etc. What is the choice of n that minimizes t_n?

We have

$$t_n = n/\rho_c = \frac{s}{d} \cdot \frac{2(1-q^{\alpha+d})}{pq^{\alpha+d-1}}.$$

Differentiating $t'_n = \frac{dt_n}{dc}$, setting $t'_n = 0$, and solving for c gives the following formula for the optimum c:

$$\frac{W(-q^\alpha/e)+1}{\ln(1/q)}$$

Here W is the function satisfying $W(x)e^{W(x)} = x$. The value of the numerator is approximated by $+(1-2q^\alpha/e)^{1/2}$.

For example, let $q = 0.9$, and $a = 10$. Then the optimum value of c is approximately 8.1. Then the time to complete the job is approximately $13s$. If we run the tasks in parallel on p processors, we expect to get linear speedup.

6 Examples

The task ordering imposed by a problem and represented in its task graph constrains its potential solutions. Below we examine three important classes of scientific algorithms. We consider problems ranging from totally unconstrained (Monte Carlo) to highly constrained (domain decomposition).

Algorithm designers must consider the following questions: What is the definition of a task? How does the feeder traverse the task graph? How do the Piranha traverse it? How does the algorithm cope with the loss of a node (i.e. what is the retreat function)? How are nodes added to computation? What information is stored locally and what is stored globally (in tuple space)?

In our discussion below, we will attempt to answer the questions posed above. Since we are primarily interested in the coordination framework of the problem, details about the computation algorithms will be omitted.

6.1 Monte Carlo Simulations

Monte Carlo simulations are ideal for the Piranha environment. Tasks are easily defined (one trial). Initialization data for each trial can be stored in tuples. Task order is not relevant since the task graph contains no arcs. The feeder and the Piranha repeatedly grab and compute any unfinished trial. Retreat simply requires that the task be returned to tuple space for some other process to complete. New nodes join the computation by consuming tasks. Previously active processes need not be aware of the presence of a new worker. Since state is not relevant between trials, efficiency dictates that as much state be stored locally as possible (without causing excessive paging). State is flushed before each trial.

One advantage of Piranha is that easy applications are easily programmed. Pseudocode for a Monte Carlo program is given in figure 4.

With the simulation template given below, it is possible to construct numerous Piranha programs simply by defining the `do_simulation` function and the result structure. Programmers can also include additional data in the "task" and "result" tuples as needed for the simulation.

	Workers	Run Time	Aggr Time
Piranha	4.26	3.59	14.5
Net Linda	4.5	3.17	14.3
Sequential	1	14.2	14.2

Workers is the average number of workers used. The time spent by the feeder (Piranha) and master (Network Linda) were omitted since these processes do not consume tasks in this program. Run Time is the wall clock time from start to finish. Aggregrate time (Aggr Time) is the sum of all of worker time used. All times given are in hours. Six Piranha, eight network Linda and two sequential runs were averaged.

Table 1: Rayshade Performance

One "Monte Carlo" type (loosely speaking) Piranha program is Rayshade.[4] Rayshade is a ray tracing program written by Craig Kolb. It renders an image in horizontal "scanlines." Each scanline is a task and tasks are completely independent. Table 1 shows a comparison of the Piranha, Network Linda[5], and sequential versions of Rayshade. The Piranha version is 98% efficient compared to the sequential version. Only 68 total seconds (on average) were wasted to retreat.

6.2 LU Decomposition

LU decomposition is interesting in a Piranha environment because it is a "middle-weight" problem: its ordering constraints are harder than (e.g.) Monte Carlo simulations and easier than domain decompostion problems. Unlike Monte Carlo problems, task dependencies limit the range of possible solutions. Furthermore, data must be transferred from task to task. But unlike domain decomposition, the graph is not so limiting as to dictate a single solution.

The LU decomposition (LUD) of an $m \times n$ matrix A, $m \leq n$, is a pair of matrices L and U such that $A = LU$, L is $m \times n$ unit lower triangular, and U is $m \times n$ upper triangular.[2]

In our Gaussian-like approach to LUD, we select the leftmost column i of the matrix, and zero the elements in i above the diagonal. The same linear transformation applied to i are then applied to the columns to the right of it $(i+1, i+2 \ldots)$. i is incremented and the process continues.

