Dynamic Migration of PVM Tasks *

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

Department of Computer Science, Universiteit van Amsterdam, Kruislaan 403,
1098 SJ Amsterdam, The Netherlands
email: (kamil,zegerh,dick,bjo,sloot)@wins.uva.nl

Keywords: cluster computing, load balancing, task migration, PVM

Abstract: *The total computing capacity of the workstations that are present in many organisations today is often under-utilised, as the performance for parallel programs is unpredictable. These computing resources 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. These tasks are part of a parallel program using a message passing library such as PVM or MPI. Currently Dynamite supports the PVM library only, but it can be extended to support the MPI library. 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-connections.*

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.

Dynamite [1] provides a dynamic load balancing system for parallel jobs running under PVM [2] when run on the aforementioned clusters of workstations. The load balancing is realised through the migration of tasks. For technical reasons which become clear later on, the (processor) architecture and the operating system version need to be the same.

Dynamite is currently operational under SunOS 5.5.1, SunOS 5.6 and Linux 2.2.x[1]. It aims to provide a complete solution for dynamic load balancing, see Section 6.

Dynamite is an acronym for DYNAMIc Task migration Environment and is also known as DPVM [3] (Dynamic–PVM), since it is based on PVM, version 3.3.11. Although Dynamite currently supports PVM-based programs only, the principles of Dynamite should be easily portable to MPI [4]. The modular design of Dynamite supports this portability as well. For MPI, task migration has already been studied in Hector [5]. Various PVM variants supporting task migration have been reported, such as tmPVM [6], ChaRM [7], DAMPVM [8] and MPVM [9]. Systems that migrate sequential jobs have also been studied, *e.g.* Codine [10] and Condor [11, 12].

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 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.

*This paper appeared in: K.A. Iskra; Z.W. Hendrikse; G.D. van Albada; B.J. Overeinder and P.M.A. Sloot: Dynamic Migration of PVM Tasks, in L.J. van Vliet; J.W.J. Heijnsdijk; T. Kielmann and P.M.W. Knijnenburg, editors, ASCI 2000, Proceedings of the sixth annual conference of the Advanced School for Computing and Imaging, pp. 206-212. ASCI, Delft, June 2000. ISBN 90-803086-5-x.

†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: J. Gehring, A. Streit, F. van der Linden, J. Clinckemaillie, A. H. L. Emmen.

[1] Only libc5 and glibc2.0 libraries are currently supported, glibc2.1 is expected to be supported soon.

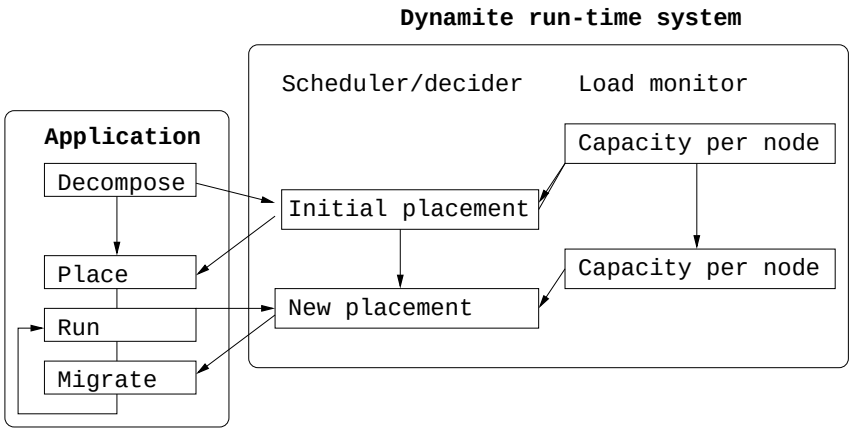

Figure 1: *Dynamite run-time system. An application is decomposed into several subtasks first. An initial placement is determined by the scheduler, one that needs not be optimal yet. When the application is run, the monitor checks the capacity per node. If it is decided that the load is unbalanced (above a certain threshold), one or more task migrations might be necessary to establish a new and more optimal load distribution.*

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. The following items which make up the state of a process are preserved:

   - direct and indirect PVM connections,
   - handling of shared libraries,
   - open files.

