Experiments with Migration of Message-passing Tasks

K. A. Iskra[1], Z. W. Hendrikse[1], G. D. van Albada[1], B. J. Overeinder[1], P. M. A. Sloot[1] and J. Gehring[2]

[1] Informatics Institute, Universiteit van Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{kamil,zegerh,dick,bjo,sloot}@science.uva.nl
[2] Paderborn Center for Parallel Computing,
Fürstenallee 11, 33102 Paderborn, Germany
joern@uni-paderborn.de

Abstract. The combined computing capacity of the workstations that are present in many organisations nowadays is often under-utilised, as the performance for parallel programs is unpredictable. Load balancing through dynamic task re-allocation can help to obtain a more reliable performance. The Esprit project Dynamite[1] provides such an automated load balancing system. It can migrate tasks that are part of a parallel program using a message passing library. Currently Dynamite supports PVM only, but it is being extended to support MPI as well. The Dynamite package is completely transparent, *i.e.* neither system (kernel) nor application source code need to be modified. Dynamite supports migration of tasks using dynamically linked libraries, open files and both direct and indirect PVM communication. Monitors and a scheduler are included. In this paper, we first briefly describe the Dynamite system. Next we describe how migration decisions are made and report on some performance measurements.

1 Introduction

With the introduction of more powerful processors every year, and network connections becoming both faster and cheaper, distributed computing on standard PCs and workstations of an organisation becomes more attractive and feasible. Consequently, the interest in special purpose parallel machines is declining in favour of the clusters of workstations.

In such environments, performance optimisation and load balancing by off-loading work to other nodes in the cluster is highly desirable. For sequential programs, this has long been solved (*e.g.* in Condor [10, 9] and Codine [17]). For tasks in parallel programs, this still is a research issue. Dynamite [1, 6–8] provides a dynamic load balancing system for parallel jobs running under PVM

[1] Dynamite is a collaborative project, funded by the European Union as Esprit project 23499. Of the many people that have contributed, we can mention only a few: A. Streit, F. van der Linden, J. Clinckemaillie, A. H. L. Emmen.

[5] when run on clusters of workstations. The load balancing is realised through the migration of tasks.

Dynamite is an acronym for DYNAMIc Task migration Environment and is also known as DPVM [11] (Dynamic–PVM), since it is based on PVM, version 3.3.11. Dynamite currently supports PVM-based programs only, but its modular design greatly facilitates the creation of an MPI [16] version. Work on this is currently under way in cooperation with the people from Hector [12]. Various PVM variants supporting task migration have been reported, such as tmPVM [15], ChaRM [4], DAMPVM [3] and MPVM [2].

Dynamite is currently operational under Sun Solaris/UltraSparc 2.5.1 through 8 (32 bit) and Linux/i386 2.0 and 2.2[2]. It aims to provide a complete solution for dynamic load balancing, see Section 6. The strengths of Dynamite are its powerful and versatile checkpointing mechanism, its transparency, its modularity and its robustness.

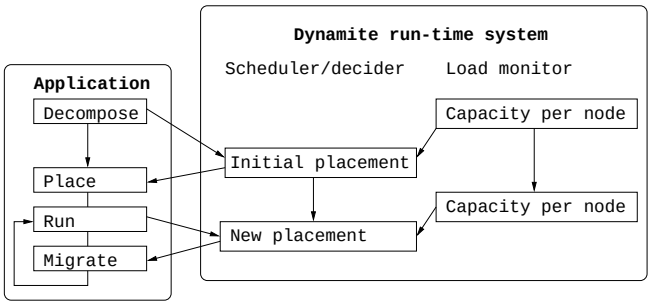

Fig. 1: Running an application with Dynamite. An application has to be decomposed into several subtasks already, and it must be linked with the Dynamite libraries. The run-time system places these on nodes in the cluster, starts execution and monitors the system. If it decides that the load is unbalanced (above a certain threshold), one or more task migrations may be performed to establish a new and more optimal load distribution.

