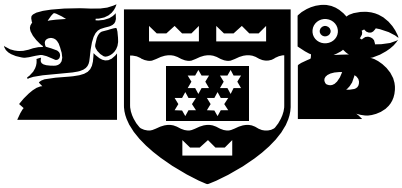
VICTORIA UNIVERSITY OF WELLINGTON



Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Load Sharing in a Distributed Environment

Andy Bond

CS-TR-92/1
October 1992

Abstract

A network of UNIX [Ritchie, 1974] workstations is becoming a common sight in modern computing environments. Each workstation provides powerful computing resources which are periodically in strong demand by the local user. However even in busy environments, a significant proportion of these machines will be idle or underutilized at any one time. By supplementing the local computing resources through offloading tasks to idle workstations, better utilization can be made of the distributed computing environment. Such a strategy is commonly known as **load sharing** or **load balancing**. This paper discusses an experimental task allocation system called STARS. Using a distributed database of resource usage measurements, STARS allocates tasks to systems based on the availability of resources within the network. We will look at how such a system can be integrated into a users computing environment and some results based on our experience with its use.

Publishing Information

This report appeared in the *Proceedings of UniForum NZ '92*, New Plymouth, 14-16 May 1992.

Introduction

Today's research computing environments often comprise a variety of workstations designed to provide computing resources to their local user. These resources include CPU processing, IO bandwidth, memory, and network access. While this local computing capacity manages to fulfil the requirements of most users, there are users who require either more or less of their local computing capacity. For those on the lower end of the scale, their workstations can abound with idle computing resources which are going to waste. In contrast, there are those users who could quite easily soak up any extra computing resources if a simple means of access was available.

The interconnection technology that is now common place in the UNIX environment provides users with just such a simple means of using extra remote computing resources. Combined with a shared file system, the user is able to perform tasks remotely with little more effort than is required to compute locally. The allocation of spare computing resources to those users requiring more computing capacity is provided by a "load sharing" environment. By striving to balance the computing load on our network of workstations, we share the extra computing requirements of some users amongst the remaining workstations with spare capacity. However we must ensure that no local workstation user loses access to required local resources by a task allocation from some external workstation. This situation can lead to a load sharing policy being ostracized by the local computing community.

This paper looks at load sharing mechanisms and how they integrate into a user community. Any such service should pose little over head on the general computing environment and make access to remote computing resources as transparent as possible. Initially, we shall look at some traits common to load sharing mechanisms including how such facilities are provided. The following sections will look at two implementations of load sharing mechanisms. CMU's Butler provides an off-the-shelf set of programs that allow users to gain access to idle workstations. This is a good example of how a simple load sharing mechanism can be eased into a standard UNIX computing environment. Condor is the second example we will look at. It is somewhat more complex than Butler as it provides task mobility features that transparently move tasks amongst idle workstations until the task completes. This provides a useful comparison between a "strap on" task sharing paradigm and one that requires modification to the computing environment (i.e. the kernel).

The Load Sharing Paradigm

Load sharing in a UNIX environment seems to be something that should be there by default. As we have noted, the simple remote execution mechanism and the now common shared file system seem to beg the load sharing question. Why are such mechanisms still only provided in research environments? Probably because commercial organizations have not yet entered into workstation environments with force. However, the load sharing paradigm is applicable to any environment employing a number of UNIX hosts not just those with a one-to-one assignment of workstations to users.

There is extensive motivation for the use of load sharing mechanisms apart from simple implementation. In the Computer Science Department at Victoria University, we have seen that even at the busiest of times more than 50% of workstations are idle (see FIGURE 1 on page 2). As we would expect, the use of the computing environment is busiest during the working hours with a steady drop off as night approaches. By ranking each host out of 100 (where 100 is no users with little load and 0 is many users with a high load), we can see from FIGURE 2 on page 3 the expected daily trend in workload. The load increases as the workday commences and slows down as the 5pm approaches[1]. With such extensive idle computing resources available consistently over the day, it seems fateful that load sharing mechanisms will emerge from the woodwork.