The dependencies for a column c at iteration j of the algorithm are on column j (the column being scaled) and on the value of c at the previous iteration. The task dependency graph for LUD is given in figure 5.

We have considered a number of Piranha algorithms.[4] They all define a task to be updating one column by one pivot. While the feeder cannot consume tasks in any order as it could above, it still has quite a few options. In figure 8 we present two alternatives.

[4] Rayshade is not a Monte Carlo simulation. Since there are no intertask dependencies; we call it a Monte Carlo type Piranha program.

[5] Linda is a registered trademark of Scientific Computing Associates, New Haven.

```
#define DONE -999

feeder ()
{
  int i;
  struct Result result;

  /* put out the tasks */
  for (i=0; i < TASKS, i++)
    out (‘‘task’’, i);
  /* help compute results */
  piranha ();
  /* collect results */
  for (i=0; i < TASKS; i++)
    in (‘‘result’’, i, ?result_data);
}

/* declared globally to be available for retreat */
int i;
piranha ()
{
  struct Result result;

  while (1) {
    in (‘‘task’’, ?i);
    if (i == DONE) {
      /* all tasks are done */
      out (‘‘task’’, i);
      break;
    }
    else {
      /* do the task */
      do_simulation (i, &result);
      out (‘‘result’’, i, result);
      in (‘‘tasks done’’, ?i);
      out (‘‘tasks done’’, i+1);
      if ((i+1) == TASKS) out (‘‘task’’, DONE);
    }
  }
}

retreat ()
{
  /* replace the current task */
  out (‘‘task’’, i);
}
```

Figure 4: Monte Carlo Pseudocode

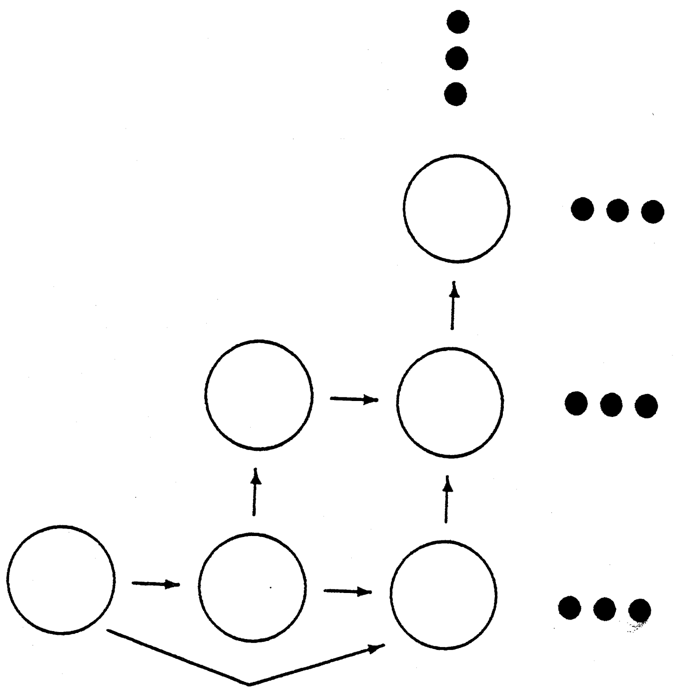


Figure 5: LUD Task graph

In the "left-to-right" algorithm, we exploit the fact that a column does not depend on columns to its right. We store the columns in a queue in tuple space. Each process p (the feeder mimics the Piranha) claims the leftmost unclaimed column c_i. p then reads and applies pivot columns c_1 to c_{i-1}.[6] p creates pivot vector c_i, publishes it, then claims another column. One a pivot is read, it can be cached for later use.

If process p retreats while working on column c_i after having applied j pivots, p returns c_i to tuple space with a count of the number of pivots applied. Some process notices this retreated tasks, claims it, and performs the necessary operations on it. If all active processes are blocked waiting for pivot c_i (i.e. all processes are working on columns c_j, $j > i$), some process abandons its current tasks and completed c_i instead. Newly active processes simply begin consuming columns.

