Bibliographic reference for this paper:

G.D. van Albada; J. Clinckemaillie; A.H.L. Emmen; J. Gehring; O. Heinz; F. van der Linden; B.J. Overeinder; A. Reinefeld and P.M.A. Sloot:
Dynamite - blasting obstacles to parallel cluster computing,
in M. Boasson; J.A. Kaandorp; J.F.M. Tonino and M.G. Vosselman, editors,
Proceedings of the fifth annual conference of the Advanced School for Computing and Imaging ASCI,
June 15-17, 1999, ASCI, pp. 31-37.
ASCI, Delft, June 1999.

An earlier (more accessible) version appeared in:

G.D. van Albada; J. Clinckemaillie; A.H.L. Emmen; J. Gehring; O. Heinz; F. van der Linden; B.J. Overeinder; A. Reinefeld and P.M.A. Sloot:
Dynamite - blasting obstacles to parallel cluster computing,
in P.M.A. Sloot; M. Bubak; A.G. Hoekstra and L.O. Hertzberger, editors,
High-Performance Computing and Networking (HPCN Europe '99), Amsterdam, The Netherlands,
in series Lecture Notes in Computer Science, nr 1593 pp. 300-310.
Springer-Verlag, Berlin, April 1999. ISBN 3-540-65821-1.

Dynamite - blasting obstacles to parallel cluster computing

G.D. van Albada[1], J. Clinckemaillie[2], A.H.L. Emmen[3], J. Gehring[4], O. Heinz[4],
F. van der Linden[1], B.J. Overeinder[1], A. Reinefeld[5], P.M.A. Sloot[1]

[1] Department of Computer Science, Universiteit van Amsterdam, Kruislaan 403,
1098 SJ Amsterdam, The Netherlands
[2] Engineering Systems International, 20 Rue Saarinen,
F-94578 Rungis SILIC 270, France
[3] Genias Benelux BV, James Stewartstraat 248, 1325 JN Almere, The Netherlands
[4] Paderborn Center for Parallel Computing, Fürstenallee 11, 33102 Paderborn, Germany
[5] Konrad-Zuse-Zentrum für Informationstechnik, Takustrasse 7, D-14195 Berlin, Germany

Keywords: cluster computing, migration, scheduling

Abstract. Workstations make up a very large fraction of the total available computing capacity in many organisations. In order to use this capacity optimally, dynamic allocation of computing resources is needed. The Esprit project Dynamite addresses this load balancing problem through the migration of tasks in a dynamically linked parallel program. An important goal of the project is to accomplish this in a manner that is transparent both to the application programmer and to the user. Our approach adds the migration of dynamically linked tasks, of tasks with open files and with direct PVM connections to earlier versions of DPVM. The system is now provided with monitoring and scheduling software. As a test bed, the Pam-Crash software from ESI is used.

1 Introduction

Workstations have become ubiquitous in many organisations. By their nature, they are often used intensively during normal working hours, and are often largely idle otherwise. They represent a huge reservoir of computing capacity that can be used much more efficiently.

Thus, we currently witness a shift of emphasis in high-performance computing from expensive, special-purpose monolithic systems to the use of clusters of workstations or PCs.

When using time-shared workstation clusters as HPC compute servers, however, one has to cope with the dynamical behaviour of the compute nodes, the network load and the application tasks. These can lead to local load imbalances, which hamper the application's execution speed and the overall system performance.

The application itself can also exhibit dynamic behaviour due to changes in the load per task (e.g. contact problems in car crash simulations). This leads to serious load imbalances, which are difficult to resolve, even on dedicated parallel platforms that offer a constant performance per node. When the node performance changes dynamically, as in workstation clusters, the situation becomes even more difficult.

Also, running a HPC task on a workstation may jeopardise its primary purpose of providing computing capacity to a particular employee.

