Scheduling Large-Scale Parallel Computations on Networks of Workstations

Robert D. Blumofe and David S. Park
MIT Laboratory for Computer Science
Cambridge, Massachusetts

Abstract

Workstation networks are an underutilized yet valuable resource for solving large-scale parallel problems. In this paper, we present "idle-initiated" techniques for efficiently scheduling large-scale parallel computations on workstation networks. By "idle-initiated," we mean that idle computers actively search out work to do rather than wait for work to be assigned. The idle-initiated scheduler operates at both the macro and the micro levels. On the macro level, a computer without work joins an ongoing parallel computation as a participant. On the micro level, a participant without work "steals" work from some other participant of the same computation. We have implemented these scheduling techniques in Phish, a portable system for running dynamic parallel applications on a network of workstations.

1 Introduction

Even with the annual exponential improvements in microprocessor speed, a large body of problems cannot be solved in a reasonable time on a single computer. One method of reducing the time for solving such problems is to develop algorithms to be used on parallel supercomputers. Much effort has been expended in improving the performance of parallel supercomputers. Another resource for performing large-scale parallel computations is networks of workstations. Workstation networks are increasingly prevalent, and since much of a typical workstation's computing capacity goes unused [20], a workstation network presents a large source of compute power on which to run large-scale parallel applications. Furthermore, a typical workstation is far less expensive than a compute node of most massively parallel supercomputers [19]. Because of these characteristics, networks of workstations are worthy of study as a way of performing large-scale parallel computations.

When compared to a massively parallel supercomputer, such as the CM-5 from Thinking Machines, a network of workstations suffers an obvious weakness: network performance. In particular, the software overhead incurred when sending a message on a typical workstation is often at least two orders of magnitude greater than the corresponding overhead on a parallel supercomputer. Also, the bisection bandwidth of a typical workstation network is again often at least two orders of magnitude less than the bisection bandwidth of a parallel supercomputer's interconnect. Despite these limitations, we demonstrate later that for some applications a good scheduler running on a network of workstations can reduce the interprocessor communications to the point where the modest communication performance does not degrade the overall application performance. Furthermore, and more importantly, improving workstation networks with low-latency and high-bandwidth interconnects (such as ATM) is a very active field of research [3, 15, 17], which is closing the gap between workstation networks and supercomputer interconnects.

Parallel computing on a network of workstations presents a vast array of scheduling choices. Specifically, when given a set of parallel jobs to execute, a scheduler must answer several questions for each processor:

- Should the processor work on a parallel job at all?
- When should the processor work on parallel jobs?
- Which parallel job should the processor work on?

This paper appeared in the Proceedings of the Third International Symposium on High Performance Distributed Computing, August 2–5, 1994, San Francisco, California.

E-mail: rdb@lcs.mit.edu or dspark@lcs.mit.edu.

This research was supported in part by the Advanced Research Projects Agency under Grant N00014-91-J-1698 and by Project SCOUT (ARPA Contract MDA972-92-J-1032). Robert Blumofe is further supported by an ARPA High-Performance Computing Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Many supercomputers answer these questions with a very limited set of possibilities. For example, the CM-5 divides its processors into a small number of fixed-size partitions. Each partition is run either in dedicated mode, where the processors complete one job after another, or in time-sharing mode, where the processors are gang-scheduled to the jobs in round-robin fashion. Thus, all the processors work on parallel jobs all the time, and all the processors in a given partition work on the same parallel job. For example, if 4 jobs wish to run in a 32-node time-shared partition, then each job runs on all 32 processors for some quantum of time until the job is preempted by the round-robin scheduler.

Clearly, this technique of allocating the 32 processors to the 4 jobs may not be the most efficient choice. First, empirical evidence (though in a somewhat different context) [26] indicates that better throughput may be achieved by space-sharing rather than time-sharing — in other words, assign each of the 4 jobs to 8 of the processors. In this manner, each job gets a dedicated set of processors, and all context-switching overheads are avoided. Also, with space-sharing comes another possibility. Suppose the available parallelism in one of the jobs decreases. In this case, assigning some processors to another job with excess available parallelism is better than letting the processors sit idly.

To address these scheduling issues, we have developed and implemented some new scheduling techniques in a prototype system. We refer to our scheduling techniques as *idle-initiated* because idle computers actively search out work to do rather than wait for work to be assigned. The idle-initiated scheduler works on both the macro and the micro levels.

