Appeared in Proc. of the 2nd IEEE Conf. on Computer Workstations, March 1988, pp. 12-22.

The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster

Michael Stumm
University of Toronto
Toronto, Canada M5S 1A4

Abstract

This paper describes a decentralized scheduling facility designed for a large, heterogeneous, workstation-based distributed system. It assumes a system with existing facilities for remote execution and (possibly) task migration. The solution proposed is not only performance effective, but also robust against processor and communication failures, scalable to large systems with a high frequency of scheduling needs, and stable. It has been implemented on a large distributed system based on the V kernel.

This paper also exposes several scheduling requirements of parallel computations executing as task groups, where the individual tasks execute in parallel on different workstations. These requirement significantly complicate the scheduling process. We present a possible solution for effectively scheduling these task groups.

1 Introduction

The availability of remote execution facilities in distributed systems allows its users to offload their workstations by executing some tasks at remote sites and it allows users to effectively exploit remote resources. For example, a computationally intensive application will complete significantly faster on an idle host than on a busy one. Similarly, users on workstations based on older technology will observe a response time improved by an order of magnitude if their programs are executed on newer, faster workstations — for example, a text formatting job will run 3 times faster on a Sun-3 than on a Sun-2. Remote execution also allows the use of several workstations as a parallel machine with portions of a parallel computation executing in parallel on different workstations.

The availability of migration facilities adds an additional degree of flexibility. The system can more easily adapt to changes in the load and in the requirements of the users. For example, a user may want to eject all running tasks from his workstation in order to reboot it, or in order to run special applications.

Remote execution and migration facilities are available in a number of systems [7,12,20,18,22,21]. However, users generally need to explicitly indicate which programs should run remotely and when to migrate which programs (see for example [16]). This paper describes the design and implementation of a global, decentralized scheduling facility that automates this process. For every program that is started, it chooses an appropriate execution site, either local or remote, depending on the implemented policy. In addition, if migration is supported, the scheduling facility may decide to migrate a task (although this happens very infrequently in practice). Note that the facilities described here only decide on which host a task should run and do not implement scheduling (or dispatching) within a host. We assume that mechanisms for that purpose already exist. The solution we propose is fully distributed among a server process on each host and some application tasks in execution. It is designed to work in large, heterogeneous, workstation-based distributed systems. It has been implemented for the V system [2,6].

This paper is not theoretically oriented; we do not propose new scheduling algorithms, nor do we analyze their performance in any depth. Instead, we wish to focus on implementation and systems issues and rely on performance studies undertaken by others. Our solution had to satisfy several basic requirements:

- Scalability:
  It had to work in reasonably large systems with 50-500 workstations or more. We also wanted it to be capable of scheduling tasks at a relatively high frequency, at least so that every task could be globally scheduled before starting execution. Hence, our solution had to be capable of making quick scheduling decisions, with minimal overhead.

- Performance:
  It had to be able to improve performance noticeably by reducing the average response time.

- Heterogeneity:
  It had to work in a heterogeneous environment. It had to differentiate not only between hosts of different architectures, but also between compatible hosts running

at different speeds. We felt it should be possible to let everyone benefit from the newer and faster workstations in the cluster. (The most expensive portion of a workstation is its user interface. One of the goals of our facility was to be able to prolong the usefulness of older hardware, while providing the performance of the newer hardware.)

The scheduling facilities described can be implemented in any distributed system that supports remote execution where tasks run in a network-transparent environment. We assume the availability of (possibly unreliable) multicasting facilities[1] and that the cost of file access is equal across all hosts[2].

Further, we also assume that the target system is a cluster of workstations. This is important with respect to the policies we implemented, since such systems have a large amount of computational power that goes mostly unused. For example, our implementation site had over 70 workstations with an aggregate processing power of more than 150 MIPS, yet the average processor load was only about 20% during the busiest times. Only a fraction of the total number of hosts are used at any given time. We expect this low load level to become more pronounced as faster processors (10-80 MIPS) and multiprocessor workstations come to the market in the next few of years.

This paper is divided into three main sections. Section 2 presents and defends the policies chosen for our design and implementation to schedule single, independent tasks before they start executing. It discusses design issues and identifies a number of problems encountered during implementation and proposes solutions. Section 3 addresses migration and shows that a significant amount of extra overhead is involved in determining whether to migrate a task. Section 4 focuses on research in progress. It shows that some tasks that are part of a parallel computation executing as a task group may have special scheduling requirements. For example, we show that some of these tasks cannot share processors with other processes without significantly degrading the performance of the parallel computation. We then present a decentralized method for allocating hosts to accommodate tasks with these special requirements.

2 Scheduling Independent Tasks

In this section we first describe the scheduling policy we chose for scheduling independent tasks and then discuss several design issues, such as load characterization, choice of policy, file system issues, religion, etc. We also describe our implementation and its performance.

[1]Multicasts can be emulated by using broadcasts if multicasting facilities are not available.

[2]Since the disk seek time is the dominant factor in accessing a file page, it is generally better to have a few expensive, but fast disks available on the network than having a less expensive, but slower disk on each workstation [13].

2.1 Scheduling Based on Publishing State Information

The policy chosen for scheduling single, independent tasks is based on hosts publishing their state information: Every host constantly monitors its own state and multicasts it to all interested parties whenever it changes significantly. The state information of a host includes attributes that define the configuration (i.e. processor type, devices, coprocessors, etc.) and the expected utilization of its resources, such as processor and memory. Each host maintains a cache containing the state information of all other hosts. This cache is updated every time new information is received and is consulted every time a task is scheduled. If the system contains very many hosts, then the information of only the N "best" hosts needs to be cached.

