Performance Measurements on Dynamite/DPVM

K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder,
P. M. A. Sloot

Informatics Institute, Universiteit van Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
kamil,zegerh,dick,bjo,sloot@science.uva.nl

Paper to appear in
Recent Advances in PVM and MPI.
7th European PVM/MPI User's group Meeting
editors *J. Dongarra and P. Kacsuk and N. Podhorszki*
Lecture Notes in Computer Science, (c) Springer Verlag

Performance Measurements on Dynamite/DPVM

K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot

Informatics Institute, Universiteit van Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
kamil,zegerh,dick,bjo,sloot@science.uva.nl

Abstract. The total computing capacity of workstations can be harnessed more efficiently by using a dynamic task allocation system. The Esprit project Dynamite provides such an automated load balancing system, through the migration of tasks of a parallel program using PVM. The Dynamite package is completely transparent, *i.e.* neither system (kernel) nor application program modifications are needed. Dynamite supports migration of tasks using dynamically linked libraries, open files and both direct and indirect PVM communication. In this paper we briefly introduce the Dynamite system and subsequently report on a collection of performance measurements.

1 Introduction

With the continuing increases in commodity processor and network performance, distributed computing on standard PCs and workstations has become attractive and feasible. Consequently, the availability of efficient and reliable cluster-management software supporting task migration becomes increasingly important.

Various *PVM* [5] variants supporting task migration have been reported, such as *tmPVM* [12], *DAMPVM* [3], *MPVM* (also known as MIST) [2], *ChaRM* [4] and CoCheck [11]. For *MPI* [14], task migration has been studied in Hector [8].

Building on earlier *DPVM* work by L. Dikken et al. [7], we have developed Dynamite[1]. Dynamite [1] attempts to maintain optimal task allocation for parallel jobs in dynamically changing environments by migrating individual tasks between nodes. Task migration also makes it possible to free individual nodes, if necessary, without breaking the computations.

Dynamite supports applications written for *PVM* 3.3.x, running under Solaris/UltraSPARC 2.5.1, 2.6, 7 and 8. Moreover, it supports Linux/i386 2.0 and 2.2 (libc5 and glibc 2.0 binaries; glibc 2.1 is not supported at this point).

[1] Dynamite is a collaborative project between ESI, the Paderborn Center for Parallel Computing, Genias Benelux and the Universiteit van Amsterdam, partly funded by the European Union as Esprit project 23499. Of the many people that have contributed, we can mention only a few: J. Gehring, A. Streit, J. Clinckemaillie, A.H.L. Emmen.

The principal advantages of Dynamite are its API-level transparency, its powerful, dynamic loader based checkpoint/migration mechanism and its support for the migration of both direct and indirect *PVM* connections. We have found Dynamite to be very stable. Its modular design greatly facilitates the port to *MPI* [14], which is currently underway.

2 Dynamite overview

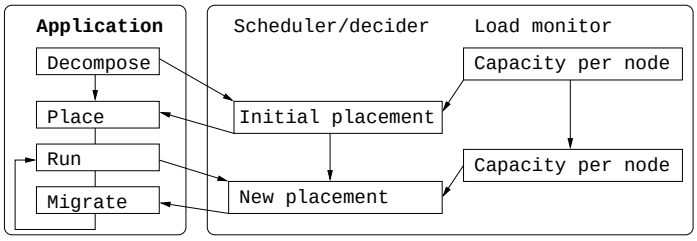

Fig. 1: Dynamite run-time system. An application has to be decomposed into several subtasks already. An initial placement is determined by the scheduler. When the application is run, the monitor checks the capacity per node. If it is decided that the load is unbalanced (above a certain threshold), one or more task migrations may be performed to obtain a more optimal load distribution.

The Dynamite architecture (see Figure 1) is built up from three separate parts:

1. The load-monitoring subsystem. The load-monitor should leave the computation (almost) undisturbed.
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. Basically, the checkpoint software makes the state of a process persistent at a certain stage.

Parallel *PVM* applications consist of a number of processes (*tasks*) running on interconnected nodes constituting a *PVM virtual machine*. A *PVM daemon* runs on every node and communicates with other daemons using the UDP/IP protocol. *PVM* tasks communicate with each other and with *PVM* daemons using a message-passing protocol. *PVM* message passing is reliable: no messages can be lost, corrupted or duplicated and must arrive in the order sent.