The problem of optimally allocating a set of tasks to a set of hosts while minimizing response time is NP-complete[2]. There are several successful allocation algorithms [Merkenscoff and Liaw, 1986] which approximate solutions to the problem but in real life we usually try and share load while disregarding the need for any strict

1. Various glitches in the curve indicate periods where coordinated system activity takes place (such as file system checks at 3am).
2. An NP-complete problem is hard.

FIGURE 1 Percentage of active workstations over an eight week period.

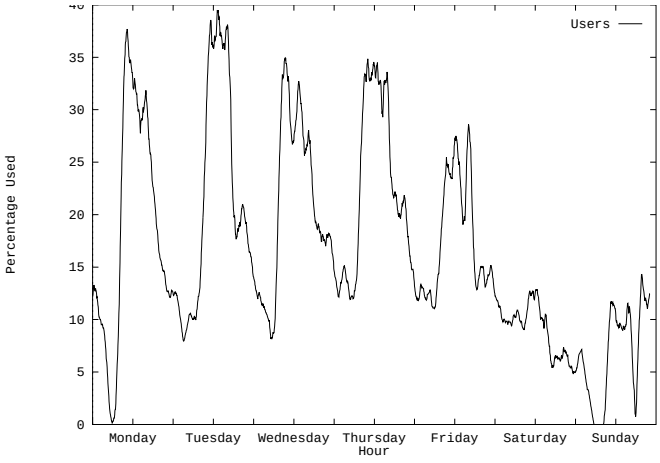
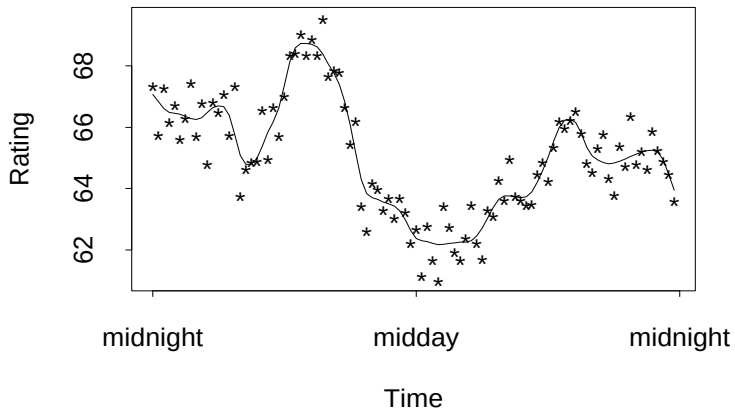


FIGURE 2 Average host ratings over the 24 hour weekday period for 6 weeks.



minimization. Our solution will attempt to reduce response time by automatically and transparently assigning tasks to processors. At the extremes, this can be as optimistic as assigning a task to a number of idle workstations or as pessimistic as being confronted with a set of very busy machines. Most allocation situations will occur somewhere within these extremes but at all times we should ensure that our mechanism provides no worse response time than would be present without the intervention of load sharing.

Load sharing requires some criteria on which the allocation of tasks to hosts must be based. Some load sharing mechanisms are based on **static** allocation techniques, i.e. separate the tasks so all tasks of type α get sent to host β, etc. Essentially we are dividing the host set into a collection of task servers. A **dynamic** load sharing mechanism bases the allocation on some changing criteria using current system performance as a metric. Commonly, this metric turns out to be something like the load average. The next job to be allocated is sent to the host with the lowest load average. In addition, we can introduce another dimension into the equation by using a different allocation policy for different types of jobs. A system that changes the load sharing policy may be thought of as **adaptive**.

Sometimes the allocation of a task to a host turns out to be less than desirable. The system load may increase due to some unforeseen local task or the allocation mechanism may make a misinformed allocation choice. An allocation mechanism may have the ability to pre-empt a task and migrate it to a more suitable site. Instead of load sharing, we now have an optimal method for load balancing at the extra cost of task migration. This usually requires modifications to the UNIX kernel or the application to support such moves [Freedman, 1991]. Without a migration mechanism, the load sharing system may choose to terminate any offending allocated tasks and restart them elsewhere or simply ignore the situation and hope it goes away. The system implementations described later include examples of each of these choices.

