# Scheduling Parallel Applications in Networks of Mixed Uniprocessor/Multiprocessor Workstations

Olaf Arndt[1], Bernd Freisleben[1], Thilo Kielmann[2], Frank Thilo[1]

[1]Dept. of Electrical Eng. and Computer Science, University of Siegen,
Hölderlinstr. 3, D-57068 Siegen, Germany
{arndt|freisleb|thilo}@informatik.uni-siegen.de

[2]Dept. of Math. and Computer Science, Vrije Universiteit, Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
kielmann@cs.vu.nl

## Abstract

Trying to exploit the idle computing power of workstation networks for parallel applications requires means for dynamic workload scheduling. In this paper, we present the features of the WINNER resource management system developed for this purpose. WINNER relies on an elaborate technique for accurately measuring the currently available computing speed of a workstation, particularly in the presence of workload. Empirical investigations demonstrate that this technique is applicable to both uniprocessor and multiprocessor workstations. WINNER's approach to scheduling is shown for a parallel version of GNU make and for the PVM parallel programming platform. Performance measurements for both cases are presented to illustrate the benefits of our approach.

## 1 Introduction

The price/performance ratio of networks of workstations (NOWs) has fostered their deployment as system architectures for parallel processing applications. Several projects have been initiated to deliver parallel supercomputing power by stacking up and connecting together large numbers of dedicated off-the-shelf workstations [1, 3, 14]. In addition, the omnipresence of enterprise NOWs in today's computing infrastructures and the fact that workstations are idle or only lightly loaded for significant fractions of time have increased the desire to exploit this accrued computational power for the parallel evaluation of computationally intensive tasks.

However, scheduling parallel applications on enterprise NOWs is quite different from scheduling them on dedicated parallel computers. The scheduling strategies used in the latter rely on exclusively accessible and identical processing elements. Scheduling in enterprise NOWs is more challenging, since the actual computing power available for parallel applications is a highly dynamic entity. The reasons are that (a) the speed characteristics of the machines are typically different, (b) machines may fail or may be switched on or off by their primary users, and (c) primary users generate workload which should be processed without interference by additional parallel computations.

To cope with these problems, so–called resource management systems for NOWs have been developed [7, 9, 10, 11]. The incarnation of our ideas on managing the resources of a NOW is the WINNER system[1]. Its design rationale and basic functionality were already described in a previous paper [2]. In the present paper, we particularly focus on WINNER's features that support parallel applications. Therefore, we present WINNER's technique for measuring the dynamically available computing speed of a workstation in the presence of current workload. We show that, using this technique, symmetric multiprocessor workstations (SMPs) can be seamlessly integrated into task placement decisions. We present the behavior of WINNER's scheduling for a parallel version of GNU make [13] and for the PVM parallel programming platform [5]. WINNER's flexibility in providing support for scheduling in these different scenarios stems from its modular design which also opens it for new kinds of applications as well as new hardware components.

[1] **W**orkstations **I**n **N**etworks **N**ow **E**xploit **R**esources

## 2 The WINNER System Design

WINNER has been designed for typical Unix NOW environments, consisting of a central server and several workstations. For a WINNER NOW, the server is required to provide shared file systems and user accounts for all connected workstations. The various tasks of the WINNER system are performed by three kinds of *manager* processes: system managers, node managers, and job managers. The distribution of these manager processes across the workstations is shown in Figure 1. Additionally, there are several user–interface tools e.g. for status reports and for influencing the usage of a user's workstation.

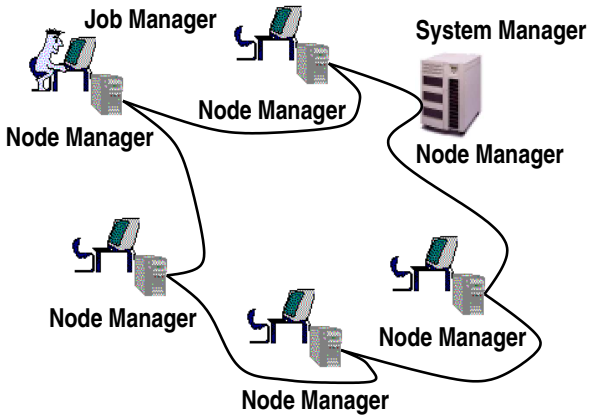


Figure 1: Manager processes in a WINNER cluster.