In Dynamite, a *monitor* process is started on every node of the *PVM* virtual machine. This monitor communicates with the local *PVM* daemon and collects information on the resource usage and availability, both for the node as a whole and individually for every *PVM* task. The information is forwarded to a central *scheduler*, which makes migration decisions based on the data gathered. *PVM* daemons assist in executing these decisions.

For migration, first, the running process must be checkpointed, i.e. its state must be consistently captured on the source node. Next, the process is *restored* on the destination node; its execution resumes from the point at which the source process was checkpointed. Typically, the original process on the source node is terminated.

Processes that are part of the parallel *PVM* application present additional difficulties. 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. 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 [6].

3 Performance measurements

In order to evaluate Dynamite's performance, a number of tests have been conducted. Some of these are concerned with the performance of the components of the system, such as the modified *PVM* library. Others attempt to quantify the performance of the Dynamite system as a whole, in a controlled dynamic environment.

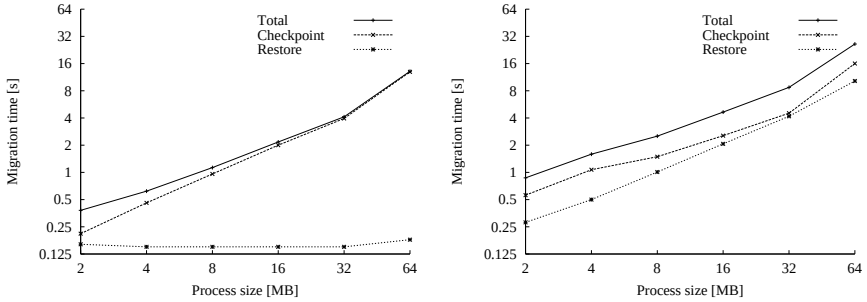

Fig. 2: Migration performance of *DPVM* for (a) Linux and (b) Solaris.

3.1 Performance of System Components

In a system like Dynamite, there are two easily measurable performance factors:

– the time it takes to migrate a task of a given size,

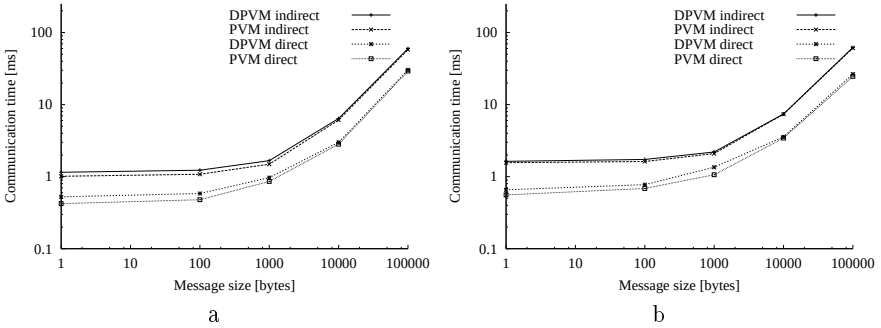

Fig. 3: Communication performance in *DPVM* and *PVM* for (a) Linux and (b) Solaris.

– the difference in communication performance compared to standard *PVM*.

Experiments have been performed to measure these two factors, both under Linux and Solaris. In case of Linux, reserved nodes of a PC-cluster have been used, equipped with PentiumPro 200 MHz CPU and 128 MB RAM, running kernel version 2.2.12. In case of Solaris, idle UltraSPARC 5/10 workstations have been used, equipped with 128 MB RAM and running kernel version 5.6. In both cases, 100 Mbps Ethernet was used. In both cases the NFS servers used for checkpoint files were shared with other users, which could affect the performance to some extent.

Figure 2 presents the performance of migration in *DPVM* for various process sizes. A simple ping-pong type program communicating once a few seconds via direct connection was migrated, process size was set with a single large `malloc` call. Execution time of each of the four migration stages (see [6]) was measured. In general, it was found that the major part of the migration time is spent on checkpointing and restoring, the remaining stages amount to approximately 0.01 – 0.03s, and hence are not shown. The speed of checkpointing and restoring is limited by the speed of the shared file system. On our systems this limit lies at 4–5 MB/sec for NFS running over the 100Mbps network. It can be observed, however, that the restoring phase under Linux takes an approximately constant amount of time, while it grows with process size under Solaris, resulting in twice larger migration times for large processes. This is a side effect of differences in the implementation of `malloc` between the two systems. For large allocations, Linux creates new memory segment (separate from the heap) using `mmap`, whereas Solaris always allocates from the heap with `sbrk`. When restoring, the heap and stack are restored with `read`, which forces an immediate data transfer. However, for the other segments our implementation takes advantage of `mmap`, which uses more advanced *page on demand* technique. Since the allocated memory region is not needed to reconnect the task to the *PVM* daemon, the time it takes to restart the task is constant under Linux. Clearly, delays may be incurred later, when the `mmap`ped memory is accessed and loaded.