The motivation for a continuous optimal task allocation is three-fold:

- overall performance is determined by the slowest task,
- dynamic run-time behaviour of both task (the amount of computational resources needed by a task) and node (computational resources offered by a node) may vary in time,
- computational resources used by long-running programs might be reclaimed on demand.

The Dynamite system (see Fig. 1) consists of three separate parts:

1. The load-monitoring subsystem. The load-monitor should leave the computation (almost) undisturbed.

[2] Only the libc5 and glibc2.0 libraries are currently supported.

2. The scheduler, which tries to make an optimal allocation.
3. The task migration software, which allows a process to checkpoint itself and to be restarted on a different host. This also has a significant impact on the message-passing libraries. An extensive and detailed description of this part of the system can be found in [8].

Dynamite is required to be as transparent to the user as possible. This implies that the checkpoint/migration mechanism must be implemented completely in user-space and no additional changes to the code of the program may be required. Indeed, the user only has to link to the Dynamite dynamic loader[3] (which contains the checkpoint/restart mechanism and is a shared library itself; it is based on the Linux ELF dynamic loader 1.9.9) and the DPVM library. From then on, the complete Dynamite functionality is available. It is also necessary to use Dynamite's infrastructure (daemons, group server, console and such) as functionality has been added and protocols have been adapted.

Users of sequential programs that do not use PVM can merely link their applications using the Dynamite dynamic loader, thus taking advantage of the checkpoint facility.

First we will describe the architecture of Dynamite in Sections 2 and 3. Thereafter quantitative results will be presented, which have been obtained with Dynamite running on a Linux cluster. These data will be compared to standard PVM runs. Subsequently, we briefly discuss the limitations of Dynamite and come to our conclusions.

2 Checkpointing mechanism

Checkpointing a process boils down to writing the address space of a process to a file and retrieving its contents afterwards. This includes the shared libraries which may be used by the process. In addition, the contents of (some of the) processor registers must be taken care of, such as the program-counter and the stack pointer. Moreover, a proper implementation must also consider various communication channels.

In Dynamite, the checkpointing functionality was implemented in the dynamic loader, to which the following changes were made:

1. it can handle a checkpoint signal (`SIGUSR1`),
2. it can treat a checkpoint file just like any other executable,
3. it *wraps* certain system and library calls:
   - for open files (a.o. `open`, `write`, `creat`),
   - for memory allocation (`mmap`, `munmap`, `mremap`[4]),
4. it preserves certain cross-checkpoint data separately,
5. it provides handles for additional processing before and after checkpoints (*e.g.* for communication libaries).

[3] The dynamic loader can be specified by using the appropriate compiler option.
[4] Linux specific.

PVM tasks communicate with each other. During the migration process, care must be taken to ensure that the communication is retained and that no messages are lost. Such tasks thus present additional difficulties. A PVM daemon must run on every node that participates in a PVM environment. The same holds true for Dynamite. The network of PVM daemons plays a central role in the communication between tasks and in initiating and co-ordinating the migration of tasks. Every PVM task has a socket connection with the local PVM daemon. This connection is used for the *indirect routing*. PVM tasks can also establish point-to-point *direct* TCP/IP communication channels with each other, to improve the performance. Task migration is mediated by the local PVM daemon, which sends the checkpoint signal to the task, and subsequently monitors the checkpointing process. After checkpointing, the daemon on the target node restarts the process. Extra care must be taken when migrating PVM tasks to ensure that they do not permanently lose the connection with the rest of the parallel application, and that the PVM message protocol is not violated.

In Dynamite robust mechanisms for address translation, connection flushing and connection (re-) establishment have been incorporated that have been demonstrated to survive thousands of consecutive migrations.

For a detailed description of the implementation, the reader is referred to [8].