The system manager is the central server process of a WINNER network. Its duties include (a) collecting the load information of all respective workstations, (b) managing the currently active jobs, and (c) deciding which hosts are assigned to a particular job request.

On every host participating in a WINNER network, a node manager performs the tasks related to the machine it runs on. First of all, it periodically measures the host's utilization and reports it to the system manager. Furthermore, node managers are responsible for starting and controlling WINNER processes on their node, like e.g. reducing process priorities whenever a console user shows up. WINNER has a rich set of features for 'protecting' console users. A description of these features is beyond the scope of this paper and can be found in [2].

System and node managers run as daemon processes. In contrast, job managers are invoked by a WINNER user in order to execute a sequential or parallel job. Thus, job managers are part of WINNER's user interface. Their duties are (a) acquiring resources from the system manager, (b) starting processes on the acquired nodes via the respective node managers, and (c) controlling and possibly redirecting input and output of the started processes.

WINNER's modular structure yields a system which is easily extensible and adaptable to further workstation platforms or kinds of jobs. Its manager processes and user–interface tools are implemented in C++. They communicate with each other by a reliable message–exchange layer on top of UDP sockets. Currently, WINNER runs on Digital Unix, Linux, and Solaris. Two job managers are available for sequential tasks; one for batch execution and another one for interactive programs (including X–Window applications) [2]. The job managers currently available for scheduling parallel applications will be described in the remainder of the paper.

## 3 Determining Processor Speed in the Presence of Workload

In order to perform suitable task placement decisions, the system manager must have an accurate global view of the processor speed and current utilization of the workstations. To achieve this, each node manager provides the system manager with the information related to its own workstation. In this section, we will explain how WINNER's system manager computes $S_c$, the currently available speed a workstation can offer to a process to be scheduled. After first presenting the case of single-processor machines, we will extend WINNER's scheme for computing $S_c$ to SMP workstations containing multiple (identical) processors.

### 3.1 The Duties of Node Managers

At startup, each node manager performs a simple benchmark loop, evaluating the machine's speed (of a single processor) in integer operations, floating point calculations, and memory access. This benchmark's result is a single number proportional to the host's sequential performance, relative to every other workstation in a network. This value (called the *base speed* $S_b$) is reported to the system manager along with the node name, IP address, and other static data such as the amount of main memory and the number of CPUs.

Afterwards, the node manager regularly queries several load characteristics of its local host and reports them to the system manager either if they differ significantly from the last set of data sent or after a certain

time interval, indicating that the node manager is still "alive".

To measure the currently existing workload of a machine, Unix kernels provide a so-called *load average* value averaging the number of processes in the run queue within certain time intervals, the fastest of which is typically averaged over the last 60 seconds. Due to this averaging procedure, these load values follow the real load situation only very slowly. To get more recent data, WINNER computes the current run queue length from two consecutively measured load-average values as follows. The Unix *load average* value $a_t$ at time $t$ is computed by the kernel every $i$ seconds from the current length of the run queue $q$ and the load average $a_{t-i}$, $i$ seconds ago:

$$a_t = \beta\, a_{t-i} + (1-\beta)\; q$$

Here, $\beta$ is the smoothing factor determining by which the old average $a_{t-i}$ contributes to $a_t$. For the 60 second load average, a value of $\beta = e^{-i/60}$ is used. Neglecting variations of the run queue length $q$ during an interval and assuming an interval of $i = 10$ seconds between two consecutive load measurements performed by WINNER, we get an approximation for the average run queue length $a$ from two consecutive load values:

