A Dynamic Load Balancing System for Parallel Cluster Computing

B. J. Overeinder, P. M. A. Sloot, R. N. Heederik and L. O. Hertzberger

Parallel Scientific Computing and Simulation Group,
Department of Computer Science, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract

In this paper we discuss a new approach to dynamic load balancing of parallel jobs in clusters of workstations and describe the implementation into a Unix run-time environment. The efficiency of the proposed methodology is shown by means of a number of case studies.

Key words: Dynamic load balancing, task migration, distributed computing systems, clusters of workstations.

1 Introduction

With the advent of high-speed networks (most prominently FDDI and ATM), clusters of workstations achieve the same scalable parallelism as the current MPP architectures. However, software support for such parallel cluster systems still lags behind. These parallel cluster systems require new programming paradigms and environments to provide the user with mechanisms and tools to exploit the full potential of the available distributed resources. In these systems changes such as variation in demand of processor power, variation in number of processors, or dynamic changes in the run-time behaviour of the application, hamper the efficient use of resources.

If we deal with load balancing on parallel cluster systems, we can identify three levels that determine the efficient use and utilization of the distributed resources, namely:

– domain decomposition, because workload should be evenly divided over a set of tasks;

– global scheduling, since the tasks must be mapped to distributed resources;
– local scheduling, for the fact that multiple tasks must be scheduled on one processor.

Next, we focus on the consequences with respect to load balancing in case the amount of workload and available distributed resources are not static.

Consider, for example, an application that after a straightforward domain decomposition, can be mapped onto the processors of a parallel architecture. If the hardware system is homogeneous and allocated to only one application program, then the execution will run balanced until completion: we have mapped a static resource problem to a static resource system. However, if the underlying hardware system is a cluster of multi-user workstations we run into problems because the available processing capacity per node may change: in this case the static resource problem is mapped to a system with dynamic resources, resulting in a potentially unbalanced execution. Things can get even more complicated if we consider the execution of an application with a dynamic run-time behaviour to a cluster of workstations, i.e., the mapping of a dynamic resource problem onto a dynamic resource machine.

One way to deal with this dynamic changing resource requirement would be an intelligent system that supports the migration of processes from overloaded to under-loaded processors at run-time, without interference from the programmer. In addition, the resulting adaptive system hides the complexity of the load balancing from the programmer/end-user. It implies the following constraints:

– since we assume that the computational resource is a scalable cluster environment, the application programming model must be based on message passing;
– it is essential we support a generic operating system, therefore the machine platform operating system should be Unix;
– by hiding the complexity in libraries, the dynamic load balance run-time support system must be incorporated at user level.

The first two constraints directly relate to the target hardware platform, i.e., a cluster of workstations interconnected by a network and providing *de-facto* the Unix operating system. The third constraint stems from the fact that dynamic load balance facility is supplied on top of Unix, and not by enhancing the operating system. This facilitates the acceptance by individual users in academia and industry.

To fully exploit the potential of clusters of workstations, a detailed comprehensive understanding of the underlying mechanisms must be obtained. Therefore, a good understanding of the interplay between the dynamic parallel application systems and the adaptive computing systems is essential.

The work presented here reports on a pilot implementation of such an experimental adaptive system, for which we have tossed the name *Dynamic-PVM* [4,15]. The paper is outlined as follows. In Section 2 we introduce in general terms the necessary components for dynamic load balancing within the context of the formulated constraints. Given the functional design outlined in this section, the resulting implementation of the run-time support system is described in Section 3. Experiments and results are presented in Section 4. The results of the experiments are discussed and summarized in Section 5. Section 6 concludes with future work.

2 Background and Design Aspects

Within the design of a self-contained experimental environment for dynamic load balancing of parallel application systems, at least the following three functionalities should be included: (i) parallel programming environment, (ii) parallel run-time support system, and (iii) checkpoint/migration facility. The parallel programming environment enables the programmer to decompose the application problem into parallel subtasks. The parallel run-time support system allows for the parallel execution of the parallel application system; and the checkpoint/migration facility extends the run-time support system with functionality necessary for dynamic load balancing.

The first two facilities are provided by the PVM system [12]. The PVM system includes an application programming interface for parallel program development and a run-time support system to allow for parallel execution of the application. The task checkpoint/migration functionality extension must be integrated with the PVM run-time support. The choice to use PVM as the basic parallel programming environment is motivated by the free availability of the source code and the extendibility of the run-time support. The application programming interface incorporates the dynamic addition and deletion of hosts (resources) and processes. Moreover, PVM is the most widely used message passing environment to date. Therefore, we are able to test our system for various existing PVM-based applications.

With respect to the checkpoint/migration facility, two operation levels are distinguished: *operating system level* and *user level*. In operating system level implementations the resource management facilities are supported by the OS kernel. Examples of such systems are Mach [9], V-Kernel [14], Sprite [5], and Charlotte [2]. User level designs and implementations of adaptive systems include dynamic resource management facilities by providing their own dynamic load balancing run-time support. Examples of user level designs are Condor [8] for sequential, and MPVM [3] for parallel application systems. The research presented here is a typical example of this last category.

Table 1 shows a comparison of the different aspects of granularity and load managing required by the three systems under considerations: Condor, PVM, and the adaptive system we discuss here: DynamicPVM. Condor is included here as a representative example of a sequential job migration system. We use the term *job* to indicate the largest entity of execution (the application) consisting of one (viz., a sequential program) or more cooperating *tasks* (viz., a parallel program).

	Condor	PVM	DynamicPVM
intended usage	long running background jobs	distributed parallel programs	
unit of execution	job	task	
load managing objective	load distribution	load decomposition	both
schedule policy	dynamic load balancing	round-robin allocation	dynamic load balancing
schedule objective	resource utilization	application turnaround time	both
performance objective	efficiency	effectiveness	both

Table 1
Granularity and workload management strategies for Condor, PVM, and DynamicPVM.

