Adaptive Parallelism with Piranha

David Louis Kaminsky

YALEU/DCS/RR-1021
May 1994

Abstract

Adaptive Parallelism with Piranha

David Louis Kaminsky
Yale University
May 1994

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

In the Piranha model of adaptive parallelism, idle network nodes are used to run explicitly parallel programs. Idle nodes host transient processes called *piranha.* A piranha process is started automatically when a node becomes idle and continues while it remains so. When a node becomes busy, its piranha process executes a short *retreat* function and exits. One *feeder* process is created to manage the ongoing computation. The feeder does not retreat.

Piranha processes transform a distributed data structure describing work into one describing results. Distributed data structures are stored in a Linda tuple space where they are accessible to all participating processes.

A Piranha programmer writes a program by specifying three coordination functions—`feeder`, `piranha` and `retreat`—and an arbitrary number of computation functions. The coordination functions define the interaction among the processes. The computation functions perform the task at hand.

We present the Piranha model for adaptive parallelism, a methodology for developing Piranha applications and a software infrastructure that supports the model. We conclude by presenting our view of Piranha's future.

This work was partially supported by NASA Training Grant NGT-50719.

Adaptive Parallelism with Piranha

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

David Louis Kaminsky
May 1994

Acknowledgments

I must thank a number of people who helped me during my studies.

Throughout my work, guidance from David Gelernter, my advisor, and Nick Carriero was indispensable. Nick also modified the SCA Network Linda compiler to support Open Tuple Spaces, and spent countless hours helping prepare this dissertation.

Rob Bjornson provided technical information that sped my work and supplied useful hints on being a graduate student. Jeff Westbrook helped define the more theoretical aspects of Piranha. Steve Weston developed much of the Piranha graphical interface, and answered numerous Linda questions. Scientific Computing Associates provided access to their Network Linda source code.

My officemates, Shakil Ahmed and Susanne Hupfer, made graduate school tolerable—maybe even fun.

Finally, special thanks to my parents and to my sister for providing moral support in good times and in bad. Success is easier attained when failure is not feared.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problems and Opportunities

The quest for knowledge drives the search for solutions to ever more complex questions. An answer to an existing question often leads to additional questions, some more complex than the original. As we push for solutions to ever more complex questions, we often need more powerful computers.

Hardware manufacturers have provided increasingly powerful machines on which to solve our problems. Over the past 10 years, peak speeds have increased by an average 50% per year.[1] However, it is expensive to push the bounds of hardware technology; today's premier supercomputers (e.g. Cray's C90) cost over $20 million.

Rather than upgrading current hardware, people are turning to their pre-existing networks of workstations as sources of computing power. *Network parallel systems* such as Network Linda [CG90],[2] PVM [BDG+91], Express [FKB91] and p4 [BL92] allow a network of workstations to be used as a parallel processing engine.

However, network based parallel computations must co-exist with other uses of the workstations. Often a workstation belongs to an owner who will not tolerate outside applications degrading the performance of his node. Network parallel applications are thus limited to idle nodes.

Figure 1.1 illustrates node use in a typical day at the Yale Department of Computer Science. Each line in the figure represent the use of one node. Shaded areas denote idle time and white areas denote busy time. Thirty nodes are shown running over a 24 hour period that begins at 6:00 pm. While there is a substantial amount of idle time (represented by the aggregate shaded area), the time is fragmented into numerous, variable-length bursts.

Network parallel computations are ill-suited to the irregular fragments of idle time

[1]Cray 1S (1983) to Cray C90 with 16 processors using the Linpack benchmark on a 1000×1000 problem [Deu93].