Solving these problems requires that work somehow be migrated from one node to another. This can be done internally to the parallel application, but such an approach requires a major adaptation of each individual program. Various solutions have been developed to improve the load distribution for workstations. These range from systems that schedule parallel or sequential jobs on free workstations, such as LSF [1], via systems that can also migrate sequential jobs, such as Codine [2] and Condor [3, 4], to systems that also aim to migrate tasks in parallel jobs. MPVM/MIST [5, 6] does this for PVM based jobs, Hector [7] for MPI.

In the ESPRIT project 23499 “DYNAMITE”, we develop a dynamic execution environment that handles the load balancing of parallel applications in a dynamically changing cluster environment by migrating individual tasks in a manner that is robust, efficient and transparent to the user and the application programmer. The DYNAMITE software is based on PVM 3.3.11 and is called Dynamic PVM [8] or DPVM for short. DPVM is totally transparent to the user's application: existing PVM codes need only be linked to the DPVM library. The DYNAMITE system is intended for environments requiring a relatively infrequent redistribution of workload for large applications that can run for several days. We strive for a response time of at most a few minutes and a minimal overhead, but give an absolute priority to reliability and stability.

In constructing such an environment, the following problems need to be addressed:

- migration of dynamically linked tasks,

- migration of communication endpoints,
- load monitoring,
- task (re-)allocation,
- job preparation

In this paper, we describe the ongoing work in the DYNAMITE project. The first two issues will be addressed in the next section. Subsequently we will address load monitoring, task allocation and job preparation in separate sections, before coming to our conclusions.

2 Migration

Migration of tasks requires that the state of the task is captured, after which a new task is started on the target machine, initialised with the captured state. Correct migration is difficult because the interactions of the task with its environment need to be taken into account. A completely transparent migration, which cannot be detected by the task or its communication partners, is almost impossible to realise, but is not usually necessary either. We strive to migrate dynamically linked tasks with open files, communicating with other tasks solely through PVM.

For the migration of tasks with open files, we impose the additional requirement that these files can be accessed using the same path on both the source and target machine.

We have implemented the migration mechanism making use of a full checkpoint of the task. Though it requires additional communication and I/O compared to a mechanism based on a direct transfer of the task image from source to target machine, we have decided to use this approach for reasons of robustness and clarity of implementation.

Pilot versions of the checkpointer and migrator were implemented for the SUN Solaris operating system, and have been tested on OS versions 2.5.1 and 2.6 on UltraSparc workstations.

2.1 Migration of dynamically linked code

As stated, as the first step in the migration of a task, a checkpoint dump is made.

The checkpointing implementation used in DYNAMITE differs from existing implementations in two ways. Firstly, the checkpointing code is not linked into the program itself. Instead, it is present in the dynamic loader, a piece of code loaded before the actual program is run. The task of the dynamic loader is to load the shared libraries required by the program. Most Unix systems implement shared libraries using a dynamic loader, and have an option to specify a different loader for each program. This option is used to specify our own dynamic loader.

In DYNAMITE, this dynamic loader will perform these tasks as usual, but will also contain code to handle checkpointing signals, and to keep information on the used shared libraries. This means that it can take care of creating the checkpoint, and restoring it, using the exact same memory mappings for shared libraries. This is important, because shared libraries are normally not guaranteed to be mapped on the same memory address, which would make restarting the application impossible.

The other new aspect in the checkpointing code is the propagation of checkpointing signals. This means that the dynamic loader will, before creating the checkpoint, signal the application to allow it to save state that can possibly not be saved within the framework of the normal checkpointing procedure. Normally, programs need not be aware of this signal, so that transparency is maintained, but the DPVM library uses it for the migration of files and connections.

The current implementation of the checkpointer has the following limitations:

- The checkpointed task should not be multithreaded. This limitation applies to all PVM programs anyway.
- The checkpointed task should not have any network connections open, save for those serviced by PVM. Migration of open files is supported as long as the path name is valid on both hosts.