We ran a simulation of the left-to-right algorithm. Figure 6 show the aggregate and wall clock time used as a function of the number of processors. In each case (1000, 2000, and 3000 cols) wall clock time is reduced by adding processors only up to a point. For example, after about 12 processors, adding more does not greatly reduce the wall clock time.

Figure 7 shows the efficiency and speedup as we add more processors. The 3000 column case show that up to a point, processors can be added with near constant resulting in near linear speedup. After that point, efficiency drops as processors spend more time waiting for

[6] When a processor claims one column, it is reserving multiple tasks (i.e. it will apply each pivots to the column, barring retreat). This optimization saves the cost of repeatedly removing the column from tuple space and replacing it there.

See Back

Figure 6: Processors vs. Time

See Back

Figure 7: Processors vs. Efficiency

pivots from other processors. This efficiency breakpoint is higher for larger problems.

Unlike the Monte Carlo model and the "left-to-right" algorithm, The "bottom-up" algorithm completely partitions the tasks among the workers. The columns of the matrix are assigned arbitrarily to the active Piranha. When a new pivot is published, each Piranha p applies it to each of its columns. Once pivot $i-1$ had been applied to column i, the Piranha passes column i to the feeder. The feeder creates the pivot from column i and publishes it. The feeder also stores the result. The algorithm is called "bottom-up" from its traversal of the task graph given above — each Piranha applies one pivot to all of its columns before applying the next pivot to any of them.

Dynamic rebalancing is used to maintain load balance among the Piranha. We store the count of the unfinished and active Piranha in tuple space. Each Piranha can compute its "fair share" of the work. If a Piranha has more than its fair share (above a certain tolerance), its dumps columns into the "overflow pool" (in tuple space) until it has the correct number. Underloaded Piranha remove columns from the pool to boost their load.

The overflow pool is also used when Piranha enter and leave the computation. New Piranha increment the count of the number of active workers and claims columns from the pool until they have sufficient work.[7] A retreating Piranha decrements the number of active workers and dump its columns to the pool.

If the feeder is waiting for column i, but i is in the overflow pool (due to retreat or rebalancing), the feeder claims i. If i has had fewer than $i-1$ pivots applied, the feeder itself applies the requisite pivots. Unlike the Piranha, the feeder traverses the task graph left-to-right.

In preliminary measurements of the bottom-up version of LUD, we have shown efficiency of about 70% on a 2000×2000 matrix when compared to sequential times extrapolated from smaller matricies. The sequential version, when run on a 2000×2000 matrix, performed poorly due to excessive paging. This problem occurs on matrices smaller than 1000x1000.

6.3 Domain Decomposition

We consider a heat transfer program as representative of a class of algorithms with nearest neighbor dependencies.[8] A task is to update the value of one column of gridpoints at an iteration. As described above, each gridpoint depends on its four nearest neighbors. The task graph is presented in figure 2.

Since each task depends on the adjacent columns (or column), unoptimized solutions

[7] They do not wait for a full complement of columns before they begin working.

[8] The implementation of the domain decomposition algorithm is being completed.