A dynamic load sharing method needs access to some sort of performance data from all the hosts that are potential remote execution sites. By simple extension, each host cooperating in the load sharing paradigm must both have access to load information from each other host as well as provide load data to these hosts. As is common in most distributed systems, this information may be provided by a central server or may be distributed amongst all contributing hosts. If such servers are also responsible for allocation decisions, a central server approach will ensure that global allocation decisions are available at minimal cost (i.e. no messages need be passed between cooperating hosts). However should the central server be unavailable, no load sharing is possible. In contrast, a distributed approach benefits from increased reliability as no single node failure will restrict load sharing at anywhere other than the point of failure. This reliability comes at an increased cost of communication between the cooperating server sites.

Load sharing mechanisms can come in many forms. The decision criteria or policy for making the load sharing decisions can range from static table driven to dynamic with adaptive task dependent decision criteria. By introducing a pre-emption and migration paradigm we are able to turn our load sharing tool into an optimal load balancing mechanism. However this usually requires reduced integration flexibility (through kernel or application modifications) and additional migration costs. The reliability of any distributed mechanism is only as good as it's weakest link. A distributed server approach decreases the failure probability by increasing the required points of failure for total system loss. In contrast, the more traditional centralized server approach reduces communication overhead but at the same time decreases total load sharing reliability.

A Butler for Idle Workstations

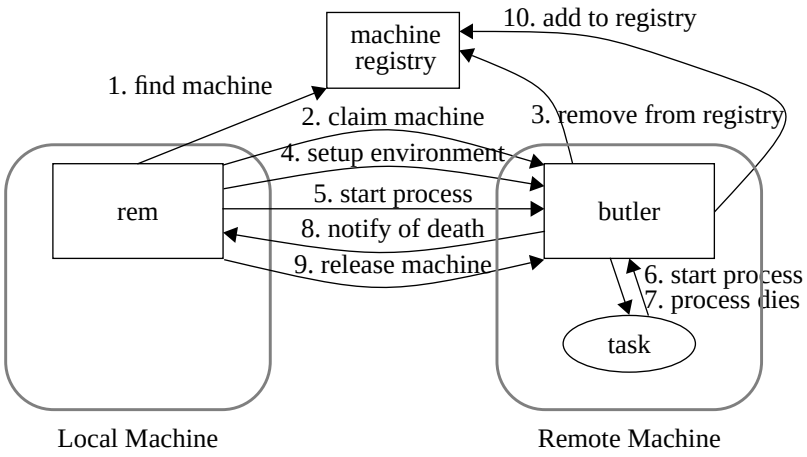
Butler [Nichols, 1987] is a system for using idle workstations that was developed at CMU. It requires some form of networked file system (such as NFS [Lyon et al., 1985]) and a windowing environment (like X Windows [Scheifler and Gettys, 1986]). The system is completely detached from the kernel and runs off-the-shelf with neither kernel nor application modifications. Butler is only used for the allocation of idle machines and does not interest itself in spare capacity on underutilized workstations.

FIGURE 3 on page 4 shows a diagram of the three mechanisms making up the Butler system and how they interact in the invocation of a process.

A pool of idle hosts is maintained in a global machine registry. By running a Butler server on a workstation, the workstation is added or removed from this machine registry as remote jobs are executed on the host. Originally, the machine registry was simply implemented as a shared directory where the Butler servers added or removed file entries from the directory. Later as this became a bottleneck (with up to 350 workstations accessing it), the machine registry was cached in registry servers that migrate amongst idle hosts (gypsy servers).

To invoke a process using Butler, a local Butler client (`rem`) is run requesting an idle host. This client contacts a machine registry server and then attempts to contact the Butler server on the provided host. If the idle host has already been allocated or the machine has been reclaimed by it's local user then the client tries for another idle host from the machine registry server. Upon successful allocation, the remote Butler removes itself from the machine registry and exchanges execution information with the client. If the local user returns while the host executes the requested process, the client is informed of the process's impending termination and a short time later the process is terminated. No attempt is made at either restarting nor migrating the process. Following pre-emption and the user once again logging out, a Butler server is run on the workstation. Initially, these servers were manually started by the departing user but later were automatically invoked when the user count reached zero.

FIGURE 3 Process invocation using Butler.