The checkpointer writes a full checkpoint to a file, including any mapped dynamic libraries and the complete data segment. An earlier version of DPVM used a migration approach in which most of the data segment was transferred directly from the old to the new task image through a socket. While this approach has a speed advantage, it hampers a robust implementation.

It is not necessary for the original task still to be active for the restart, as would be the case when part of the image would be transferred directly from the old to the new task. The checkpoint file is an executable in its own right, and can thus be restarted in the usual way.

The job of the migrator, which is part of the DPVM library, is to start a new task on the target machine, using the checkpointed executable.

2.2 Migration of communicating tasks

A main objective of the DPVM migration facility is transparency of the migration protocol. With respect to the 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 causing failures triggered by migration of one of the tasks. The work upon which our implementation is based is described in [8]

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 PVM daemon (pvmd) representing that host. Next, the master pvmd updates its routing table to reflect the new location of the task. 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 task. If a message is currently being sent by the task, migration is delayed until the transmission has been completed. The task is then checkpointed and the newly created process on the destination host is requested to restart the checkpoint.

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. Any message that arrived at the pvmd during the checkpoint/migration phase is then delivered to the restarted task.

2.3 Packet Routing

In PVM the task identifier, task id for short, is a unique identifier that 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, virtualising the task ids. This is accomplished by maintaining additional routing information tables in all pvmds. 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, so that each routing table reflects the actual location of all migrated tasks in the system.

2.4 Direct Connections

The basic mode of communication in PVM is through the daemon. For reasons of efficiency, PVM allows tasks to request a direct connection to another task. This complicates the rerouting of the communication. The main problem of direct communication connections is making sure that all communication has been flushed. Simply breaking the connections may result in loss of messages and is not acceptable. Several approaches are possible. Some involve shutting down communication for the whole system temporarily, but this may cause unnecessary delay. Another approach is to leave an agent in place that takes care of the connection as long as it has not been confirmed as flushed by the other side.

In the new version of DPVM the direct routing problem has been solved, essentially by making a transaction out of each PVM communication (send/receive of a message). If it fails due to the connection being shut down, the transmission will have no effect and will be retried after the partner task has been migrated, and the connection restored, using the address information obtained through the PVM daemons.

A related problem occurs in the implementation of task migration in MPI. MPI is a specification that is often implemented using direct connections, but without daemons, as in MPICH, a popular MPI implementation. See [7] for one possible solution for the problems that occur when implementing task migration for this MPI version

3 Resource Monitoring

Any migration decision has to be based on the information that is currently available about the cluster. This refers to the state of the hardware as well as to the runtime behaviour of the applications. The typical approach taken by most cluster management systems is to measure the load on each available host and of each application process. The busiest tasks are then moved to the least loaded nodes until a satisfactory state is achieved. This strategy has been proven well suited for running independent jobs on networks of workstations, but it performs less well for parallel applications as it completely neglects communication between interdependent tasks. This drawback is especially apparent in environments with significant performance differences between the nodes. In such

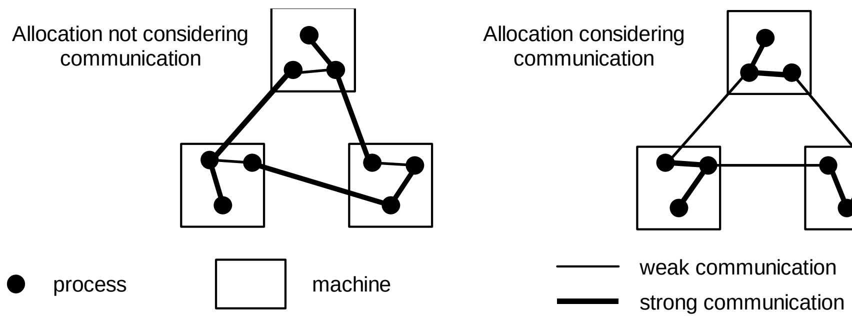
Allocation not considering communication
Allocation considering communication
process
machine
weak communication
strong communication