3 Migration Decisions

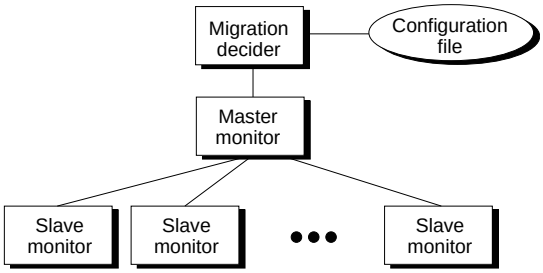

Fig. 2: Monitoring and migration decision sub-system

On each host of a Dynamite-managed resource pool there runs a small and lightweight module which we call the *slave monitor* (Fig. 2). This small program is responsible for frequently (every minute or as specified by the user) collecting status information about its host and sending it to a global *master monitor*. The master monitor therefore receives incoming messages from all hosts and uses this data for creating a history of resource consumption and availability of the complete Dynamite pool. This information is then forwarded to the *migration decider* which determines, when a migration becomes necessary, which tasks will be involved, and where they shall be moved to. Currently, the migration decider uses the following information:

N : number of PVM tasks currently managed by Dynamite

P : number of hosts in the Dynamite pool

C^j_{CPU}, C^j_{Mem} : relative speed and memory capacity of host j

l^i_{CPU}, l^i_{Mem} : CPU load and memory usage of PVM task i on a virtual host with $C^j_{\mathrm{CPU}} = C^j_{\mathrm{Mem}} = 1$

L^j_{CPU}, L^j_{Mem} : load on host j that is not under the control of Dynamite

W_{CPU}, W_{Mem}, W_{Mig} : relative importance of balancing CPU and memory load and minimising migrations as specified in the configuration file

H^i : host on which task i currently resides

W_{CPU}, W_{Mem}, and W_{Mig} are provided by the user whereas the remaining parameters are determined automatically by the system (except for C^j_{CPU}, C^j_{Mem} which in the current implementation still have to be entered manually).

The migration decider uses two different cost functions for different types of parallel applications. Below is the default function we use for parallel applications which depend mainly on the overall performance of all of the nodes they run on. As a consequence, this cost function steers Dynamite towards an evenly loaded pool. It defines the costs of placing tasks $1, \ldots, N$ onto hosts $h^1, \ldots, h^N$ as:

$$\begin{aligned}
\mathrm{Cost}(h^1, \ldots, h^N) = W_{\mathrm{CPU}} & \underbrace{\sum_{j=1}^{P}}_{(*)} \underbrace{\left[L^j_{\mathrm{CPU}} + \sum_{i \in \{1,\ldots,N\}, h^i = j} \frac{l^i_{\mathrm{CPU}}}{C^j_{\mathrm{CPU}}} \right]}_{\text{CPU load on host } j} \\
+ W_{\mathrm{Mem}} & \underbrace{\sum_{j=1}^{P}}_{(*)} \underbrace{\left[L^j_{\mathrm{Mem}} + \sum_{i \in \{1,\ldots,N\}, h^i = j} \frac{l^i_{\mathrm{Mem}}}{C^j_{\mathrm{Mem}}} \right]}_{\text{Memory load on host } j} \\
+ W_{\mathrm{Mig}} & \cdot \underbrace{\mathrm{Card}\left\{ i \in \{1, \ldots, N\} \mid h^i \neq H^i \right\}}_{\text{Necessary task migrations}}
\end{aligned}$$

This cost function has also been used for the experiments presented in Sec. 4. If tasks have to be managed that show a more synchronous behaviour, it is less important to achieve an equally balanced situation. Instead, the primary goal should then be to reduce the load of the mostly loaded host. Hence, in such a situation the user can choose to apply a second cost function in which the two sums marked with (*) are replaced by the maximum function.

The problem of mapping an arbitrary set of tasks onto a heterogeneous set of hosts is known to be NP-hard. Hence, we decided to use a simple hill climbing heuristics for the first prototype of the migration decider. Unfortunately it turned out that hill climbing failed in some very common Dynamite scenarios. Fig. 3

depicts such a case where two large tasks are running on the same host and one of these is not under the control of Dynamite. This situation represents a local minimum to hill climbing.