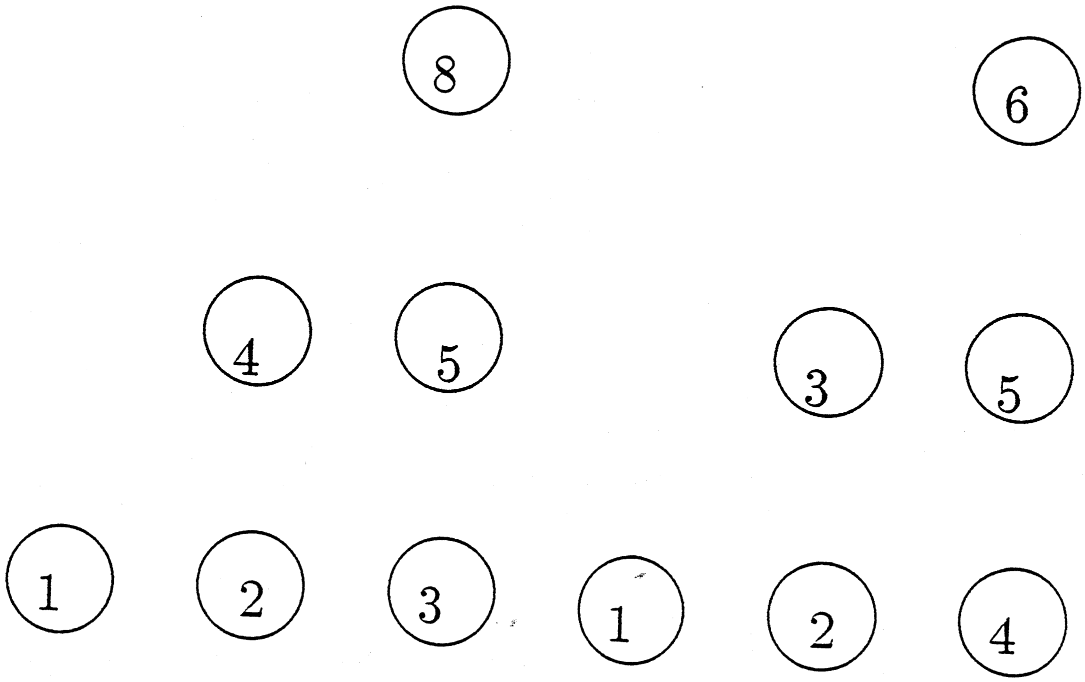


Figure 8: LUD Task graph
The LUD task graph is shown labelled "left-to-right" (left) and "bottom-up" (right).

perform an unacceptably large amount of communication. If we consider iterating M times over an $N \times N$ matrix, where each column and its (one or) two neighbors are accessed at each iteration, we globally access $(3 \times N^2 - 2 \times N) \times M$ gridpoints.

Our Piranha algorithm assigns a contiguous set of columns to each active processor. This reduces the communication to $2 \times K_i$ per iteration, where $K - i$ is the number of workers at iteration i. Summing over all iterations i yields the entire communication cost. In addition, initially distributing the grid incurs N^2 sends.

Both the feeder and the Piranha (they behave identically) traverse the task graph bottom-up. Each active process p claims a column set $x_{a,1}$ to $x_{b,1}$. Then, at each iteration i, p reads $x_{a-1,i-1}$, $x_{b+1,i-1}$, updates its columns, and publishes $x_{a,i}$, $x_{b,i}$.

As in LUD, we use a rebalancing algorithm to cope with retreats and new workers. When a node p retreats, it dumps its columns into tuple space. p's neighbors claim the columns at the next iteration. New nodes are added by splitting the columns possessed by some worker. Imbalances are corrected between iterations by passing columns among workers possessing adjacent column sets. By using local exchanges we believe that this algorithm will maintain a load balance, tolerate retreats, and include new Piranha without fragmenting the grid and thus perform efficiently in the Piranha environment. Measurement of actual performance will be taken in the near future.

7 Future Work

We have presented Piranha algorithms for a three important types of problems. We have yet to investigate problems with irregular task graphs or graphs that cannot be determined exactly until run-time. We are currently studying rebalancing algorithm for domain decomposition problems like the one presented above. Finally, we are studying the sensitivity of our algorithm to retreat frequency.

References

[1] Agha, G. *Actors: A Model of Concurent Computation in Distributed Systems.* MIT Press (Cambridge: 1986).

[2] Aho, A., Hopcroft, J., and Ullman, J, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974.

[3] Baker, H. and Hewitt, C., "The incremental garbage collection of processes." In *Conference Record of the Conference on AI and Programming Languages, pp 55-59, ACM, Rochester, New York, 8/77.*

[4] Bjornson, R., Kaminsky, D., and Weston, S., "LU Decomposition in an Adaptive Parallel Environment", in progress, 1993.

[5] Carriero, N and Gelernter, D. How to Write Parallel Programs: A First Course. MIT Press (Cambridge: 1990).

[6] Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M., Littlefield, R.J. "The Amber System: Parallel Programming on a Network of Multiprocessors," Proceedings of the Twelth ACM Symp. on Operating Systems Principles, pp 147-158, 12/89.