In Figure 3, comparison of communication performance between *DPVM* and *PVM* is presented. Both indirect and direct communication performance has been measured. A ping-pong type program was used, exchanging messages between 1 byte and 100KB in size. With *DPVM*, a slowdown is visible in all cases. It stems from two factors:

- signal (un)blocking on entry and exit from *PVM* functions (function call overhead),
- an extra header in message fragments (communication overhead).

The first factor adds a fixed amount of time for every *PVM* communication function call, whereas the second one increases the communication time by a constant percentage. For small messages the first factor dominates, since there is little communication. An overhead from 25% for direct communication under Linux to 4% for indirect communication under Solaris can be observed. While particularly the first difference in speed is significant, it must be pointed out that it represents a worst case scenario. The overhead percentage is larger for direct communication, since the communication is faster while the overhead from signal blocking/unblocking stays the same.

As the messages get larger, the overhead of signal handling becomes less significant, and the slowdown goes down to 2–4% for 100KB messages.

Tests have been made to compare the communication speed in *DPVM* before and after the migration, but no noticeable difference was observed ($\pm 1\%$).

3.2 Stability of the system

Care has been taken to prove the robustness of the environment. Thousands of migrations have been performed both under Solaris and Linux, for processes ranging in size from light-weight, 2 MB processes to heavy, 50 MB and larger. Delays between individual migrations ranged between a fraction of a second and several minutes, in order to test for race conditions. Similarly, different communication patterns have been tested, including tasks using very small and very large messages, using direct and indirect communication, communicating point-to-point and using multicasts. These proved to be very revealing tests.

In one test performed 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 using direct connections.

3.3 Performance of the Integrated System

Benchmarks In order to assess the usefulness of the integrated system, reserved nodes of a cluster have been used to run a series of parallel benchmarks under several different conditions. The benchmarks in question originate from the NAS Parallel Benchmarks suite [13]. The individual benchmarks have been adjusted to use four computation tasks each, running for aproximately 30 minutes in an optimal situation. Where necessary, code has been added to provide intermediate information on the execution progress of each task.

Eight nodes of a Linux cluster were reserved, each equipped with PentiumPro 200 CPU and 64 MB RAM, running Linux kernel version 2.0.36. 100 Mbps FastEthernet was used as the communication medium. The number of nodes exceeds the number of tasks, so this is a *sparse* decomposition, and consequently during the execution of the benchmarks some nodes are idle. The Dynamite scheduler works best in such a situation, since it can migrate tasks away from overloaded nodes to idle nodes.

	No load	Load	
		Dynamite	No Dynamite
cg (smallest eigenvalue approximator)	1795	2226 (+24%)	3352 (+87%)
ep (embarrassingly parallel)	1620	1773 (+9%)	1919 (+18%)
ft (discrete Fourier transform)	1859	2237 (+20%)	2693 (+45%)
is (integer sort)	1511	1758 (+16%)	1688 (+12%)
mg (discrete Poisson problem)	1756	1863 (+6%)	2466 (+40%)

Table 1: Execution times of NAS parallel benchmarks, in seconds.

Table 1 presents the execution times of the NAS parallel benchmarks. The numbers in the *No load* column were obtained by running the individual benchmarks in the ideal situation, when all the nodes were totally idle otherwise. Of course, the results obtained this way are the best. In case of the other two *Load* columns, an external load has been applied. The external load was generated by running a single computationally intensive process for 5 minutes on each node used by the benchmark. One node at a time was overloaded in this way, and the external load program worked in a cycle, going back to the first node when it was done with the last one. Two kinds of measurements have been carried out: one with *Dynamite* running, and one without. In both cases, the benchmarks ran slower than without external load. However, in case of all but one of the benchmarks, the results obtained with Dynamite significantly outperform the other case, reducing the percentage of slowdown by a factor of 2 to 6.