A program is scheduled by first determining what files are available for executing that program. There may be several, each for a different configuration or host type. Also, any special requirements an executable file may have, such as minimal memory requirements, is determined. A set is then constructed, consisting of the M most lightly loaded hosts (as indicated by our cache) that are eligible execution sites, as determined by the requirements of the program or specified by the user. The load measure maintained for each host is appropriately scaled to reflect its processing power.

If the local workstation belongs to this set, then it is chosen, biasing the choice towards local execution. (This bias can artificially be made arbitrarily large.) Otherwise, workstations are randomly chosen from this set and probed to verify that the cached information is still valid (to within a certain degree of accuracy). If a probe identifies an inaccurate cache entry, then the cache is updated and the set is modified appropriately; otherwise that host is chosen for execution. (Our implementation indicates that the cache will be accurate enough so that a second probe will very rarely become necessary. The number of probes can therefore be limited to 2 or 3.) The randomness in the selection process helps avoid *scheduling clashes* where several hosts simultaneously select the same execution site for their programs.

This scheme uses a policy similar to the *Threshold* policy analyzed by Eager, Lasowska and Zahorjan [8], except that the decision whether to execute a program remotely is made based on approximate global information as opposed to the state of the local system alone. Also, instead of probing hosts completely at random, we attempt to make an intelligent guess as to which ones would perform best.

2.2 Load Characterizing Parameters

A key issue is how to measure the load of a host. Two important factors are the load on the processor and the load on the memory system as indicated by the amount of ongoing paging activity (or the amount of free memory on a system without demand-paged virtual memory). Although the processor load is more important with respect to its effects on response time, the effects of the memory load

should not be underestimated. Adding a new task can significantly increase the page swapping activity, thereby decreasing the performance of all tasks running on that host. However, we consider memory utilization to become a less important factor as memory sizes increase with decreasing memory costs in the future. (Another possible load factor, a host's networking activity, is not considered significant enough, especially since much of it is already captured by the processor load.)

At issue is how to characterize and measure processor load. Often a count of the processes assigned to a processor is implicitly used to measure the load of the processor [8,19]. Similarly, Ferrari proposes to use a combination of queue lengths[9]. Unfortunately, observation in our system indicate that this is not a good measure; we could find no correlation between individual queue lengths and processor utilization. The workload characteristics on workstations may differ from those of more traditional time-sharing systems.

A better way of measuring the load of a processor is to periodically poll it to determine if it is idle or busy, that is, measure processor utilization[3] directly. This can be done by having an *idle-process* (an eternal process always running at lowest priority) regularly increment a counter. This counter can be periodically polled to see what proportion of the processor is allocated to the idle-process. For systems with higher loads, a more accurate indication of processor load can be obtained by periodically polling the length of the queue of ready-to-run processes, but at a higher cost. Ferrari and Zhou conclude from a measurement study of load balancing that queue length averages will result in better load balancing for systems with higher loads (e.g. an average utilization of 50% and more) [10].

The processor utilization undergoes short term fluctuations that are largely irrelevant to the overall trend. Sudden short-term changes are due to external events and can happen frequently without having any significance to the average load. One typically applies a smoothing function to the values of the processor utilization to be able to identify significant changes in the overall trend. One inexpensive possibility is to use an exponential smoothing function:

$$AveUtil_t = (w \times AveUtil_{t-1}) + ((1-w) \times Util_t)$$

where the weight, $w \in (0,1)$, determines the amount of smoothing. In effect, this is a poor man's way of predicting the future behavior of the processor.

2.3 Publishing vs. Other Schemes

A number of other policies besides Publishing are also suitable for decentralized scheduling. For example, a Querying policy can be used to query the current load of hosts and schedule a new task to the most lightly loaded host that responds. Other policies do not make use of load information when choosing an execution site for a task. For example, a host is selected randomly with the Random policy and a Cyclical policy attempts to assign tasks to hosts in a round robin fashion.

[3]Note that processor utilization is important only because the system is heavily underutilized. If it were not, then the processor utilization would remain constant at about 100%.

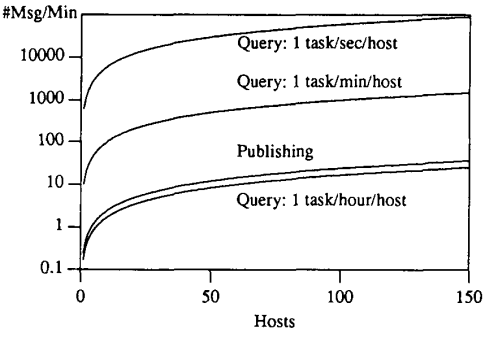

Figure 1: Messages per 100 Hosts.

All of these algorithms have been well studied using both analytical models and simulations [8,24,23]. We also analyzed these policies through simulations. They indicated that Publishing and Querying performed comparably under light to medium system loads and that the Querying policy performed slightly better under higher loads and for computationally intensive tasks. Moreover, it was found that the Random and Cyclical policies performed significantly worse.

The major reason for choosing a Publishing-based scheme over Querying was scalability. In Publishing-based schemes, the number of messages is a function of the number of hosts. The overhead can be tuned to an acceptable level. Our implementation indicated that approximately 20 messages per minute are generated per 100 hosts during busy periods a day, when hosts multicast their state information when the load changes by more than 10%. About four times as many messages are generated when state information is multicast with load changes of 5%. Our simulations indicated that the number of messages published would not increase significantly under higher system loads.