The macro-level scheduler is responsible for assigning processors to parallel jobs and operates with the following goals:

1. Space-share rather than time-share.
2. Accommodate dynamically changing amounts of parallelism among jobs.
3. Allow owners to retain sovereignty over their machines.

These goals and the resulting macro-level, idle-initiated scheduler are all discussed in Section 2.

The micro-level scheduler is responsible for assigning the tasks that comprise a parallel job to the participating processors of that job. The micro-level scheduler operates with the following goals:

1. Preserve communications and memory locality.
2. Accommodate dynamic parallelism.

These goals and the resulting micro-level, idle-initiated scheduler are all discussed in Section 2 after the macro-level scheduler.

The idle-initiated scheduler is implemented in the *Phish* prototype system. Phish provides a simple programming model and an execution vehicle for parallel applications with unstructured and/or dynamic parallelism — the type of applications that are difficult to implement in the data-parallel or message-passing styles. The Phish runtime system allows applications to utilize a dynamically changing set of workstations. In particular, workstations, when left idle, may join an ongoing computation and then leave the computation when reclaimed by their owners. Also, workstations may join and leave an ongoing computation in response to the availability of parallelism within the computation. The Phish implementation operates with the following goals:

1. Execute large-scale parallel applications on large numbers of processors.
2. Utilize compute resources that would otherwise go idle.
3. Provide fault tolerance so that applications can run for long periods of time.
4. Do not interfere with the normal day-to-day use and management of the workstation network.
5. Achieve linear parallel speedup with only modest degradations in efficiency.

These goals and our first prototype implementation are all discussed in Section 3.

This prototype is currently operational at the MIT Laboratory for Computer Science, and we have run a couple of large-scale applications: a protein-folding application and a ray tracer. In Section 4 we report preliminary performance data from these and two other "toy" applications. These data demonstrate the high degree of efficiency delivered by the micro-level scheduler's ability to preserve communication and memory locality. For example, in an execution of the protein-folding application with 8 workstations, despite the over 10 million tasks executed, only 133 were ever migrated from one workstation to another, and yet the execution achieved almost perfect 8-fold speedup.

The remainder of this paper is organized as follows. Section 2 discusses the idle-initiated scheduler at both the macro and micro levels, and Section 3 discusses

the implementation of this scheduler in Phish. Preliminary application performance is presented in Section 4. This paper closes with a discussion of related work in Section 5 and conclusions in Section 6.

2 Idle-initiated scheduling

We have developed idle-initiated scheduling techniques in order to effectively schedule large-scale, dynamic parallel computations on a network of workstations. Idle-initiated scheduling operates on both the macro level and the micro level.

Macro-level scheduling

The role of the macro-level (or inter-application) scheduler is to determine which workstations are idle and to assign these idle workstations to parallel jobs.

Each workstation owner can set his or her own policy on "idleness" versus "busyness." For example, some owners may decide that their machines are idle — that is, available to be used for parallel jobs — only when nobody is logged in. Other owners may make their machines available so long as the CPU load is below some threshold. We believe that maintaining the owner's sovereignty is essential if we want owners to allow their machines to be used for parallel computation.

When a workstation becomes idle, it requests a job from a pool of parallel jobs managed by the macro-level scheduler. If parallel jobs are available, the scheduler assigns a job to the workstation. Note that when it assigns a job to a workstation, the scheduler keeps that job in its pool so that the job can also be assigned to other idle workstations. If no parallel job is available, the workstation continues requesting jobs until either a job becomes available or the workstation becomes busy. Once a workstation receives a job, it runs a participating process under the control of the micro-level scheduler described below until the process dies.

The participating process can die for several reasons. In the simplest case, the job may terminate. Second, the owner's idleness policy may determine that the workstation is no longer available for parallel computation. In this case, the process's data migrates before termination to another process of the same parallel job. Third, the amount of parallelism in the job may decrease to the point where a participant is unable to keep busy. As the parallelism in an application shrinks, some of its participating processes die, and the macro-level scheduler accommodates this time-varying parallelism by reassigning the freed workstations to other jobs. Finally, the macro-level scheduler may preempt the process due to scheduling priority. This preemption is the only case in which the macro-level scheduler performs time-sharing. Whenever possible, however, the macro-level scheduler shares compute resources among parallel jobs by space-sharing.