$$a \;=\; \frac{a_t - e^{-10/60}\, a_{t-10}}{1 - e^{-10/60}}$$

Using this equation, WINNER calculates the average run queue length $a$ every 10 seconds. Fig. 2 illustrates the benefits of using $a$ instead of the values $a_t$ as they are reported by the Unix kernels. In this figure, the load of a (hypothetical) machine changes between the values 0, 1, and 2. WINNER's $a$ value follows the load to its exact value every 10 seconds, as soon as the next value of $a$ is computed. For 5–seconds averaging, the load average $a_t$ follows the real load slightly slower than $a$. Unfortunately, this averaging interval is only available on Digital Unix. Various other flavors of Unix (e.g. Solaris, Linux, AIX, BSD/OS) only provide a load value $a_t$ that is averaged over the last 60 seconds. It is obvious from the figure that this value lags far behind the real load situation. Hence, computing the load average $a$, WINNER can react much faster and hence much more accurate to changing load situations.

Unfortunately, the load average values $a_t$ as reported by the Unix kernels may be misleadingly high. This can happen when many shortly running processes are in the run queue. In that case, the CPU utilization can be observed more accurately using the fraction of time the processor(s) spent in the *idle* CPU state. In the single-processor case, the fraction of time spent

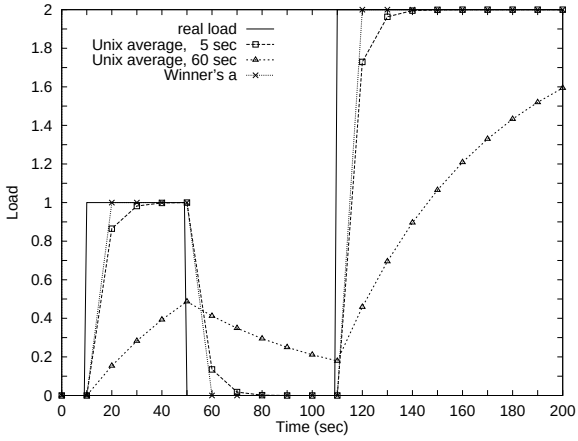


Figure 2: Load averages computed by WINNER and different Unix kernels.

in this state is reported by the operating system to be within the interval $[0,1]$. In the case of a machine with $p$ processors, an interval of $[0,p]$ is used instead. However, once the CPU idle time is close to zero, it is impossible to distinguish whether only one process is fully utilizing the CPU or whether several processes (per processor) are present in the CPU run queue. Hence, both types of information have to be used by WINNER to determine processor utilization exactly.

### 3.2 The Single-Processor Case

Whenever a job manager requests a new node for its job, the system manager has to select the most appropriate machine. Besides checking for the presence of a console user and verifying static properties such as requested memory sizes, the system manager basically takes the currently available speed into account.

Based on the workstation's base speed $S_b$, its current speed $S_c$ is calculated as $S_c = S_b/\lambda$, where $\lambda$ denotes the fraction of available processing power in the presence of the current workload. Assuming a constant load, $\lambda$ could be calculated as $\lambda = a + 1$ (where $a$ is the load average as computed by the node manager). This reflects the fact that after starting a new process, there are $\lambda$ active processes sharing the CPU.

As explained above, computing the available processor speed based on the load average may be inaccurate. Hence, for achieving more precise values, WINNER's system manager instead calculates $\lambda$ by using $t_i$ (the percentage of time the processor was idle): $\lambda = 2 - t_i$, yielding $\lambda \approx 1$ for high percentages of idle time and $\lambda \approx 2$ for higher CPU usage.

In the case of small values of $t_i$, it can be assumed that the workload consists of more than one process.