Although Querying has the advantage that the most current load information can be obtained, it delays the start of a task until responses are received from a query. Also, in Querying-based schemes the number of messages is a function of the number of tasks scheduled. Not all hosts have to queried each time a task is scheduled — for example, hosts can join a group depending on their load and clients can first query lightly loaded hosts [21] — but the number of messages generated in large systems can easily limit the number tasks that can be scheduled. Figure 1 compares the number of network messages in Publishing to the number of messages in Querying for different scheduling frequencies, assuming only 9 hosts receive and respond to an average query. (Note, however, that the messages due to Publishing are all broadcasts and hence interrupt each host.)

Finally, with Publishing, all packet events are evenly distributed across all hosts, whereas hosts that issue a query

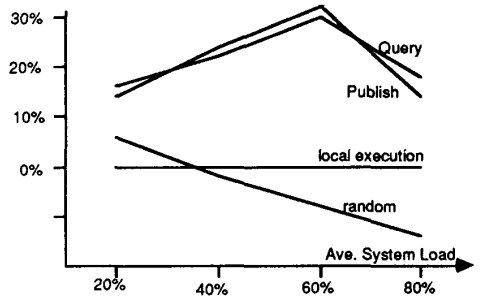

Figure 2: Average speedup.

incur most of the packet events themselves. Hence, precisely those hosts that want to offload some of their load will momentarily have an even higher load (due to packet processing) every time they need to schedule.

Our simulations were carried out by using a subset, i.e. 10 workstations, of the workstation cluster, by implementing the scheduling scheme under consideration and actually scheduling and executing tasks. However, the workload and tasks used were artificial, since the natural workload was not high enough. Other results of our simulations were:

- The scheme based on Querying performed slightly better than one based on Publishing at higher loads and for processor-intensive tasks.
- It is worthwhile to globally schedule tasks under all system load conditions, including very high and low ones.
- It is also worthwhile to schedule processor-unintensive (interactive) tasks.

Figure 2 contains a sample result of our simulation. It depicts the speedup obtained for scheduling in a system with 8 Sun-2's and 2 Sun-3's an average task that would run for 10 seconds, with 2 seconds CPU time, on a Sun-2.

Some studies indicate that centralized schemes may outperform decentralized ones [21,24]. They can reduce communication overhead, since less broadcasting is necessary and they can more easily prevent scheduling clashes. However, we never seriously considered them. We felt that centralized solutions could be (aesthetically) tolerated only if they significantly outperformed the decentralized schemes, which, according to our simulations, they did not. Also, they would not scale to systems with a very high scheduling frequency, without dedicating server hosts for this purpose.

2.4 Search Path Semantics

In a heterogeneous system, a simple command, such as `cat`, will actually refer to multiple executable files, possibly one for each host type and configuration. There are various approaches to representing multiple related files in a file system. Locus [22] uses the hidden directory approach, where a given file name actually refers to a directory (e.g. `/bin/cat`) containing the executable files: `m68k`, `vax`, etc. In V, a naming convention is followed where the host type is appended directly to the file name; i.e. `cat` exists as a set of *relatives*: `cat.m68k`, `cat.vax`, etc. The convention followed by V has several unfortunate consequences. For one, unless entries in a directory are maintained in sorted order (which they are usually not) then every entry in the directory must be read to obtain the set of *all* relatives of a given file. Since each program will now have several executable files, directories will become much larger.

A further disadvantage is that the search path semantics are confusing. For example, if we have a search path `"A B . C"`, and a user would like to execute file `foo` on an M680X0 and there is such a file in '.', his current directory, but there is a Vax executable file `foo.vax` in `A`, what does the user really want to execute, his file or the one in directory `A`? The search path semantics in Locus appear clear: The first hidden directory with the name `foo` is chosen and an attempt is made to execute a file from this directory. An error is signaled if an appropriate file cannot be found there.

We have chosen to interpret the search path similarly: An error is signaled if an appropriate file cannot be found in the first directory containing a relative of the program the user want to execute. However, we are still uncertain how much this interpretation will cause confusion to the casual user.

Note that both systems, V and Locus, use the ad-hoc technique of encoding information about files in the file name. Although this may be an efficient method for storing a small amount of information, such as host type specification, it is not suitable for storing larger amounts of information. Examples of information about files that may be of interest with respect to scheduling include: minimal, average or maximum memory requirements, that fact that it should be executed locally, expected execution time, expected processor intensity, etc. Several proposals for mechanisms to store information about files exist, including auxiliary pages [1] and property lists [14]. It remains to be seen how mechanisms of this type can be exploited for scheduling purposes. One would like to be able to quickly obtain a set of files together with associated properties of interest with minimal overhead, that is without having to go through the overhead of opening additional files.

2.5 The Communists and the Niggards

The introduction of remote execution (and migration) facilities invariably leads to an intensive religious debate between those members of the user community that view a workstation as a personal computer that should remain under the absolute control of the user, and those that view a cluster of workstations as a single machine, in effect, a loosely coupled multiprocessor.

For example, as the remote execution facilities were first introduced in the V system at Stanford, a lengthy de-

bate ensued whether a user should have to explicitly add his workstation to the processor pool or if he should should have to explicitly disallow others from using the workstation when he would like to do some "serious" computing. It took several months to convince the user community that

1. the number of idle workstations at any given time is relatively large so only rarely would a program be scheduled on a workstation on which someone was working,
2. a user would not notice if the workstation was also used for a foreign program, since his programs always ran at a higher priority,
3. foreign programs do not crash the workstation, and
4. that most people would "forget" to put their workstations in the processor pool if the default case disallowed remote execution.