In this article we will focus on the latter, technically most challenging part of this system.

From the beginning, Dynamite was required to be as transparent to the user as possible. This implies a.o. 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[2] (which contains the checkpoint/restart mechanism and is a shared library itself; it is based on the Linux 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 pay some attention to the architecture of Dynamite in Sections 2 and 3. Thereafter quantitative results will be presented, which have been obtained with Dynamite running on a small Linux cluster. These data will be compared to standard PVM runs.

[2]The dynamic loader can be specified by using the appropriate compiler option.

2 Checkpointing mechanism

The checkpointing of a process basically boils down to writing the address space of a process to a file and retrieving its contents afterwards (`mmap`ping it to memory). This includes the shared libraries, which may be used by the process. In addition, the contents of (some of the) processor registers have to be taken care of, such as the program-counter and the stack pointer. Moreover, a proper implementation should also consider communication channels such as open files and TCP/IP sockets.

The checkpointing functionality was implemented in the dynamic loader, to which the following changes have been made:

1. it can handle a checkpoint signal (`SIGUSR1`), see Section 2.1,

2. it can treat a checkpoint file just like any other executable, see Section 2.2,

3. it *wraps* certain system and library calls, see Section 2.3:

   - for open files (a.o. `open`, `write`, `creat`),
   - for memory allocation (`mmap`, `munmap`, `mremap`[3]),

4. cross-checkpoint data is stored separately, see Section 2.4.

[3]Linux specific.

2.1 Checkpoint signal

When a checkpoint signal is sent to the process, control is passed to the checkpoint handler.

First of all a `sigsetjmp` call is made in order to save the current signal status and the contents of the processor registers (on Linux this is a `setjmp` call). The return value of `sigsetjmp` (zero if it is called for the first time) distinguishes between the checkpoint and restart procedure.

Next the name and location of the checkpoint-file is determined. If the application has been linked against the DPVM library, this name is determined by the `dpvm_usersave` routine in the DPVM library. The checkpoint file is placed in a directory which must be accessible from all the nodes in the cluster (indicated by the `DPVM_CKPTDIR` environment variable).

After saving the signal mask and the status of the open files, the checkpoint itself is created in the routine `ckpt_create`. Basically, this routine saves the address space of the process:

- the `.txt`-segment,
- the `.bss`-segment,
- the stack used by the process,
- the dynamically allocated pages,
- shared libraries used.

In addition, some extra sections are stored as well, such as the section containing the checkpoint filename and the section containing the cross-checkpoint data pointer, see Section 2.4.

2.2 Restoring from the checkpoint

When a binary is run, the dynamic loader is executed first. As soon as the dynamic loader has finished, control is passed to the actual program. Of course, this holds also for the Dynamite dynamic loader. One of the first things this loader tries to locate is the special section containing the name of the checkpoint file. If such a section is present, it knows that it is restoring from a checkpoint, and specialised subroutines take care of a proper handling of the process' segments.

Eventually, signal status and processor registers are restored, after which the process returns from `sigsetjmp` taking the appropriate branch for restored programs.

Finally, the process resumes its execution at the point where it left off.

2.3 Wrapped system calls

The reason for wrapping certains system- and library-calls is that the checkpointing/restart (*i.e.* migration) facility should be able to deal with open files. Basically, these wrapper routines invoke the original C-library calls, doing some extra administration, which allows the open file connections to be restored properly.

The reason for implementing the syscall-wrapper `mmap` is different, however. Of course, the memory allocated by this syscall must be restored too, when restarting a checkpointed process. This implies that all the memory allocations done by `mmap` have to be monitored as well[4]. This holds also for the mapping of the shared libraries by the process.

2.4 Cross-checkpoint storage

A data structure is defined as a container for those objects which need to be preserved across a checkpoint/restart, such as the mapping of the shared libraries used by the process or the status of the open files. This data structure is also part of the checkpoint file.

3 The DPVM library

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.