The preference for space-sharing over time-sharing is supported both by intuition and to some extent by empirical data. In the context of shared-memory multiprocessors, Tucker and Gupta [26] found that utilization and throughput are improved by space-sharing instead of time-sharing. They cite context-switch overhead as a significant factor. In the realm of message-passing parallel computing, Brewer and Kuszmaul [5] found another reason to avoid time-sharing. They found that achieving performance in message passing is critically tied to the rate at which messages can be received. When a process is swapped-out, it cannot receive messages — messages fill up available buffers and potentially clog the network.

Micro-level scheduling

On the micro level, each parallel job consists of a pool of discrete tasks. Tasks are dynamically created because the execution of a task can spawn new tasks. Furthermore, tasks can have synchronization requirements in that some tasks may need to wait for other tasks to be executed. The role of the micro-level (or intra-application) scheduler, then, is to assign tasks to participating processes.

Each participating process maintains its own list of *ready* tasks whose synchronization requirements have been met. For example, Figure 1(a) shows the state of a participating process's list of ready tasks. While the queue is not empty, the process works on ready tasks in a LIFO order. It works on tasks at the head of the list and inserts any newly spawned ready tasks at the head of the list. In Figure 1(b), the process has executed task D, which spawned tasks E, F, and G.

At some point, the participating process finishes executing all of its ready tasks. Of course, there may still be ready tasks that need to be executed in the lists of other participants. In order to get one of those ready tasks to work on, the participant without ready tasks becomes a *thief*. The thief chooses uniformly at random a *victim* participant, from which to steal a ready task. If the victim's list of ready tasks is not empty, it gives the thief the task at the tail of the list. Thus, stealing tasks is done in a FIFO manner. In Figure 1(c), the participant process has become a victim. A thief has stolen task A, which was at the tail. If, on the other hand, the victim's queue of ready

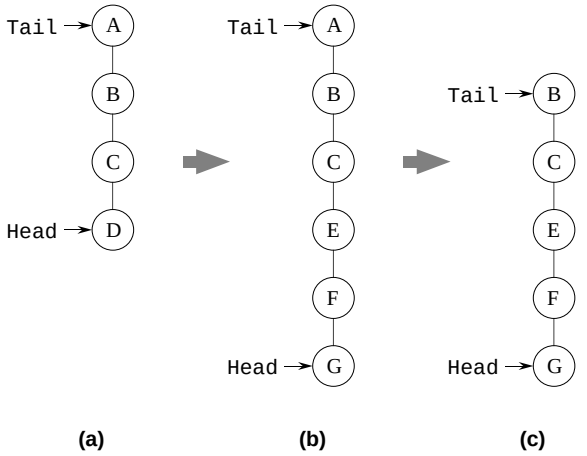

Figure 1: The local ready task list of a participant. **(a)** The queue contains four tasks: A, B, C, and D. **(b)** After executing task D which spawns three child tasks: E, F, and G. **(c)** After some other participant steals task A.

tasks is empty, the thief chooses another victim from which to steal a task. If no task can be found even after many attempted steals, the amount of parallelism in the job must have decreased. In response to this decrease in parallelism, the thief process terminates, and the terminated process's workstation goes back under the control of the macro-level scheduler to be assigned another job.

With CPU speeds increasing faster than network and memory speeds, the overhead incurred by network communication, cache misses, and page faults becomes ever more significant to application performance. Thus, preserving communication and memory locality is essential to performance. Our micro-level scheduler preserves communication and memory locality by working on tasks in LIFO order and stealing tasks in FIFO order.

This claim is supported by intuition, analytic results, and empirical data. Intuitively, executing tasks in LIFO order preserves memory locality by keeping the process's working set small, because whenever a task is executed, the next task to be executed is often closely related to the first task. Stealing in FIFO order has an intuitive payoff in preserving communication locality, because for computations with a tree-like structure, the task at the tail of the ready list is often a task near the base of the tree, and therefore, a task that will spawn many descendent tasks. Analytic results of Blumofe and Leiserson [2] show that for a large class of dynamic computations, the randomized work stealing strategy combined with LIFO execution order and FIFO steal order achieves linear speedup (with high probability) as well as tightly bounded communication and memory requirements. In Section 4, we present empirical evidence that our micro-level scheduler preserves both communication and memory locality.

3 Phish

Phish is a portable package for running dynamic parallel applications on a network of workstations. In this section, we present an overview of the Phish system. Because of space constraints, we are unable to present a detailed description of the entire Phish system. Rather, we focus on how Phish implements the idle-initiated scheduler.