Note that we have mainly described mechanisms for scheduling which can be used to realize many types of policies. For our own site, we realized an extreme policy, namely, that of viewing the workstation as a component of a loosely coupled multiprocessor and scheduling (almost) every task on a global basis. We believe this policy is reasonable, since typically a workstation cluster is paid for by an organization and hence its resources should be allocated so as to benefit the entire organization and not individual members. Also, it is in the interest of the average user to globally schedule tasks, since the average execution time decreases.

But we realize that with our policy, we ask the user to cross an additional psychological barrier in that not only is "his" workstation used transparently by others, he does not know where his tasks are executing. (He can determine this, however, if he so wishes and the "shell" can be switched into a verbose mode that always displays the execution site.) We still need to further educate users not to reboot their workstations without first migrating away all running tasks. Unfortunately, we still find it necessary to execute all programs that maintain a lot of non-reproducible state, such as editors, locally by default.

2.6 Implementation

We describe the implementation for the V-system [2,6] of our scheduling facility based on the Publishing policy. At the time of implementation, V did not cache file pages in memory [3] between program execution. Section 2.8 addresses scheduling in systems with such caching facilities.

The V-system consists of a distributed kernel and a distributed collection of server processes. Each host has a *program-manager* that provides program management for programs executing on that host and an *exec-server* that provides a shell-like service for interpreting command lines and starting programs for the user. The program-manager monitors the host on which it is executing and maintains the cache containing state information of all other hosts in the system. It makes this information available to all other local tasks.

The load of a host is measured by monitoring the CPU time allocated to the idle-process and is scaled by a factor representing the processing power of the host. Two exponential smoothing functions with different weights are applied to this value. One tracks long-term trends and is smoother (i.e. gives more weight to past values), while the other tracks short-term trends. The available memory is monitored in a similar fashion. The computed long-term values (together with configuration information) are multicast to all other program-managers, if they differ significantly from the values that were last multicast (i.e. by more than 10% of maximum value). The group communication facilities of V [5] are used for this purpose. Each host that receives the multicast message updates its cache appropriately. A newly started host can initialize its cache by obtaining a copy from any one of the other hosts.

To schedule a task, the exec-server (or any other program wishing to schedule a task) reads the cache from the program-manager and selects one of the hosts represented therein. Assuming a remote host is selected, remote execution proceeds basically as described in [20]. A probe is sent to that host by sending a load-request message to its program-manager, asking it to load and start executing the specified program. The load value used to select a host is included in the load-request message. If this value is much lower than the short-term load value, then the request is turned down, and if it is much lower than the long-term value, then besides turning down the request the requester is also asked to correct its cache entry.

Once started, the program is run in its own address space and is provided with a network-transparent execution environment. Except for the time needed for scheduling, the fact that a task may be executing remotely, is transparent to the user[4]. The difference between loading a task locally or remotely is equal to the difference between a local and remote message transaction (ca. 1.5 ms), since all files are loaded over the network. Hence, even small tasks can safely be scheduled remotely. Also, a user will not notice any sluggishness when interacting with remotely executing interactive programs, since all inter-process communication is based on message passing that differs only by a factor of two for the local and remote cases (ca. 1.5 ms vs. 3 ms). In fact, we were surprised to find users on slower workstations notice a significantly improved responsiveness in running interactive tasks, such as editors, remotely on newer, more powerful workstations.

Note that the multicasts used for distributing load information need not be reliable. Stale cache entries are detected and updated on use. Orphaned cache entries (those of crashed hosts) indicating light load are also detected on use: The first host to detect an orphan multicasts that information to all other program-managers.

Orphaned entries indicating heavier load are seldom detected on use. Hence, these entries must periodically

[4]Some programs have location-dependent semantics.

be found and disposed of. Unfortunately, it is not reasonable in larger systems to periodically validate each entry by querying the host in question, as this would cause considerable network traffic. One possibility is to include a permanent host identifier, such as a unique physical network address, with each entry in the cache. Each host then periodically checks for duplicate entries from one host. Alternatively, one can require each host to multicast its state information at least every so often (say, once an hour). Then, if the state information of a host is not updated for a lengthier period of time (e.g. several hours), that entry can be removed.

The most serious problem encountered in implementing these scheduling facilities was the overhead in determining the set of executable files for a given program and determining its scheduling requirements. Searching through directory structures along a given search path on a file server can lead to a large number of messages. For example, in our first implementation almost 100 messages were generated per binary file lookup. This can be alleviated, however, either by introducing a hashed bit-map indicating which executable files exist in a user's search path, or by caching directory entries locally.

2.7 Measurements

Measurement of our implementation indicate that for a system with 50 hosts, 10 state-information messages are broadcast to all hosts per minute during busy periods. This is with a normal, real workload with a load average of about 20%. Our simulations indicate that the number of broadcast message would increase to about 15 with a higher load average of 60%. Each host will service each message broadcast at a cost of approximately 1.5 to 2 milliseconds in processing overhead (depending on host type). In order to compute the load average, the program-manager must query the kernel for the time allocated to the idle process. It does so every 5 seconds at a cost of approximately 1 ms.

The scheduling facility described provides the mechanisms for implementing many kinds of policies. Our implementation supported the policy of scheduling every task that is started in the system, unless the user directs the "shell" to follow a different policy. Initial measurements indicate that the response time of large programs decreased by about 10% on average with this policy. This measure alone is not very meaningful, as it depends heavily on the type of hosts running at the time, the distribution of the arrival rate at the individual hosts, the distribution of the load, the type of tasks that are executed, etc. Because of the low load averages, most tasks execute locally. The benefit of remotely executing programs is due mostly to offloading tasks from the older, slower workstations to the newer faster ones. For example, to *TeX* a 20-page document on an otherwise unloaded Sun-2 requires approximately 240 seconds, but only 100 seconds on an otherwise unloaded Sun-3, 150 seconds on a Sun-3 that already has a load average of 50% (as measured by CPU utilization) and 200 seconds on a 75% loaded Sun-3.