Fig. 1. : Grouped Tasks

scenarios, it is often the case that larger machines (typically SMPs or NUMAs with 4 to 16 processors) are assigned multiple processes. It is then desirable to have frequently communicating tasks grouped together on big machines (Figure 1).In the first case, the sequential load is equally balanced but the communication is not. Therefore, the monitoring tool must also keep track of the communication between the tasks. In order to make an optimal migration decision, the following information is needed:

- available capacity on each node (CPU, memory, disk space),
- current load of each node,
- required capacity for each task,
- network connectivity and capacity,
- communication pattern for each task.

Each of these items can be measured at execution time by monitoring software, but we assume that node capacity and network properties are sufficiently stable that they can best be specified beforehand by the system administrator. Therefore, we have chosen a textual representation of the static resources (see [9] for further details).

Detailed information about the network topology can be obtained from a “Network Resource Description” file that is used for migration decisions. Tasks should preferably be migrated to nodes in the same subnet. This provides locality for the messages and prevents that a large amount of data has to be routed from one subnet to the other. If it is not possible to fulfil the requirement for locality then nodes in adjacent subnets are selected. Because of the assumed dynamic behaviour of the application and the system load, the other items need to be obtained by monitoring software. Information about load and capacity must be collected from all nodes of the cluster, also those where currently no task of the parallel application is running. This is accomplished by running a small monitor program (monitor slave) on each node (Figure 2).

The statistics obtained by the monitor slaves are sent to the monitor master process that is not only responsible for maintaining the whole cluster statistics but also has to make migration decisions. The information on communication patterns is obtained directly from the DPVM environment. Therefore, DPVM has been enhanced by a message monitoring thread. This thread keeps track of each message sent and received. These communication statistics are also sent to the monitor master process that is depicted in detail in Figure 3.

The monitor master process consists of five threads that operate concurrently. The message dispatcher thread identifies each message received and appends it to the appropriate queue. There exist three different queues:

- a node capacity queue to store the information from the monitor slaves (CPU, memory, I/O, …),
- a DPVM capacity queue to store the information about CPU and memory utilisation of the DPVM processes and