Butler is very useful for implementing gypsy servers. These servers roam idle workstations performing their assigned task (such as machine registry or help servers) while only using spare capacity within the distributed environment. Such mobile servers are located via a shared directory structure as was the early implementation of the machine registry.

The average start-up time for remote executions using the Butler system is seven seconds. It was found to be long enough to discourage people from using the mechanism for small tasks but was popular for long interactive tasks (such as remote shells). Butler is a good example of a simple, non-intrusive load sharing system which provides an elegant solution to remote execution in a distributed UNIX environment. The scheduler uses simple dynamic allocation information (a boolean host idle metric) as the load sharing policy and has sidetracked the pre-emption/migration option by a warn and kill approach. By using Butler servers on each host, the only point of failure for the entire load sharing system is the machine registry server set. The machine registry servers increase the reliability of this point of failure by duplicating themselves. Each machine registry server monitors other servers, starting another when the number falls below a specified minimum.

The Condor Scheduling System

Condor [Litzkow et al., 1988] was developed at the University of Wisconsin-Madison on top of a specialized remote execution mechanism[1] for UNIX. The system is designed to maximize host utilization while minimizing the interference between the workstation users and the allocated tasks. By incorporating a pre-emption and migration mechanism, interference to returning workstation users is minimized. Condor is essentially an elaborate background service which supports execution on remote hosts. All these features come at a minimal overhead with less than 1% of the local CPU used.

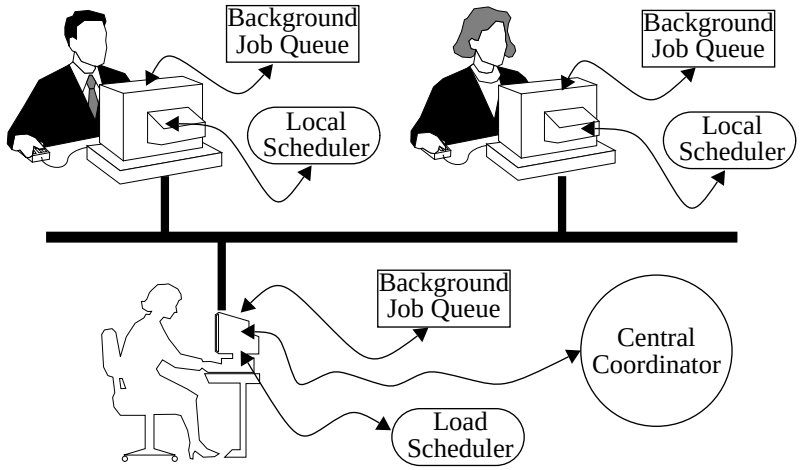
The underutilization of workstation resources and abundance of large computing tasks led to the development of Condor. The long idle periods that characterize workstation processing cycles make the workstations an ideal environment for the scheduling of long running jobs. By guaranteeing the sanctity of the local user, a checkpointed remote execution system can ensure that pre-empted tasks will always run to completion. The Condor subsystem has solved several issues associated with load distribution:

1. Remote Unix (RU) has a checkpointing feature which allows a program to recover its state and be restarted on another machine. As was mentioned in the migration discussion, this feature requires extensive system support.

- task placement is transparent
- on remote site failure, a task is automatically restarted with no user interaction
- there is fair and equal access to available idle remote sites
- scheduling overhead is minimal, ensuring wide user acceptance

FIGURE 4 on page 6 shows how task scheduling is affected from within Condor. Each workstation maintains a

FIGURE 4 Scheduling within Condor.



local job scheduler and background job list. In addition, one site maintains a central coordinator. A user submits background jobs to their local job queue which is in turn periodically polled by the central coordinator. The central coordinator collects these background jobs and also the site states from the local schedulers. A site state is determined by the local scheduler and is set according to the availability of the site for remote processing. The central coordinator then allocates the collected jobs amongst the available local schedulers. While a job is running, the local scheduler monitors whether the local user has returned and if so, pre-empts the executing job. The pre-empted task is then placed in the background job queue again, ready to be picked up by the central coordinator on the next poll. The job will then be restarted at another available site.