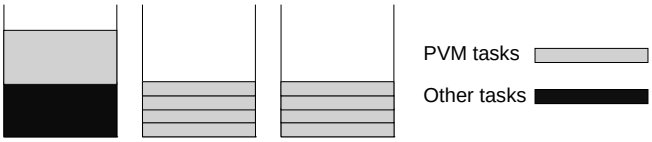

Fig. 3: A common local minimum for hill climbing

If Dynamite was to be used for managing large computing environments with hundreds or thousands of hosts, it probably would be best to use more advanced search heuristics like for instance genetic algorithms or simulated annealing. However, since the original design was made with smaller installations in mind, we decided to take a more problem oriented approach. In our target scenario it is often possible to do an exhaustive search which then results in an optimal solution and therefore avoids non-promising but still expensive migration steps.

The search algorithm that is now used in the migration decider is based on the well known branch and bound algorithm. This technique was adapted to Dynamite by two modifications:

1. The search tree is ordered in a way that increases the probability of early cut-offs. To achieve this, the current placement is made the root of the search tree. This also ensures a very rapid search, if the current situation is already balanced. Furthermore, the first levels of the tree involve placements of the heaviest tasks. As a consequence, many wrong decisions can be detected early in the search.
2. The user can specify an upper time bound for the exhaustive search. If the global optimum could not be determined in time, the algorithm will then return the best local optimum it has encountered. Due to the pre-processed search order, it is likely that the still unexplored areas of the search space contain mostly placements of small tasks and are therefore less important for the overall quality of the solution.

4 Performance measurements

In order to prove that Dynamite delivers what it promises, a number of tests have been conducted.

Some stability testing has been done. Under Solaris, Dynamite was able to make over 2500 successful migrations of large processes (over 20 MB of memory image size) of a commercial PVM application Pam-Crash [19] using direct connections, after which the application finished normally. Similar results have been obtained under Linux.

A series of performance measurements was made on the selected nodes of the DAS cluster [18], which run Linux kernel 2.0 and 2.2 on PentiumPro 200 MHz CPUs. The scientific application Grail [13, 14], a FEM simulation program, has been used as the test application.

	Parallel environment	Execution time
1	PVM	1854
2	DPVM	1880
3	DPVM + sched.	1914
4	DPVM + load	3286
5	DPVM + sched. + load	2564

Table 1: Execution time of the Grail application, in seconds.

Table 1 presents the results of these tests, obtained using the internal timing routines of Grail. Each test has been performed a number of times and an average of the wall clock execution times of the master process (in seconds) has been taken. The parallel application consisted of 3 tasks (1 master and 2 slaves) running on 4 nodes. To obtain the best performance, when using plain PVM, it would be typical to use a number of nodes equal to the number of processes of the parallel application. In this example, for DPVM, one node is left idle (DPVM chooses to put the group server there, but this uses only a minimal amount of CPU time). Such a decomposition would be wasteful for standard PVM.

In the first set of tests presented in Table 1, standard PVM 3.3.11 has been used as the parallel environment.

In the second row, PVM has been replaced by DPVM. A slight deterioration in performance (1.5%) can be observed. This is mostly the result of the fact that migration is not allowed while executing some parts of the DPVM code. These *critical sections* must be protected, and the overhead stems from the *locking* used. Moreover, all messages exchanged by the application processes have an additional, short (8 byte) DPVM fragment header.

In the test presented in the third row, the complete Dynamite environment has been started: in addition to using DPVM, the monitoring and scheduling subsystem is running. Because in this case the initial mapping of the application processes onto the nodes is optimal, and no external load is applied, no migrations are actually performed. Therefore, all of the observed slowdown (approx. 2%) can be interpreted as the monitoring overhead.

In the fourth set of tests an artificial, external load has been applied. This has been achieved by running a single, CPU-intensive process for 600 seconds on each node in turn, in a cycle. Since the monitoring and scheduling subsystem was not running, no migrations could take place. A considerable slowdown of about 75% can be observed.

The final, fifth set of tests is the combination of the two previous tests: the complete Dynamite environment is running, and the external load is applied. Dynamite clearly shows its value: by migrating the application tasks away from

the overloaded nodes, it manages to reduce the slowdown from 75% to 34%. The following factors contribute to the remaining slowdown:

- it takes some time for the monitor to notice that the load on the node has increased and to make the migration decision,
- the cost of the migration itself,
- the master task, which is started directly from the terminal window, is not migrated; when the external load procedure was modified to skip the node with the master task, the slowdown decreased by a further 10%.