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

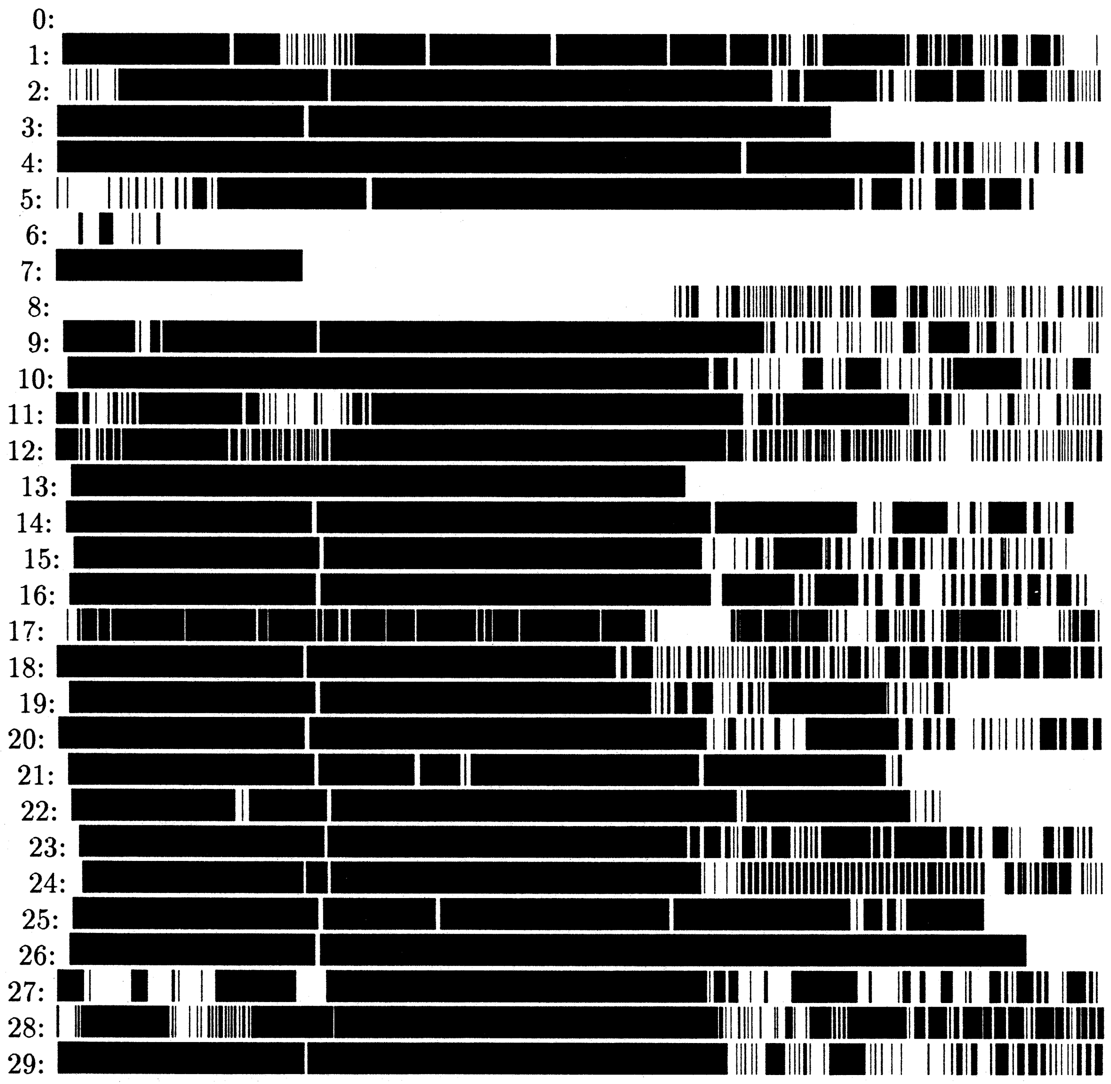

Figure 1.1: Node-by-node idle time

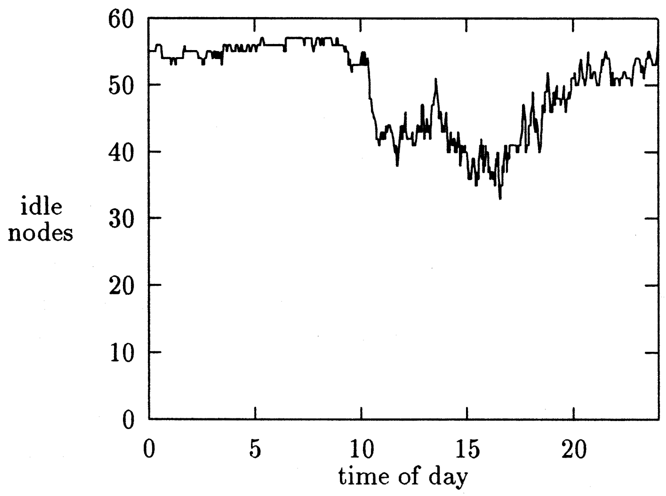

Figure 1.2: Number of unused nodes over the course of a day. The graph starts at midnight. Sixty nodes were observed.

typically available in workstation clusters. To avoid intruding on node owners when running a network parallel computation, a user must locate nodes that can host his program's processes, and ensure that each of the nodes will remain idle for the duration of his computation. Short of individually polling owners, it is impossible for him to learn which nodes will be idle long enough (and at the right time) to host his processes. Furthermore, even if he does find enough nodes, but an owner changes his plans or the computation runs longer than expected, the computation could interfere with that owner's work. And even then, he still wastes the fragments of idle time available on other nodes. These fragments, when aggregated, represent a large amount of computer time.

Figure 1.2 shows a graph of the number of nodes that were idle over the course of a day. While individual nodes toggle between idle and busy, the overall number of idle nodes remains high. If the number of idle nodes is large, the amount of idle time must also be large. Thus, figure 1.2 supports our assertion that the slices of idle time shown in figure 1.1 include a large amount of potential compute time. Our aim is to harness the power of the bits and pieces of idle workstation time, and to focus it on explicitly parallel computations.[3]

[3]We have called this "supercomputing out of recycled garbage" [GK91].

1.2 Adaptive Parallelism

Definition: An *adaptively parallel computation* is a parallel computation that can execute on a dynamically changing set of nodes.

An adaptively parallel program adjusts to changes in the set of nodes hosting it. It can respond to changes both in the location and number of its processes. A change in location occurs when one node terminates a process and another initiates one. The number of processes remains the same, but the location of the processes changes. A change in the number of processes occurs when a node terminates a process and no other node initiates one, or when a node initiates a process and no other node terminates one. In the former case, the number of processes grows, while in the latter it shrinks. Despite growing, shrinking or moving, an adaptive computation proceeds.

An adaptively parallel computation can execute using bits and pieces of network idle time (such as the ones depicted in figure 1.1). When a node becomes idle, it starts a process that assists an ongoing adaptive computation. When a node becomes busy, it kills any processes participating in adaptive computations, and the computations continue using their remaining processes. If all nodes become busy, the number of nodes working on an adaptive computation drops to zero. An adaptively parallel computation will stall when no nodes are working on it, but it will restart as soon as some node joins.

1.3 Previous Uses of Idle Time

Many of the previous attempts to compute using idle cycles are degenerate cases of adaptive parallelism. They are either not adaptive—i.e., the number of processes is not dynamic—or not true parallel computations.

If an applications consists only of independent tasks—i.e. no task depends on any other—its tasks can be executed in any order or in parallel. A *hunter*[4] executes the tasks of an independent-task application on idle nodes. When a hunter locates an idle node, it starts a process on the node. The process then performs tasks on behalf of an application, and the results are passed back to a master process. If more nodes are idle, more tasks can be executed concurrently, and more parallelism can be achieved. Condor [LLM88], Godzilla [Cra90] and RES [Car92] are examples.

Hunters are a degenerate case of adaptive parallelism. Since hunters do not support IPC among the the processes consuming tasks, they support only a limit class of parallel programs (those in which tasks are independent). They are, however, fully adaptive: the degree of parallelism increases and decreases with the number of free processors.

Migrable network parallel (MNP) systems use idle cycles to run parallel programs.

[4]We borrow the term "hunter" from Litzkow who uses it to refer to Condor [LLM88].

MNP systems do this by assigning pieces[5] of a network parallel program to idle nodes and migrating them off nodes that become busy. If the number of pieces exceeds the number of idle nodes, multiple pieces are assigned to the same node. When there are more idle nodes than pieces of the program, some nodes are not used. Amber [CAL+89] and The Benevolent Bandit [FSK89] are representative.

Unlike hunters, MNP systems support IPC and are thus complete parallel systems. However, they are not fully adaptive since the degree of parallelism in such systems is not related to the number of available nodes. While pieces of the program move in response to changes in the environment, the *number* of pieces is selected when the program is written or when it is run and is unrelated to the computing environment.

1.4 The Piranha Model of Adaptive Parallelism

The Piranha model is a nearly complete realization of adaptive parallelism. The number and location of processes change in response to changes in the environment, and IPC is supported. Two types of processes exist in Piranha: A single, persistent *feeder* and multiple, transient *piranha.* A piranha process executes a short *retreat* function, then terminates when its processor is withdrawn from the computation. New piranha processes are started when processors become idle. Processes exchange data through a globally-accessible object memory. Piranha is not completely adaptive since it requires a single, persistent feeder; thus the number of Piranha processes never falls to zero.

1.5 Advantages of Piranha

The Piranha model holds a number of advantages over the models presented in section 1.3. Since Piranha is a transient process model, it avoids the problems of process migration incurred by MNP systems. Rather than moving processes from node to node, Piranha simply terminates them (after their retreats complete) when necessary.

Since process migration is difficult to implement without modifying the operating system [DO91], the use of MNP systems is somewhat limited. Node owners are, understandably, reluctant to permit modifications to their operating system. Piranha runs over Unix and requires no operating system changes. Implementing migration is more difficult still in a heterogeneous environment [ZWZD92]. We are not aware of any MNP systems that run heterogeneously. By its nature, Piranha is strongly heterogeneous: Piranha never migrates processes, and Piranha's globally-accessible object memory can be shared by dissimilar nodes using well-established data conversion techniques (e.g. XDR).