The Condor dynamic load sharing system uses an idle metric that is provided by the distributed local schedulers. The system relies on an underlying checkpointed remote execution system to provide migration. Once the local user returns to an idle workstation, any executing allocated tasks are pre-empted and migrated to another idle host. This underlying remote execution relies upon kernel and application changes to provide the checkpointing. Even though the local schedulers are fully distributed, the central coordinator is required to make the final allocation decisions. This leaves a single point of failure through which the load sharing scheme can fail. This problem is minimized as only yet to be allocated tasks are effected. Since the central coordinator has a rather simple job, it can be restarted at another site if required.

STARS[1] — A Distributed Allocation Environment

STARS is a mechanism that provides distributed task allocation based on the demand for and supply of resources in a heterogeneous UNIX workstation environment. The system has been in use (in various forms) at the Compu-

1. Shrewd Task Allocation via Resource Scheduling

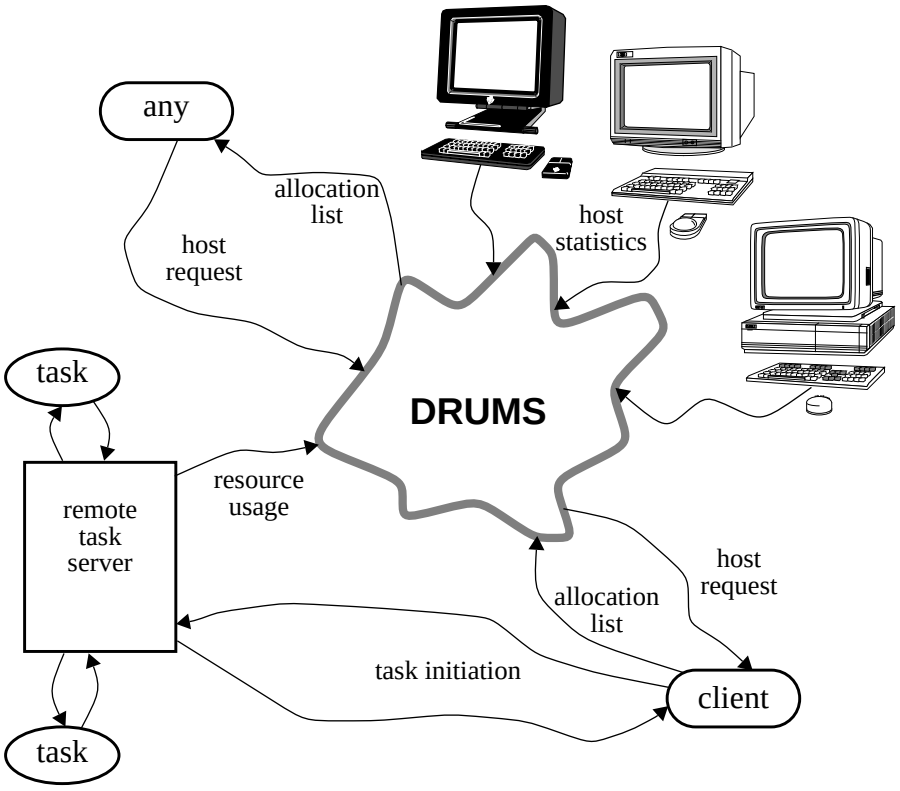
ter Science Department, Victoria University, for over three years providing allocation for a variety of clients. In the previous sections, we have described various forms of allocation for idle workstations. The STARS allocation philosophy is somewhat different. Our method attempts to model more closely the economics of resources in a typical distributed workstation environment. By matching available host resources and task resource requirements, we hope to make more informed decisions on which hosts are more suitable for different tasks.

Before embarking on a tour of the allocation scheme that we use in STARS, we look at an overview of the subsystems that make up the allocation system. Following a brief description of the allocation scheme, some examples are given of how a distributed allocation mechanism such as STARS fits into a typical users environment.

An Overview of STARS

STARS is a multifaceted system that supports distributed task allocation through a set of cooperating servers. FIGURE 5 on page 7 shows how the various parts interact to provide this allocation service.

FIGURE 5 The STARS task allocation environment.