At the macro-level scheduling, Phish consists of the *PhishJobQ* and the *PhishJobManager* as shown in Figure 2. The PhishJobQ, an RPC server, resides on one computer and manages the pool of parallel jobs. When a Phish application begins execution, it is submitted to the PhishJobQ. When an idle workstation requests a job, the PhishJobQ assigns one of its parallel jobs to the idle workstation. Our current implementation of the PhishJobQ uses a non-preemptive round-robin scheduling algorithm to assign jobs.

The PhishJobManager, a background daemon, resides on every workstation that is part of the Phish network and tries to obtain a job from the PhishJobQ when the workstation becomes idle. Our current implementation of the PhishJobManager uses a very conservative policy — a workstation is deemed idle only when no users are logged in. While users are logged in, the PhishJobManager checks every five minutes to see if they have logged out. As soon as the PhishJobManager discovers that its workstation is idle, it requests a job from the PhishJobQ. If the PhishJobQ responds negatively because the parallel job pool is empty, then the PhishJobManager continues to request a job every thirty seconds until it gets a job from the PhishJobQ. If the PhishJobQ responds positively by assigning a job, the PhishJobManager starts a *worker* process to participate in the job and waits for the worker to terminate. In the meantime, the PhishJobManager checks every two seconds to see if anyone has logged in. If the PhishJobManager discovers that the workstation is no longer idle, it terminates the worker process. Future implementations of Phish will provide opportunities for using and studying more sophisticated job assignment algorithms and different idleness policies.

At the micro-level scheduling, an executing Phish

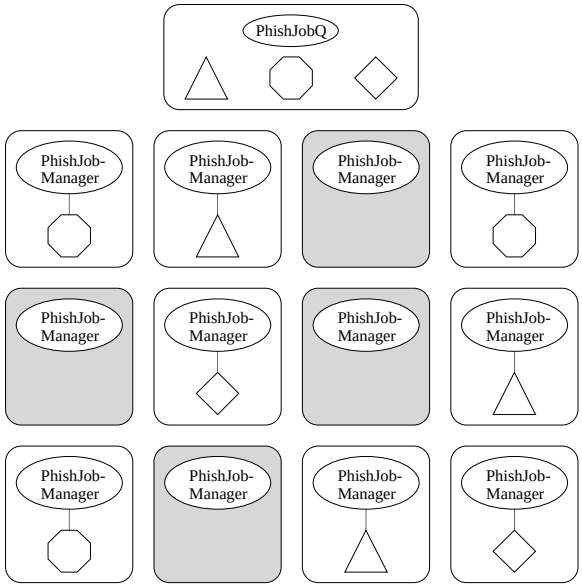

Figure 2: Parallel jobs are submitted to the PhishJobQ. Each workstation in the network runs the PhishJobManager, and when the PhishJobManager determines that its workstation is available to run a parallel job, it gets a job from the PhishJobQ. In this example, the PhishJobQ has 3 jobs, and each workstation with no user logged in (those not shaded) is participating in one of the jobs.

job consists of a *Clearinghouse* and one or more workers as illustrated in Figure 3. The Clearinghouse is a special program (independent of the particular application) that is responsible for keeping track of all worker processes participating in the job and providing various services to the workers. A worker is a participating process, which is an instance of the actual application program. When a worker starts, it registers with the Clearinghouse, and when a worker quits, it unregisters. Workers can find out about the other workers participating in the job by obtaining periodic updates from the Clearinghouse. Workers can perform I/O through the Clearinghouse, so a user need only watch the Clearinghouse to see job output. When a parallel job is started, a Clearinghouse must also be started. Often, a worker process is also started on the same workstation as the Clearinghouse.

The PhishJobQ and Clearinghouse represent potential bottlenecks to the scalability of the Phish system, but these concerns are largely mitigated by the coarse granularity of the services they provide. The PhishJobManager on a given workstation communicates with the PhishJobQ at most once every 30 seconds. Likewise, a worker process communicates with the Clearinghouse once to register, once to unregister, and once every 2 minutes to obtain an update. The only other communication between the Clearinghouse and its workers is for I/O which is buffered as much as possible. Although we have not empirically tested the system's scalability, we conjecture that Phish can be scaled to over a thousand workstations.