Then, the load average value should be used for calculating $\lambda$ in order to reflect the higher load. An empirically determined threshold of $t_i = 0.15$ is used for switching between the two cases. $\lambda$ is hence computed as follows:

$$\lambda = \begin{cases} a+1, & t_i < 0.15 \\ 2 - t_i, & t_i \geq 0.15 \end{cases}$$

### 3.3 The Multi-Processor Case

For seamlessly integrating SMP machines into WINNER networks, the system manager has to adapt its computation of $S_c$ accordingly. There are two basic differences that have to be taken into account. First, on a machine with $p$ processors, the fraction of CPU idle time is reported in the interval $[0, p]$. Second, the number of running processes (constituting the load average $a$) will be serviced by all $p$ processors. Assuming an ideal scheduler (in the operating system), their load will be equally distributed across all $p$ processors. Nevertheless, a single process can only be served by a single processor. Hence, whenever there are less processes than processors, the available speed must be derived from the capacity of only one processor.

This situation changes in the case of multithreading. But since it is impossible to predict whether a given program binary will use multiple threads of control, a resource "multi-processor machine" has to be requested by the user explicitly in this case. Although the scheme presented here does not help to automatically select multiprocessor workstations for multithreaded applications, the workload generated by multiple threads will still be observed correctly.

The computation of $S_c = S_b/\lambda$ is performed analogously to the single-processor case with the exception that $\lambda$ is computed via an intermediate value $\lambda'$. For $t_i \geq 0.15$, $\lambda'$ is computed as $\lambda' = 1 + p - t_i$. For $t_i < 0.15$, $\lambda' = a + 1$ as with a single processor. Finally, $\lambda$ is computed as $\lambda = \lambda'/p$ while excluding values $\lambda < 1$ whenever there are less processes than processors. The computation of $\lambda'$ and $\lambda$ can be summarized as follows:

$$\lambda' = \begin{cases} a+1, & t_i < 0.15 \\ 1 + p - t_i, & t_i \geq 0.15 \end{cases}$$

$$\lambda = \begin{cases} 1, & \lambda' < p \\ \lambda'/p, & \lambda' \geq p \end{cases}$$

It is easy to see that the computation for $\lambda$ coincides with its single-processor counterpart for $p = 1$. Consequently, WINNER's always uses this enhanced scheme for computing $S_c$. Fig. 3 illustrates how $S_c$ models CPU capacity available to new processes in SMP systems, depending on the number of CPUs and the given load average. Given $p$ CPUs and load averages of up to $p - 1$, 100% of a single CPU's capacity is available to a new process. With higher load, this capacity decreases while CPUs are shared between processes.

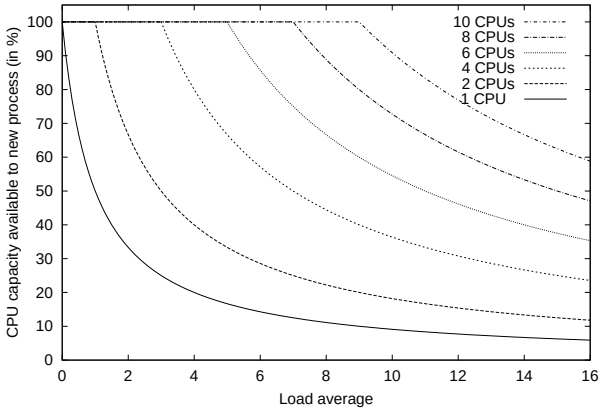


Figure 3: Available CPU capacity in SMP systems as modelled by $S_c$.

In order to evaluate WINNER's model of process scheduling on SMP machines, its actual behavior has been tested on two SMP platforms. One machine is equipped with two 200 MHz Sun Ultra Sparc processors running Solaris 2.6, the other with four 200 MHz Pentium Pro processors, running Linux 2.1.89. The benchmark program computed the prime numbers up to $2 \cdot 10^6$. It merely uses CPU power without performing I/O operations. As background load, up to ten long running, CPU intensive processes (approximating the value of $\pi$) have been used which where present throughout the benchmark. Basically, our measurements compare the actual process scheduling with the runtimes expected via WINNER's $S_c$ value without disturbance by I/O operations.

As shown in Figure 4, WINNER's $S_c$ value perfectly models the real behavior of the Linux SMP scheduler. On Solaris, $S_c$ still produces gradually correct results while the costs of process switching seem to be significantly higher than on Linux. Therefore, the actual benchmark runtime is up to 10% slower than expected. It would be interesting to investigate whether these results carry over to larger SMPs where memory access becomes a bottleneck.