- **DRUMS** is a robust and adaptive server set that roams the network providing access to statistical information about hosts and tasks. Two sets of servers are used, repositories of replicated data and collectors of statistical measurements. The servers minimize their impact on the computing environment by moving from busy hosts to those that have available resources[1]. As the servers move, they are also able to increase or decrease their numbers according to their current processing loads. By a complex set of checks and balances, each server set is robust to all but the most catastrophic multiple host failures.

  DRUMS performs two main tasks.

1. In fact, both sets of servers use STARS to allocate new server instances.

1. The duplicated collector set is responsible for gathering the resource statistics from each host in the distributed environment. This data is then passed on to the fully replicated set of databases. Databases store this host resource information as well as task resource usage data passed on by the remote execution unit (described below). The databases also maintain a list of currently executing tasks that have been invoked through STARS.

2. The most important task of DRUMS is to respond to client allocation queries. The client provides a host requirement description and is given back a ranked list of hosts which best fulfil this criteria (this allocation scheme is described in "The Adaptive Host Allocation Scheme" on page 8).

- **Statistical daemons** are present on each host in the networked environment. These provide the statistical information gathered by DRUM's collectors. The resource statistics are obtained by periodically by gathering data from the local UNIX kernel.

- **Any** is an example of a primitive interface to DRUMS. It packages client queries and ships them off to a DRUMS database for processing. The database returns a list of ranked hosts ordered by how well they fulfil the query. Options to `any` specify the order and quantity of hosts to be printed on the standard output. Typically, `any` is used to select a host for remote execution using the remote task interface. More explanation of this interface is given in "Incorporating STARS into a Distributed UNIX Environment" on page 9.

- A **remote task server** runs on each host in the networked environment. A client front end provides user access to remote computing facilities through these remote task servers. The mechanism performs a job analogous to `rsh` with a few minor additions:

  I. The client front end uses an alias file to find

     a. the exact location of the executable file on the remote host

     b. the environment variables that need transportation to the remote host

     c. which streams of I/O should be transported from the client to server and back

     This elaborate process means no shell need be invoked on the remote host, thus saving on remote task setup time.

  II. On connection, the remote task server registers the initiation of all tasks with the DRUMS databases. The databases then have a list of all tasks currently running in the STARS environment. Upon task completion, the resource usage information is sent to the databases which maintain this task resource usage history for future use by the task allocation algorithm (see "The Adaptive Host Allocation Scheme" on page 8).

The failure of a node within STARS is handled easily. If a DRUMS database was running on the failed host, the other databases will notice an increase in query load and start a new database. Since the database contains only replicated information, the loss is not vital. However, the unique information in collectors is lost if their host fails. The loss of host statistic updates is eventually noticed by the databases and consequently, a database begins to re-register the hosts with out of date statistics. This re-registration process will lead to the creation of a new DRUMS collector (see [Bond and Hine, 1991] for more details). Obviously remote processes executing on the failed node will be lost (but no Condor-like restarting is attempted). The databases will notice that tasks have stopped on the failed host when no task resource usage information is returned and consequently, their execution entries will be deleted from the databases.

The Adaptive Host Allocation Scheme

The task allocation scheme used in STARS is based on dynamic resource statistics gathered by DRUMS collectors. This information is updated every few minutes and when new tasks are started. As well as dynamic resource availability information, the allocation is based on dynamic task resource requirement data. This feature is unique to STARS.

A client to the allocation scheme passes a task alias to a DRUMS database requesting a ranked host list. This alias indexes a table listing characteristics of hosts able to run that task. For example,

```
latex
COMPSCI && (SUN4 || HP300) && hostname != embassy
```

specifies that latex must run on a computer science machine which is a sun4 or hp300 but isn't called embassy. In addition to the characteristics mask to select hosts, the allocation scheme requires a method for ranking hosts. This is accomplished via a resource weight vector specifying how important each resource is to the task. If DRUMS has not seen another task like this before it uses a default set specified in the alias table. For example,

```
max free_virtmem 0.2 , min load_1 0.3, min num_users 0.2,
max mips 0.3
```

Here we name each resource and specify how much relative emphasis is to be placed on it's value. In this example, 20% emphasis on free virtual memory, 20% on the number of users, 30% on the 1 minute load average, and 30% on the relative processor speed[1] of the host.

