MPI Farm programs on non-dedicated clusters

Nuno Fonseca[1], João Gabriel Silva[2]

[1]CISUC - Polytechnic Institute of Leiria, Portugal, nfonseca@estg.ipleiria.pt
[2]CISUC - DEI - University of Coimbra, Portugal, jgabriel@dei.uc.pt

Abstract. MPI has been extremely successful. In areas like e.g. particle physics most of the available parallel programs are based on MPI. Unfortunately, they must be run in dedicated clusters or parallel machines, being unable to use for long running applications the growing pool of idle time of general-purpose desktop computers. Additionally, MPI offers a quite low level interface, which is hard to use for most scientist programmers. In the research described in this paper, we tried to see how far we could go to solve those two problems, keeping the portability of MPI programs, but drawing upon one restriction - only programs following the FARM paradigm were to be supported. The developed library - MpiFL - did provide us significant insight. It is now being successfully used at the physics department of the University of Coimbra, despite some shortcomings.

1. Introduction[1]

MPI has made possible a very important development among scientists: parallel program portability. Up to not many years ago, parallel number crunching programs in areas like e.g. particle physics were made for particular combinations of operating systems and communication libraries, severely hindering their widespread use. With MPI, things changed. An MPI program made for a very expensive massively parallel machine also runs on a cheap cluster made of PCs. Other middleware communication packages had the same potential, but none managed to become a de facto standard like MPI did.

Unfortunately, MPI does not tolerate node failures. If a single machine in a cluster fails, the whole computation is lost. Non-dedicated clusters, built by bringing together the idle time of workstations with other uses, are out-of-reach of MPI programs, since in those ad-hoc clusters nodes become unavailable quite often. The reason can be a network cable that is plugged out, or a computer switched off, or simply because the owner of a node launches a task that uses up all available CPU time. Even in dedicated clusters failures do happen, although much less frequently, forcing many programmers to resort to "hand-made" checkpointing, regularly saving intermediate

[1] The authors would like to thank Hernani Pedroso and João Brito (Critical Software); Paulo Marques and Fernando Nogueira (University of Coimbra); Nelson Marques and Patricio Domingues (Polytechnic Institute of Leiria) for their help. This work was partially supported by the Portuguese Ministry of Science and Technology and by the European Union, under R&D Unit 326/94 (CISUC), projects POSI/EEI/1625/95 (PARQUANTUM) and POSI/CHS/34832/99 (FT-MPI).

data, and manually restarting application from that data when the computation crashes. There are essentially two approaches to deal with this problem.

One is to use a different middleware altogether, capable of tolerating failures and cycle stealing. The best example is probably the Condor system [Tannenbaum 02], which can handle non-dedicated clusters in a very efficient way. Still, Condor has a different API, and quite different execution model, from MPI. Switching to Condor means losing the main advantage of MPI: program portability. While Condor is becoming popular in clusters, it has little expression in dedicated parallel machines. If Condor usage becomes more widespread, it may be able to offer similar portability to MPI, but it is hard to say whether that will happen. It is true that Condor is capable of running MPI programs, but to do so fault-tolerance is lost. MPI and non-dedicated clusters are thus not brought together by Condor.

The other solution is to change MPI to make it fault tolerant. There have been several attempts for doing that, like Co-check [Stellner 96], Startfish [Agbaria 99], MPI-FT [Louca 00], FT-MPI [Fagg 00] and MPI/FT [Batchu 01]. Some of these systems provide transparent fault-tolerance and run unmodified programs, others require changes to user programs, with some (like e.g. Startfish) supporting both mechanisms. The underlying tradeoff is that there is a bigger performance penalty for transparent checkpointing, and implementation is significantly more complex. Still, none of these systems are readily available, and most did not go beyond the state of limited prototypes. This is anyhow a promising approach, but a complex one, that will still take some time before becoming mainstream.

Without denying the merits of any of these approaches, we decided to explore a different route. In order to keep full portability, we tried to build a layer above MPI that could run over any unmodified MPI implementation, but would support non-dedicated clusters by offering fault-tolerance.