3.1 PVM migration overview

The network of PVM daemons plays a central role in initiating and co-ordinating the migration of tasks. On reception of the `move` command, control is passed to the `pvm_move` function, which steps through the following stages successively:

1. `PvmMoveCreateContext`
2. `PvmMoveRouteBroadcast`
3. `PvmMoveCheckpoint`
4. `PvmMoveRestart`

The `PvmMoveCreateContext` stage is executed on the destination node, *i.e.* the node where the task is to be migrated to. A new PVM task context is created, so that the PVM daemon can accept

[4] Although under Solaris these `mmap`-calls are merely invoked by the locale/nls-related libraries, it is used frequently by the standard Linux C-library. Therefore it was and is important that these memory regions are taken care of properly.

any messages addressed to the migrating task and temporarily store them.

In the `PvmMoveRouteBroadcast` stage, all PVM daemons but the source and destination one are notified that a migration is about to take place. The daemons update their routing information, so that messages sent via the daemons to the migrating task are sent to the destination node.

The `PvmMoveCheckpoint` stage is executed on the source node, *i.e.* the node the task runs on before the migration takes place. First, routing information is updated, so that any messages sent to the migrating task via the PVM daemon are forwarded to the destination node instead of being delivered locally. Finally, the task finds out that it is to be migrated. A `SIGUSR1` signal is sent to the task by the PVM daemon, along with the end-of-connection `TC_EOC` message. Control is passed to the checkpoint signal handler in the Dynamite dynamic loader. However, before the actual checkpointing takes place, the signal handler invokes the DPVM function `dpvm_usersave`, which reads all the available data from all connections, closes the task connections and sends the final `TM_MIG` migration message to the local PVM daemon. Subsequently, the checkpoint handler creates the checkpoint file and terminates the process.

In the final `PvmMoveRestart` stage, executed on the destination node, the task is restarted at the new location using the `spawn_task` function. In the process of restarting the task from the checkpoint file, the dynamic loader invokes the DPVM function `dpvm_userrestore`, which reconnects the restored task to the PVM daemon on the destination node. Control is passed back to the application code, and the PVM daemon can finally deliver all messages addressed to the migrating task which it had to store during the migration.

3.2 Direct connections

By default, PVM tasks use indirect connections to communicate with each other. In this mode, messages between tasks are routed through two PVM daemons, local to the source and destination tasks. As a consequence, PVM application tasks do not have any remote network connections open, their only communication channel is with the daemon.

To improve efficiency, an alternative direct communication mode is available on application request. In this mode, tasks that wish to communicate with each other can establish a direct TCP/IP network connection between themselves.

Special care must be taken when migrating a task that has direct connections with other tasks, or messages that are being processed or are cached in the kernel buffers will be lost during the migration.

In stages `PvmMoveCreateContext` to `PvmMoveCheckpoint`, along with updating the routing information, DPVM notifies all PVM tasks that a migration is about to take place. This is done by sending special `TC_MOVED` control messages to all tasks. Because it is important that the tasks reply in a timely manner, PVM daemons also send a `SIGURG` signal along with the `TC_MOVED` messages. It is the responsibility of the asynchronously invoked signal handler function `dpvm_oobhandler` to get the message.

In `PvmMoveCheckpoint` stage, in the `dpvm_usersave` function, the migrating task sends the `TC_EOC` message via all open direct connections. The peer tasks read all data from the connection until they receive `TC_EOC`, at which point they send the `TC_EOC` message back. The migrating task reads all data on its side of the connection, and closes the connection upon reception of `TC_EOC`. The peer tasks receive `EOF` at this point, and can close the connection on the other side.

Any messages that were only partially sent by the migrating task are fully resent after the task is restarted. Any messages that were partially sent by the peers of the migrating task are fully resent via PVM daemons, *i.e.* indirectly. The direct connection is reestablished as soon as the migrating task restarts and there are new messages to be sent.

4 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, e.g., is practically unsolvable.

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.