Message-passing MNP systems face other challenges. If a process P_a sends a message to process P_b on node N_1 and P_b is relocated to node N_2 before receiving the

[5]We use the term piece to refer to objects in an object-oriented system and to processes in a process-oriented system

data, the message must be forwarded from N_1 to N_2. Object-oriented systems have a similar problem in locating method calls in mobile objects. Message forwarding can result in inefficiency and additional system complexity.

In Piranha, data are not passed directly from one process to another. Instead, Piranha communication is achieved through modifications to *distributed data structures*—collections of data objects that are accessible to all processes [CG90]. Thus, communication is uncoupled and anonymous: two processes need not synchronize to exchange data or even know of each other's existence. Any process can modify a data structure, and a change made to a data structure by one process is visible to all other processes. Thus, processes can communicate without regard to lifetime, location or name. Uncoupled and anonymous communication is important in Piranha since processes are transient. If communication were not uncoupled, unless processes overlapped temporally, they could not communicate. If communication were not anonymous, data sent to a specific process could be lost if that process terminated.

Another problem with MNP systems is that the number of processes is unrelated to the number of available processors. An application starts with a fixed number of processes. When the number of processes exceeds the number of idle processors, the system must assign multiple processes to some processors. To avoid one node shouldering too heavy a load, MNP systems require a load balancing mechanism. Such a mechanism will increase system complexity and may leave residual imbalances that degrade performance. Conversely, when the number of processors exceeds the number of processes, resources will be wasted. Piranha parallelism is limited only by the number of free processors and uncompleted tasks.

While hunters (e.g. RES) do not suffer from the drawbacks listed above, they are useful only for a limited problem domain—problems with no intertask dependencies. As Carlson writes [Car92]:

> Direct communication between peer processes executing in RES is generally unworkable. This is because the the scheduler may choose not to run any particular set of processes at one point in time. Even more difficult is the fact that, because a process may be killed at any point in its execution and restarted on another system, it would be infeasible to set up any form of reliable inter-process communication between two processes.

Many of the applications discussed in chapter 2 (e.g. DNA sequence similarity assessment; see section 2.1.3) include intertask dependencies. Since Piranha is built on Linda (see section 1.7), each Piranha process can communicate with all other processes. Results of computing one task can be passed from the worker that completed the task to a worker requiring the result. Thus, applications with intertask dependencies are within Piranha's domain.

A final advantage of the Piranha is the leverage we can gain from the master/worker [CG90] programming style. As we will demonstrate in chapter 2, Piranha programming is simplified by using a number of anonymous, identical compute servers.

1.6 The Model in Practice

A Piranha programmer writes a program by specifying three coordination functions—**`feeder`**, **`piranha`** and **`retreat`**—and an arbitrary number of computation functions. The coordination functions define the interaction among the processes. The computation functions perform the task at hand.

Piranha processes transform a distributed data structure describing work into one describing results. Distributed data structures are stored in tuple space where they are accessible to all participating processes.

When a Piranha job is submitted, the *feeder* is automatically executed by the Piranha run-time system on the *home node.* It starts a computation by creating a distributed data structure consisting of *tasks* that describe work to be done. We call this the *task data structure.*

The **`piranha`** function acts as a template for piranha processes. When a node becomes idle, a new process executing the code in the `piranha` function is automatically started on it. There is no explicit process creation in Piranha. While it is active, a typical piranha process repeatedly removes a task from the task data structure, computes its result, and places that result into a distributed *result data structure.* When all tasks have been consumed, the computation is complete and the feeder collects the results.

If the node hosting a piranha process becomes busy (e.g. an owner presses a key), the piranha process calls its *retreat* function. A simple retreat function places the process's current task back into the task data structure. In more complicated retreats, a retreating process might make more extensive changes to one of the program's distributed data structures. The process exits when retreat completes, and the computation proceeds without it. If the node becomes idle again, it starts a new piranha process.

In more complex Piranha algorithms, some tasks depend on output produced after computing other tasks. A piranha process must gather all of a task's *dependency data* before it can complete the task. Dependency data are produced by piranha processes as a result of completing tasks and may be stored along with the tasks in the task data structure, with the results in the result data structure, or in some other distributed data structure.

A task is called *enabled* when it can be completed. A task can be completed when all of the data on which it depends are present. If some data are missing, the task is non-enabled and cannot be completed until the remaining data are supplied. A non-enabled task becomes enabled when some piranha process (or the feeder) supplies the remaining dependency data.

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

Piranha programmers need not write each program from scratch. We have iden-

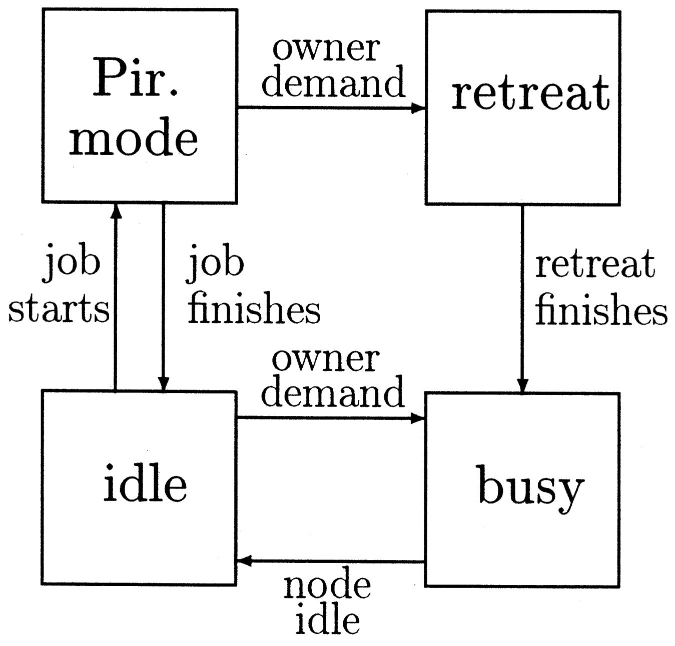

Figure 1.3: Piranha state diagram

tified a number of Piranha coordination structures that can be used to guide future application development. By extracting the coordination structures from existing programs, we can simplify coding of similar applications. To encourage code reuse, we plan to insert a number of Piranha *templates* into the Linda Program Builder, a template based editor [ACG91].

Figure 1.3 shows a state transition diagram for Piranha nodes. Nodes not hosting Piranha jobs toggle between idle and busy based on owner demand. If a node is idle when a Piranha job is submitted, the node starts a piranha process and transitions to Piranha mode. If the application completes, the node becomes idle. If the owner demands his node while it is in Piranha mode, the piranha process executes its retreat function and the node transitions to busy.

1.7 Fit to Linda

The Piranha model makes no assumptions in the abstract about the coordination language (or parallel language or coordination library) used to build Piranha applications. However, because of its nature, Piranha demands specific capabilities from a coordination language.