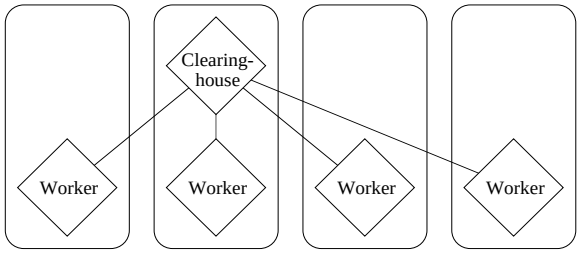

Figure 3: The Clearinghouse provides services to all the workers participating in a parallel job.

Phish applications are coded using a simple extension to the C programming language and a simple preprocessor that outputs native C embellished with calls to the Phish scheduling library. We support this programming model on both the CM-5 with the Strata [4, 13] scheduling library and on a network of workstations with Phish. More details of this programming model can be found in [13].

Once a program has been compiled and bound with the Phish library, a user can set off a flurry of parallel computation on workstations throughout the network by simply invoking the program on his or her workstation. For example, simply typing "`ray my-scene`" will run our parallel ray tracer on the data given in the file `my-scene`. By default, this simple command starts up the Clearinghouse and the first worker on the local workstation, so the computation begins right away. Also by default, it automatically submits the job to the PhishJobQ. Thus, as other workstations become idle, they automatically begin working on the ray-tracing job.

We conclude this section with a couple of comments about the Phish implementation. Since the round-trip latency of the network is very high, almost all communications are done with split-phase operations; that is, the runtime system almost always works while waiting for a reply message. In order to achieve split-phase communications, all communications are implemented on top of UDP/IP messages. Finally, Phish is fault tolerant. Enough redundant state is maintained so that lost work can be redone in the event of a machine crash.

4 Application performance

Currently, we have 2 toy applications and 2 real applications with more on the way. The toy applications are `fib` and `nqueens`. The `fib` application is a naive, doubly-recursive program that computes Fibonacci numbers. The `nqueens` application counts by backtrack search the number of ways of arranging n queens on an $n \times n$ chess board such that no queen can capture any other. The real applications are protein folding and ray tracing. The protein-folding application finds all possible foldings of a polymer into a lattice and computes a histogram of the energy values. This application was developed by Chris Joerg of the MIT Laboratory for Computer Science and Vijay Pande of the MIT Center for Material Sciences and Engineering. The ray-tracing application renders images by tracing light rays around a mathematical model of a scene. More details of both the ray tracer and the protein folder can be found in [13].

We begin with data measuring the *serial slowdown* incurred by parallel scheduling overhead. The serial slowdown of an application is measured as the ratio of the single-processor execution time of the parallel code to the execution time of the best serial implementation of the same algorithm. For example, if the serial slowdown is 3.0 and the best serial implementation runs in 10 seconds, then our parallel implementation runs in 30 seconds on one processor. Another way of looking at this number is to say that the parallel implementation needs 3 processors — assuming linear speedup — in order to break even. Serial slowdown arises due to the extra overhead that the parallel implementation incurs by packaging tasks so they can be run in parallel (as opposed to simple procedure calls in the serial implementation), scheduling the execution of these tasks, and polling the network for messages.

Serial slowdown data for 3 applications are given in Table 1. In general, we see that the serial slowdown incurred by Phish on our network of workstations is slightly greater than that suffered by Strata on the CM-5. Phish must work harder in its scheduling because it operates with a dynamic processor set while Strata operates with a static processor set.

For the individual applications, on one end of the spectrum, we see rather large serial slowdown for `fib`, and on the other end, we see almost no serial slowdown for `ray`. The `fib` application incurs serial slowdown because of its tiny grain size; it does almost nothing but spawn parallel tasks, which are simple procedure calls in the serial implementation. The fairly coarse grain size of the `ray` application incurs very little serial slowdown.

	`fib`	`nqueens`	`ray`
CM-5	4.44	1.09	1.00
SparcStation 10	5.90	1.12	1.04

Table 1: Serial slowdown measured for three applications on the CM-5 using the Strata scheduling library and on a SparcStation 10 using Phish.