This table indicates the basic design considerations given the constraints we set out to met. The next subsections describe the essentials of the message passing system and the checkpoint/migration facility required to implement the functionalities outlined in Table 1.

2.1 The PVM System

The PVM (Parallel Virtual Machine) system presents an integrated environment for heterogeneous concurrent computing on a network of workstations. The computational model is process-based, that is, the unit of parallelism in PVM is an independent sequential thread of control, called a task. A collection of tasks constituting the parallel application, cooperate by explicitly sending and receiving messages to one another. The support for heterogeneity permits the exchange of any data type between machines having different data representations.

The PVM system consists of two parts: a daemon, called *pvmd*, and a library

of PVM interface routines, the *pvmlib*. The PVM daemon and library enables a uniform view of the network of workstations, called hosts in PVM, as a parallel virtual machine.

Each host in the virtual machine is represented by a daemon that takes care of task creation and dynamic (re-)configuration of the parallel virtual machine. PVM tasks are assigned to the available hosts using a round-robin allocation scheme. Once a task is started, it runs on the assigned host until completion, i.e., the task is statically allocated.

The PVM library implements the application programming interface that includes primitives for process creation and termination, host addition and deletion, coordinating tasks, and message-passing primitives. The underlying communication model can be classified as asynchronous message-passing, where the messages are buffered at the receiving end. An important aspect of the communication model is that the message order from each sender to each receiver in the system is preserved. The PVM message-passing interface supplies both point-to-point communication primitives and global communication primitives based on dynamic process groups. To enable the use of heterogeneous host pools, messages can be encoded using an external data representation (XDR [11]).

A relevant issue in the context of the forthcoming discussion, is message routing. PVM supports two routing mechanism for messages, namely *indirect* and *direct* routing. By default, the messages exchanged between tasks are indirectly routed via the PVM daemon. With indirect routing, a task sends the messages first to the local PVM daemon. The local daemon determines the host on which the destination task resides, and sends the message over the User Datagram Protocol (UDP) transport-layer to the responsible daemon. This daemon eventually delivers the message to the destination task. For example in Fig. 1, an indirect path from task a1 to b2 goes via pvmd A and pvmd B. Direct message routing allows a task to send messages to another task directly over a Transmission Control Protocol (TCP) link, without interference of the PVM daemons and thereby enhancing communication performance (see for example TCP connection between tasks a1 and c1 in Fig. 1).

2.2 Aspects of Checkpoint and Migration

Systems supporting dynamic load balancing, such as Charlotte [2], Condor [8], or V-System [14], stem from the observation that many of the—constantly increasing number of—workstations in academic and industrial institutions are lightly loaded on the average. In general, workstations are intended for personal usage, which has a typical activity pattern that machines are only

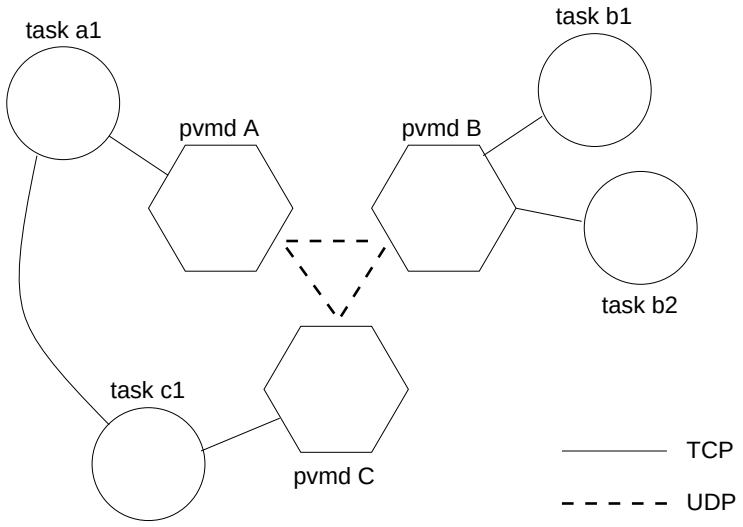

Fig. 1. The PVM system composed of daemons and tasks.

used for a small part of the day. Typical figures of large pools of workstations have a mean idle time of 80% [8]. Thus, there is an opportunity to use these idle workstations as computation servers to increase the processing power available to active users and such to improve the utilization of the hardware. The problem however, is the complexity involved to make efficient use this idle time.

To address this problem, global scheduling based on dynamic load balancing by process migration is implemented. In order to make scheduling decisions, the dynamic load sharing system monitors the workstations in the network by keeping track of their load parameters. Workload is balanced over the network by placing new jobs on lightly loaded nodes and migrate jobs from heavily loaded machines to less loaded ones. To guarantee unobtrusiveness, access to idle workstations and retain the sympathy of the workstation's owner, the system can detect interactive usage of a workstation and evacuate all jobs from such a workstation.

Process *migration* is realized by the movement of an active process from one machine to another in a parallel or distributed computing system. The process is suspended and detached from its environment, its state and data (the *checkpoint*) transfered to the destination host, where it is restarted and attached to the destination environment. The major requirement for providing a migration facility is *transparency*: the execution of a process should proceed as if the migration never took place. In parallel application systems, this transparency should hold also for the migrated process's communication partners. The application programs then do not to have take account for possible complications of checkpointing and migration. Migration is mainly applied to long running jobs to counterbalance better load balance for suspend, migration, and restart overhead.

The effective global scheduling of application programs on a cluster of workstations is essential to efficiently use the potential of the system: it should achieve an efficient mapping between an application program and the parallel cluster. In general, global scheduling consists of three components: load data acquisition and distribution, and a load balance algorithm. For example, the Condor scheduler consists of both a centralized and a distributed part. Each node in the pool runs a small daemon that gathers statistics about the node and forwards this information to the central scheduler. This information is used to maximize the exploitation of the available processing power.