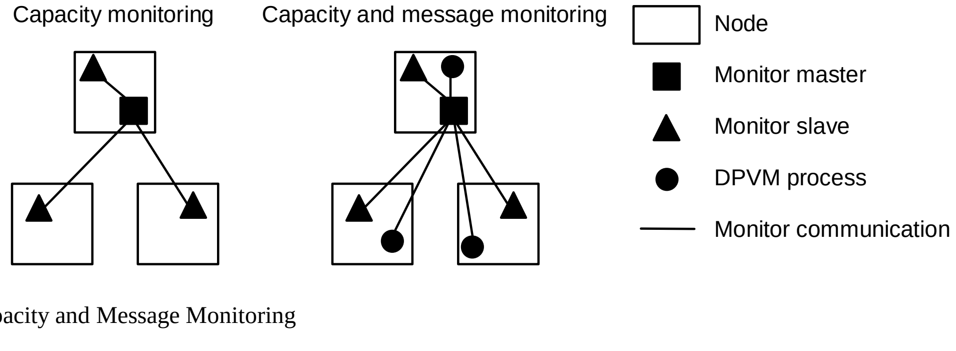

Fig. 2. : Capacity and Message Monitoring

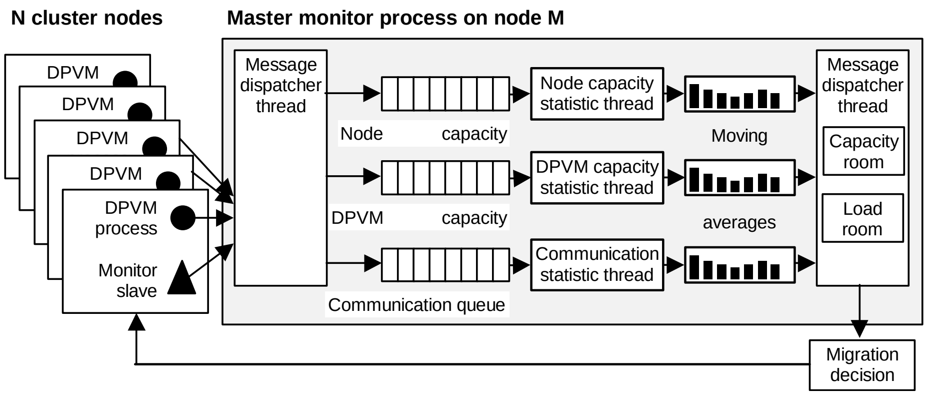
N cluster nodes
Master monitor process on node M
DPVM
DPVM
DPVM
DPVM
DPVM process
Monitor slave
Message dispatcher thread
Node capacity
DPVM capacity
Communication queue
Node capacity statistic thread
DPVM capacity statistic thread
Communication statistic thread
Moving averages
Message dispatcher thread
Capacity room
Load room
Migration decision

Fig. 3. : Architecture of the Monitor Master Process

- a communication queue to store the information about the communication activity between the DPVM processes.

The queues act as an intermediate store because the statistics threads are only active every *j* seconds, where *j* can be adapted to the application monitored. Long running applications do not need a short monitoring interval and, therefore, the statistics need not be updated regularly. Each statistic thread maintains a ring-buffer (not shown) where the last *l* entries are stored. Each entry in the ring-buffer corresponds to a snapshot of the monitored data at a certain point in time. It is obvious that it is not practicable to store all values since monitoring has begun. Therefore, we have chosen to implement a moving average scheme that keeps track of the last *l* entries.

This scheme has the advantage that we can apply a recursive formula that depends only on the newest and oldest value of the ring-buffer. This speeds-up calculation of the moving averages and decreases the monitoring overhead. To allow further processing, e. g. to visualise the data sets, the statistical data is also written to disk (not shown).

4 Migration Decider

The migration decider is the main part of the scheduler thread that is executed periodically by the monitor master process. Based on the monitored data, the migration decider has to judge about where and when to migrate a task from an overloaded node. Additionally the task to be moved causes some constraints on the migration decision. Therefore, the master load monitor has to supply some normalised values about the attributes CPU, memory, and disk swap space of each node and additionally the available network capacity.

The increasing interest in distributed computing has lead to intensive scientific research in load balancing schemes for distributed memory systems [2, 5, 10, 11, 12, 13, 14, 15]. Because not every load-balancing scheme is applicable to every application, the migration decider has been designed in a flexible manner to support a broad range of applications.

CPU capacity on the one hand, and memory and disk swap space on the other, are inherently different in their scheduling requirements. Where, up to a certain point, more CPU capacity is always better, swap space and memory either are enough, or insufficient (you do not want to use virtual memory).

For the first prototype we have implemented a straightforward solution with a greedy-like algorithm and constraints lists.

We call $c_{i,j}$ the available capacity of the attribute *i* (currently only CPU capacity) of the node *j*. In conjunction with priority coefficients k_i for each attribute we are able to calculate the local available capacity C_j of the node *j* which is given by (1)

$$C_j = \sum_i k_i \times c_{i,j} \qquad (1)$$

Using the priority coefficients k_i we can adapt the load-balancing scheme to the needs of different applications. Applications with a high demand in CPU and memory capacity like Pam-Crash will use a high value for these priority coefficients. All C_j will then be sorted. Sorting C_j in ascending order provides us a data-set which comprises the *capacity room C*. Sorting C_j in descending order provides the data-set for the *load room L*. Each of these data sets are managed as priority queues (heaps) as indicated by Figure 4.