As discussed in 1.5, since Piranha is a transient-process model, communication among processes must be anonymous and uncoupled: data cannot be passed directly from one process to another. Instead, data must be stored in a medium accessible to all processes—i.e. in distributed data structures. To support high-performance applications, interaction with the data structures must be efficient.

Linda's tuple space supports anonymous, uncoupled communication. Tuple based communication is anonymous since, to access a tuple, a process needs to know only its structure, not its producer's name:

> Processes deal only with tuple space, not with each other. Thus, no process has any direct dependence on the...identity of other processes... [BCG+91b]

Tuple based communication is also uncoupled. Once a process creates a tuple, the process might exit, but the tuple will remain in tuple space.[6] Until explicitly removed from tuple space, tuples can be accessed by any process. Again from [BCG+91b]:

> Processes don't care when the information they produce will be used; they don't care when the information they consume was produced. Two processes whose lifetimes are in fact wholly disjoint may communicate via tuple space.

Thus, processes need not overlap temporally to exchange data.

Thus, since Linda communication is both anonymous and uncoupled, Piranha is a tight fit to the Linda model.

1.8 Related Work

In section 1.3, we discussed previous attempts to use idle time. Below we present a more thorough survey of work related to Piranha.

The literature abounds with papers describing attempts to use idle cycles. Among the first to use idle time were systems that offload sequential jobs from loaded nodes to idle ones. Typically, a *controller* process in such systems keeps a list of idle nodes. Nodes inform the controller when they transition between idle and busy. When an application is submitted for remote execution, the controller assigns it to an idle node. Systems such as Butler [Nic87], Condor [BL91, LL90, LLM88], and DAWGS [CM92] are representative. When a node becomes busy, remote processes must be removed from it. Condor and DAWGS migrate jobs when an owner reclaims his node; Butler kills jobs and restarts them on another nodes.

As discussed in section 1.3, hunters use idle time to run independent tasks comprising a single application. Systems such as Godzilla [Cra90], RES [Car92], LSF

[6]While some Linda implementations tie tuple space to the processes accessing it, this is not inherent to the Linda model.

[WZAL93, ZWZD92], SPAWN [WHH+89], and V [Che88, Che84, TLC85] are examples. Silverman reports that a 116-digit number was factored using idle nodes located at sites nation-wide. Work was parceled out using electronic mail [Sil91]. Resource managers such as DQS [GS92] and NQS [DeR92, DeR93] can also schedule a number of independent trials.

Resource managers can also be used to schedule network parallel applications on idle nodes. For example, DQS and LSF will schedule Network Linda and PVM jobs. However, there is no guarantee that the resource needs of the network parallel jobs will be met by the idle nodes. For example, only ten nodes might be available for a parallel job requiring fifteen. Furthermore, nodes do not necessarily remain idle. Even if fifteen nodes are available when the job is started, some of those nodes might become busy as the computation progresses. No effort is made by resource managers to alter an application's computing pool to match a changing set of idle nodes.

MNP systems extend the network parallel approach by assigning pieces of a network parallel program to idle nodes and migrating them off nodes that become busy. (See section 1.3.) Amber [CAL+89], The Benevolent Bandit [FSK89], Charlotte [AF89], Clouds [DLAR91], The Parform [CS93] and Sprite [DO91, OCD+88] take this approach.

1.9 The Dissertation

In this dissertation, we will show:

- a model for adaptive parallelism that accommodates a broad range of coarse-grained, asynchronous program structures (presented above),
- a method for applications development that exploits the model, and
- a software system to support the model effectively.

The remainder of this dissertation proceeds as follows: chapter 2 describes the Piranha programming methodology, chapter 3 describes the use of idle time as a resource, chapter 4 presents the software infrastructure developed to support Piranha, chapter 5 discusses our view of Piranha's future and chapter 6 contains concluding remarks.

Chapter 2

Program Development Methodology

2.1 Introduction

Piranha programmers, like sequential code developers, must consider correctness (does the program compute the correct answer?) and efficiency. Additionally, Piranha programmers must ensure that execution will not stall[1] despite an arbitrary pattern of retreats. For example, if $N - 1$ piranha processes are blocked waiting for data from a task held by process P, and P retreats, execution will stall until the task is completed by some other process.

To understand the implications of an application's task interdependencies (hence the potential to stall), we construct a dependency graph (depgraph). A depgraph consists of vertices representing tasks and directed edges representing intertask dependencies. If, in an application A, task t_2 cannot complete without data from task t_1, the depgraph for A contains a directed edge from the vertex representing t_1 to the vertex representing t_2.[2] A depgraph is a partial ordering on an application's tasks. Depgraphs are acyclic.

To reveal potential task-completion orderings, we label each vertex in a depgraph with its depth in the partial ordering. If a vertex has no incoming edges, the task represented by that vertex can be completed without completing any other tasks, and the vertex is labeled zero. If a task depends only on tasks with vertices labeled zero, its vertex is labeled one, and the task cannot be completed until all tasks on which it depends have been completed. In general, we label a vertex V one greater than the

[1]We use the term "stall" instead of "deadlock" since a stalled Piranha computation might restart if a new worker joins.

[2]In many cases, the size of a problem (hence the number of tasks) depends on an input parameter. In such cases, the size of the depgraph may vary, but the general pattern of task interdependency tends to be the same.

maximum of all labels on vertices with edges into V.

A depgraph evolves as a computation proceeds. When a task is completed, its vertex is removed and the graph is relabeled. When the label on a vertex representing a task reaches zero, that task can be completed without completing any other task. Tasks labeled zero are called *enabled.* Since progress is made when tasks are completed, and only enabled tasks can be completed, active Piranha processes generally claim enabled tasks before non-enabled ones. It is the programmer's responsibility to impose this ordering (or to use an alternative one).

Below we present a taxonomy of static depgraphs and propose task execution orderings for each member of the taxonomy. As the depgraphs include more complex edge patterns (i.e. the programs contain more complicated intertask dependencies), the rules for ordering task completion become more complex. Our goal is to create programs from these orderings that are correct, efficient and stall-free. In presenting our taxonomy, we begin with problems with no dependencies and proceed through cases of increasing complexity.

2.1.1 Unordered Depgraphs

We begin our presentation with a simple case: all tasks are independent. In this case the depgraph is *unordered*—i.e. it contains no edges—and all tasks are initially enabled. Piranha processes can compute tasks in any order. Pseudocode for this type of program is given in figure 2.2. (We discuss `task_start` and `task_done` below.)

Pseudocode presented in this chapter is not strictly correct. In the examples presented, the feeder does not consume tasks. If all of the piranha processes were to retreat, no processes would consume tasks, and the application would stall. Since the feeder remains active, there is no reason for this stall to occur—in fact, Piranha programs should never stall.

In practice, this flaw is not significant: the number of piranha processes seldom, if ever, drops to zero. As we demonstrate in section 3.2.1, a substantial fraction of our nodes tend to be idle. Since idle nodes are available to host piranha processes, our applications have alway had at least one worker process. Thus, even if the feeder does not consume tasks, some process will. For simplicity, we tend not to have the feeder consume tasks. Feeders could consume tasks, but at the cost of increased code complexity.