Of course, if our applications are going to suffer any serial slowdown, there ought to be some parallel speedup forthcoming. Figure 4 shows the average execution time and Figure 5 shows the parallel speedup achieved by the protein-folding (`pfold`) application running on a network of SparcStation 1's using Phish. In general, measuring speedup with Phish is complicated by the fact that the computers participating in a computation do not start up at the same time. Therefore, even if they stay with the computation until the end and terminate at (very nearly) the same time, each participating computer runs for a different amount of time. Also, the participating computers may differ in computing power. To circumvent this heterogeneity, we did our measurements using only SparcStation 1's. To deal with participants running for different amounts of time, we attempted to start each participating computer at as close to the same time as possible. (Actually, starting all the participants at exactly the same time is impossible since each participant must begin by registering with the Clearinghouse) We then measured the speedup with P participants as the ratio of the execution time of the parallel implementation running with one participant to the average execution time of the P participants. Specifically, let T_1 denote the execution time of the parallel implementation running with one participant, and let $T_P(i)$ for $i = 1, 2, \ldots, P$ denote the execution time of the ith participant in a parallel execution with P participants. Then the P-processor speedup $\mathcal{S}_P$ is given by

$$\mathcal{S}_P = P \frac{T_1}{\sum_{i=1}^{P} T_P(i)} \ .$$

(To simplify presentation, this definition of speedup is slightly generous. When we perform an execution with P participants, rather than consider it as a P-processor execution, we should consider it a ρ-processor execution with ρ defined as the time average of the number of processors participating in the execution. In practice, this modified definition does not change our results by very much since we were able to start all the participants at reasonably close to the

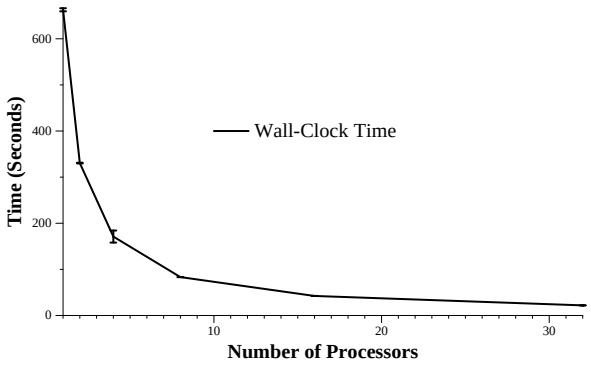

Figure 4: Average execution time of the Phish `pfold` application running on a network of SparcStation 1's versus the number of participants. The average execution time of a P-participant execution is given by $(\sum_{i=1}^{P} T_P(i))/P$ where $T_P(i)$ is the wall-clock execution time of the ith participant. When doing this experiment, we used idle workstations, so the wall-clock time is virtually the same as the sum of the user and system times given by the "rusage" system call.

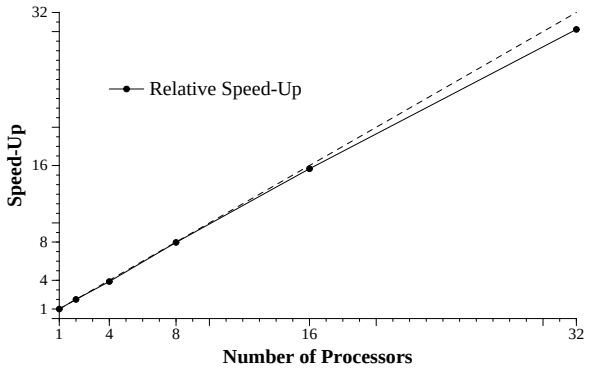

Figure 5: Speedup of the Phish `pfold` application running on a network of SparcStation 1's versus the number of participants. The P-participant speedup is computed as $\mathcal{S}_P = PT_1/\sum_{i=1}^{P} T_P(i)$, where $T_P(i)$ is the wall-clock execution time of the ith participant and T_1 is the wall-clock execution time of the parallel program with one participant. The dashed line represents perfect linear speedup.

same time.)

The graph of Figure 5 shows that we are getting close to perfectly linear speedup for the `pfold` application, even with 32 participants. With 32 participants, the execution time is getting short enough that some of the fixed overheads, especially registering with the Clearinghouse, are becoming significant. In fact, all 4 of our applications demonstrate similar speedups, but for lack of space we only present the `pfold` data. We realize that achieving linear speedup is often easy — just make the problem size large enough. Therefore, the serial slowdown data are far more significant in assessing Phish's performance. In particular, the fact that an application with a tiny grain size such as `fib` suffers less than a factor of 6 loss in efficiency (and achieves linear speedup), shows that the idle-initiated scheduler can effectively execute fine-grain parallel applications. This efficiency in the face of highly limited network performance comes from the idle-initiated scheduler's ability to preserve locality, as we now show.

Table 2 presents some data that indicate the extent to which our idle-initiated scheduler is able to preserve locality. These data present several message and scheduling statistics for a 4-participant and an 8-participant execution of the `pfold` application.