As a secondary goal, we wanted to offer a higher-level API, better adapted to the type of problems scientists were facing, e.g. at the Physics department of the University of Coimbra. After analyzing their programs were concluded that all of them were already implemented in the farm paradigm (also known as master/worker), or could be easily changed to fit that paradigm. In the farm paradigm there is one master node that distributes subtasks to the other nodes (known as workers or slaves) The slaves do not communicate among themselves, only with the master, which collects the slave outputs and from them computes the collective result of the computation (for a discussion of the main paradigms of parallel programs see [Silva 99]).

The farm paradigm is very interesting for fault-tolerance purposes because, since everything goes through the master, that node becomes an obvious place to do checkpointing. Since non-dedicated clusters are generally heterogeneous, load balancing is also required; otherwise the slowest node would limit the whole computation. Fortunately, in the farm paradigm this balancing is obtained almost for free: faster nodes will finish their jobs first, and request more jobs from the master, thus getting a bigger share of the job pool than slower nodes, which will get less jobs to perform. This works well as long as the number of jobs to distribute is significantly above the number of available nodes.

The library we built was thus called MpiFL (MPI-Farm Library).

Any library so devised necessarily has a drawback, similar to what was pointed before to Condor - it has a different API than MPI. This is the price to pay to provide the user a higher level API. But, contrary to Condor, we have the potential not to lose the main advantage of MPI - portability. If, of course, we are able to build such a library to run over any MPI implementation, from big machines to small clusters.

The idea to build such libraries is not new. Among them is MW, built over Condor [Goux 01]; we have developed in the past something similar for the Helios OS [Silva 93]; and in the Edinburgh Parallel Computing Centre a number of such libraries was also developed some years ago over MPI, but with no fault tolerance [Chapple 94]. The *algorithmic skeleton* community has also started to work on libraries for MPI, which promise some level of standardization of the particular skeletons used, and allow the programmer to go on using their familar environments, like C++ and MPI [Kuchen 02], but do not address the complex issues of faults, errors and failures.

In spite of several other variations reported in the literature, we could not find any other farm library over MPI that had been built with our set of requirements in mind: high portability over MPI implementations, support for non-dedicated clusters (i.e. fault-tolerance) and an high-level, easy to use API. These are the goals of MpiFL.

The paper is structured as follows. After this introduction, section 2 discusses how far fault-tolerance can be obtained under the chosen constraints. Section 3 then discusses the interface the library offers to programmers. Section 4 describes the internals of the library and some results of its usage. Section 5 closes the paper, presenting some conclusions and future research directions.

2. Fault-Tolerance over MPI

Loss of nodes. In both version 1 and 2 of the MPI standard [mpi 1.1][mpi 2.0], fault tolerance is specified only for the communication channels, which are guaranteed to be reliable, but not for process faults. If a process or machine fails the default behaviour is for all other nodes participating in the computation to abort. The user may change this by providing error handlers, but the system may not even be able to call them, and even if it does they are useful only for diagnostic purposes, as only sometimes (implementation dependant) will it be possible to continue the computation after the execution of the error handler, as the standard does not specify the state of a computation after an error occurs.

In a master worker configuration we would like to be able to continue using the remaining workers if one of them fails. Although the standard does not guarantee that, a particular implementation may sometimes make it possible to just ignore the failed processor and go on using the others, depending on the type of fault. We can benefit from that possibility whenever it is available, through what we call "lazy fault-recovery": the default behaviour of aborting everything when a single process is lost is switched off by defining new error handlers and, as long as most of the worker nodes still make progress, the computation is kept alive. When it is judged that the price of a global restart of the application is outweighed by the benefit of adding to the virtual machine the nodes that in the meantime became available (possibly including some or all of the failed workers, if the problem that led them to stop was transient) the computation should be stopped and relaunched with the full set of available machines. With this approach we manage to use up as much capacity to work in a degraded mode as each MPI implementation provides.

"Lazy fault-recovery" is made possible because of another feature of MpiFL, which is also useful for load balancing. Even in the absence of faults, since in non-dedicated clusters nodes can have quite different processing capabilities, when the list of tasks to be distributed to the workers is exhausted, MpiFL distributes to the idle workers the jobs that have already been distributed to some worker but whose result is not yet available. In this way, a fast node may finish a job faster than the slow node it was originally distributed to. This also works when a task is assigned to a node that fails - since that node will not return a result, the task will eventually be sent to a different worker, and the application reaches completion.