The problem at hand is an experimental adaptive system, where we concentrate on the integration of a checkpoint/migration facility within PVM to enable global scheduling of parallel tasks. Global scheduling itself is a vast area of research, but will not be discussed in this paper.

3 Implementation Aspects of the Extensions in the DynamicPVM Experimental System

This section describes the extensions to PVM that are necessary to support dynamic load balancing within the run-time support system. In order to implement task migration, as eluded in Section 2.2, functionalities in the *pvmd* and *pvmlib* need to be enhanced with checkpoint/migration mechanisms.

It is essential to note that the intertask communication, viz., message routing by the *pvmd*, is strongly affected by the added functionality of task migration. Therefore, we need to develop a methodology to guarantee the transparency and correctness of this intertask communication.

The extensions to the *pvmd* and *pvmlib* must not change the PVM programming interface and semantics, such that source code portability is guaranteed. The packet routing by the *pvmd* ensures migration transparency. With this approach, any standard PVM application can be linked and executed with the DynamicPVM system without a modification to the source code of this application, thus hiding the complexity for the end-user.

3.1 The Scheduler

Although the scheduler as such is not considered in the experimental DynamicPVM system, its role and interface is mentioned here. The scheduler in DynamicPVM is the initiating process of all load balancing activities. The scheduler acts as a resource manager of the distributed system, that is, it de-

cides when to migrate a task and to which host it is moved. In this respect, the scheduler largely determines the effectivity of the DynamicPVM system in its aim for load sharing. The development of good algorithms or heuristics for load sharing is a study on itself and is beyond the scope of this article. The current simple scheduler decides on (re-)allocation of processors for tasks, based on gathered load information of the workstation pool.

The scheduler is implemented as a normal PVM task. This approach makes the incorporation of new scheduling strategies flexible and provides for a flexible experimental platform for studying the effectivity of the different load balancing disciplines. A consequence of implementing the scheduler as a PVM task, is that an additional interface must be provided to enable the scheduler to interact with the DynamicPVM system, in particular with the PVM daemons. To this end, the *pvmlib* is extended with an interface routine, `pvm_move(tid, host)`, that initiates the migration of task `tid` to the specified host.

From an operational point of view, the activities of the scheduler consist of gathering distributed load data of the hosts in the pool, and decide on initial task placement and task migration. Initial task placement is the allocation of newly created tasks to hosts. The actual creation of tasks is the responsibility of the *pvmds*. If the scheduler decides to migrate a task to another host, it issues the library function `pvm_move()` to activate the migration of a task to a selected host. Section 3.3 describes the migration protocol in more detail.

The implementation of the DynamicPVM scheduler, discussed here, collects load information from the hosts in the host list. From the load information and the list of tasks, it selects candidates for migration and decides on the destination hosts. In this ranking process the task/processor workload is taken into account to strive for load sharing. Initial placement of tasks is still carried out in a round-robin assignment by the *pvmd* at which the task is spawned.

3.2 Consistent Checkpointing Through Critical Sections

To implement dynamic load balancing by task migration, the run-time support system must be able to create an image of the running process, the so-called *checkpoint*. A checkpoint of an active process consists of the state and data of the process, together with some additional information to recreate the process. To incorporate file I/O migration, the state vector also includes information about open files together with their modes, file descriptors, etc. The text segment of the active process is taken from the executable file, and is therefore not part of the checkpoint.

A complication with checkpointing communicating PVM tasks, is that the state of the process also includes the communication status of the socket con-

nections. Thus, to save the state of the process, the interprocess communication must also be in a well-defined state. Since suspension of the related communicating task is not desirable, the task should not be communicating with another task at the moment a checkpoint is created. To prohibit the creation of process checkpoints during communication, we apply the notion of *critical sections* and embed all interprocess communication operations in such sections. Checkpointing can only take place outside a critical section. When a checkpoint signal arrives during the execution of a critical section, the checkpointing is deferred till the end. We have implemented the checkpoint facility with two different strategies for storing the process's state and data: *direct* and *indirect*.

With direct checkpointing, the destination host opens a TCP connection to the host where the checkpoint is migrated from, and reads the process's state and data. Indirect checkpointing on the other hand, creates a dump of the process's state and data to a file on a shared (e.g., NFS-mounted) file system. With the current network bandwidth limitation, the direct checkpointing strategy is twice as fast as the indirect checkpoint strategy, because it involves only one transfer of the migrating process, compared to two transfers (write/read) when using a file system checkpoint (see also Section 4.3). The advantage of using a checkpoint to a file system is that the process can be restarted on another host at a later stage.

3.3 The Migration Protocol

The main objective of the DynamicPVM migration facility is transparency of the migration protocol, i.e., to allow for the movement of tasks without affecting the operation of other tasks in the system. With respect to the individual task selected for migration this implies transparent suspension and resumption of execution: the task has no notion that it is migrated to another host, and the communication can be delayed without failure triggered by migration of one of the tasks.

In the task migration protocol we distinguish five phases:

(i) create new process context at destination host;
(ii) disconnect task from its local *pvmd*;
(iii) checkpoint task;
(iv) move task to its new host;
(v) restart and reconnect the task to its new *pvmd*.

The first step in the migration protocol is the creation of a new process context at the destination host by sending a message to the *pvmd* representing that host. Next, the master *pvmd* updates its routing table to reflect the new

location of the task, see also Section 3.4. Before the task selected for migration is suspended, the communication between this task and its *pvmd* has to be flushed. Then the task is disconnected from its local *pvmd* and messages arriving for that task are refused by the task's original *pvmd*. The master *pvmd* will now broadcast the new location to all other *pvmds*, so that any subsequent message is directed to the task's new location.