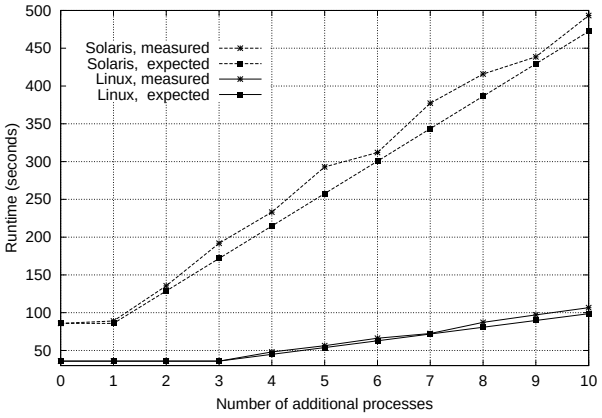


Figure 4: Process scheduling on SMP machines.

## 4 A Parallel GNU make

The standard Unix tool make is typically used for building large software systems from their source code files. Therefore, a so-called makefile contains all dependencies between source code, the final application binary, and possible sets of intermediate files. For building an application binary, make reads the respective makefile and creates a dependency graph between all files. Typically, the making of intermediate files (or of multiple targets) can be performed independently, according to the graph structure, and may hence be constructed in parallel.

The freely available implementation of GNU make [13] already allows to make independent targets quasi-parallel within multiple, concurrent processes on a single machine. To utilize this feature, the user has to specify a maximum number of concurrently operating processes. wmake (WINNER make) extends GNU make by plugging in code to select the most suitable (e.g. fastest available) workstations as targets for process creation, thus transforming the make process into a parallel and distributed application.

wmake's runtime behavior was evaluated on a cluster of Sun Sparc workstations sharing a NFS file system over 10 MBit/s ethernet and running the Solaris operating system, two of which are equipped with two processors (see Table 1). These runtime measurements were performed in order to evaluate two aspects: WINNER's efficacy for parallel jobs, and the policy of utilizing SMP workstations like multiple single-processor machines. wmake's test targets were the building of the ACE toolkit consisting of 167 C++ source files, the Emacs editor with 76 C source files, and the POV–Ray raytracer with 69 C source files.

Table 1: Sparc workstations for evaluating wmake.

| Name | Architecture | MHz | CPUs | $S_b$ |
|---|---|---|---|---|
| pi47 | UltraSPARC | 200 | 2 | 307 |
| pi45 | UltraSPARC | 167 | 1 | 279 |
| pi16 | SuperSPARC | 60 | 2 | 113 |
| pi24 | MicroSPARC | 70 | 1 | 69 |
| pi25 | MicroSPARC | 70 | 1 | 68 |
| pi31 | MicroSPARC | 70 | 1 | 66 |

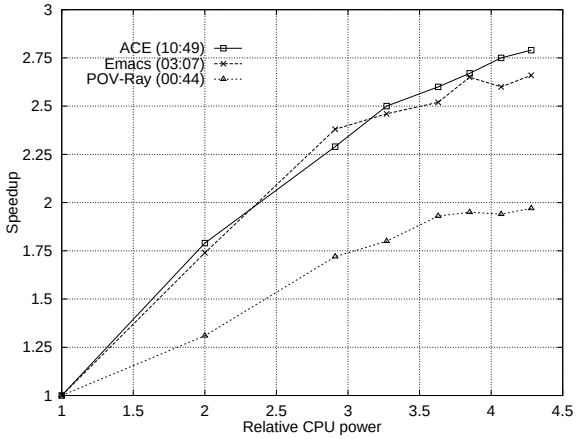


Figure 5: Speedup of wmake, up to 8 parallel processes.

