Experiments in Dynamic Load Balancing for Parallel Cluster Computing

J.J.J. Vesseur R.N. Heederik B.J. Overeinder P.M.A. Sloot
Parallel Scientific Computing and Simulation Group
Department of Mathematics and Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ, Amsterdam

June 21, 1995

1 Introduction

In academic and industrial institutions, a shift of emphasis in High Performance Computing from parallel monolithes to clusters of high performance workstations is taking place. These loosely coupled parallel systems require new programming paradigms and environments that provide the user with tools to explore the full potential of the available distributed resources.

Although such cluster computing systems provide the user with large amounts of processing power, their usability and efficiency is mainly determined by environmental changes like variation in the demand for processing power and the varying number of available processors.

To optimize the resource utilization under these environmental changes it is necessary to migrate running tasks between processors, i.e., to perform dynamic load balancing. We introduced a scheduling mechanism in PVM that supports such load balancing for parallel tasks running on loosely coupled parallel systems. The enhanced system is called *DynamicPVM*.

Our primary objective is to study models describing adaptive systems like DynamicPVM. To validate these models, experiments with actual implementations of such dynamic systems are required. The work presented here reports on a pilot implementation of DynamicPVM.

The choice for PVM [5] as the basic parallel programming environment is motivated by the fact that PVM is the most widely used environment to date and is considered the *de facto* standard. The process migration primitives used in DynamicPVM were initially based on the checkpoint-restart mechanisms found in a well established global scheduling system, Condor [3] but have been replaced our own routines order support our pool of Solaris workstations and to reduce checkpoint overhead.

Table 1 shows different aspects of load managing for the three systems discussed in this paper. We use the term *job* to indicate the largest entity of execution (program) consisting of one (serial program) or more cooperating *tasks* (parallel program).

	Condor	PVM	DynamicPVM
intended usage	longer running background jobs	parallelized distributed application programs	
unit of execution	job	task	
load managing objective	load distribution	load decomposition	both
schedule policy	dynamic load sharing	cyclic allocation	dynamic load balancing
schedule objective	resource utilization	application response time	both
performance objective	efficiency	effectiveness	both

Table 1: Different aspects of load managing for Condor, PVM, and DynamicPVM.

2 PVM: Runtime Support System for Parallel Programs

PVM (Parallel Virtual Machine) provides primitives for remote task creation and Inter Process Communication (IPC). It supports both point to point and global communication primitives. Tasks are assigned to available processors using a cyclic allocation scheme. Jobs are placed statically, i.e., once a job is started, it runs on the assigned processors until completion.

Each processor in the PVM pool is represented by a *daemon* that takes care of task creation and all IPC to and from tasks running on the processor. To enable the use of heterogeneous processor pools, messages are encoded using an external data representation (XDR [4]). With the current version PVM (3.3.x) direct IPC between two PVM processes, without interference of the PVM daemons is supported, thereby enhancing communication performance.

3 Condor: Runtime Support for Job Scheduling

The Condor system stems from the observation that many of the—constantly increasing number of—workstations in academic and industrial institutions are lightly loaded on the average. Most workstations are intended for personal usage, which has a typical activity pattern where machines are only used for a small part of the day. As a consequence many computing cycles are unused during the day. Typical figures of large pools of workstations have a mean idle time of 80% [3].

To address this problem, Condor implements a global scheduling based on dynamic load balancing by job migration. Condor monitors the nodes in its pool by keeping track of their load. New jobs are spawned on lightly loaded nodes and jobs from heavily loaded machines can be migrated to less loaded ones. When Condor detects interactive usage of a workstation all Condor jobs can be evacuated from that workstation in order to retain the sympathy of the workstation's owner. To implement this job migration Condor creates checkpoints on a regular basis, which can be restarted on another machine.

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 optimize the available processing power.

By automatically redirecting all system calls made by a Condor job to the machine that initiated the computation, the migration of jobs is made completely transparent to the programmer. In this way the programmer is freed from the complications of checkpointing.

Using Condor, it is not possible to migrate jobs consisting of cooperating parallel tasks since Condor does not provide any support for IPC primitives.

Combining PVM with an extended version of Condor's checkpoint-restart facility makes it possible to apply global scheduling to parallel tasks.