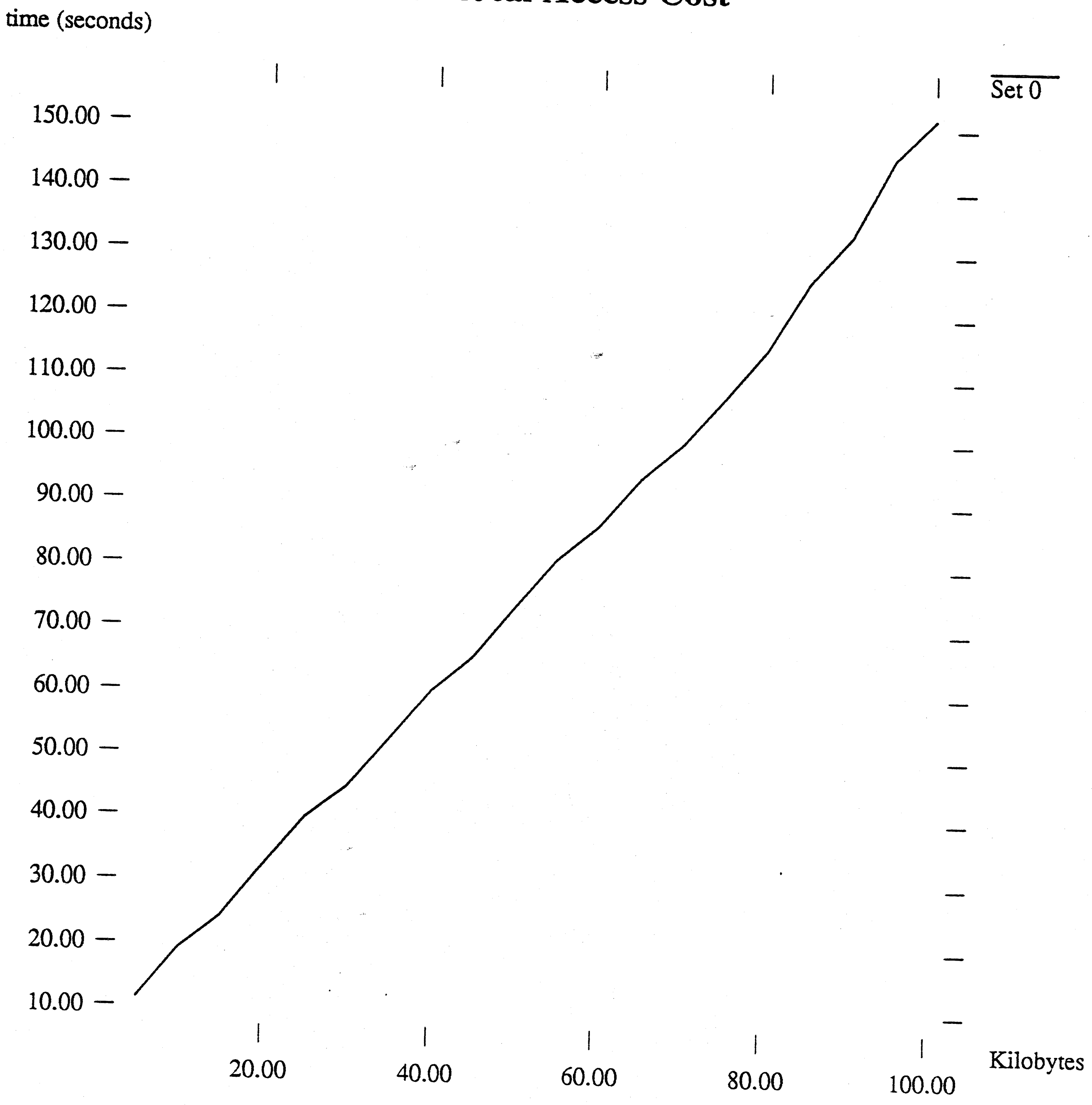
[7] Gelernter, D. and Kaminsky, D., "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha," Sixth ACM International Conference on Supercomputing, Washington D.C., July 19-23, 1992.

[8] Lenstra, A.K., and Manasse, M., "Factoring by Electronic Mail," Proc. Eurocrypt '89, Lecture Notes in Computer Science 434, Springer-Verlag, 1990.