Even when the feeder does consume tasks, Westbrook showed that, given current Linda semantics, there is no guarantee that this code will complete—i.e. there is a potential for livelock [CGKW93]. The intuition behind the proof is that a piranha process can claim a task, perform some computation and retreat. Another process could claim that same task, compute and retreat. These exchanges could continue indefinitely. Linda does not ensure fairness,[3] so even though the feeder remains alive

[3]Fairness here means that if N processes $P_1 \ldots P_n$ repeatedly in and out a tuple T, then each process will access T eventually.

throughout the computation, there is no guarantee that it will ever receive and complete the task.

Livelock is unlikely to occur in practice. We have found that nodes tend to be available for about 10 minutes (see section 3.2.1) while tasks tend to complete in less than a minute. Consequently, processes can usually complete at least one task (and typically more) before retreating. The probability that a single task will be repeatedly claimed and retreated is extremely small, and we have not observed any instances of livelock.

Atearth, written by Martin White of the Yale Physics department, is an example of a program with an unordered depgraph [GKW91]. Atearth is a simulation of the flight of neutrinos from the sun toward the Earth. The simulation consists of a number of trials, and each trial simulates the flight of a single neutrino with given characteristics (e.g. mass and energy). The trials are independent.

Table 2.1 shows the performance of Atearth. We present data for sequential, Piranha and Network Linda versions of the program.

Figure 2.1 shows how each value in the efficiency chart is calculated. For the Piranha version, we measure run time and worker time, and the number of workers is calculated. 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 nodes will shift the curve, but not change its shape. Thus, we believe omitting feeder time is, on balance, reasonable.

For the Linda version, we fix the number of nodes and measure the run time. We calculate worker time by multiplying the run time by the number of workers. The Linda versions of our sample programs use the master/worker programming model. Like the feeders in the Piranha versions, Linda master processes do not consume tasks, and they are not counted in the numbers of workers.

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. In a homogeneous network, the average number of nodes used during a run determines its average compute intensity. 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.[4] When we could not equalize the compute intensities, we used a slightly lower Network Linda intensity. Since our applications do not scale superlinearly, using fewer nodes (thus lower intensity) improves application

[4]Fractional numbers of Network Linda nodes were obtained by averaging results from runs using different numbers of nodes.

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. While we can run a Piranha application at any time, secure that it will ferret out available idle cycles, we can run Network Linda only when we can find a sufficient number of free nodes. To avoid interfering with interactive work, we should run Network Linda only at night or only on reserved nodes. If a node is idle unexpectedly, its compute power is wasted. Furthermore, Network Linda nodes must be idle not only when a job starts, but when it ends. Guaranteeing that a node will remain idle for the duration of a computation can be difficult, especially when running during the day.

In some Linda programs, calculating the worker time by using a fixed number of workers will overstate the amount of time workers actually spend working. If the number of workers exceeds the number of tasks, some workers will be wasted. Such imbalances often occur at the end of a computation when only a few tasks remain. If we allowed taskless workers to exit, the overall worker time would be reduced. In the examples we present, the number of tasks is much larger than the number of workers. Thus, while we may waste some worker time at the end of a computation, the waste is very small with respect to the overall worker time.

We use a standard Network Linda version of each program for our benchmarks. This version does not include any code required to support Piranha's retreat.

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 coarse grained (56,000 tasks in over 25 hours, or an average of 1.6 seconds per task) and its task descriptor tuples and result tuples are small (less than 50 bytes each), the application has a high computation to communication ratio, and high efficiency results.

Atearth demonstrates a general point: network parallel programs with high computation to communication ratios are often very efficient with respect to their sequential versions. Inefficiency typically results from spending a large amount of time exchanging data over the network. This point also applies to Piranha.

Consistency in the Presence of Retreat

A retreat can potentially cause an inconsistency in a distributed data structure. To allow programmers to ensure data-structure consistency, we introduce two system calls that restrict the delivery of retreat.

Consider the code fragment in figure 2.3 taken from figure 2.2. If a retreat strikes at the line labeled `A`, the piranha process would not have published the result of its current task. The proper action in the retreat function would be to return the task descriptor to tuple space. At the line labeled `B`, the result would have been produced, and the proper retreat action would be to do nothing. Returning the task descriptor to tuple space would cause a duplicate result to be produced.

- 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 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 2.1: 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 2.1: 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.

```
feeder ()
{
  task_count = 0;
  while (get_task (&data)) {
    out ("task", task_count, data);
    task_count++;
  }

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

int current_task_id;
piranha ()
{
  while (1) {
    in ("task", ? current_task_id , ? data);
    task_start ();
    compute_result (current_task_id, & data, & result_data);
    task_done ();
    out ("result", current_task_id, result_data);
  }
}

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

Figure 2.2: Unordered depgraph code skeleton

```
int current_task_id;
piranha ()
{
  while (1) {
    in ("task", ? current_task_id , ? data);
    compute_result (current_task_id, & data, & result_data);
    /* A */
    out ("result", current_task_id, result_data);
    /* B */
  }
}
```

Figure 2.3: The duplicate tuple problem

The problem in this scenario is that there is no way to determine whether the out has completed when a retreat strikes. Bakken and Schlichting encountered a similar problem when designing a fault-tolerant Linda system and call this the "duplicate tuple problem" [BS93].

To prevent the duplicate tuple problem, Piranha supplies system calls to block and enable the retreat signal. Upon entry to the **piranha** function, the retreat signal is blocked. After it claims a task, a process enables retreat by calling **task_start**. The retreat signal remains enabled until immediately before the result is published. The process disables retreat by calling **task_done**. Figure 2.4 shows the program rewritten.

Blocking retreat solves the duplicate tuple problem. Since a programmer can keep retreat blocked except when a process is actually working on the task (i.e. between the **task_start** and the **task_done**), he knows that retreat will only strike his process after a task is claimed, and before its result is produced. Thus, he knows that replacing the task will not cause a duplicate result.

Of course, a malicious programmer could omit the **task_start** call, and retreat would never strike. We discuss this problem in sections 4.3.5 and 5.2.4.

In some of the code fragments given in this dissertation, we omit **task_start** and **task_done** for clarity.

Loose Orderings

Loose orderings are a slight twist on dependency free problems. In some applications, we wish to collect results in a specified order. For example, we might want to display the results of an infinite stream of tasks. Processing the tasks randomly would make difficult the orderly display of results.

To impose a general order on result production, we impose a loose ordering on the