2.8 File Caching

The scheduling system described does not account for the caching of file pages in the memory of individual hosts between program execution. At the time of implementation V did not support demand-paged virtual address spaces and every file had to be down-loaded in its entirety before starting execution. Some systems, however, provide some form of automatic caching of files. For example, many Unix systems allocate new pages on a least recently freed basis, allowing read-only pages to be reused (without having to go to a file server) if they can still be found in the free-list. It is also possible to implement caching RAM file systems on each host. More recently, virtual memory capabilities have been added to the V kernel, where large memories are exploited to cache file pages [3].

Clearly, a program will start executing more quickly if scheduled to execute on a host where the executable file is being cached, since access to a file server can be avoided. However, all file caching schemes rely on both spatial and temporal locality of file access and spatial locality is defeated by any scheduling mechanism that spreads the load across the system.

A number of strategies that allow the Publishing policy to better cope with caching memories appear to be worth exploring, something we plan to do in the future. First, the scheduler could remember the host that was last selected for remote execution and choose that host for remote execution again, if its load remains relatively light (as indicated by the cache of load information). Thus, assuming that certain programs are repeatedly run by a user or that certain files are repeatedly accessed by the programs run by a user, the probability of it being cached on a host selected by the scheduler is higher.

Second, it is possible to define a mapping from program names to a small subset of the hosts. When scheduling a program, an effort is made to execute it on one of the hosts to which its name maps to. A host outside of this set is chosen only if it clearly has a lighter load. This should increase not only the locality of file reference on a per-user basis, but on a per-system basis. (It should be possible to let the size of these sets vary dynamically, adapting to the relative load of each set.) However, this scheme only takes executable files into account and not data files.

Finally, the overhead of down-loading a file is a relevant factor only for short jobs. A policy that forces all short jobs to execute locally (possibly at a higher priority) would restrict remote execution to programs where file loading time is insignificant. However, this requires an a priori estimate of the execution times, as maintained, for example, by the file system. Also, this policy still defeats one of the purposes of caching file pages, namely, that of offloading file servers.

The existence of caching memorys may also require us to reevaluate the merits of Querying-based schemes. Each query could identify the executable file and the respondents could indicate to what extent they are caching the file (assuming the respondents can gather this information efficiently). The overhead of the query will offset the cost of

having to go to the file server.

3 Load Balancing through Task Migration

Migration facilities are absolutely necessary if tasks are executed remotely. A user must be able to eject all foreign tasks if he needs exclusive use of his workstation (or would like to take it down).

Migration can also be beneficial for load balancing purposes; a task will complete more quickly if it can be migrated to a host with a significantly lighter load. This may be necessary, for example, after a scheduling clash, when several hosts simultaneously select the same site for processor-bound tasks. The scheduling facilities described so far can be extended to also balance the load by migrating tasks from heavily loaded hosts to more lightly loaded hosts. The global state is periodically reevaluated for each task to decide whether to migrate it to another site. This is associated with more overhead, however, since when comparing the load of the local host with that of a remote one, the local load must be discounted by the proportion due to the process being considered for migration. Hence, the processor load inflicted by each candidate task must also be measured and monitored.

There are several complications associated with load balancing through task migration. First, there is a danger of instability where tasks migrate from one host to another. This can be solved by requiring the difference in load between the two hosts to be larger than a threshold and by requiring that a task may only migrate if its expected time to completion is large enough. Having these requirements has a number of additional other benefits. It decreases the number of tasks that must be monitored, since only few tasks have an expected remaining life time that is large enough. And it increases the probability that the overhead of migration will be offset by improved performance at the new site.

A second complication is that the measure of the load inflicted by a task must be compatible with the measure of the processor load for the arithmetic to be meaningful. Again, having a large enough threshold allows for (unavoidable) inaccuracies in these calculations.

Finally, the decision whether to migrate and to where to migrate cannot always be made by the "system" without additional information from the application. In some instances it is easier to let the application make the migration decisions. For example, some tasks may require efficient access to certain devices and should not be migrated at all or only to a certain subset of the hosts. Others may have residual dependencies [20] (e.g. a compilation that uses local temporary files) that should also be moved. As another example, some tasks function as watchdogs that watch other tasks for reliability reasons and therefore should never be migrated to the same host.

On the other hand, if the application makes the migration decisions then a separate thread of control is needed to periodically evaluate the load situation. It must also provide a means for the system to ask it to migrate away (e.g. when a user ejects all tasks) by, for instance, listening on a predetermined port.

In order to prevent the system module from having to manage a lot of application data, yet at the same time allow existing, unaltered programs to be migratable, both approaches need to be supported. This is possible as follows: By default, each task is migratable by the system, using the same constraints that were available for the initial scheduling. A task may optionally register itself as *non-migratable* or *self-migratable*. Tasks that do not register themselves are assumed to be freely migratable by the system (in our case, by the program-manager). A task registering itself as self-migratable must also issue a process (or port or procedure) identifier, to allow the system to issue a migration request, should the system want to initiate the migration.

In practice, we expect that most tasks will not need special considerations and will not register. We expect tasks that register themselves as non-migratable to be local servers or applications executed on special hosts.