Figure 4 presents the execution progress of the NAS parallel benchmarks (due to space restrictions, only 3 of them could be included). In each case, the data for one of the tasks of the parallel application is shown. The left graph presents the time spent on executing each individual step (ideally, this should be a constant); the right graph presents the total time spent so far.

In Figure 4 (a), results for *cg* benchmark are shown. This benchmark slows down 87% when subjected to external load. Such a significant slowdown is an indication of two things. First, large part of execution time must be spent on computation, otherwise the external load would not affect the local task so significantly. Second, the communication pattern of the benchmark (global communication) forces other processes to wait for the one lagging behind, with all the unpleasant consequences to the performance.

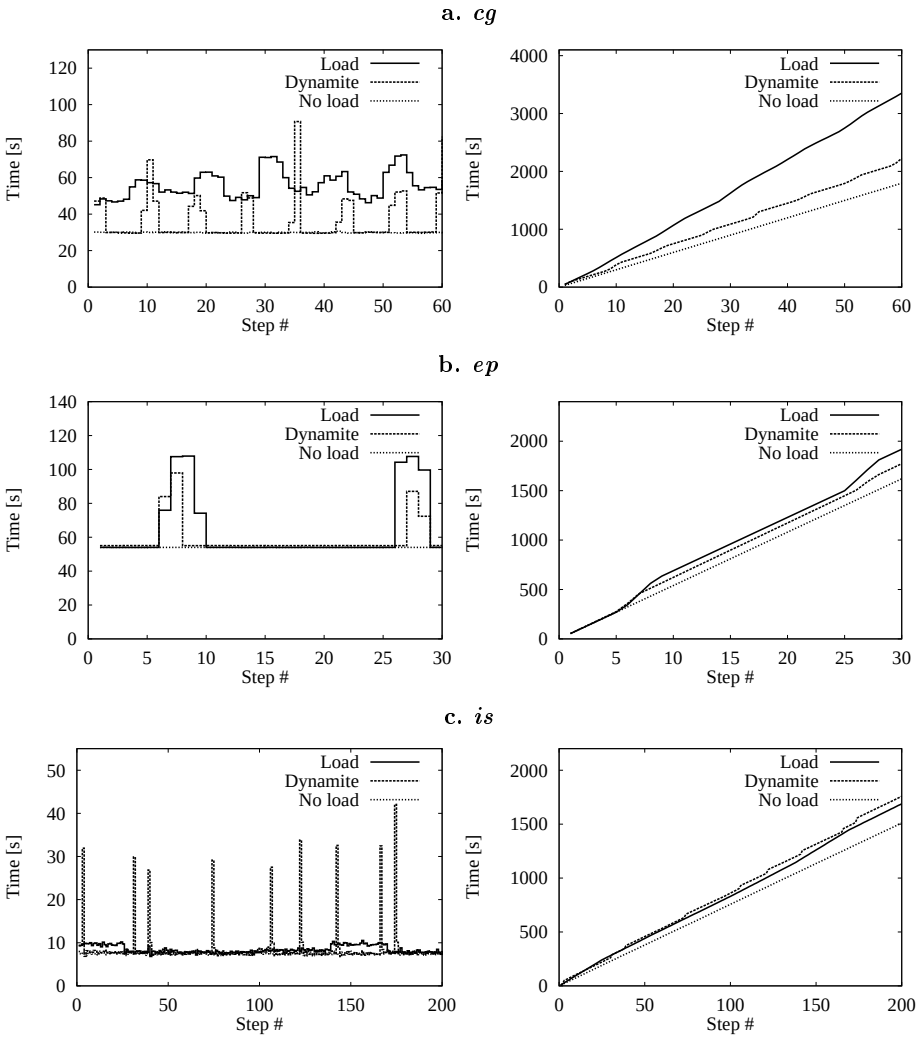

Fig. 4: Execution progress of NAS parallel benchmarks: the time to execute one step (left) and the total time (right).

The results of *ep* benchmark, as presented in Figure 4 (b), are different. The computation tasks of the *ep* benchmark do not communicate with each other at all, and consequently all of the execution time is spent on computation. In such a case, external load significantly hampers the performance of the affected task, but, due to lack of communication, has no influence on other tasks (the line on the left picture is flat in the area where other tasks of the application are affected by the external load).