When comparing the number of tasks executed to the maximum tasks in use, which effectively measures the size of the largest working set of any participant, we see that even though more than 10 million tasks are executed, no participant ever has more than 59 tasks in use. Thus, our LIFO scheduler keeps the working set small, which is vital to achieving good performance because of the hierarchical memory organization of modern RISC workstations. Furthermore, increasing the number of participants does not increase the size of the working set. This property provably holds [2] for an algorithm that is only slightly different from the one we use.

	4 participants	8 participants
Tasks executed	10,390,216	10,390,216
Max tasks in use	59	59
Tasks stolen	70	133
Synchronizations	10,390,214	10,390,214
Non-local synchs	55	122
Messages sent	1,598	1,998
Execution time	182 sec.	94 sec.

Table 2: A variety of Phish message and scheduling statistics taken from a 4- and an 8-participant execution of the `pfold` application.

The remaining data in Table 2 show the extent to which the idle-initiated scheduler is able to avoid expensive network communication. Very few tasks need to be stolen. Also, an overwhelming majority of synchronizations are local and therefore do not require any network communication. Ultimately, very few messages are sent.

5 Related work

Much recent work in operating systems has gone to improving the utilization of workstation networks. To varying degrees, distributed operating systems such as V [9], Sprite [21], and Amoeba [25] view the workstation network as a pool of processors on which processes are transparently placed and migrated. This transparent process placement improves throughput and utilization by more evenly spreading the load across processors. Unlike Phish, which focuses on dynamic parallel applications, these systems are concerned primarily with static distributed applications.

Among systems that do directly address parallel computing on workstation networks, some focus primarily on message-passing. Of particular note in this category is PVM [22]. PVM provides a collection of message passing and coordination primitives that an application can use to orchestrate the operation of its various parallel components. PVM does not, however, provide much support for scheduling beyond a basic, static scheduler. In contrast, Phish provides a relatively high-level programming model that relieves the programmer of the need to schedule at the message-passing level.

The Parform [6] is a message-passing system with an emphasis on dynamic load balancing. The Parform employs load sensors to determine dynamically the relative load of the various machines that make up the Parform. This information is then used to divide and distribute the various parallel tasks. In contrast, the idle-initiated scheduler does not move a task unless an idle machine requests work.

The EcliPSe system [23] and the DIB system [11] employ workstation networks to run parallel applications from specific domains. EcliPSe performs stochastic simulation, and DIB performs backtrack search. DIB is of particular relevance to us, since backtrack search exhibits dynamic parallelism. In fact, DIB's scheduler inspired our idle-initiated scheduler. This type of scheduling technique actually goes back before DIB to MultiLisp [14] and has become known as *work stealing* [2].

Other systems address parallel computing on a network of workstations by maintaining shared global state. In the Ivy system [18], the global state is a paged virtual address space. Pages migrate between processors on demand while a protocol ensures the consistency of multiple copies of a page. As an alternate approach, the shared global state in systems such as Emerald [16], Amber [8], Amoeba/Orca [24] and Network Objects [1] is a collection of abstract data types or objects. Objects can be placed on and migrated between the network nodes. Operations can be invoked on an object no matter where the object is located, and protocols ensure the consistency of duplicate objects. These systems support varying degrees of concurrency and fault tolerance.

These systems with global-state are largely orthogonal to ours, because they focus on the efficient implementation of shared global state while mostly ignoring scheduling issues. We have taken the reverse tact by concentrating on scheduling issues without implementing any kind of shared global state. As a consequence, the current Phish implementation is somewhat limited in the types of applications that can be run. On the other hand, there are important applications that don't need a shared global state for which Phish delivers tremendous performance. In the future, we plan to add shared global state to Phish.

Linda [7] combines shared global state and scheduling issues into one simple paradigm: *generative communication*. The basic idea is that objects called "tuples" can be placed in, removed from, or simply read from a common "tuple-space." This simple notion turns out to be surprisingly expressive. Although no particular scheduling is actually built into Linda, our scheduling techniques — or any scheduling technique for that matter — could be implemented with Linda.

In fact, Piranha [12] is a system built on top of Linda with design goals very similar to those of Phish. (The fact that these systems share a piscene name is purely coincidental.) Like Phish, Piranha allows a parallel application to run on a set of workstations that may grow and shrink during the course of its execution. In particular, as workstations become idle, they may join an ongoing computation, and when reclaimed by their owners, workstations may leave a computation. Piranha's creators call this capability "adaptive parallelism." This capability is also present in the Benevolent Bandit Laboratory [10], a PC-based system. Phish also possesses this capability, and Phish's macro-level scheduler is very similar to these other systems. Phish's micro-level scheduler, however, is very different and works with the macro-level to give Phish the added capability of adapting parallelism to internal, as well as external, forces. In particular, Phish allows workstations to join and leave a computation in response to growing and shrinking levels of parallelism within the computation.