4 Initial Thoughts on Scheduling Parallel Programs

4.1 Scheduling Requirements

A cluster of workstations can be used as a parallel machine to run parallel applications for speedup. Parallel *make* is a common example of such an application. More tightly coupled parallel algorithms have also been successfully run on workstation clusters [4,11]; for instance, a number of parallel optimization algorithms, such as branch-and-bound, are well suited to run in this environment. In many cases, parallel computations will need special scheduling support. Unfortunately, scheduling schemes developed for (shared memory) multiprocessors, such as those studied by Ousterhout [17] do not apply to more loosely coupled distributed systems since they schedule at too fine a granularity.

Scheduling requirements of parallel programs can vary from program to program, depending on their internal structure. Some parallel programs containing (relatively) independent tasks can be scheduled with the facilities described so far. For example, a distributed make can schedule its tasks independently (making sure that multiple tasks are not simultaneously assigned to the same host). Some parallel programs contain tasks that are well suited to pick up any available processing "slop". These programs are typically asynchronous and exhibit (very close to) linear speedup, and their tasks can also be scheduled using the described scheduling facilities. Parallel branch-and-bound and Monte Carlo simulations are examples of such programs.

However, classes of parallel programs also exist with more specific scheduling requirements. In this section, we identify two such classes and describe the scheduling requirements they have. We then propose appropriate

scheduling mechanisms. The two program classes we consider are synchronous parallel programs and asynchronous parallel programs with less than perfect speedup behavior. Both classes constitute a significant enough proportion of all parallel programs to warrant the introduction of appropriate support.

Synchronous parallel programs execute in lock-step, synchronizing at the end of each iteration. Algorithms for dynamic programming or Gaussian elimination, for example, are often structured in this way. Tasks belonging to such synchronous computations, should not share the processor assigned to them with any other tasks; that is, it should be possible to allocate entire processors to the individual tasks. The reasoning behind this is that even slight variations in the processing power made available to a member task can significantly decrease the performance of the entire computation, since all member tasks must periodically wait for the slowest task to complete its iteration and a single, slowly running task can prevent the computation from attaining the expected speedup. For example, consider a parallel synchronous computation (with perfect speedup behavior) executing as 5 tasks, where 4 of them have exclusive use of the processor they are executing on, but one of them has only access to 70% of its processor's capacity. This computation will complete more quickly if executed using 4 tasks if they can run alone on the processors to which they are assigned.

On the other hand, tasks of asynchronous parallel computation with less than perfect speedup behavior can be scheduled to run with other tasks as long as these tasks are not part of another parallel computation. The reasoning behind this is that the speedup curve of almost all parallel computations is concave[5] (i.e. the efficiency of a parallel computation cannot increase with the number of processors), so the efficiency of two computations suffer when they both increase their degree of parallelism but start to share processors. Consider, for example, a parallel matrix multiplication of two 48×48 matrices with the speedup as obtained on a cluster of Sun-2's running the V kernel and depicted in figure 3. The speedup with 5 processors (4.5) will always be greater than or equal to half the speedup with 10 processors (3.75). Therefore, if, for example, we wish to run two such computations at the same time, then it is more efficient (the computations will complete faster on average) if each computation uses 5 separate processors, rather than if each computation time-shared the same 10 processors. The two computations will also complete slower on average if 10 processors first run the tasks of the first computation followed by those of the second computation.[6]

Hence, we identify two scheduling requirements:

[5] Programs with anomalous behavior exist.

[6] Using the same line of reasoning, one can also argue that parallel computations should use the most powerful processors in the system, since a parallel computation will always complete faster when using two fast processors than if run on 6 processors one third the power of the fast ones. This may, however, contradict scheduling policies concerning interactive jobs.

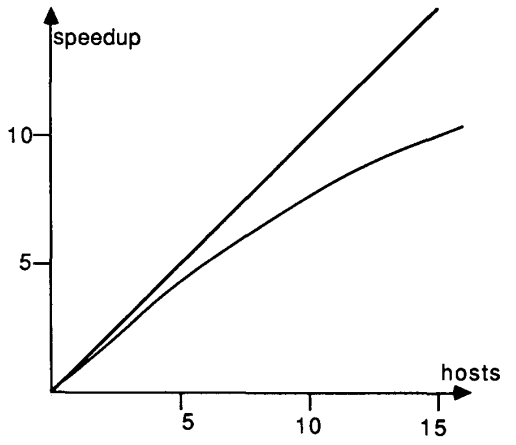

Figure 3: Matrix multiplication speedup on a workstation cluster.

1. A task of a synchronous parallel computations should not share a processor with any other tasks.

2. A task of an asynchronous parallel computation should not share a processor with a task belonging to any other asynchronous parallel computation.

Section 4.2 shows how the scheduling facilities described previously can be modified slightly to provide the necessary support to meet these two requirements.

Both of these requirements make the scheduling of parallel programs, executing as task groups, very difficult. Either the number of processors a program may use is restricted to a small number in anticipation that other programs will also run. Or, alternatively, one may allow a program to "grab" as many hosts as it wishes. However, the first case will limit some programs from fully exploiting the available processing power and the second case may block out other parallel programs, since each host may run at most one task belonging to a parallel application.

A third alternative is to require each program to be capable of dynamically adjusting the number of hosts it is using. This would allow the implementation of a policy that is fair, but at the same time allow programs to exploit as much parallelism as is currently available. This approach requires parallel computations to be capable of internally restructuring and redistributing their workload to be able to adapt to changes in the number of processors available to them. Clearly, the overhead for doing this can be significant if the structure of the parallel program is not designed from the start to allow for such reorganizations. Having self-adjustable parallel programs may not be as difficult as it may sound at first. Cheriton and Stumm [4] describe a model for structuring parallel computations for a workstation environment that is capable of running while the number of processors vary. Many parallel programs, such as simulations described by Nicol and Reynolds [15], must be

capable of dynamically changing the distribution of computational work to adapt to changing processing requirements within its (internal) problem domain. The methodology used in these programs can be generalized for releasing and adding processors to a parallel computation.