The next phase is the actual migration of the process. As stated in the previous section, there are two checkpoint strategies to experiment with: direct and indirect. The newly created process on the destination host is requested to restart the checkpoint. If direct checkpointing is used, it opens a TCP socket and waits for the original task to begin transmission of the checkpoint. Using indirect checkpointing, the task opens the checkpoint file and reads the checkpoint from disk.

Finally, after the checkpoint is read, the original state of the task (among which data, stack, signal mask, and registers) is restored and the task is restarted with a `longjmp`. Any message that arrived during the checkpoint/migration phase is then delivered to the restarted task.

3.4 Packet Routing

In PVM the task identifier, *task id* for short, is a unique identifier which serves as the task's address and therefore may be distributed to other PVM tasks for communication purposes. For this reason the *task id* must remain unchanged during the lifetime of a task, even when the task is migrated.

This has implications for the packet routing of messages. The *task id* contains the host identifier at which the task is enrolled and a task sequence number. This information is used by the *pvmd* to route packets to their destination, i.e., to the appropriate *pvmd* and task. When a task is migrated to another host, this routing information is not correct anymore. Therefore, an additional routing functionality must be incorporated in the *pvmd* routing software in order to support the migration of tasks. An important design constraint is that the routing facility must be highly efficient and should not impose additional limitations on the scalability.

To provide transparent and correct message routing with migrating tasks, the *task ids* must be made location independent, thus by virtualizing the *task ids*. This is accomplished by maintaining additional routing information tables contained by all *pvmds* (see Fig. 2). These routing tables are consulted for all inter-task communication. Upon migration of a task, first the routing table of the master *pvmd* is updated to reflect the change in location of the migrated task. Next, the master *pvmd* broadcasts the routing table change to all other

pvmds, such that each routing table reflects the actual location of all migrated tasks in the system. Figure 2 depicts the migration of a task attached to pvmd B and the subsequent routing table update.

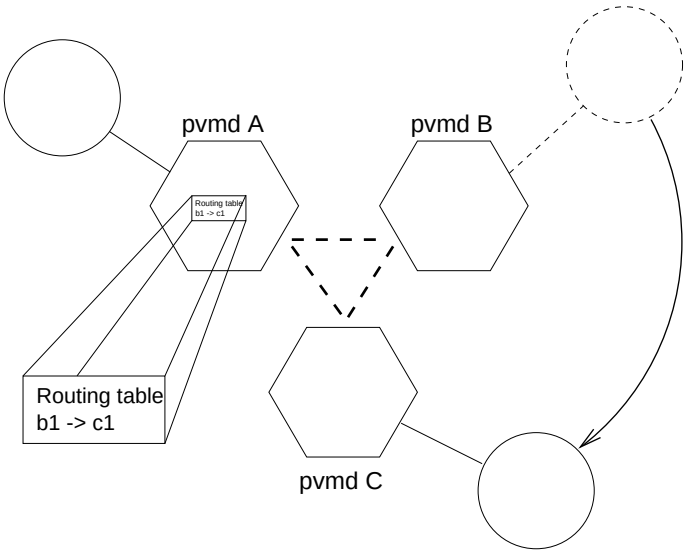

Fig. 2. Routing tables keep track of the migrated tasks.

4 Experiments and Results

DynamicPVM is currently implemented for networks of IBM AIX/32 machines [4], Sun workstations operating under SunOS4 and Solaris [15], and PC's running Linux. It supports only homogeneous checkpointing and migration, because the formats of the checkpoints (the "layout" of the processes) for AIX/32, SunOS4, Solaris, and Linux can not be interchanged. As a result, a task running on a Sun workstation operating under Solaris can only be migrated to another Sun workstation operating under Solaris. Migration between a Solaris workstation and a SunOS4 workstation is not supported (neither between Solaris and AIX/32).

4.1 The One-Factor Experiment

One-factor designs are used to compare two systems that differ one categorical variable, here the standard PVM system and the DynamicPVM system. Techniques to analyze this one-factor experiments, in order to decide whether the observed difference is due to induced differences among the systems or due to experimental errors, are presented in this section.

The model used for single-factor design is [6],

$$y_{ij} = \mu + \alpha_j + e_{ij} \tag{1}$$

Here y_{ij} is the ith response of design j, μ is the mean response, α_j is the effect of design j, and e_{ij} is the experimental error.

The model enables analysis of the origin of the variance, whether it stems from α_j or e_{ij}. The total variation of y in a one-factor design accumulates in the effect factor α_j and the error term e_{ij}. If we square both sides of the model equation, the sum of squares can be written as,

$$\text{SSY} = \text{SS0} + \text{SSA} + \text{SSE}$$

where $\text{SSY} = \sum_{i,j} y_{ij}^2$, $\text{SS0} = \sum_{i,j} \mu^2$, $\text{SSA} = \sum_{i,j} \alpha_j^2$, and $\text{SSE} = \sum_{i,j} e_{ij}^2$. If we design our experiment such that the effects of α_j and e_{ij} add up to zero, then the cross-product terms of Eq. (1) are also equal to zero.

Now we define the quantity *total sum of squares* (SST) by:

$$\text{SST} = \sum_{i,j} (y_{ij} - \mu)^2 = \text{SSY} - \text{SS0} = \text{SSA} + \text{SSE} \tag{2}$$

Although SST is different from the variance of y, it is a measure of y's variability and is called the *variation* of y. Equation (2) shows that the total variation is determined by two parts: SSA representing the known part (due to different systems) and SSE representing the unknown part (due to experimental errors) of the variation.

The *significance* of the known part of the variation is determined by comparing its contribution to the total variation with that contributed by the errors. The F-test is applied to check if SSA is significant larger than SSE and to derive whether the observed difference is due to significant differences among the systems rather than experimental errors.