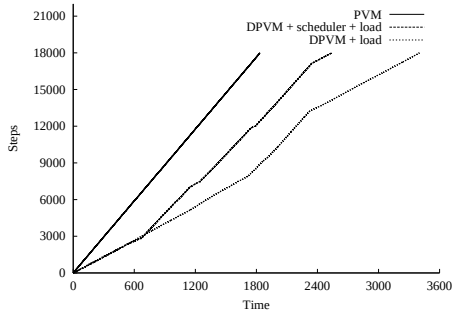

Fig. 4: Execution progress of Grail for three cases. Note that the plain PVM run was made without an external load, whereas both DPVM runs were done with such a load (see text).

Figure 4 presents the execution progress of Grail for three of the five cases. For the standard PVM with no load applied this is a straight, steep line. The other two lines represent DPVM with load applied, with and without the monitoring subsystem running. Initially, they both progress much slower than PVM – because the load is initially applied to the node with the master task, no migrations take place. After approximately 600 seconds the load moves on to another node. Subsequently, in the case with the monitoring subsystem running, the migrator moves the application task out of the overloaded node, and the progress improves significantly, coming close to the one of the standard PVM. In the case with no monitoring subsystem running, there is no observable change at this point. However, it does improve between 1800 and 2400 seconds from the start: that is when the idle node is overloaded. After 2400 seconds from the start, the node with the master task is overloaded again, so the performance deteriorates in both DPVM cases.

Experiments presented in [7] demonstrate that Dynamite works best for programs with a moderate to high computation to communication ratio and moderate task sizes.

Initial experiments using the Dynamite scheduler for cluster management are inconclusive: it functions for small to medium number of tasks, but when

the number of tasks significantly exceeds the number of available nodes, the scheduler doesn't seem to be able to balance them properly.

5 Limitations

The Dynamite system has a number of limitations, most of which are the limitations of the checkpointing mechanism itself. The checkpointer is designed to preserve the memory image of the process and its open files, but nothing more than that. For example, processes that use any of the following features will not be migrated properly:

- pipes,
- sockets,
- System V IPC, like shared memory,
- kernel supported threads,
- `mmap`ping/`open`ing of special files, like `/dev/`..., `/proc/`..., etc.

Some of these, like sockets, might eventually be supported, but supporting shared memory is practically unsolvable in general.

Another limitation, specific to the DPVM subsystem, is an inability to migrate the master PVM task if it is started from the terminal window. Such a task checkpoints correctly, but in order to restart properly, it would have to be restarted manually from a terminal window, whereas it is started by the PVM daemon on the destination host, without standard input and with redirected standard output/error streams. Because of these limitations, the restarted process hangs.

As regards the scheduler, we find that sub-optimal decisions are sometimes made in complex situations. Further research into scheduling methods for dynamic task re-allocation is needed.

6 Conclusions and future prospects

Concluding, task migration in parallel programs has been shown to provide a viable solution to the load-balancing problem. We have succeeded in implementing such a system completely in user space. Since the system is stable now, it can now be used as a research and production tool.

It has also been demonstrated that Dynamite can realise an optimal utilisation of system resources for long-running jobs (a couple of hours and more).

The mean system load is not always a good criterion for load balancing. Future experiments will include the maximum load criterion and possible others.

Dynamite is transparent (existing object files can be relinked to create a Dynamite-enabled executable). Its checkpointing mechanism is easily portable to other operating systems using the ELF binary format.

Dynamite aims to provide a complete integrated solution for dynamic load balancing. In order to accomplish this, the following challenges are still to be solved:

– support for MPI,
– generic support for the migration of the TCP/IP sockets,
– support for Linux GNU libc 2.1 library.

Meanwhile, Dynamite will be used as a research tool, in order to do experiments on dynamic task scheduling, which is an area of active research. Currently, attempts are being made to assess the usefulness of Dynamite as a cluster-management type software. The results of these experiments will be presented in the future.