Figure 5 shows the speedup achieved by wmake for which one up to eight parallel processes have been requested. The sequential runtimes are given in the format *(mm:ss)*. By assigning the fastest available CPU (one inside pi47) the relative speed of 1.0, the accumulated speed of all CPUs has a value of 4.28. In the absence of additional workload, when invoked with one active process, WINNER will place it on the fastest machine, pi47. With two parallel processes, both will be placed on pi47, exploiting both CPUs inside this machine. Accordingly, with three parallel processes, two will be placed on pi47 and one on pi45. Further on, all eight available CPUs will be used in this manner. Taking into account that compiler runs typically cause significant fractions of I/O time and that the final linking phases are inherently sequential, the maximally achieved speedup values clearly indicate the usefulness of wmake's parallelization. Furthermore, the runtime improvements achieved by employing the second CPUs of the two dual-processor workstations emphasize our results on WINNER's scheduling policy for SMP machines as presented in the previous section.

So far, we investigated WINNER's scheduling policies only in the absence of additional workload. In this case, the machines' static base–speed values $S_b$ deter-

mine the task placement decisions. Because WINNER keeps track of dynamically changing workload conditions, it also avoids scheduling tasks on temporarily slow machines. This behavior is analyzed in the next section along with WINNER's PVM support.

# 5 Scheduling PVM Tasks

WINNER provides a job manager (wpvm) for being able to schedule PVM tasks. This job manager automatically selects a number of appropriate hosts for the PVM virtual machine (usually a tedious task when performed manually), starts the user's parallel application, and improves PVM's scheduling by utilizing WINNER's load distribution mechanism.

## 5.1 System Integration

In order to avoid changes to the PVM system and to application programs, WINNER follows the approach introduced along with the CARMI system [11] for integrating a resource management system into PVM. This is achieved by registering several processes into the PVM infrastructure (for details see [2]): The so-called PVM resource manager is the most important of these processes. Its main task is to replace PVM's default round-robin scheduling by WINNER's workload scheduling schemes. The hoster and tasker processes are responsible for starting the PVM daemons and PVM user processes, respectively, thus enabling wpvm to start and control all processes of the PVM application via the WINNER interfaces.

A user invokes wpvm specifying the desired number of hosts and the name of the user application's master program. wpvm then contacts the system manager in order to get assigned the most suitable (i.e. the fastest available) workstations. The fastest of these hosts is selected to act as the master host for the virtual machine. wpvm spawns the resource manager task on the master host. This task initializes the virtual machine by starting the master PVM daemon. Next, the hoster task is started, and the virtual machine is expanded to include all assigned hosts. Finally, one tasker process is started on each workstation.

After all taskers are active, wpvm spawns the application's master task on the master host. The master task then will presumably spawn some PVM child processes (via pvm_spawn()). The task placement decisions for these processes are then automatically redirected to the resource manager which uses WINNER for choosing a suitable host for each task.

When all user tasks have terminated, the parallel job has obviously finished. wpvm will now shut down the PVM virtual machine, wait for all auxiliary tasks (resource manager, hoster and taskers) to finish, and finally terminate itself.

## 5.2 Runtime Evaluation

To illustrate WINNER's effect on application runtimes, a parallel algorithm for solving the *knapsack problem* was implemented using the PVM system. The knapsack problem is defined as the problem of finding a set of items each with a weight $w$ and a value $v$ in order to maximize the total value while not exceeding a fixed weight limit. In the implemented *divide–and–conquer* algorithm [4], the problem for $n$ items is recursively divided into two subproblems for $n-1$ items, one with the missing item put into the knapsack and one without it. Whereas the first subproblem is handed over to another processor, the second one is recursively computed within the same node, yielding a dynamically shaped task tree. The assignment of tasks to processors and the related workload distribution is the primary problem of such tree computations.