Please note that the notion of "failed node" includes the case when the rightful owner of a machine starts using it heavily and stalls the process running on behalf of MPI, which runs with low priority. A slow node is thus equivalent to a failed node. If later the slow node still produces the result of its assigned job, it will simply be ignored. There is one drawback to this scheme - we are not able to stop a worker that is processing a job whose result we already know is not needed. The only way to do that is abort the whole computation and restart it. MpiFL gives the user the possibility of specifying this behaviour.

System monitoring. As should result clear from the previous section, MPI has no mechanism no determine when a node fails, or when a new one becomes available. An external monitoring service is needed. This service is operating system (OS) specific, and so is not portable across different MPI implementations running on different OSes, constituting the biggest limitation to a full MpiFL portability. Even if such a monitor could be based only on MPI calls, we would need to have it separate from the main computation, so that the monitor could survive when the MPI computation aborts, not only to be able to signal that a particular MPI process died, but also to be able to relaunch an MPI application that aborted because a remote machine died.

The type of diagnostic made by this monitor must be more fine-grained than just the usual detection of machine crashes - even if a machine is up, it has to detect whether all the requirements for an MPI process to run are satisfied, and also whether the machine has sufficient idle time to be able to constructively participate in a computation.

In the current version of MpiFL, the loss of the monitor in the master machine is the sole case that requires manual intervention to restart the MPI computation. A mechanism to do this automatically is not difficult to build, and may be included in a future version.

Recovery and restart. To recover a computation after a crash we use periodic checkpoints, which are written to disk by the master (optionally to several disks in different machines, to make the system tolerant to crashes of the master machine). The content of the checkpoints is basically the results of the already executed tasks, which makes it easy to migrate the checkpoints, even across different architectures and OSes.

Since there are still very few implementation of the MPI-2 standard, we decided to base MpiFL on MPI version 1.1 [mpi 1.1]. On MPI-2 we could add dynamically new processes, but on MPI 1.1 a restart is needed to add new machines, which results in some wasted processing. When a restart is needed because of a computation crash, we obviously seize the opportunity to include all available nodes. Otherwise the monitor has to force an abort to include new machines, a decision that is only taken when the estimate of the time to complete the application with the new machines is clearly lower than without them, even taking into account the cost of the abort and the restart.

Master

```
#include <mpifl.h>
main() {
int a,b;
MpiFL_MasterInit();

// Handle Non-deterministic
//  code here (optional)
if (MpiFL_FirstStart()) {
   scanf("%d",&a)
   b=rand(); }
MpiFL_Restart (&a, sizeof(a));
MpiFL_Restart (&b, sizeof(b));

//deterministic code
// Produce jobs
for ()
   MpiFL_SendJob(..);
// Consume
for ()
   MpiFL_GetResults(..);
…
MpiFL_MasterClose();
}
```

Slave

```
#include <mpifl.h>
main() {

MpiFL_SlaveInit();


while
 (MpiFL_GetJob()>0){
  //Process
   …
  MpiFL_SendResults()
 }
MpiFL_SlaveClose();
}
```

Figure 1 - Templates for master and slave. Other more complex variations are also possible, like generating new jobs based on the results of the previous ones.

3. High-Level interface for Farm programs

The programmer only has to provide the master, that produces the jobs for the workers and collects the results, the workers that process the jobs (see figure 1), and the configuration file, that indicates where the master, workers and checkpoints should be. The library does everything else.

When the MPI computation crashes, the monitor restarts it. The master executes from the beginning, and at the call to *MpiFL_MasterInit* the library reads the checkpoint contents. The master then proceeds to generate the jobs again, but the library will recognize those that have already been processed before and replays their results to the master without calling any worker. Only those that had not been processed before will be forwarded to the workers. The master redoes all its processing after each restart, which can be inconvenient; but to do otherwise, we would have to ask the user to do additional calls to the library to set checkpoints, or do transparent checkpoints and lose portability [Silva 95][Silva 98]. We chose portability and the simpler interface.