In the example of our two system experiment, described in the next subsection, we have the following instantiation of Eq. (1):

- $j = 1$ indicates the PVM system;
- $j = 2$ indicates the DynamicPVM system;
- i indicates one experiment, consisting of 1000 observations;
- the values of e_{ij} follow a normal distribution with mean $\bar{e}_{ij} = 0$ and standard deviation $\sigma_{e_{ij}}$; $\sigma_{e_{ij}}$ is tested in the F-test;
- α_j indicates the difference between the $j = 1$ and the $j = 2$ experiment, and the μ; therefore $\sum_{i,j} \alpha_j = 0$.

A well-known method to measure the basic communication properties of a message-passing systems is the *ping-pong* experiment. With the ping-pong experiment, series of messages of different sizes are sent between two tasks: one master and one slave. The master sends the message to the slave, the slave receives the message into a buffer, and immediately returns it to the master. Half the time of this message ping-pong is recorded as the time t to send a message of length n.

In this sense, the ping-pong experiment is a suitable benchmark to determine the overhead introduced by the DynamicPVM implementation. A serious problem in benchmarking systems in dynamically changing environments such as a network of workstations, is that one does not have control over all the factors influencing the measurements, for example network and processor load. Here however, we can design the experiment such that it circumvents this problem by performing the measurements in series of equally loaded workstation environments. In addition, detailed analysis of the results is necessary to preclude experimental errors.

The ping-pong experiment was performed for both the public domain PVM implementation as well as the DynamicPVM implementation. The experiments were executed on a lightly loaded system of SparcStation4 workstations connected by a 10Mb/s Ethernet. The data were analyzed according to the techniques described in Section 4.1. All reported experiments passed the null hypothesis of the F-test.

For each message size, 30 observation were collected for both PVM and DynamicPVM. Each individual observation consists of 1000 ping-pong measurements during "low" network load. The ratio behind this is that the network load qualification of "low" is not well defined. One series of 1000 ping-pong measurements with low network load, results in a observation for message size n for one specific low network load. By repeated series over different low network loads, we obtain different observations with some variation. The results of the ping-pong experiments shown in Table 2 are the *grand mean* of these observations.

The same experiments were performed for "medium" network load. Again for each message size, 30 observations for both PVM and DynamicPVM were collected; the resulting grand means are shown Table 3.

In Fig. 3 we show the α_2 values for DynamicPVM obtained from the one-factor experiment. The data used in the model are obtained from the low network load ping-pong experiment (see Table 2). The figure depicts the categorical difference in ping-pong results between PVM and DynamicPVM for increasing

Size (bytes)	PVM (μsec)	DynamicPVM (μsec)	Size (bytes)	PVM (μsec)	DynamicPVM (μsec)
1	7268	7619 (4.8%)	2048	9853	10594 (7.5%)
4	7170	7568 (5.6%)	4096	14904	15856 (6.4%)
8	7196	7680 (6.7%)	8192	22091	23248 (5.2%)
16	7170	7626 (6.4%)	16384	36980	38437 (3.9%)
32	7172	7794 (8.7%)	32768	65150	67808 (4.1%)
64	7236	7723 (6.7%)	65536	120756	126199 (4.5%)
128	7288	7913 (8.6%)	131072	232304	242082 (4.2%)
256	7676	8137 (6.0%)	262144	453843	475323 (4.7%)
512	7844	8454 (7.8%)	524288	900094	937909 (4.2%)
1024	8482	9048 (6.7%)	1048576	1785052	1854352 (3.9%)

Table 2
PVM and DynamicPVM ping-pong results for low network load. The percentages are the overhead induced by DynamicPVM.

Size (bytes)	PVM (μsec)	DynamicPVM (μsec)	Size (bytes)	PVM (μsec)	DynamicPVM (μsec)
1	7425	7612 (2.5%)	256	7817	8017 (2.6%)
4	7372	7660 (3.9%)	512	8311	8519 (2.5%)
8	7367	7635 (3.6%)	1024	9232	9426 (2.1%)
16	7402	7588 (2.5%)	2048	11151	11322 (1.5%)
32	7432	7611 (2.4%)	4096	17081	17398 (1.8%)
64	7489	7652 (2.2%)	8192	25623	26149 (2.0%)
128	7570	7806 (3.1%)	16384	43328	43689 (0.8%)

Table 3
PVM versus DynamicPVM ping-pong results for medium network load. The percentages are the overhead induced by DynamicPVM

message length. Due to the definition of α_j with respect to μ, we have $\alpha_1 = -\alpha_2$.

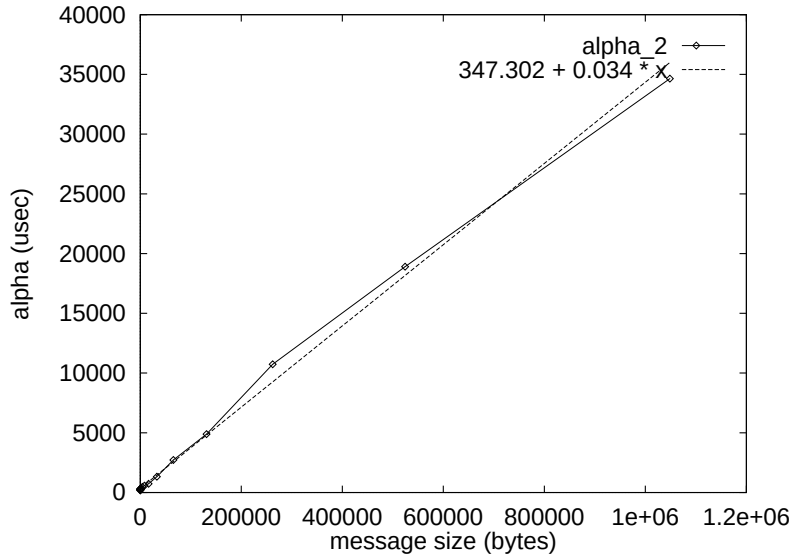

Fig. 3. The α_2 values for DynamicPVM from the one-factor ping-pong experiment.