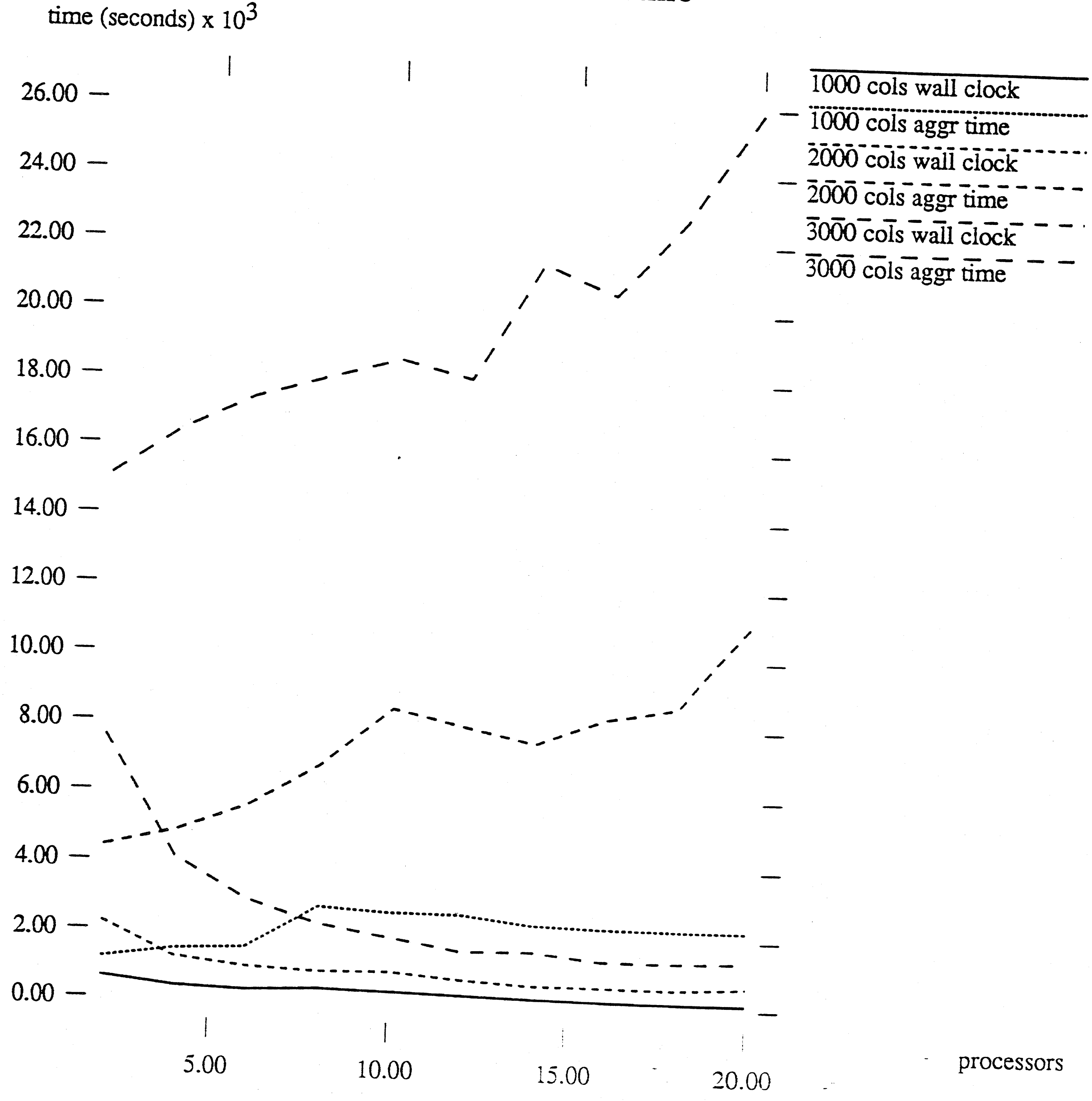
[9] Mutka, M.W. and Livny, M. "Profiling Workstations' Available Capacity for Remote Execution", Performance '87, pp 529-544, Elsevier Science Publishers B.V. (North Holland), 1988.

[10] N. Reingold, private communication, 1993.