5 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 [16] 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 [13], which run Linux kernel 2.0 and 2.2 on PentiumPro 200 MHz CPUs. The scientific application Grail [14, 15], a FEM simulation program, has been used as the test application.

	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 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 tests can be grouped into two categories, depending on the decomposition used:

- **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, because for the sparse decomposition only 3 nodes at a time are used, just like for the redundant one.

In the first set of tests presented in Table 1, 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. 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 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. 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 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 shell, cannot be 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 desperate attempts to migrate the tasks actually exacerbated the situation, due to the lack of nodes with significantly lower load. It can be argued that the migrator should have refrained from making any migrations in this case, though.

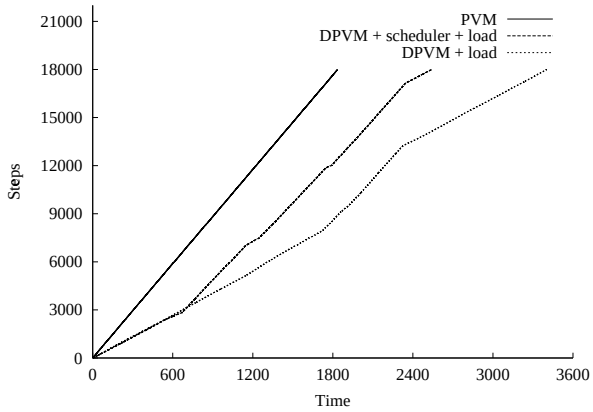

Figure 2: *Execution progress of Grail for sparse decomposition. Note that the PVM run was done without any external load on the system. Allowing Dynamite to migrate tasks results in a clear performance gain, when there is a time-varying external load (see text).*

Figure 2 presents the execution progress of Grail for sparse decomposition. For the 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 from the start, the node with the master task is overloaded again, so the performance deteriorates in both DPVM cases.

6 Conclusions and future prospects

Concluding, the concept of load balancing by task migration has been shown to work. Moreover, we have succeeded in implementing such a system completely in user space. Since the system is stable now, further study on the scheduler can be carried out.

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

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 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.

References

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

[2] 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, Massachusetts, 1994. http://www.epm.ornl.gov/pvm/

[3] 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, 101–115, 1996.

[4] MPI: A Message-Passing Interface Standard, Version 1.1. Technical Report, University of Tennessee, Knoxville, TN, June 1995. http://www-unix.mcs.anl.gov/mpi/

[5] 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, 61–68, 1996.

[6] C. P. Tan, W.F. Wong, C.K. Yuen, tmPVM — Task Migratable PVM, Proceedings of the 2nd Merged Symposium IPPS/SPDP. pp. 196-202.5. April 1999.

[7] P. Dan, W. Dongsheng, Z. Youhui, S. Meiming, Quasi-asynchronous Migration: A Novel Migration Protocol for PVM Tasks, Operating Systems Review v. 33 n. 2, ACM, 5–14, April 1999.

[8] P. Czarnul, H. Krawczyk, Dynamic allocation with process migration in distributed environments, in J. J. Dongarra and E. Luque and Tomas Margalef, editors, Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999, in series Lecture Notes in Computer Science, nr 1697, 509–516

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

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

[11] J. Pruyne, M. Livny, Managing checkpoints for parallel programs, Proceedings IPPS second workshop on job scheduling strategies for parallel processing, 1996.

[12] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and migration of Unix processes in the Condor distributed processing system, Technical Report 1346, University of Wisconsin, WI, USA, 1997.

[13] The Distributed ASCI Supercomputer (DAS). http://www.cs.vu.nl/das/

[14] J.F. de Ronde, G.D. van Albada, P.M.A. Sloot, High Performance Simulation of Gravitational Radiation Antennas, in L.O. Hertzberger, P.M.A. Sloot, editors, High Performance Computing and Networking '97, in series Lecture Notes in Computer Science, 200–212. Springer–Verlag, April 1997.

[15] J.F. de Ronde, G.D. van Albada, P.M.A. Sloot, Simulation of Gravitational Wave Detectors, Computers in Physics, vol. 11, nr 5, 484–497, September 1997.

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