We performed comparative runtime measurements on a cluster of workstations (based on Linux 2.1.89) which consists of 16 machines with one Pentium II processor each, running at 300 MHz, and the abovementioned SMP server with 4 Pentium Pro CPUs, running at 200 MHz. As the communication platform, we used PVM version 3.4.beta6. To simplify the following analysis, we normalized the base speed of the Pentium II processors to $S_b = 1.0$. Relative to these processors, a single Pentium Pro of the SMP machine has $S_b = 0.65$. We performed three series of measurements, using 8, 12, 16, and 20 CPUs; each with standard PVM and with wpvm. Since the number of tasks created by the knapsack application is a power of two, the number of created processes was 8, 16, 16, and 32, respectively. Hence, in the case of 12 and 20 CPUs, some of them had to accomodate two tasks.

For the wpvm runs, the machines were selected dynamically by wpvm. For the PVM runs, we assumed a "smart" user with knowledge about the static machine speeds. Hence, in both cases the Pentium II machines were used, and the SMP server was only added for the 20-CPU tests. Furthermore, we assumed that the PVM user would ignore dynamic load situations for selecting hosts, leading to randomly overlapping PVM configurations with background load processes. We reckon this kind of user behavior to be typical for manually setting up PVM configurations.

In the first series of experiments, no machine did

have any additional workload. In the second series, we inserted 6 CPU-intensive processes (the above-mentioned approximation of $\pi$) prior to the start of the parallel application. Again, the background load was still present when the application terminated. These six processes were also placed by WINNER, hence some of the Pentium II machines were randomly picked. The third series of experiments was analogous to the second one, except that 12 background processes were inserted into the system.

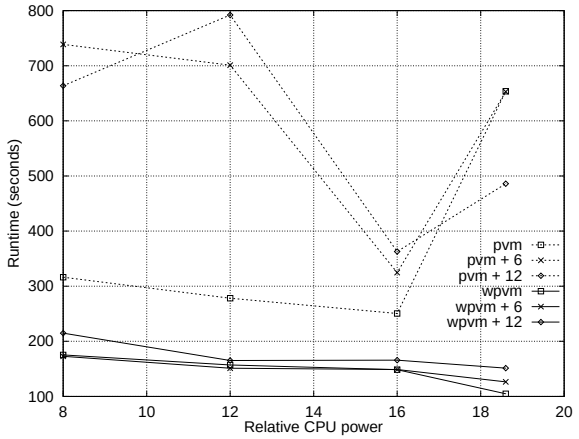


Figure 6: Runtimes of the knapsack application as scheduled by PVM and wpvm, depending on additional work load.

Figure 6 presents the results of our measurements. The values shown are average values of multiple runs. First of all, Figure 6 demonstrates that the application runtimes using WINNER are significantly smaller than those resulting from using the PVM scheduler. Since in the absence of additional load wpvm selects equally fast machines as those used in the PVM experiments, the differences in the runtimes must be due to the differences in assigning tasks to hosts. An investigation of PVM's placement decisions indeed revealed that on some machines multiple tasks had been placed, while on other machines no tasks at all were running. It seems that PVM still has problems performing round-robin scheduling correctly when tasks get created dynamically on different machines at the same time.

At least without additional workload, the PVM runs using 8 to 16 uniprocessor machines yield results that are consistent with each other (although they are still much higher than the corresponding wpvm results). But when the SMP machine is added, the 16 additionally created processes are distributed poorly, which is due to the above-mentioned problems as well as because PVM has no knowledge about the number of CPUs in the SMP machine. It hence treats it like a uniprocessor workstation. Furthermore, in the case of background workload, PVM fails to distribute workload efficiently, leading to much higher runtimes.

wpvm, however, builds PVM configurations by collecting the currently fastest hosts. An unloaded Pentium II ($S_c = 1.0$) is preferred over a single Pentium Pro ($S_c = 0.65$), which is preferred over a Pentium II with a background process ($S_c = 0.5$). For runs with 20 CPUs, wpvm has to use all available machines. Hence, the performance degrades when the background workload is increased. When using less CPUs, wpvm avoids the slower hosts and places several tasks onto the SMP machine, thus minimizing the background load's impact on the parallel application.