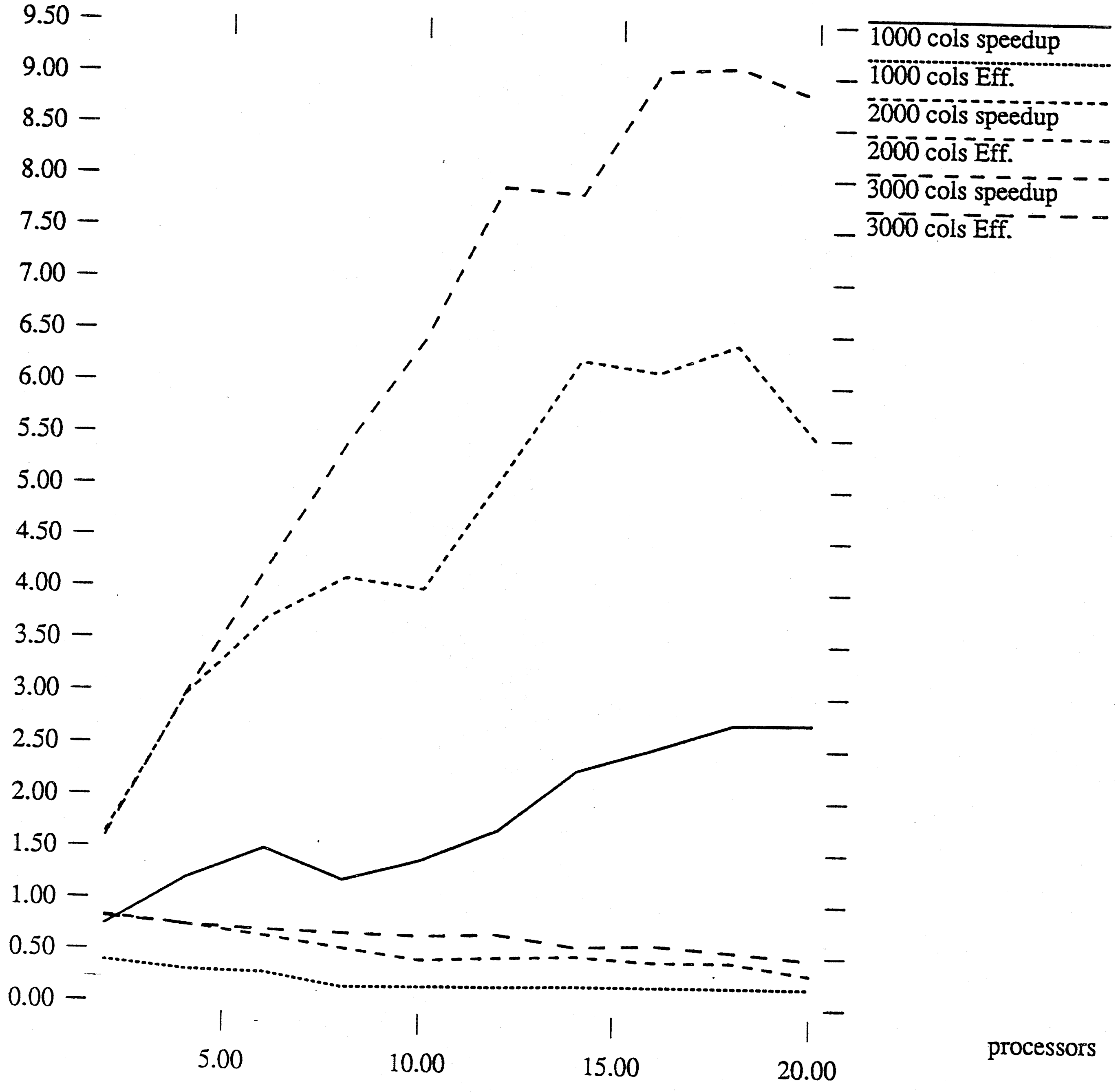
Non-local Access Cost



Processors vs. Time



Processors vs. Efficiency


processors
1000 cols speedup
1000 cols Eff.
2000 cols speedup
2000 cols Eff.
3000 cols speedup
3000 cols Eff.