4 DynamicPVM: Runtime Support System for Job Scheduling Parallel Tasks

In DynamicPVM we add checkpoint-restart mechanisms to the PVM environment. Most of PVM's features are compatible with the checkpoint-restart mechanism we use and can be incorporated in DynamicPVM without problems. The Inter Process Communication is an exception to this rule.

We present a protocol that ensures that no messages get lost whenever a task is migrated. This protocol involves a special role for the PVM daemon that initiated the computation, the *Master daemon*. We also present an extension to the PVM IPC routing mechanism in order to redirect messages to tasks that are migrated.

DynamicPVM's task migration facility consists of four principal components:

1. A global scheduler that initiates job migration (not addressed in this paper).
2. Task checkpointing, including a method to indicate checkpoint save moments.
3. Actual task migration.

4. Task restart and updating of routing tables to reflect the task's new location.

These components are briefly described below.

4.1 Task Checkpointing

In order to migrate a process, a dump of the process' data and state, together with some additional information to recreate the process, has to be made. We have implemented two different strategies for dumping this information: *direct* and *indirect*.

Using direct checkpointing, the host where the checkpoint is migrated from opens a TCP connection to the destination host and writes the process' data and status to the destination host.

With indirect checkpointing, a dump of the process' state and data is made to a shared (NFS-mounted) file system. In this way, the process can be restarted by a machine at a later stage.

Since direct checkpointing involves only one transfer of the migrating process, compared to two transfers (write/read) when using NFS it is approximately twice as fast.

Checkpointing cooperating tasks introduces new conditions as compared to checkpointing stand alone tasks. For instance, checkpoints should be avoided when a task is communicating with another task. To safely checkpoint DynamicPVM tasks, we introduce the notion of a *critical section* and embed all IPC operations in such sections. Checkpointing is prohibited whenever the task is in a critical section; checkpointing can only take place when the task is *not* participating in a communication operation.

4.2 Task Migration

The main demand on the DynamicPVM task migration facility is transparency, i.e., to allow the movement of tasks without affecting the operation of other tasks in the system. With respect to a PVM task selected for migration, this implies transparent suspension and resumption of execution. With respect to the total of cooperating PVM tasks in a job, communication can be delayed due to the migration of one of the tasks.

The first step of the migration protocol is to create a new, empty, process context at the destination processor by sending a message to the daemon representing that node.

Next, the Master-Daemon updates it's routing tables to reflect the new location of the process. The task to be migrated is suspended and messages arriving for that task are refused by the task's original daemon. Such messages are queued by the sending daemon, to be processed after the new location has been broadcasted.

In the next phase, the Master-Daemon broadcasts the new location to all nodes, so that any subsequent messages are directed to the task's new location.

The last phase is the actual migration of the process. As stated in the previous section, there are two strategies implemented and the user can choose the appropriate mechanism.

4.3 Task Restart

The newly created process on the destination processor is requested to restart the checkpoint. If direct checkpointing is used, it opens a TCP socket and waits for the checkpointing task to begin transmission of the checkpoint.

Using indirect checkpointing, the task opens the checkpoint file and reads the checkpoint from disk.

After the checkpoint is read, the original state of the process is restored (data/stack/signal mask/registers) and the process is restarted with a `longjmp`. Any messages that arrived during the checkpoint-restart phase are then delivered to the restarted process.

5 Current Status and Results

DynamicPVM is currently implemented on a cluster of IBM RS/6000, AIX32 machines [1, 2] and cluster Sun workstations, operating under SunOS4 and Solaris. The pilot implementation presented is not yet able to migrate tasks that perform file-I/O.

5.1 Checkpointing

Table 2 shows some results obtained by migrating a 75Kbyte process with data segments of various sizes in both direct and indirect mode. As can be seen in the table, the time needed for the migration is linear to the size of the program. The migration using NFS takes 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. The data is displayed in Fig. 1.

Data Size (Mb)	Direct (sec)	Indirect (sec)
0	0.1	0.3
0.5	0.6	1.0
1	1.1	1.6
2	2.1	2.9
4	4.1	5.8
8	7.9	12.5
12	11.9	19.8
16	15.9	29.0

Table 2: Times needed for migration of tasks for various sizes. These times were acquired on lightly loaded machines (Sun SPARCs running Solaris 2.4), with little network traffic.