References

1. Albada, G.D. van, Clinckemaillie, J., Emmen, A.H.L., Gehring, J., Heinz, O., Linden, F. van der, Overeinder, B.J., Reinefeld, A., Sloot, P.M.A.: Dynamite – blasting obstacles to parallel cluster computing. in Sloot, P.M.A., Bubak, M., Hoekstra, A.G., Hertzberger, L.O., editors, High-Performance Computing and Networking (HPCN Europe '99), LNCS **1593** 300–310
2. Casas, J., Clark, D.L., Konuru, R., Otto, S.W., Prouty, R.M., Walpole, J.: MPVM A Migration Transparant Version of PVM. Computer Systems **8 nr 2** (1995) 171–216
3. Czarnul, P., Krawczyk, H.: Dynamic Allocation with Process Migration in Distributed Environments. in Dongarra, J.J., Luque, E., Margalef, T., editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface: 6th European PVM/MPI Users' Group Meeting, LNCS **1697** (1999) 509–516
4. Dan, P., Dongsheng, W., Youhui, Z., Meiming, S.: Quasi-asynchronous Migration: A Novel Migration Protocol for PVM Tasks. Operating Systems Review **33 nr 2** (1999) 5–14
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., Sunderam, V.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994) `http://www.epm.ornl.gov/pvm/`
6. Iskra, K.A., Hendrikse, Z.W., Albada, G.D. van, Overeinder, B.J., Sloot, P.M.A.: Experiments with Migration of PVM Tasks. in ISThmus 2000, Research and Development for the Information Society, Conference Proceedings, Poznan, Poland (2000) 295–304
7. Iskra, K.A., Hendrikse, Z.W., Albada, G.D. van, Overeinder, B.J., Sloot, P.M.A.: Performance Measurements on Dynamite/DPVM. in Dongarra, J., Kacsuk, P., Podhorszki, N., editors, Recent Advances in PVM and MPI. 7th European PVM/MPI User's Group Meeting, LNCS **1908** (2000) (in press)
8. Iskra, K.A., Linden, F. van der, Hendrikse, Z.W., Albada, G.D. van, Overeinder, B.J., Sloot, P.M.A.: The implementation of Dynamite – an environment for migrating PVM tasks. Operating Systems Review **nr 3** (2000) 40–55
9. Litzkow, M., Tannenbaum, T., Basney, J., Livny, M.: Checkpoint and migration of Unix processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin, WI, USA (1997)
10. Livny, M., Pruyne, J.: Managing Checkpoints for Parallel Programs, in Rudolph, L., Feitelson, D. G. editors: Proceedings IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing, LNCS **1162** (1996) 140–154

11. Overeinder, B.J., Sloot, P.M.A., Heederik, R.N., Hertzberger, L.O.: A dynamic load balancing system for parallel cluster computing. Future Generation Computer Systems **12** (1996) 101–115
12. Robinson, J., Russ, S.H., Flachs, B., Heckel, B.: A task migration implementation of the Message Passing Interface. Proceedings of the 5th IEEE international symposium on high performance distributed computing (1996) 61–68
13. Ronde, J.F. de, Albada, G.D. van, Sloot, P.M.A.: High Performance Simulation of Gravitational Radiation Antennas, in L.O. Hertzberger, P.M.A. Sloot, editors, High Performance Computing and Networking, LNCS **1225** (1997) 200–212
14. Ronde, J.F. de, Albada, G.D. van, Sloot, P.M.A.: Simulation of Gravitational Wave Detectors. Computers in Physics, **11 nr 5** (1997) 484–497
15. Tan, C.P., Wong, W.F., Yuen, C.K.: tmPVM — Task Migratable PVM. Proceedings of the 2nd Merged Symposium IPPS/SPDP. (1999) 196–202
16. MPI: A Message-Passing Interface Standard, Version 1.1. Technical Report, University of Tennessee, Knoxville (1995) `http://www-unix.mcs.anl.gov/mpi/`
17. `http://www.genias.de/products/codine/`
18. The Distributed ASCI Supercomputer (DAS). `http://www.cs.vu.nl/das/`
19. `http://www.esi.fr/products/crash/`