With this model, the programmer must assure that in the case of a restart, the master deterministically recreates identical jobs. If it has non-deterministic code, it will have to execute that code only once, using the *MpiFL_FirstStart* function (see figure 1). Since the *MpiFL_FirstStart* function only returns *true* in the first execution of the program, the code inside the *if* will only be executed once. The variables that store the outcome of that non-deterministic code (*a* and *b* in the example of Figure 1), are saved by the call to *MpiFL_Restart* in the first execution, and restored in executions that result from a restart caused by a failure.

4. MpiFL Library

The internal structure of MpiFL is presented in figure 2. All communication is done through MPI, except between the master and the monitor, that use pipes, although they could use TCP sockets. The checkpoints can be replicated in several machines to ensure their survivability.

We have made two implementations of MpiFL, one for Windows (using WMPI from Critical Software - http://www.criticalsoftware.com/HPC/), and one for Linux (using MPICH - http://www-unix.mcs.anl.gov/mpi/mpich/). Both versions of the library support C, C++ and Fortran90.

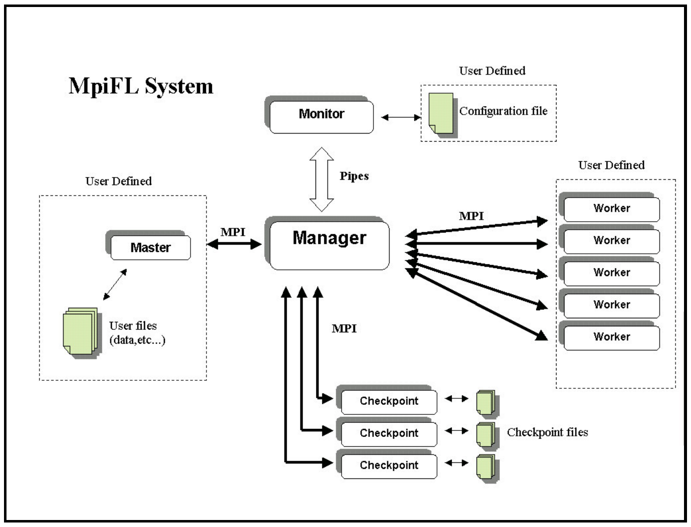

Figure 2 - MpiFL

We have subject both versions of MpiFL to many faults, like the abrupt killing of MpiFL process, many kinds of communication problems (removal of network cables, disabling of NIC's, stopping networking services, etc.), machine problems (reboots, stopping MPI services, etc), checkpoint problems (inconsistent checkpoint files, removal of checkpoint files, etc), and the library recovered always, except when the monitor in the master machine was affected.

The library was also tested for two weeks in a non-dedicated cluster composed of 30 machines in computer rooms used by computer engineering students, a particularly hostile environment (only the master machine was in a restricted access lab), and it was never down, in spite of the frequent node losses and additions.

To judge the performance penalty the library might represent, many test were made with a traveling salesman optimization problem, with different mixes of machines and granularity of tasks, and the overhead introduced by the library was always negligible (below 1%). The experience of usage in real problems in the Physics department points in the same direction, although measurements were not done systematically there. Generically, with different programs the performance overhead can become relevant, being determined essentially by the amount of data that has to be saved to disk, as is well known for a long time [Elnozahy 92].

5. Conclusions and Future Work

The main conclusion of this research is indeed the strong need for a clean failure semantics in MPI. The parallel machines that were dominant when MPI was first devised have now been largely replaced by clusters that have high failure rates that must be dealt with. In the case of non-dedicated clusters because these are inherently unstable environments; in the case of dedicated clusters because of the rapidly growing number of machines. If MPI is not able to solve this problem, it will probably soon lose the dominant position it now has as the preferred execution environment for parallel programs.

In spite of this shortcoming of MPI, we have shown that very useful levels of fault-tolerance can be achieved without significantly compromising portability, while at the same time offering the average scientist programmer a much easier programming model for the farm paradigm.