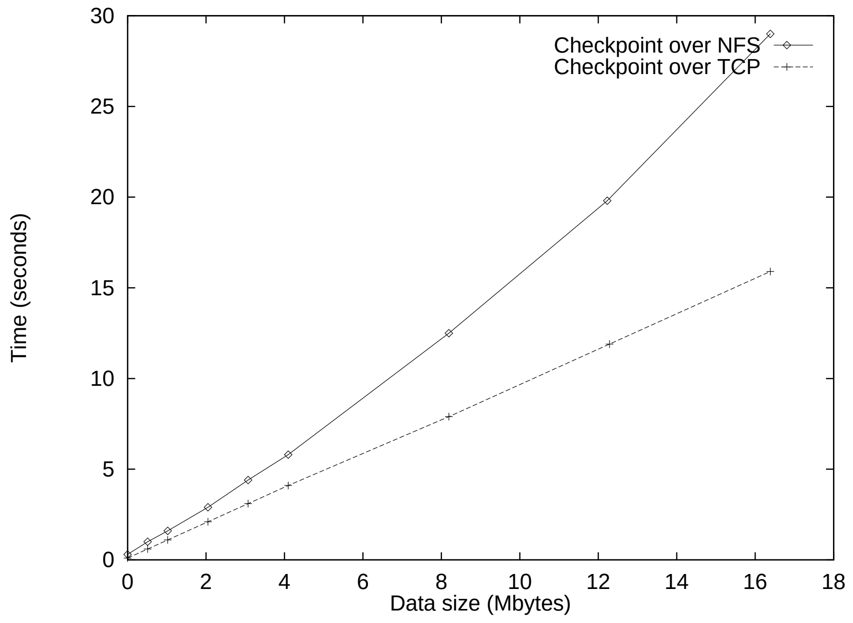

Figure 1: Migration times (in seconds) for checkpointing using NFS and direct TCP/IP.

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 with approximately 50%.

5.2 Real Applications

Our first test results are obtained with PVM implementations of the NAS Parallel Benchmarks (NPB), see Table 3. The typical applications within the NPB are:

EP Embarrassingly Parallel

FT 3-D Fast Fourier Transformation

MG 3-D Multigrid Solver

CG Conjugate Gradient

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/IP checkpoint.

Benchmark	PVM	DynamicPVM			Speedup
	time	time	migrations	chkp. size	
EP	13:34	10:55	4	1300K	1.24
FT	3:09	3:13	0	9500K	0.98
MG	48:01	42:40	2	2500K	1.13
CG	21:26	19:37	5	9000K	1.09

Table 3: Execution times of PVM versus DynamicPVM.

From Table 3 one can see that during the execution of the FT benchmark no migrations were performed. This is due to the fact that no workstations were available to migrate to, but also because the execution run is actually to short to take advantage of the characteristics of DynamicPVM. What is does show, is that the overhead of DynamicPVM is only within 2%. In general, better results in terms of speedup and resource utilization can be achieved by introducing a more advanced schedular that makes well-thought decisions on the placement of tasks over the workstations, and by experimenting with long running jobs of several days instead of minutes.

6 Conclusions

Tests performed with the pilot implementation indicate the usability of the integrated approach. In the near future we will design additional experiments for quantitative testing of the behaviour of DynamicPVM as well as extend the checkpoint mechanism to support checkpointing of tasks performing file-I/O.

Our final goal is to develop probabilistic models of jobs in a dynamic environment and validate them by experiments with an actual implementation. We have added several primitives to PVM that allow us to monitor PVM tasks. Using these primitives we will implement a scheduler based on these models. Simulations using our models should provide detailed insight in the dynamics of the behaviour of systems like DynamicPVM and can be used to improve scheduling mechanisms. The actual scheduler is a topic of future research.

References

[1] L. Dikken, “DynamicPVM: Task migration in PVM,” Tech. Rep. TR-ICS/155.1, Shell Research, Nov. 1993.

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

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

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

[5] V. S. Sunderam, “PVM: A framework for parallel and distributed computing,” *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, Dec. 1990.