To summarize our results, we observe that compared to the plain PVM system, wpvm nicely maps processes to available CPUs while taking dynamically changing CPU speeds as well as SMP machines into account, which finally yields faster application runtimes. By automatically building a PVM virtual machine consisting of the actually fastest CPUs, wpvm also removes this tedious task from the PVM user.

## 6 Related Work

There are several alternative resource management approaches for NOWs that provide support for the scheduling of parallel applications. Their properties with respect to this support are briefly described in the following. To the best of our knowledge, none of them explicitly supports SMP workstations.

CARMI [11] operates on top of Condor [9] and provides a resource management API to PVM applications, allowing them to exploit dynamically changing sets of hosts. However, existing PVM applications have to be rewritten which can be rather tedious as programmers have to explicitly handle the addition and removal of hosts. This problem is somewhat alleviated by the accompanying WoDi library, which provides a comfortable interface for implementing manager/worker applications on top of CARMI.

The Prospero Resource Manager (PRM) [10] lends its modular structure to WINNER. However, PRM's evaluation of load information is rather simplistic in that it is only used to decide whether or not a workstation is available for job assignment. Moreover, while a host is assigned to a job, this machine is not available for further jobs. PRM supports PVM applications by providing its own version of PVM, yielding compatibility problems with current and future PVM releases.

Finally, the load balancing extension to PVM introduced in [7] collects real load information which it presents to PVM applications by means of a load vector as well as a dedicated, load balancing spawn function. Unlike WINNER, this system is restricted to a single instance of a PVM virtual machine and therefore only performs load balancing for one PVM application at a time. In contrast, WINNER works on the middleware level and is hence capable of simultaneously scheduling several (sequential as well as parallel) applications without undesirable interferences between their load distribution decisions. Furthermore, wpvm does not require any changes to existing application binaries in order to make its services accessible.

## 7 Conclusions

In this paper, we presented the features of the WINNER resource management system that provide support for scheduling parallel applications. We explained how WINNER seamlessly integrates symmetric multiprocessor workstations in its scheduling decisions, a feature that presently does not seem to be supported by any other resource management system. Furthermore, we have presented WINNER's scheduling behavior for a parallel version of GNU make and for the PVM parallel programming environment. Runtime measurements have shown the suitability of our approach in general, as well as the superiority of WINNER compared to the default PVM scheduler.

There are several areas of ongoing and future research: support for further programming platforms like MPI [6] and Objective Linda [8], development of a batch queueing system for controlled and fault–tolerant execution of long–running applications, and the integration of checkpointing and migration facilities for parallel applications like to ones implemented in the CoCheck system [12].

## References


[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[2] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Dynamic Load Distribution with the Winner System. In *Proc. Workshop Anwendungsbezogene Lastverteilung (ALV'98)*, pages 77 – 88, Munich, Germany, 1998. Published as Technical Report TUM-I9806, SFB 342/01/98 A

[3] D. J. Becker et al. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[4] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[5] A. Geist et al. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. MIT Press, 1994.

[7] D. J. Jackson and C. W. Humphres. A Simple Yet Effective Load Balancing Extension to the PVM Software System. *Parallel Computing*, 22:1647–1660, 1997.

[8] T. Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. *Proc. 8th Int. Conf. of Computing and Information*, Waterloo, Ontario, Canada, June 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).

[9] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. *Proc. of the 8th Int. Conf. on Distributed Computer Systems*, pages 104–111. IEEE, 1988.

[10] B. C. Neuman and S. Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distr. Systems. *Concurrency: Practice and Experience*, 6(4):339–355, 1994.

[11] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *LNCS 949*, pages 259–278, USA, 1995. Springer.

[12] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. *LNCS 1162*, pages 140–154, Honolulu, Hawai'i, 1996. Springer.

[13] R. M. Stallman and R. McGrath. GNU *Make*. Free Software Foundation, Boston, MA, USA, 1997.

[14] M. S. Warren et al. Parallel Supercomputing with Commodity Components. *Proc. Int. Conf. on Parallel and Distrib. Processing Techniques and Applications*, pp. 1372–1381, 1997.