For the MpiFL library we are considering three enhancements: a fully automatic restart also for the case of loss of the monitor in the master machine; using the dynamic process creation capability of MPI 2.0 to add new nodes without having to restart the whole computation; and perfecting the "lazy fault recovery" mechanism to better use all capability that each particular MPI implementation may have to go on working with lost nodes.

References

[Agbaria 99] A. Agbaria and R. Friedman. "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations". In the 8th IEEE International Symposium on High Performance Distributed Computing, pages 167-176, August 1999.

[Batchu 01] Rajanikanth Batchu, Jothi P. Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, Yoginder Dandass, Manoj Apte, "MPI/FT(tm): Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing," The Third International Workshop on Software Distributed Shared Memory (WSDSM' 01), May 16 18, 2001, Brisbane, Australia.

[Chapple 94] Simon Chapple and Lyndon Clarke "PUL: The Parallel Utilities Library" Proceedings of the IEEE Second Scalable Parallel Libraries Conference, Mississipi, USA, 12-14 October, 1994, IEEE Computer Society Press, ISBN 0-8186-6895-4.

[Elnozahy 92] E.N. Elnozahy, D.B.Johnson, W.Zwaenepoel "The Performance of Consistent Checkpointing", Proc. 11th Symposium on Reliable Distributed Systems, pp. 39-47, 1992, IEEE Computer Society Press.

[Fagg 00] Graham E. Fagg, Jack J. Dongarra "FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world", ; EuroPVM/MPI, User's Group Meeting 2000, Spring-Verlag, Hungary, September 2000, pp. 346-353.

[Goux 01] J.-P Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder, "Master-Worker: An Enabling Framework for Applications on the Computational Grid' '*Cluster Computing* 4 (2001), pp. 63-70.

[Kuchen 02] H. Kuchen, "A Skeleton Library" Proceedings of Euro-Par 2002, LNCS, Springer-Verlag, 2002.

[Louca 00] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas And Paraskevas Evripidou "MPI-FT: Portable Fault Tolerance Scheme for Mpi" Parallel Processing Letters, Vol. 10, No. 4 (2000) 371-382.

[mpi 1.1] "MPI: A Message-Passing Interface Standard", Version 1.1, Message Passing Interface Forum, June 12 1995, http:\\www.mpi-forum.com

[mpi 2.0] "MPI-2: Extensions to the Message-Passing Interface", Message Passing Interface Forum, July 18, 1997, http://www.mpi-forum.com

[Silva 93] Luís Moura e Silva, Bart Veer e João Gabriel Silva "How to Get a Fault-Tolerant Farm", In R. Grebe, J. Hektor, S.C. Hilton, M. R. Jane, P.H. Welch (eds.) "Transputer Applications and Systems ' 93" Vol. 36 in the Series "Transputer and Occam Engineering", IOS Press, 1993, Amsterdam. Vol. 2, pp 923-938, ISBN 90-5199-140-1.

[Silva 95] Luís M. Silva, João Gabriel Silva, Simon Chapple, Lyndon Clarke "Portable Checkpointing and Recovery" 4th IEEE International Symposium on High Performance Distributed Computing (HPDC-4), Pentagon City, Virgínia, USA, August 2-4, 1995, pp188-195, IEEE Computer Society Press, ISBN 0-8186-7088-6

[Silva 98] Luís M. Silva, João Gabriel Silva. "System-Level versus User-Defined Checkpointing". 17th IEEE Symposium on Reliable Distributed Systems, 1998, October 20-23, West Lafayette, USA, IEEE Computer Society, p. 68-74, ISBN 0-8186-9218-9

[Silva 99] Luis Moura Silva, Rajkumar Buyya, "Parallel programming models and paradigms", in R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems: Volume 2", Prentice Hall PTR, NJ, USA, 1999.

[Stellner 96] Georg Stellner. "CoCheck: Checkpointing and Process Migration for MPI". In Proceedings of the International Parallel Processing Symposium, pages 526-531, Honolulu, HI, April 1996. IEEE Computer Society Press.

[Tannenbaum 02] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny, "Condor - A Distributed Job Scheduler", in Thomas Sterling, editor, Beowulf Cluster Computing with Linux, The MIT Press, 2002, ISBN 0-262-69274-0.