In section 4.3 we describes how task groups can compete for processors distributing the processing power evenly among competing groups.

4.2 Exclusive Processor Allocation

The scheduling facilities described in Section 2 can be used, with only minor modifications, to implement a policy where at most one task belonging to a parallel computation may run on a host at one time. As soon as one of these tasks is started, it registers itself as such with the local program-manager. The fact that a processor is running a registered task becomes part of the host state information that is disseminated to all other hosts. The task responsible for starting up a parallel program can schedule its tasks by consulting the local cache of state information, making sure that at most one task belonging to a parallel computation is run per host. (Note that it is in the interest of these tasks to register themselves, so enforcing mechanisms are not necessary.)

Similarly, synchronous tasks, requiring exclusive use of a processor, register themselves as synchronous when they start up. When this happens, the program-manager evicts all other locally running tasks by migrating them to other hosts. The fact that a host is executing a synchronous task also becomes part of the state information that is published. This ensures that no other task is scheduled to run on a host already running a synchronous task.

If migration is not available, then it is possible to always keep a few hosts in reserve for executing synchronous tasks. That is, the set of all hosts can be partitioned into two disjoint subsets, one reserved for executing tasks belonging to synchronous computations and the other for all other tasks. Keeping a number of hosts idle, in reserve for synchronous tasks, is tolerable in large systems, due to the assumed overabundance of processing power. (In order not to waste these cycles completely, short tasks[7] can still be allowed to run on the reserved, but not yet allocated hosts. These short tasks would always be allowed to run to completion.)

The number of hosts reserved for executing synchronous tasks should not be static, but should vary dynamically, adapting to the current state of the system (i.e. the relative number of reserved hosts, the average load on the non-reserved hosts, etc.). A dynamic partitioning can be implemented in a decentralized fashion if the two bits added to the state information associated with each host are interpreted such that one bit identifies a host as being reserved for executing synchronous tasks and the second bit indicating if such a task is currently executing. Since the state information of each host is published whenever one of these bits change, every host can independently and periodically reevaluate the global and local state to determine if it should change sides and become a member of the other subset. (But one must be careful to ensure that hosts do not oscillate between the two subsets.)

[7]The definition of short here is relative and taken to mean not more than a few minutes. But it does require an a priori estimate of a task's execution time.

4.3 Task Groups Competing for Resources

By allowing each parallel application to exploit as much parallelism as it can and wishes, it is possible that a new program will want more processors than are immediately available. In this case, it may be necessary to force other parallel programs to "relinquish" some of the processors on which they are running tasks, in order to be able to fairly distribute the hosts among all parallel programs. In theory, a system server, such as the program-manager, could take on the responsibility for doing this. In our case, however, the "system" does not know which tasks belong to which parallel computations, making it difficult for the system to take on this responsibility. Hence, we let each parallel application schedule its own tasks and let the parallel applications negotiate for processors directly with each other. We define a simple protocol that defines the interactions between negotiating task groups.

We assume that each parallel program in execution is being controlled by a master process (that belongs to the parallel application). Each such master process joins a well known group (so that it will receive messages addressed to this group) and is responsible for negotiating with other master modules and for restructuring the computation if required to do so. If an application wants to use more processors than are currently available, then it sends a query message to the group of master processes to determine their identity and the number of tasks in their task group. Using this information, it may request some of these computations to release processors. It must do this in a way that distributes the processors more evenly among the task groups. A parallel computation must reduce the number of tasks if asked to do so. The master process of a parallel computation may also periodically reevaluate the global state to determine if it should expand, to adapt to changes in the global state.

Hence, in this scheme, task groups compete for processors. If each parallel application attempts to maximize its number of parallel executing tasks, then each task group will be approximately of equal size. It is possible to refine this scheme by having the master modules specify the speedup characteristics during the negotiation process, to allow computations with better speedup to have more processors. It would also be straightforward to modify the scheme to accommodate task group priorities. In practice, we expect very little competition between task groups because of the large number of available processors and the fact that most parallel applications can make effective use of only a small number of processors. Moreover, parallel computations tend to run for a long time.

There is a tradeoff in the choice of how often task groups reevaluate the global state and subsequently self-regulate. If done often, the groups will quickly adapt to changes in the system, but at the cost of additional overhead and possible instability. In our case, a slow regulation process is tolerable, since significant changes in the overall global state are very infrequent and only few parallel programs are ever run.

Tasks belonging to parallel computations compete not only among each other; the available processing cycles must be divided among normal, sequential tasks and those belonging to parallel applications requiring special consideration. In the previous subsection, we described how processing cycles are assigned on a per processor basis to synchronously executing tasks. The situation with tasks belonging to asynchronous parallel computations is more complicated, however, since they can run on a processor together with sequential tasks. A spectrum of policies that regulate the priority of sequential tasks over those of parallel computations can be implemented by modifying the definition of a processor's load to also include a portion attributed to the execution of the parallel task. The definition of processor load (see section 2.2) is modified to be equal to the weighted sum of processor utilization consumed by "normal" tasks and processor utilization consumed by a task belonging to a parallel computation. The weights can be chosen appropriately to implement a desired policy.

5 Conclusions

We described the design of a decentralized global scheduling facility in the context of a large, workstation-based distributed system that can remotely execute and (possibly) migrate tasks. The solution proposed is fully decentralized and is

- robust against processor and communication failures, including network partitions;
- stable in that tasks do not continuously migrate from host to host;
- performance effective in that overall the average response time decreases;
- scalable to many hosts and capable of scheduling many tasks at a high frequency;
- designed to accommodate heterogeneous hosts.