4.3 Checkpoint Overhead

Figure 4 shows some results obtained by migrating a 75 Kbyte process with data segments of various sizes in both direct and indirect checkpointing mode. As can be seen in the figure, the time needed for the migration is linear to the size of the program.

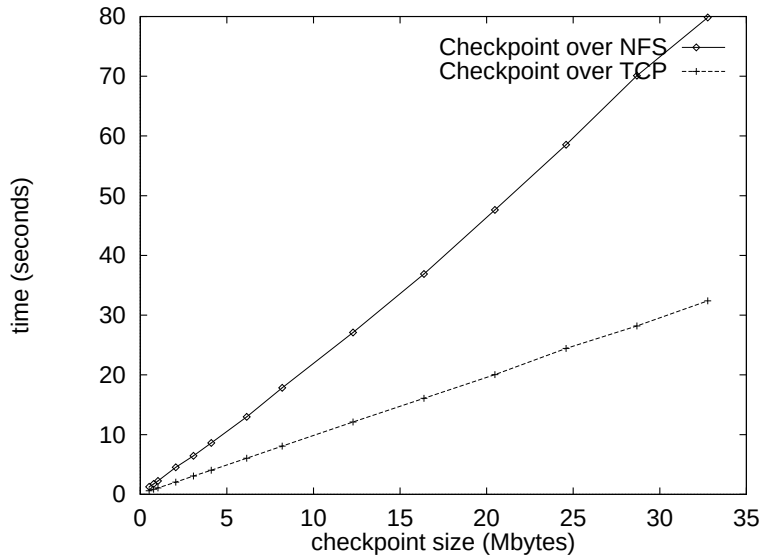

Fig. 4. Migration times (in seconds) for checkpointing using NFS and direct TCP.

The systems used in these tests had enough free physical memory to restart the checkpoint without swapping pages to disk. If a process needs to swap pages to secondary storage, performance drops dramatically (data not shown).

The NAS Parallel Benchmarks (NPB) is a suite of applications used by the Numerical Aerodynamic Simulation (NAS) Program at NASA for the performance analysis of parallel computers. The benchmark suite consists of five "kernels" and three simulated applications which mimic the computational behaviour of large scale computational fluid dynamics applications. A unique property of the NPB is that the applications are specified algorithmically. The implementation of the NPB kernels used in the experiments with DynamicPVM are described in [18].

The specific NBP kernels used in the performance analysis of PVM and DynamicPVM are:

EP The Embarrassingly Parallel kernel is based on a trivial partitionable problem requiring little or no communication between processors.

MG The 3-D Multigrid Solver is characterized by highly structured short- and long-distance communication patterns.

CG The communication patterns in the Conjugate Gradient kernel are long-distance and unstructured.

FT In the 3-D Fast Fourier Transformation, the communication patters are structured and long distance.

The experiments were performed on two sets of eight "approximate equally loaded" Sparc Classic's during daytime. One set was reserved for PVM measurements and one set was reserved for DynamicPVM. The DynamicPVM tasks are migrated to lightly loaded workstations, if available. The checkpoints are made to disk, thus two times slower than the TCP checkpoint.

Benchmark	PVM	DynamicPVM			Speedup
	time	time	migrations	chkp. size	
EP	19:11	16:51	4	100 KB	1.14
FT	57:27	52:09	1	25 MB	1.10
MG	48:01	42:40	2	2.5 MB	1.13
CG	21:26	19:37	5	9 MB	1.09

Table 4
Execution times of PVM versus DynamicPVM.

Simulated Annealing (SA) is a technique for optimization problems of very large scale. In many typical optimization problems one wants to find among many configurations one configuration which minimizes or maximizes a certain *cost function*. In our application problem we study the crystallization of N randomly placed particles on a virtual supporting sphere. The particles interact with each other according the Lennard-Jones potential [16].

Sequential Simulated Annealing The annealing process begins by creating a Markov chain, of given length, at a certain temperature. The Markov chain grows by randomly displacing particles and calculating the corresponding change in energy of the system and deciding on acceptance of the displacement.

After a chain has ended, the temperature is lowered by multiplying the temperature with the cool-rate, which is a number slightly less than 1 (typically 0.9) after which a new chain is started. This process continues until a stop criterion is met. The stop criterion in our implementation is met when the standard deviation in the final energies of the last ten chains falls below a certain value (typically 10^{-6}).

Systolic Simulated Annealing A synchronous algorithm that does not mimic sequential SA is systolic SA [1,13,17]. In systolic SA a Markov chain is assigned to each of the available processors. All chains have equal length. The chains are executed in parallel and during execution information is transferred from a given chain to its successor. Each Markov chain is divided into a number of subchains equal to the number of available processors. The execution of chain $k+1$ is started as soon as the first subchain of chain k is generated. Equilibrium is not yet established by then. Quasi-equilibrium of the system is preserved by adopting intermediate results of previous Markov chains.

The experiments with the systolic SA application were executed on two pools of Sun SparcStation LX workstations. The PVM pool consisted of three workstations, and the DynamicPVM pool of six workstations. Table 5 shows the turn-around times of the systolic SA algorithm for PVM and DynamicPVM. The systolic SA problem size is determined by the number of particles, N, and the number of iterations.

The progress of the systolic SA algorithm for PVM and DynamicPVM is depicted in Fig. 5. Progress is defined in terms of the number of temperature cooling steps, i.e., when a new Markov chain is started.

N	iterations	PVM	DynamicPVM	Migrations	Speedup
20	5	0:14:14	0:13:11	1	1.08
20	10	0:31:20	0:28:29	3	1.10
20	25	1:23:42	1:17:02	3	1.09
40	5	1:34:19	1:24:50	2	1.11
40	10	3:21:55	2:28:49	1	1.36
40	25	7:49:28	7:02:03	4	1.11
60	5	3:38:20	3:07:23	2	1.17
60	10	7:22:00	6:36:14	2	1.12
60	25	17:31:46	16:12:44	3	1.08

Table 5
Turn-around times of the systolic SA algorithm for different problem sizes.