The migration decider only looks at the first element of each heap. The first element of the *capacity room C* represents the node with the highest available capacity. Whereas the first element of the *load room L* represents the most heavily burdened node. A migration will be triggered, if the following conditions are met: $L_1 > T$ and $C_1 > 1 - T$ with i, j in $\{1, \dots, n\}$ and n = *number of nodes*, where *T* denotes an application specific threshold level for the task

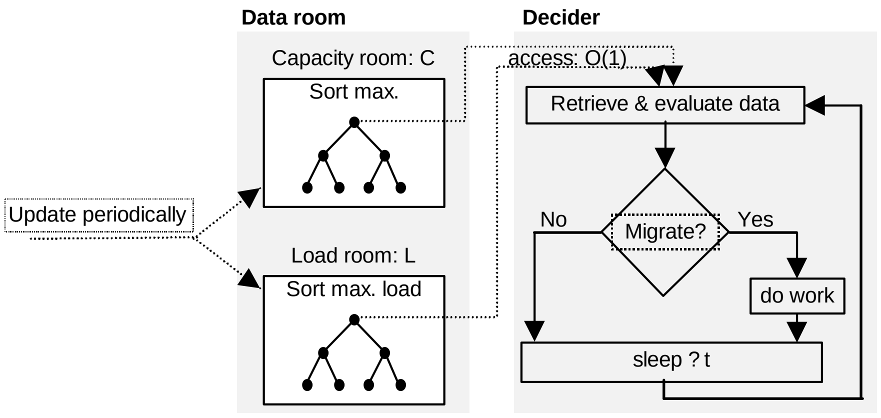

Fig. 4. : Architecture of the Migration Decider

migration. By using heaps for the data management, the migration decider task is able to retrieve the essential information with minimum effort *O(1)*. Additionally, updating elements in the data room can be done with *O(n * log n)*. Although there exist other schemes with faster access to the data elements (e. g. linked lists) if only a few number of tasks have to be considered but by using heaps we are not limited to support only a small number of tasks.

As illustrated in Figure 5, the algorithm of the decider is straightforward. The function `CheckForMigration` will be called periodically to check if the load index of the most loaded node is higher than a user defined threshold level and furthermore if a node exists which has enough remaining capacity (migration mapping). When the decision for migration is taken, the tasks are moved from the 'overloaded' node to the node with the best capacity left. Thereafter both data rooms are reordered by setting the load and capacity indices of the corresponding nodes to default values and by re-sorting the data heaps. By using a recursive algorithm, the whole migration is done in one global step. As a result, the application uses the whole workstation cluster efficiently and expensive compute time is not wasted migrating single tasks one at a time.

5 Job preparation

As is the case for every parallel application, an application using the DYNAMITE environment must be split into separate tasks. These tasks must be started on the nodes of the assigned cluster. Usually, in FEM applications, such as Pam-Crash [16], and many others, this is accomplished by partitioning the problem data over the available nodes in proportion to the capacity of a node. This will result in a tight fit, which is fine if there are no variations in load or capacity. For DYNAMITE we are considering two other approaches:

Sparse decomposition. When the aim is to allow any one node from a pool of (equal) workstations to be temporarily used for other purposes, the task should be split into fewer subtasks than the number of available nodes. In this way, flexibility is gained at a cost in performance.

Redundant decomposition. When the aim is to allow for the redistribution of work in an application that produces a dynamically changing load, it may be preferable to split the data so that every workstation gets more than one partition. In this way load can easily be shifted, albeit at a cost in communication efficiency.

Beside this additional choice in the partitioning,

```
CheckForMigration () {
    /* Will be triggered at least every t seconds */
    if (GetMaxLoadFromListOfLoadedNodes() <= Threshold) return;
    if (GetBestCapacity() > (1.0 - Threshold)) {
        /* there exists a node which is less burdened;
            do the migration stuff */
        DoMigrationStuff();
        UpdateLoadRoom();      /* effort: O(n * log n) */
        UpdateCapacityRoom();  /* effort: O(n * log n) */
        CheckForMigration();   /* do the recursion */
    }
}
```

Fig. 5. : Pseudo-Code of the Recursive Algorithm for the Decider Module

running a task under DYNAMITE also requires the monitoring tasks to be started together with the DPVM system. Though this need not require any additional effort on the side of the user, we will provide a simple GUI to assist the user in starting his DYNAMITE empowered application.