Each time a task completes, DRUMS adds it to its task resource history. Unfortunately, the resource usage values for tasks and the resource availability measurements available from hosts are not easily comparable. To solve this comparison problem, we introduce an intermediate metric vector that each measurement set can be translated to. The ranked host list for a task is found by producing a metric vector for the task (either from the default weights or from task resource usage history) and ranking how well it fits into the metric vectors computed from host resource availability. As more tasks are run, DRUMS uses the task resource usage feedback loop to learn a tasks resource requirements and eventually makes better allocation decisions for this task.

Work is currently being done to more precisely estimate task resource requirements by not only using the task alias as resource usage selection criteria but also the user running the task, their user group, time of day, host run from, etc. Each task may have several different subsets of resource usage estimates. This extra information should lead to more accurate resource usage predictions and finally better task allocations.

Incorporating STARS into a Distributed UNIX Environment

The introduction of load sharing mechanisms into user environments is sometimes regarded with suspicion. By competing for idle resources, the user feels that "their" workstation is under siege. STARS has been introduced slowly by creating simple interfaces to the allocation scheme. As mentioned before, `any` is used as a simple host allocation interface for users who want to add remote host allocation to their current working environment. Currently, a more complex interface incorporating the remote execution feedback to DRUMS is hidden behind shell scripts.

Many users have introduced `any` into shell aliases and popup menu items. A common use is an alias of the form

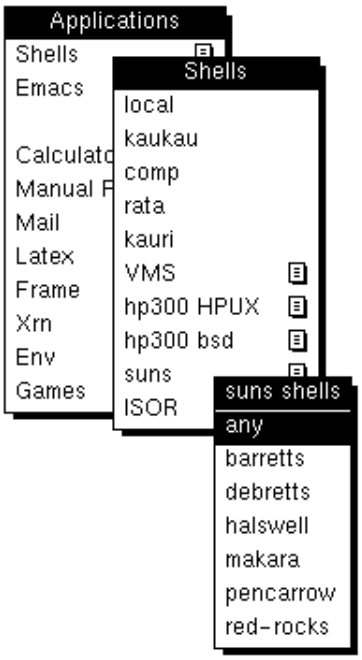
```
$ alias emacs "rsh 'any emacs' emacs </dev/null &"
```

or to use the remote execution form with resource usage feedback

```
$ alias emacs "re-client -h 'any emacs' emacs"
```

1. MIPS do not refer to Millions of Instructions per Second but rather a totally arbitrary speed rating guessed at by the author.

This might appear in a menu as



In addition to an alias table for allocation information, we have previously described an analogous table for remote execution information that is used by the remote execution client. An entry for emacs may look like:

```
emacs     /usr/local/bin/emacs    PWD;HOSTTYPE     NONE
```

In column order these are:

- the alias name
- the remote location of the executable
- the environment variables that need to be set up on the remote host
- which I/O streams need be sent to and received from the remote task[1]

If this information is unavailable for an alias then the default is to set up a remote shell and follow the standard `rsh` procedure for remote process execution. However, if the alias is found in the table, the remote execution set-up is much quicker (approximately 30% of `rsh` executable time) as no shell need be invoked on the remote host.

STARS has also been used in conjunction with various applications where multiple subtasks can benefit from running in parallel. This has included work on a distributed ray tracer which breaks the workload up and distributes server ray tracers to run on hosts allocated by STARS. Another simple application is a parallel make. GNU make supports background processing of compilations in parallel. For example,

```
$ gmake -j 5
```

will execute at most five background compiles in parallel. By assigning the default compiler to be a task allocated through STARS, we can benefit from parallel compilations on separate hosts. We might accomplish this by setting the internal `CC` variable in `gmake` to

```
CC = re-client -h ‘any gcc‘
```

1. NONE effectively isolates the remote execution and the remote execution client returns immediately.

As an example, we have compiled GNU `make` using this parallel compilation method. GNU `make` comprises over 13,000 lines of code in 23 C source files. The tests were conducted on a population of 17 HP300 type machines running `gcc` version 1.40. Each result is the average of four runs.

TABLE 1 Comparison of make times for compilation configurations.