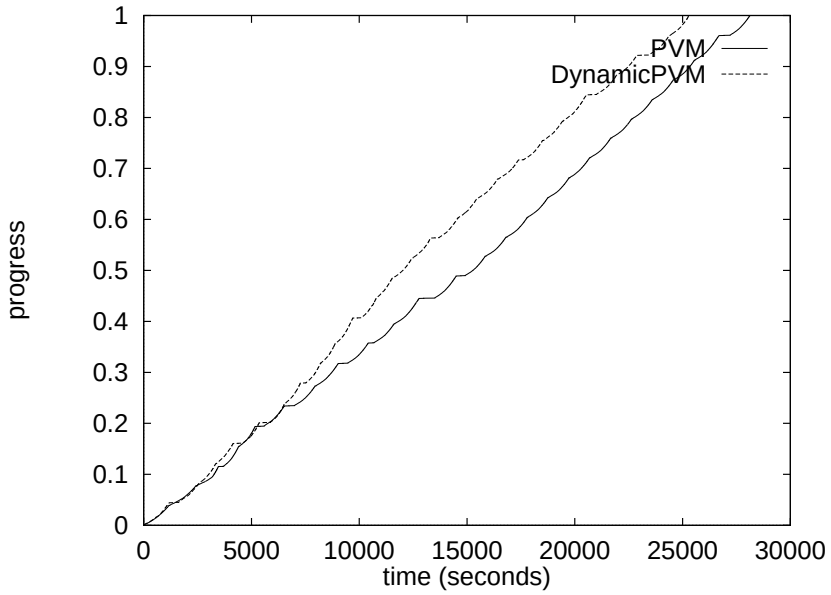

Fig. 5. Progress in time of the systolic SA application for and PVM and DynamicPVM. Problem size is $N = 40$ and iterations $= 25$.

The mean CPU load for the PVM and DynamicPVM clusters are shown in Fig. 6. For the PVM cluster, the mean CPU load is calculated by taking the *mean* of the CPU load of the three workstations. The mean CPU load for the DynamicPVM cluster is computed by taking the *mean* of the CPU load of the three workstations *currently* executing a DynamicPVM task.

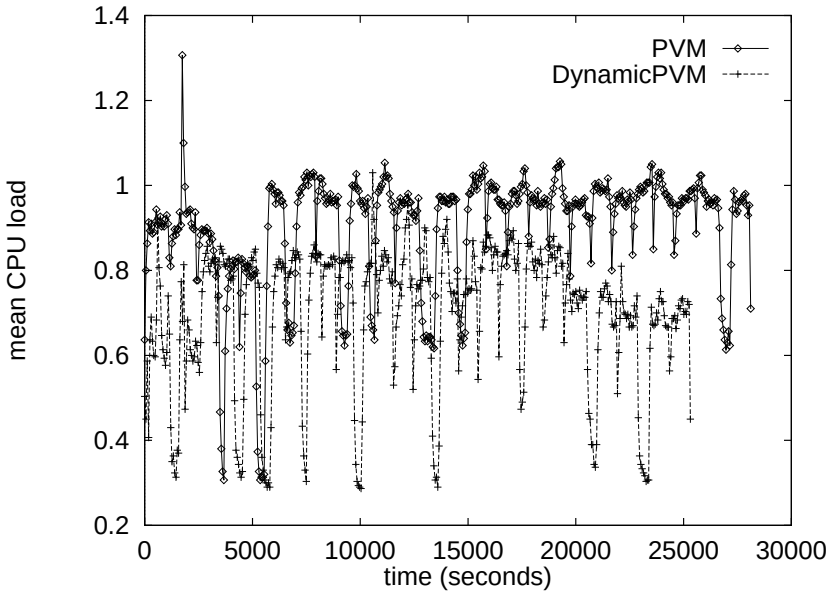

Fig. 6. Mean CPU load factors of PVM and DynamicPVM clusters.

5 Discussion and Conclusions

The results of the ping-pong experiment shown in Table 2 and Table 3 indicate that the percentage overhead of DynamicPVM is almost constant (about 5% for low network load and 2.5% for medium network load). The difference in overhead between low and medium network load can be explained since the overhead we measure is the accumulated effect of network overhead and DynamicPVM overhead. The relative DynamicPVM overhead becomes more predominant for low network load.

Figure 3 depicts how the absolute α-values increase with increasing message size. The increase in absolute overhead is due to the routing table lookup for each packet sent between two *pvmds* in DynamicPVM. As the number of bytes increases, the number of packets sent also increases (see Section 3.4). The overhead can be tuned by changing the DynamicPVM message fragment size. By increasing the packet size, a smaller number of routing table lookup operations are necessary per message sent. However, the overhead for small messages increases, as the packets sent between *pvmds* will be largely empty.

Another overhead factor introduced by DynamicPVM, is task migration. This effect is studied in the checkpoint overhead experiment. The results for indirect (NFS) and direct (TCP) checkpointing are shown in Fig. 4. The migration times for indirect and direct checkpointing are linear with respect to the size of the checkpoint. The migration using NFS takes about twice as long as the migration over TCP, which is due to the fact that migration over NFS requires a separate write and read cycle, while in direct mode the write and read are overlapped. Nonetheless, both migration modes can be efficiently imple-

mented given the underlying protocol. For direct checkpointing, the measured throughput is almost 1 MB/s, while the bandwidth of Ethernet is 1 MB/s. With indirect checkpointing a throughput of about 450 KB/s is measured.