```
int current_task_id;
piranha ()
{
  while (1) {
    in ("task", ? current_task_id, ? data);
    task_start ();
    compute_result (current_task_id, & data, & result_data);
    task_done ();
    out ("result", current_task_id, result_data);
  }
}
```

Figure 2.4: A solution to the duplicate tuple problem

way tasks are claimed. Piranha processes claim the next task in a FIFO task queue and add the result from that task to the result list. While imbalances in processor speed and task length will cause some disorder in result production, results production will be more orderly than if tasks were claimed randomly.

If a retreat strikes while a process is consuming a task, a hole will form in the list of results until the task is completed. To ensure that the hole will be filled, and thus order reestablished in result production, we maintain a second distributed data structure—an unordered bag holding retreated tasks. Tasks in this bag have priority over the unstarted tasks stored in the FIFO queue. To claim a task, a piranha process looks first to the retreated-task bag. If there are no retreated tasks, the process consumes the next unstarted task. Claiming tasks in this way insures that new tasks will not be started when any retreated task waits unattended. Pseudocode is given in figure 2.5.

As discussed above, it is possible for an unlucky task to be repeatedly claimed and retreated thereby leaving a hole in the result list. If tasks are sufficiently small with respect to the expected idle period of a node (and this has been true in our experiments), this pathological case is extremely unlikely.

Note that the depgraph for this type of problem consists of vertices and no edges. The tasks *can* be completed in any order. However, we impose a loose framework on task completion—we attempt to ensure that the earliest uncompleted tasks are claimed by some piranha process.

We use a loose ordering when running a Piranha version of Rayshade. Rayshade is a ray tracing program written by Craig Kolb of Princeton University and enhanced by Ken Musgrave of George Washington University. In sequential Rayshade, the program creates an image by reading a scene description and rendering from it a series of horizontal scanlines. The scanline at the bottom of the image is rendered first, and work proceeds up the image. The scanlines can be displayed (e.g. using `getx11`) or

```
feeder ()
{
  task_count = 0;
  out ("head", 0, 0);
  while (get_task (&data)) {
    out ("data", task_count, data);
    task_count++;
  }
  for (i=0; i < task_count; i++) {
    in ("result", i, ? result_data);
    store_result (i, &result_data);
  }
}
int current_task_id;
piranha ()
{
  while (1) {
    in ("head", ? current_task_id, ?num_retreated);
    if (num_retreated > 0) { /* get a retreated task */
      out ("head", current_task_id, num_retreated-1);
      in ("retreated task", ?current_task_id, ?data);
    }
    else {
      out ("head", current_task_id+1, num_retreated);
      in ("data", current_task_id, ?data);
    }
    task_start ();
    compute_result (current_task_id, & data, & result_data);
    task_done ();
    out ("result", current_task_id, result_data)
  }
}
retreat ()
{
  in ("head", ?j, ?num_retreated);
  out ("head", j, num_retreated+1);
  out ("retreated task", current_task_id, data);
}
```

Figure 2.5: Loose ordering skeleton

	Average Nodes	Run Time	Worker Time	Speed Up	Eff. (%)
Sequential		14.1			100
Piranha	4.3	3.6	15.5	3.9	91
Net Linda	4	3.6	14.3	4.0	99

Table 2.2: Rayshade performance. Time is given in hours. The Piranha pool consisted of 10 IBM RS/6000 model 340's. The image consists of 8192 scanlines, each containing 4096 pixels.

written to disk as they are rendered.

Network Linda Rayshade uses the master/worker paradigm. The master initially releases untraced scanlines. Each worker repeatedly claims a scanline and traces it. Workers claim scanlines from bottom to top. The master collects the traced scanlines in the same order and writes them to a file or to `stdout`.

The Piranha version defines a task to be rendering a single scanline, and scanlines are claimed using a loose ordering (working bottom-up). A loose ordering is used so that scanlines can be collected and stored in a format appropriate for visualizer programs, typically bottom to top.

Piranha processes could trace scanlines out of order. The feeder could then sort them on-line or the user could sort them in a post-processing step. However, since an image often consists of megabytes of data [Mus93a], such sorting is costly if not impossible. A loose ordering allows the feeder to collect and output the scanlines in order.

We present performance data for Rayshade in table 2.2. The data in the table are analogous to those in table 2.1. Since tasks (scanlines) are long (6.2 seconds, on average) and independent of other tasks, we expect performance to be good. Piranha is 91% and 92% efficient with respect to the sequential code and Network Linda, respectively.

Time lost to retreat did not greatly impact performance. On average, the time wasted to retreat was less than one minute. When tasks are short, as they are here, little work is wasted on each retreat. Since retreats are infrequent (see section 3.2.1), when tasks are short, the aggregate time wasted to retreat will also be small.

Despite longer tasks, Rayshade is slightly less efficient than Atearth. The loss of efficiency may result from more time spent by Rayshade outputting result tuples. Rayshade outputs an entire scanline while Atearth outputs only four floats per task. The size of the task descriptor tuple is similar in both programs.

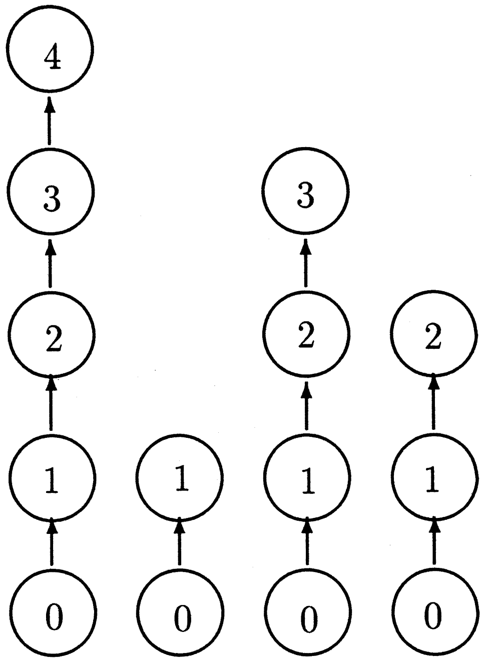

Figure 2.6: Independent sets of ordered elements

2.1.2 Independent Runs

In unordered graphs, every task is enabled initially, so any piranha process can claim any task at any time. If tasks are interdependent, this will not be the case. If only non-enabled tasks are claimed, execution will stall. Figure 2.6 shows a graph in which only the tasks at the bottom of the figure (labeled zero) are initially enabled. Once the first task in the column is completed, the second task in the column (initially labeled one) is enabled.

We could use a bag-of-tasks to solve this problem. As shown in the pseudocode in figures 2.7 and 2.8, a piranha process claims an enabled task and the dependency data for that task and computes its result and new dependency data. While this solution is correct, it performs more communication (**in**ing and **out**ing dependency data) than is necessary.

We can improve efficiency in the example above by reducing the number of times dependency data are moved. If task B depends on task A, and both A and B are executed by the same process P, P does not have to read the dependency data from A before executing B; P produced it.

We call tasks A and B *related* if there is an path in the dependency graph from