Test #	# Machines	# Parallel Compiles	Distributed/Local	Make time (sec)
1	1	1	Local	568
2	1	1	Distributed	492
3	1	5	Local	297
4	5	5	Distributed	239
5	10	10	Distributed	119

TABLE 1 on page 10 shows the results of five separate tests. Tests 1 and 2 compare sequential local compilation versus sequential distributed compilation using STARS. We only see a modest 13% decrease in execution time for the entire make since most hosts, including the local site, were relatively idle. Test 3 executes 5 parallel compilations on the local host. This is almost 50% faster than the local sequential compile indicating that the machine was not using all its resources when compiling sequentially. Eventually a bottleneck will be reached, limiting the number of parallel local processes that can be executed[1]. Test 4 also performed 5 parallel compiles but this time they were allocated using STARS. We only see a 20% decrease in total make time from the 5 local parallel compiles but a 58% decrease in total time from the original local sequential make is still impressive. This lack of significant decrease over the parallel local compilation shows that a significant local bottleneck has probably not been stressed yet. Finally, test 5 performs 10 parallel distributed compilations allocated using STARS. Here we see an 80% decrease in processing time over the initial local sequential compile and an impressive 50% decrease over the parallel make using 5 machines — by doubling the number of processors, we have halved the execution time. You really couldn't ask for much more than this!

Conclusion

Load sharing is a simple and effective means of maximizing computational resource usage in a distributed computing environment. By applying simple allocation strategies, the average response time is reduced. A load sharing mechanism may be implemented as a centralized scheduling server as in Condor but a single host failure can effectively cripple this load sharing scheme. By distributing the server as in Butler or STARS, the mechanism becomes more robust to such host failures. Butler, Condor, and STARS all use dynamic system information to make effective allocation decisions. The first two are only concerned with idle host allocation while STARS attempts to allocate hosts matching task resource requirements and host resource availability. Condor goes beyond the other two systems in providing preemption and migration facilities at the cost of extensive kernel modifications to support this feature. The other two systems are able to slot into conventional UNIX implementations with no such kernel nor application enhancements.

Users are able to incorporate load sharing mechanisms into their working environment with little effort. By adding appropriate aliases, menu options, and shell scripts the details of load sharing calls are hidden from the user. As well as decreasing job response time, the allocation mechanism is also able to enhance any inherent parallelism in application subtasks. This feature can increase effective processing power by as much as 450% over traditional sequential methods on a single host. In conclusion, a load sharing environment increases global system performance for little overhead or user effort.

1. In fact, it was not possible to perform 10 parallel compiles on the local host as it ran out of virtual memory.

References

Bond, Andy; Hine, John (January 1991) “DRUMS: A Distributed Statistical Server for STARS”, Proceedings of the Winter 1991 USENIX Conference, Dallas, Texas, pp. 335-348.

Freedman, Dan (January 1991) “Experience Building a Process Migration Subsystem for UNIX”, Proceedings of the Winter 1991 USENIX Conference, Dallas, Texas, pp. 349-356.

Litzkow, J. Michael; Livny, Miron; Mutka, Matt W. (June 1988) “Condor — A Hunter of Idle Workstations”. Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, pp. 104-111.

Lyon, B.; Sagar, G.; Chang, J. M.; Goldberg, D.; Kleiman, S.; Lyon, T.; Sandberg, R.; Walsh, D.; Weiss, P. (January 1985) “Overview of the SUN Network File System”, Proceedings of the Summer 1985 USENIX Conference, Dallas, Texas, pp. 1-8.

Markenscoff, Pauline; Liaw, Weikuo (August 1986) “Task Allocation Problems in Distributed Computer Systems”, Proceedings of the 1986 International Conference on Parallel Processing”, pp. 1037-1039.

Nichols, David A. (1987) “Using Idle Workstations in a Shared Computing Environment”. Proceedings of the Eleventh ACM Symposium on Operating System Principles, pp. 5-12.

Ritchie, D. M.; Thompson, K. L. (1974) “The Unix Time-Sharing System”. Communications of the ACM, 17:7,pp.365-375.

Scheifler, Robert W.; Gettys, Jim (April 1986) “The X Window System”, ACM Transactions on Graphics, volume 5, number 2, pp. 79-109.