In order to validate our claim of scalability, we implemented these facilities on a cluster of 70 workstations, running the V kernel, with a policy of scheduling every task, including small ones, on a global basis. Our solution or implementation can be seen as a further step in the direction of system integration, where a workstation cluster operates as a *single* system, rather than a set of interconnected personal and server computers. We believe that this step is in the interest of the users, because of the reduced response times, and in the interest of the organization that paid for the system, since the system resources are used more effectively.

Finally, we considered the scheduling of parallel programs executing as task groups in a workstation environment and showed that some require that their tasks have exclusive use of the processor and that in general processor intensive tasks that are part of a parallel computation should not share processors. We presented a protocol for scheduling competing task groups in an underutilized, relative infrequently changing environment.

With respect to migration and the scheduling of task groups representing parallel computations, we found that we had to perform some scheduling functions in the applications instead of in system modules. The information and algorithms needed to decide what actions to take were too complex and application dependent to be abstracted into a simplified, generic model. However, with applications performing system functions, the need to access system state efficiently is crucial and improvements to this effect are necessary. For example, suitable address mapping techniques could be exploited for this one-directional information transfer at memory speeds, reducing the number of system calls and messages. We also found that more general and more efficient querying facilities for file systems are needed.

In conclusion, we view the scheduling of tasks and task groups as an important facility of workstation-based distributed systems. With the increasing prevalence of powerful uni- and multi-processor workstations in the computing environments of most organizations, the user transparently has at his disposal computational power far in excess of that provided by a single personal workstation. The computational power is in a parallel form that is transparently exploited at the command execution level and that can also, with suitable support, be explicitly exploited by computationally intensive programs.

Acknowledgments

Many contributed to this work. P. Brundrett helped implement and find bugs. S. Shi performed the simulations. Discusions with members of the Distributed Systems Group at Stanford and S. Zhou helped improve the design and helped improve this paper.

References

[1] *Alto Operating System Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[2] E.J. Berglund. An introduction to the V system. *IEEE Micro*, 6(4), 1986.

[3] D. R. Cheriton. Effective Use of Large RAM Diskless Workstations with the V Virtual Memory System. 1987. Working paper.

[4] D.R. Cheriton and M. Stumm. The Multi-Satellite Star: Structuring Parallel Computations for a Workstation Cluster. *Distributed Computing*, to appear 1988.

[5] D.R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. on Computer Systems*, 3(2), 1985.

[6] D.R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proc. 9th ACM Symp on Operating System Principles*, 1983. Appeared in Operating System Review 17(5).

[7] F. Douglis. *Process Migration in the Sprite Operating System.* Technical Report UCB/CSD 87/343, Computer Science Division (EECS), University of California, Berkeley, California 94720, 1987.

[8] D. Eager, E. Lazowska, and J. Zahorjan. Dynamic Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. Soft. Eng.*, SE-12(5):662–675, 1986.

[9] D. Ferrari. *A Study of Load Indices for Load Balancing Schemes.* Technical Report UCB/CSD 85/262, Computer Science Division (EECS), University of California, Berkeley, California, 1985.

[10] D. Ferrari and S. Zhou. *An Emperical Investigation of Load Indices for Load Balancing Applications.* Technical Report UCB/CSD 87/353, Computer Science Division (EECS), University of California, Berkeley, California 94720, 1987.

[11] R. Finkel and U. Manber. DIB — A Distributed Implementations of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

[12] R. Hagmann. Process Server: Sharing Processing Power in a Workstation Environment. In *Proc. Principles of Distributed Computing*, 1986.

[13] E.D. Lazowska, J. Zahorjan, D.R. Cheriton, and W. Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Trans. on Computer Systems*, 4(3), 1986.

[14] J.C. Mogul. *Representing Information about Files.* PhD thesis, Stanford University, Stanford, California 94305, 1986.

[15] D.M. Nicol and P.F. Reynolds. *The Automated Partitioning of Simulations for Parallel Execution.* Technical Report TR-85-15, Dept. of Vomputer Science, University of Virginia, 1985.

[16] D. Nicols. Using Idle Wrokstations in a Shared Computing Environment. In *Proc. 11th ACM Symp on Operating System Principles*, 1987. Appeared in Operating System Review 21(5).

[17] J. Ousterhout. *Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.* PhD thesis, Carnegie-Mellon University, April 1980.

[18] M.L. Powel and B.P. Miller. Process Migration in DEMOS/MP. In *Proc. 9th ACM Symp on Operating System Principles*, 1983. Appeared in Operating System Review 17(5).

[19] J.A. Stankovic. Simulations of three Adaptive Decentralized Controlled, Job Scheduling Algorithms. *Computer Networks*, 8(3), 1984.

[20] M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptive Remote Execution Facilities for the V-System. In *Proc. 10th ACM Symp. on Operating System Principles*, 1985. Appeared in ACM Operating System Review 19(5).

[21] M.M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, Stanford, California 94305, 1986.

[22] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proc. 9th ACM Symp on Operating System Principles*, 1983. Appeared in Operating System Review 17(5).

[23] Y. Wang and R. Morris. Load Balancing in Distributed Systems. *IEEE Trans. Comp.*, C-34(3), 1985.

[24] S. Zhou. *A Trace-Driven Simulation of Dynamic Load Balancing.* Technical Report UCB/CSD 87/305, Computer Science Division (EECS), University of California, Berkeley, California 94720, 1986.