Figure 4 (c) shows the execution of the *is* benchmark, the only one that performs worse with Dynamite running. *Is* is in some ways similar to *ep* — they are both only slightly affected by the external load, but the reasons for that are different. Just opposite to *ep*, in *is* most of the execution time is spent on communication: tasks communicate frequently and in large volumes. Therefore, the application progress is limited by the internode communication subsystem, not by the CPU, so an external load has little influence on the local task, and an even smaller one on the remote tasks. The migration decisions of the Dynamite scheduler are not unreasonable, but their gain fails to exceed the migration cost, which is rather high in this case because of large process size (40 MB).

The large process size (30 MB) also affects the result of the *ft* benchmark, where Dynamite reduces the slowdown from 45% to 20%. The reduction would have been significantly larger, had the processes to be migrated been smaller.

Standard production code In this test, the scientific application Grail [9, 10], a FEM simulation program, has been used as the test application. The measurements were made on selected nodes of a cluster (see Section 3.1).

	Parallel environment	Decomposition	
		sparse	redund.
1	*PVM*	1854	2360
2	*DPVM*	1880	2468
3	*DPVM* + sched.	1914	2520
4	*DPVM* + load	3286	2947
5	*DPVM* + sched. + load	2564	3085

Table 2: Execution time of the Grail application, in seconds.

Table 2 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 tests can be grouped into two (decomposition) categories:

- **sparse** — the parallel application consisted of 3 tasks (1 master and 2 slaves) running on 4 nodes,
- **redundant** — the parallel application consisted of 9 tasks (1 master and 8 slaves) running on 3 nodes.

To obtain the best performance, it would be typical to use the number of nodes equal to the number of processes of the parallel application. Neither of the above decompositions does that. In case of the sparse decomposition, one node is left idle (*PVM* chooses to put the group server there, but this one uses only a minimal fraction of CPU time). Such a decomposition would be wasteful for the standard *PVM*. In the redundant case, each node runs 3 tasks of the application (one of

the nodes also runs the group server). Although the number of nodes used when running the two decompositions is different, comparing the timings makes sense, since 3 nodes are used at any one time in each case.

In the first set of tests presented in Table 2, standard *PVM* 3.3.11 has been used as the parallel environment. Not surprisingly, the sparse decomposition wins over the redundant one, since it has lower communication overhead.

In the second row, *PVM* has been replaced by *DPVM*. A slight deterioration in performance (1.5-4.5%) 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 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 can be observed, although it is far larger for the sparse decomposition (75%) than for the redundant one (19%), actually making the latter faster. This is a result of the UNIX process scheduling policies: for sparse decomposition, the external load can lengthen the application runtime by a factor of 2, while for the redundant decomposition by no more than 33%, since there are already 3 CPU-intensive processes running on each node, so the kernel is unlikely to grant more than 25% of CPU time for the external load process. This shows that sparse decomposition, although faster in a situation close to ideal, performs rather badly when the conditions deteriorate, while the redundant decomposition is far less sensitive in this regard.

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 in case of the sparse decomposition, where, by migrating the application tasks away from the overloaded nodes, it manages to reduce the slowdown from 75% to 34%. The remaining slowdown is caused by:

- the 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 is non-zero,
- the master task, which is started directly from the shell, 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%.

Turning to the redundant decomposition, it can be observed that the Dynamite scheduler actually made the matters worse, increasing the slowdown from 19% to 25%. This result, although unwelcome, can easily be explained. The situation

was already rather bad even without the external load: not only were all the nodes overloaded, they were also overloaded by the same factor (3). Therefore, the migrator had virtually no space for improvement, and its attempts to migrate the tasks actually worsened the situation. It can be argued that the migrator should have refrained from making any migrations in this case, though.

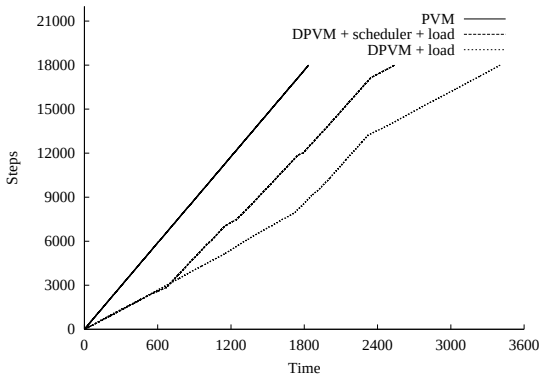

Fig. 5: Execution progress of Grail for sparse decomposition. Note that the performance of plain *PVM* was measured without any load. With a simulated background load it would have been only slightly better than the "*DPVM + load*" performance.