6 Conclusions

DYNAMITE will provide the application developer with a robust tool that makes it possible to respond flexibly to dynamic changes in the available system capacity and application workload. The DYNAMITE system will migrate (dynamically linked) tasks from a parallel program when necessary. The overhead involved will be very small compared to the possible cost of a load imbalance. The system structure is modular so that it can easily be adapted to specific application requirements. In the development phase this modularity will be used for experimentation with various migration policies.

References

[1] S. Zhou, X. Zheng, J. Wang and P. Delisle, *Utopia: A load sharing facility for large heterogeneous distributed computer systems,* Software – Practice and Experience, v. 23, n. 12, pp. 1305–1336, 1993

[2] http://www.genias.de/products/codine

[3] J. Pruyne and M. Livny, *Managing Checkpoints for Parallel Programs* - Poc. IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing, 1996

[4] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System* - Technical Report 1346, University of Wisconsin, WI, USA, 1997

[5] J. Casas, D.L. Clark, R. Konoru, S.W. Otto, R.M. Prouty and J. Walpole, *MPVM: A migration transparent version of PVM*, Usenix Computer Systems, v. 8, n. 2, Spring, pp. 171–216, 1995

[6] J. Casas, D. Clark, P. Galbiati, R. Konuru, S.Otto, R. Prouty and J. Walpole, *MIST: PVM with Transparant Migration and Checkpointing*, Third Annual PVM Users' Group Meeting, Pittsburgh, PA, 1995

[7] J. Robinson, S.H. Russ, B. Flachs, B. Heckel, A Task Migration Implementation of the Message-Passing Interface. Proceedings of the 5th IEEE international symposium on high performance distributed computing, pp. 61-68, 1996

[8] B.J. Overeinder, P.M.A. Sloot, R.N. Heederik, L.O. Hertzberger, *A dynamic load balancing system for parallel cluster computing,* Future Generation Computer Systems 12, pp. 101-115, 1996

[9] Matthias Brune, Jörn Gehring and Alexander Reinefeld, *Heterogeneous Message Passing and a Link to Resource Management,* Journal on Supercomputing, Vol. 11, Kluwer, Boston, pp. 355–369, 1997, http://www.uni-paderborn.de/pc2/services/public/1997/97012.ps.Z

[10] F. Bonomi and A. Kumar, *Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler*, IEEE Trans. on Computers, v. 39, n. 10, pp. 1232–1250, 1990

[11] J. Casas, R. Konoru, S.W. Otto, R. Prouty and J. Walpole, *Adaptive load migration systems for PVM*, Proceeedings of Supercomputing '94, Washington DC, pp. 390–399, 1994

[12] M. Hamdi and C.K. Lee, *Dynamic load balancing of data parallel applications on a distributed network*, Proceedings of 1995 International Conference on Supercomputing, Barcelona, pp.170–179, 1995

[13] R. von Hanxleden and L.R. Scott, *Load balancing on message passing architectures*, Journal of Parallel and Distributed Computing, v. 13, pp. 312–324, 1991

[14] R. Diekmann, B. Monien and R. Preis, *Load Balancing Strategies for Distributed Memory Machines*, Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, B.H.V. Topping (ed.), Saxe-Coburg, 1998

[15] T. Decker, M. Fischer, R. Lüling and S. Tschöke, *A Distributed Load Balancing Algorithm for Heterogeneous Parallel Computing Systems*, Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), H. R. Arabnia (ed.), CSREA Press, Volume II, pp. 933–940, 1998

[16] http://www.esi.fr/products/crash/index.html