6 Future work and conclusions

With Phish and the Strata scheduling library both supporting the same programming model, we natu-

rally plan to run applications using both the workstation network and the CM-5 together. Also, we plan to give Phish capabilities to run applications over wide-area networks with heterogeneous network resources. Using the CM-5 together with a network of workstations actually fits into our plans for supporting heterogeneous networks because the CM-5 is essentially a network of workstations.

We are already working on some extension of our theoretical work-stealing results to incorporate network heterogeneity. The focus of this research is accommodating heterogeneous network capability as opposed to heterogeneous computer capability. Almost all microprocessors manufactured today are within a single order of magnitude of each other in terms of performance. Interconnection networks, on the other hand, have vastly differing capabilities. Our new scheduling techniques attempt to preserve locality with respect to those network cuts that have the least bandwidth.

Besides the future work just mentioned, we have several other planned extensions. These include new applications, support for checkpointing, a graphical interface, implementation of other macro-level scheduling policies, and support for globally shared data structures.

We conclude with the following points about Phish.

- Phish's macro-level scheduler allows workstations to join and leave an ongoing computation in response to the availability of idle cycles.
- Phish's macro-level scheduler cooperates with its micro-level scheduler to allow workstations to join and leave an ongoing computation in response to the availability of parallelism within the computation.
- Phish's micro-level scheduler delivers high performance to dynamic parallel applications by preserving memory and communication locality.
- Phish is highly fault-tolerant and therefore able to run large-scale applications for long periods of time with almost no administrative effort.

Acknowledgments

Phish was intentionally designed to do on a network of workstations what Strata does on the CM-5, and therefore, Phish owes its heritage to Strata. Eric Brewer developed the Strata communications library, and Mike Halbherr, Chris Joerg, and Yuli Zhou developed the Strata scheduling library. Phish also took some ideas from work done by Bradley Kuszmaul and Ryan Rifkin. Phish owes much of its intellectual inspiration to Charles Leiserson, Frans Kaashoek, and Bill Weihl; in fact, they may not know it, but it was a research proposal they authored that first inspired us to begin this project. And if Charles, Frans, and Bill are the intellectual grandparents of Phish (we claim parenthood for ourselves), then Udi Manber and Robert Halstead (whether they want it or not) are the intellectual great grandparents. Also on the topic of lineage, Phish is named after a band from Burlington, Vermont. Major appreciation goes to Eric Brewer and Mike Halbherr for their ongoing guidance in the design of Phish. We also owe a great debt to Scott Blomquist and David Jones for their help with many Unix and Sparc system-level issues.

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, North Carolina, December 1993.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, New Mexico, November 1994. To appear.

[3] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, Chicago, Illinois, April 1994.

[4] Eric A. Brewer and Robert Blumofe. Strata: A multi-layer communications library. Technical Report to appear, MIT Laboratory for Computer Science, January 1994. [Anonymous ftp: `ftp.lcs.mit.edu:/pub/supertech/strata`].

[5] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 858–867, Cancun, Mexico, April 1994.

[6] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[7] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[8] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona, December 1989.

[9] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[10] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[11] Raphael Finkel and Udi Manber. DIB — a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[12] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., July 1992.

[13] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994. To appear.

[14] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[15] James C. Hoe. Effective parallel computation on workstation cluster with user-level communication network. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1994.

[16] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133, February 1988.

[17] H. T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Francois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-based multicomputers: An emerging parallel architecture. In *Supercomputing '91*, pages 664–673, Albuquerque, New Mexico, November 1991.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[19] Kai Li, Richard Lipton, David DeWitt, and Jeffrey Naughton. SHRIMP (scalable high-performance really inexpensive multicomputer project). In *ARPA High Performance Computing Software PI Meeting*, San Diego, California, September 1993.

[20] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988.

[21] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[22] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[23] V. S. Sunderam and Vernon J. Rego. EcliPSe: A system for high performance concurrent simulation. *Software—Practice and Experience*, 21(11):1189–1219, November 1991.

[24] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.

[25] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[26] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, Arizona, December 1989.