The computational kernels from NAS Parallel Benchmark suite represent different application behaviour in terms of computational and communication behaviour. Although the execution behaviour of the kernels are different, the advantage (speedup) gained with DynamicPVM is within the same range, see Table 4. This indicates that the experimental DynamicPVM vehicle is able to use the potential of idle workstations as computational resources, independently of the static or dynamic execution behaviour of the application. Even a memory intensive application, such as the FT kernel (25 MB checkpoint), profits from one task migration to an idle workstation.

An interesting case study is the systolic simulated annealing algorithm. In Fig. 5 the progress of the simulated annealing process is displayed in time, for both PVM and DynamicPVM. The corresponding mean CPU load of the workstations is depicted in Fig. 6. Noticeable is the correspondence between the application activity and the mean CPU load. The test runs consisted of 25 iterations that are coordinated by one task. This can be found in Fig. 5, where progress retards for a period after an iteration. Figure 6 displays this by a drop in the measured mean CPU load. The same figure shows that the mean CPU load for DynamicPVM is lower than for PVM. The net effect is a smaller turn-around time.

We conclude therefore:

- The communication and checkpoint overhead experiments show that the experimental DynamicPVM system provides efficient task migration support.
- The results of the NAS Parallel Benchmarks and the systolic SA case study show that the DynamicPVM system is able to exploit the potential of idle workstations, by (re-)mapping of a dynamic resource application onto a dynamic resource machine, irrespective of the behaviour of the kernels.
- The DynamicPVM functionalities are provided through libraries, thus hiding the complexity of the load balancing process from the end-user. The resulting transparent appearance of adaptive systems such as DynamicPVM, lowers the barrier to cluster computing.

This pilot study indicates that dynamic resource management on task level for parallel jobs is a promising approach to efficiently balance load in clusters of workstations.

6 Future Work

The research presented here indicate that DynamicPVM is an adequate research vehicle to study different approaches to dynamic resource management of parallel jobs in cluster environments. One of the open issues is the development of *true* heterogeneous DynamicPVM: tasks moving from one architecture to another. This heterogeneous task migration is a difficult problem to solve at operating system level [5]. At user level, we can take an object-oriented approach, and implement process/object migration functionalities into the *libc*. The advantage of incorporating the generic process/object migration into the *libc*, is that other message passing interfaces, such as MPI, can make use of the offered facilities.

In addition, the DynamicPVM vehicle allows for experimental research in exploring various scheduling mechanisms: the DynamicPVM system offers *efficient* support for task migration, but the *effective* use of the dynamic resources depends on an intelligent scheduler.

The experiments described in the paper can be characterized as loosely synchronous computations. For DynamicPVM (loosely) synchronous behaviour is a worst case scenario, because the overall computation stalls during the migration of one of the tasks. In the future we will explore the potentials of the experimental DynamicPVM system by a more general study with fully asynchronous massively parallel applications. Particularly, current research is directed to optimistic parallel discrete event simulation methods, such as the Time Warp protocol [7,10]. It is identified that a serious limitation in the successful application of the Time Warp protocol are the extreme computation requirements. The complete non-deterministic asynchronous execution behaviour of the Time Warp protocol makes it a highly dynamic resource problem. The load balancing of this type of asynchronous systems is a specific merit of the experimental DynamicPVM system.

References

[1] E. H. L. Aarts, F. M. J. de Bont, E. H. E. Habers, and P. J. M. van Laarhoven. Parallel implementations of the statistical cooling algorithm. *INTEGRATION, the VLSI Journal*, 4:209–238, 1986.

[2] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *Computer*, 22(9):47–56, Sept. 1989.

[3] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *Third Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

[4] L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot. DPVM: Dynamic load balancing on parallel systems. In *High Performance Computing and Networking*, pages 273–277, 1994.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice & Experience*, 21(8):757–785, Aug. 1991.

[6] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., New York, 1991.

[7] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[8] M. Litzkow, M. Livny, and M. W. Mutka. Condor—a hunter of idle workstations. In *8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[9] D. S. Milojicic, W. Zint, A. Dangel, and P. Giese. Task migration on the top of the Mach microkernel. In *MACH III Symposium Proceedings*, pages 19–21, Santa Fe, New Mexico, Apr. 1993.

[10] B. J. Overeinder and P. M. A. Sloot. Application of Time Warp to parallel simulations with asynchronous cellular automata. In A. Verbraeck and E. J. H. Kerckhoffs, editors, *Proceedings of the 1993 European Simulation Symposium*, pages 397–402, Delft, The Netherlands, Oct. 1993.

[11] Sun Microsystems, Inc. *XDR: External Data Representation Standard*, 1987.

[12] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, Apr. 1994.

[13] A. ter Laak, L. O. Hertzberger, and P. M. A. Sloot. Nonconvex continuous optimization experiments on a Transputer system. In A. Allen, editor, *Transputer Systems – Ongoing Research*, WoTUG 15, pages 251–265. IOS Press, Apr. 1992.

[14] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemtable remote execution facilities for the V-System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 2–12, Orcas Islands, Washington, Dec. 1985.

[15] J. J. J. Vesseur, R. N. Heederik, B. J. Overeinder, and P. M. A. Sloot. Experiments in dynamic load balancing for parallel cluster computing. In P. Fritzon and L. Finmo, editors, *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, pages 189–194, Linköping, Sweden, 1995.

[16] J. M. Voogd, P. M. A. Sloot, and R. van Dantzig. Simulated annealing for N-body systems. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking*, number 796 in Lecture Notes in Computer Science, pages 293–298, Berlin, Germany, 1994. Springer-Verlag.

[17] J. M. Voogd, P. M. A. Sloot, and R. van Dantzig. Comparison of vector and parallel implementations of the simulated annealing algorithm. *Future Generation Computer Systems*, 11(4–5):467–475, Aug. 1995.

[18] S. White, A. Ålund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, Apr. 1995.