```
feeder ()
{
  int run, task;
  for (run=0; run < NUM_RUNS; run++) {
   out ("enabled", run, 0);
   for (task=0; task < TASKS_PER_RUN; j++)
    out ("task", run, task, data[run][task]);
  }

  for (run=0; run < NUM_RUNS; i++)
   for (task=0; task < TASKS_PER_RUN; task++) {
    in ("result", run, task, ?result_data);
    store_result (i, &result_data);
  }
}
```

Figure 2.7: Independent runs bag-of-tasks feeder skeleton

A to B, or vice-versa.[5] If tasks A and B are related, we can reduce dependency-data communication by executing both tasks on the same processor. We further reduce communication if a single process executes all tasks along the path from A to B—dependency data along the path need never be read from tuple space.

We call such a progression of tasks a *task run.* A task run from vertex A to vertex B is a linear chain containing the vertices encountered in a path from A to B. All vertices in a run are related.

We can reduce an application's communication by assigning an entire task run to a single processor. Once a process claims a task in a run, it is responsible for completing all tasks in that run. Thus, by claiming a single task, a process implicitly reserves a number of related tasks. Dependencies are localized and interprocess communication is reduced.

As when claiming tasks, we must avoid stalling when claiming runs. We call a run *enabled* if no edge is directed from any vertex outside the run to any vertex within it. All tasks in enabled runs can be completed without data from any other run. For example, all of the runs in figure 2.6 are initially enabled. Thus, processes can claim the runs in any order without fear of stalling. An *independent run* problem is one in which all runs are initially enabled—i.e. no run depends on any other. If runs are not independent, and all active processes claim non-enabled runs, the computation will stall.

[5]We will sometimes speak of, for example, "the edge connecting tasks A and B." Strictly speaking, we should say "the edge connecting the vertices representing tasks A and B."

```
int current_run, current_task_id;
piranha ()
{
  while (1) {
    in ("enabled", ?current_run, ?current_task_id);
    in ("task", current_run, current_task_id, ? data);
    if (current_task_id != 0)
      in ("dependency data", current_run,
          current_task_id-1, ? dependency_data);
    /* calculate the result and new dependency data */
    task_start ();
    compute_result (current_run, current_task_id, &data,
                    &dependency_data,
                    &new_dependency_data, &result_data);
    task_done ();
    out ("result", current_run, current_task_id, result_data);
    if (current_task_id < TASKS_PER_RUN-1) {
      out ("dependency data", current_run,
            current_task_id, new_dependency_data);
      out ("enabled", current_run, current_task_id+1);
    }
  }
}

retreat ()
{
  out ("enabled", current_run, current_task_id);
  out ("task", current_run, current_task_id, data);
  if (current_task_id != 0)
    out ("dependency data", current_run,
          current_task_id-1, dependency_data);
}
```

Figure 2.8: Independent runs bag-of-tasks piranha and retreat skeletons

```
feeder ()
{
  task_count = 0;
  while (get_task (&data)) {
    out ("task", COARSE_SEARCH, task_count, data);
    task_count++;
  }

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

Figure 2.9: Dipole feeder skeleton

One type of independent run problem is a phased computation. Phased computations consist of a number of phases, each consisting of a number of tasks. The output from a task in one phase is the input to at most one task in the next, and each task depends on at most one other task. Tasks in the same phase are independent. For example, in a dipole localization program (Dipole) written by Srini Rao of the Electrical Engineering Department at Yale, the first phase coarsely searches for dipoles in a grid representing a magnetic field. Dipoles found in the first phase are localized in the second phase. The outputs from the first phase (coarse localizations of the dipoles) are the inputs to the second. Figures 2.9 and 2.10 shows the code skeleton for Dipole.

The depgraph for Dipole is given in figure 2.11. It has only two layers, one for each phase of the search. Since the first phase consists of 2744 tasks (a 3 dimensional grid was broken into 14^3 regions) and only 12 dipoles were found, the second layer often consisted of no work.

Table 2.3 shows the performance of Dipole. The Piranha code performed well compared to both the Network Linda and the sequential versions. This is not surprising since the computation is coarsed grained.