Figure 5 presents the execution progress of Grail for sparse decomposition. For standard *PVM* with no load applied this is a straight, steep line. The other two lines denote *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, the node with the master task is overloaded again, so the performance deteriorates in both *DPVM* cases.

4 Conclusions and future prospects

Concluding, our implementation of load balancing by task migration has been shown to be stable. The use of the Dynamite system results in a slight performance penalty in a well-balanced system, but significant performance gains can be obtained from task migration in an unbalanced system. Improvements can still be made in the scheduling.

Dynamite aims to provide a complete integrated solution for dynamic load balancing. A port to MPI is being implemented, in cooperation with the people from Hector [8]. Dynamite/DPVM can be obtained for academic, non-commercial use through the authors[2].

References

1. van Albada, G.D., Clinckemaillie, J., Emmen, A.H.L., Gehring, J., Heinz, O., van der Linden, F., Overeinder, B.J., Reinefeld, A., and Sloot, P.M.A.: Dynamite — blasting obstacles to parallel cluster computing. HPCN Europe '99, Amsterdam, The Netherlands, in LNCS, n. 1593, 300–310, 1999.
2. Casas, J., Clark, D.L., Konuru, R., Otto, S.W., Prouty, R.M., and Walpole, J.: *MPVM*: A migration transparent version of *PVM*. Usenix Computer Systems, v. 8, n. 2, 171–216, 1995.
3. Czarnul, P., and Krawczyk, H.: Dynamic allocation with process migration in distributed environments. Proceedings of the 6th European *PVM/MPI* Users' Group Meeting, Barcelona, Spain, in LNCS, n. 1697, 509–516, 1999.
4. Dan, P., Dongsheng, W., Youhui, Z., and Meiming, S.: Quasi-asynchronous Migration: A Novel Migration Protocol for *PVM* Tasks. Operating Systems Review, v. 33, n. 2, ACM, 5–14, April 1999.
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V.: *PVM*: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, 1994. `http://www.epm.ornl.gov/pvm/`
6. Iskra, K.A., van der Linden, F., Hendrikse, Z.W., Overeinder, B.J., van Albada, G.D., and Sloot, P.M.A.: The implementation of Dynamite — an environment for migrating PVM tasks. Operating Systems Review, v. 34, n. 3, 40–55, ACM Special Interest Group on Operating Systems, July 2000.
7. Overeinder, B.J., Sloot, P.M.A., Heederik, R.N., and Hertzberger, L.O.: A Dynamic Load Balancing System for Parallel Cluster Computing. Future Generation Computer Systems, v. 12, n. 1, 101–115, 1996.
8. Robinson, J., Russ, S.H., Flachs, B., and Heckel, B.: A task migration implementation of the Message Passing Interface. Proceedings of the 5th IEEE international symposium on high performance distributed computing, 61–68, 1996.
9. de Ronde, J.F., van Albada, G.D., and Sloot, P.M.A.: High Performance Simulation of Gravitational Radiation Antennas. High Performance Computing and Networking '97, in LNCS, n. 1225, 200–212, 1997.
10. de Ronde, J.F., van Albada, G.D., and Sloot, P.M.A.: Simulation of Gravitational Wave Detectors. Computers in Physics, v. 11, n. 5, 484–497, 1997.
11. Stellner, G., and Trinitis, J.: Load balancing based on process migration for MPI. Proceedings of the Third International Euro-Par Conference, in LNCS, n. 1300, 150–157, Passau, Germany, 1997.
12. Tan, C.P., Wong, W.F., and Yuen, C.K.: *tmPVM* — Task Migratable *PVM*. Proceedings of the 2nd Merged Symposium IPPS/SPDP, 196–202.5, April 1999.
13. White, S., Alund, A., and Sunderam, V.S.: Performance of the NAS Parallel Benchmarks on PVM Based Networks. Journal of Parallel and Distributed Computing, v. 26, n. 1, 61–71, 1995.
14. *MPI*: A Message-Passing Interface Standard, Version 1.1. Technical Report, University of Tennessee, Knoxville, June 1995. `http://www-unix.mcs.anl.gov/mpi/`

[2] For commercial use, contact Genias Benelux `http://www.genias.nl/`