2.1.3 Unidirectional Dependencies

Not all runs are independent. We call run B *unidirectionally dependent* on run A if there is a path from some vertex in A to some vertex in B, but there is no path from any vertex in B to any vertex in A. Intuitively, some task in B depends on a task in A, but no task in A depends (directly or indirectly) on any task in B. We call runs A and

```
int current_task_id, type;
piranha ()
{
  while (1) {
    in ("task", ? type, ? current_task_id, ? data);
    if (type == COARSE_SEARCH) {
      /* "data" updated in place */
      task_start ();
      found = compute_intermediate (current_task_id, & data);
      task_done ();
      if (found) {
        type = FINE_SEARCH; /* in case of retreat */
        task_start ();
        compute_final (current_task_id, & data, & result_data);
        task_done ();
        out ("result", current_task_id, result_data);
      }
      else
        /* tell the feeder there was no dipole */
        /* phase 2 is a no-op */
        out ("result", current_task_id, NOT_FOUND);
    }
    else { /* got a task in phase 2 */
      task_start ();
      compute_final (current_task_id, & data, & result_data);
      task_done ();
      out ("result", current_task_id, result_data);
    }
  }
}

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

Figure 2.10: Dipole piranha and retreat skeletons

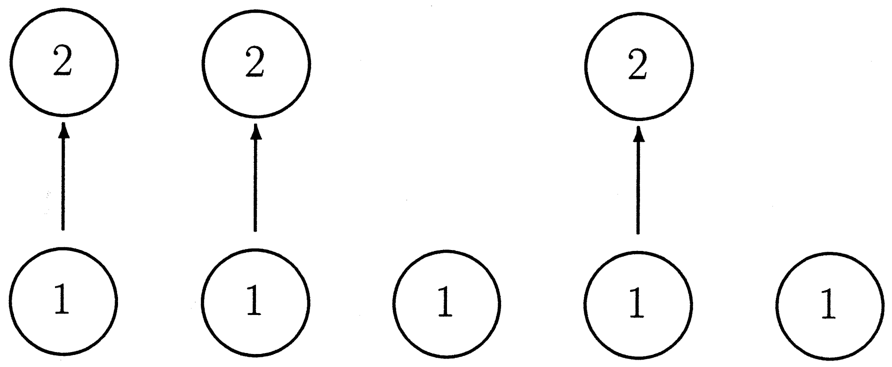

Figure 2.11: Dipole dependencies

	Average Nodes	Run Time	Worker Time	Speed Up	Eff. (%)
Sequential		22.8			100
Piranha	5.7	4.3	24.1	5.3	95
Net Linda	5	4.9	24.5	4.7	93

Table 2.3: Dipole performance. Time is given in hours. The Piranha pool consisted of 10 IBM RS/6000 model 340's. The average time wasted by retreat was .1 hours.

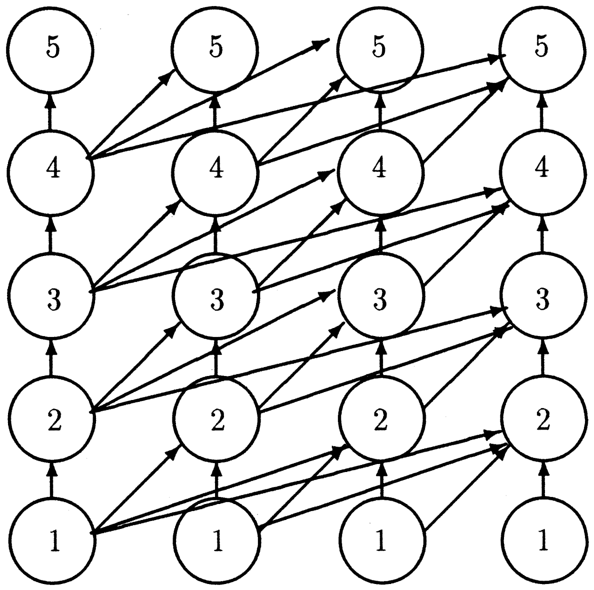

Figure 2.12: Unidirectional dependencies graph

B bidirectionally dependent if there is a path from some vertex in A to some vertex in B and a path from some vertex in B to some vertex in A. (Bidirectional dependencies are discussed in section 2.1.4.) Figure 2.12 contains only unidirectionally dependent runs. Like individual tasks, runs form a partial ordering, and each run can be labeled with its depth in the ordering. Figure 2.13 shows such a numbering for one possible group of runs.

In problems with independent runs, it does not usually matter which run is claimed by an active process.[6] However, when runs are unidirectionally dependent, improper ordering can cause execution to stall. If all active processes claim non-enabled runs (those with topological numberings greater than zero), the computation will stall. If we group the tasks of figure 2.12 into vertical runs as shown in figure 2.13, then only the leftmost run is initially enabled, and it should be claimed first.

At certain stages of a computation, it might be possible to complete some, but not all, of the tasks in a non-enabled run. The degree of parallelism is not necessarily limited to the number of enabled runs. Similarly, parallelism is not necessarily limited to the number of enabled tasks: it is sometimes possible to perform part of a task before that task becomes enabled (e.g. initializing a data structure). However, neither tasks nor runs can be *completed* until they are enabled.

As long as some enabled run is claimed, progress will continue. However, if only

[6]As in the case of loose orderings, in some cases it might be desirable to order the runs.

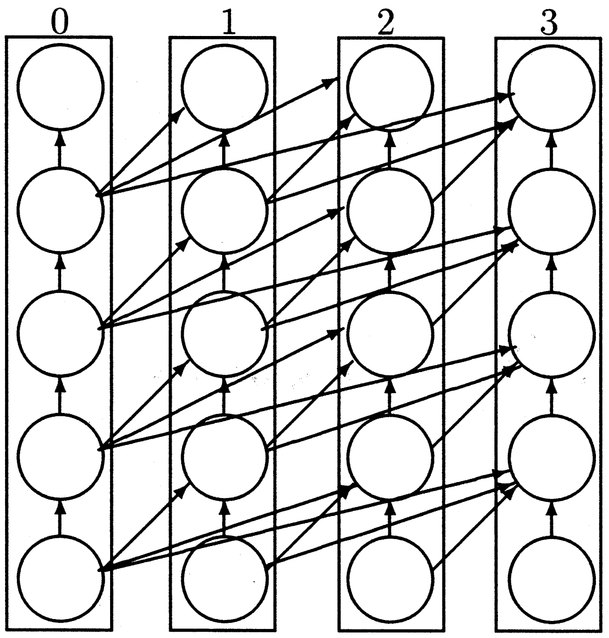

Figure 2.13: Numbered unidirectional dependencies graph

one piranha process is working on an enabled run, and that process retreats, execution will stall unless some action is taken.

We recover from a stall when some process claims and completes an enabled run. When retreat causes an enabled run to be released, some piranha process holding a non-enabled run releases it and claims the retreated one. This approach guarantees that incomplete, enabled runs will be claimed by some piranha process, and progress will be made.

This swap can be accomplished by modifying the way dependency data is transmitted. In a typical Network Linda program, a worker blocks waiting for a tuple containing data that satisfies its current dependency. In Piranha, retreat might indefinitely delay the production of a dependency tuple.

To prevent such stalling, in Piranha we add a special field to the data dependency tuple. The extra flag field allows a retreating piranha process to produce a *pseudo-dependency data tuple.* Depending on the the value of the flag, the tuple either contains actual dependency data or denotes that a retreat occurred. When a process completes a task, it produces a dependency data tuple with the flag unset. When a piranha process retreats, it produces a dependency data tuple with the flag set—i.e. it produces a pseudo-dependency data tuple. In either case, the tuple unblocks some process waiting for it.

The process that receives the tuple acts according to the flag. If the flag indicates that a retreat occurred, the receiving piranha process swaps its current task for the retreated one.[7] This process completes its new task and unblocks other processes waiting for the data. If the flag indicates that the tuple is genuine dependency data, the recipient piranha process uses the data as necessary to complete its current task. A pseudocode skeleton for LU decomposition, an application that uses this technique, is given below. Note that the LU decomposition skeleton is less complete than the ones presented previously; considerable detail has been omitted for clarity.

```
int column_index; /* the current task */
piranha ()
{
  while (1) {
    in  ("task", ? column_index);
    out ("task", column_index+1);
    in ("column data", column_index,
        ? pivots_applied, ? column_data);
    start_col: /* apply the pivots to the column */
      for (i=pivots_applied+1; i < column_index; i++) {
      if (! pivot_cached (i)) { /* pivot not cached?, get it*/
      get_pivot:
```

[7]While exchanging tasks is simple in principle, code to accomplish the swap is fairly complicated.

```
          rd ("pivot data", ? i, ? pivot_data[i], ? fake);
          /* if the pivot col was retreated, try to claim it */
          if (fake) {
            if (retreated pivot column is unclaimed) {
              /* still unclaimed, switch my col for it */
              retreat (); /* releases the current column */
              /* get the retreated column */
              in ("column data", i,
                  ? pivots_applied, ? column_data);
              column_index = i;
              goto start_col;
            }
            else goto get_pivot;  /* try to get data */
          }
          else pivot_cached(i) = TRUE;
        }
        apply pivot i to column column_index;
      }
      generate pivot from column_index;
      out ("pivot data", column_index, column_data:DIM, FALSE);
    }
}

retreat ()
{
  /* release a fake pivot */
  out ("pivot data", column_index, column_data:0, TRUE);
  /* release the column data */
  out ("column data", column_index, i-1, column_data);
}
```

In figure 2.14 we present efficiency data for a synthetic program containing only unidirectional dependencies. The program computes the results of a fixed number of tasks. Each task consists of a 1000 calls to a computation function. The computation function executes a number of operations. Each operation consists of one double precision multiplication and one add. The number of operations in the computation routine is determined by a command-line parameter. As expected, the efficiency of the program improves as the number of operations executed per task increases—i.e. the tasks become more coarse grained.