A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids

Stéphane Genaud and Choopan Rattanapoka

ICPS-LSIIT - UMR CNRS-ULP 7005
Université Louis Pasteur, Strasbourg
`{genaud, rattanapoka}@icps.u-strasbg.fr`

Abstract. This paper presents P2P-MPI, a middleware aimed at computational grids. From the programmer point of view, P2P-MPI provides a message-passing programming model which enables the development of MPI applications for grids. Its originality lies in its adaptation to unstable environments. First, the peer-to-peer design of P2P-MPI allows for a dynamic discovery of collaborating resources. Second, it gives the user the possibility to adjust the robustness of an execution thanks to an internal process replication mechanism. Finally, we measure the middleware performances on two NAS benchmarks.

Keywords: Grid, Middleware, Peer-to-peer, MPI, Java.

1 Introduction

Grid computing offers the perspective of solving massive computational problems using a large number of computers. It involves sharing heterogeneous resources located in different places, belonging to different administrative domains over a network. When speaking of computational grids, we must distinguish between grids involving stable resources (e.g. a supercomputer) and grids built upon versatile resources, that is computers whose configuration or state changes frequently. The latter are often referred to as *desktop grids* and may in general involve any unused connected computer whose owner agrees to share its CPU. Thus, provided some magic middleware glue, a desktop grid may be seen as a large-scale computer cluster allowing to run parallel application traditionally executed on parallel computers. However, the question of how we may program such an heterogeneous cluster remains unclear. Most of the numerous difficulties that people are trying to overcome today fall in two categories.

– **Middleware.** The middleware management of tens or hundreds grid nodes is a tedious task that should be alleviated by mechanisms integrated to the middleware itself. These can be fault diagnostics, auto-repair mechanisms, remote update, resource scheduling, data management, etc.
– **Programming model.** Many projects propose a client/server (or RPC) programming style for grid applications (e.g. JNGI [17], DIET [4] or XtremWeb [8]). However, the *message passing* and *data parallel* programming model are the two models traditionally used by parallel programmers.

MPI [13] is the *de-facto* standard for message passing programs. Most MPI implementations are designed for the development of highly efficient programs, preferably on dedicated, homogeneous and stable hardware such as supercomputers. Some projects have developed improved algorithms for communications in grids (MPICH-G2 [10], PACX-MPI [9], MagPIe [11] for instance) but still, assume hardware stability. This assumption allows for a simple execution model where the number of processes is static from the beginning to the end of the application run[1]. This design means no overhead in process management but makes fault handling difficult: one process failure causes the whole application to fail. This constraint makes traditional MPI applications unadapted to run on grids. Moreover, MPI applications are OS-dependent binaries which complicates execution in highly heterogeneous environments.

If we put these constraints altogether, we believe a middleware should provide the following features: a) self-configuration (system maintenance autonomy, discovery), b) data management, c) robustness of hosted processes (fault detection and replication), and d) abstract computing capability. The rest of the paper shows how P2P-MPI fulfills these requirements. We first describe (section 2) the P2P-MPI middleware through its modules, so as to understand the protocols defined to gather collaborating nodes in order to form a platform suitable for a job request. In section 3 we explain the fault-detection and replication mechanisms in the context of message-passing programs. We finally discuss P2P-MPI behavior in section 4, at the light of experiments carried out on two NAS benchmarks.

2 The P2P-MPI Middleware

2.1 Modules Organization

Fig. 1 depicts how P2P-MPI modules are organized in a running environment. P2P-MPI proper parts are grayed on the figure. On top of diagram, a message-passing parallel program uses the MPI API (a subset of the MPJ specification

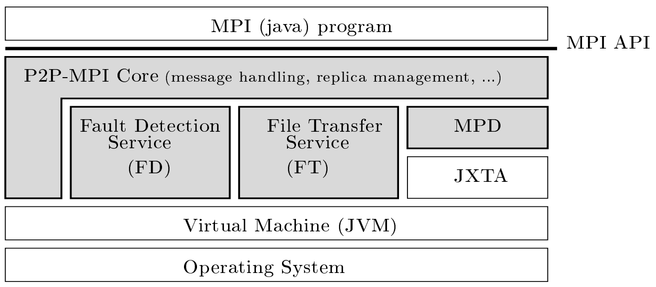

MPI (java) program
MPI API
P2P-MPI Core (message handling, replica management, ...)
Fault Detection Service (FD)
File Transfer Service (FT)
MPD
JXTA
Virtual Machine (JVM)
Operating System

Fig. 1. P2P-MPI structure

[1] Except dynamic spawning of process defined in MPI-2.

[5]). The core behind the API implements appropriate message handling, and relies on three other modules. The *Message Passing Daemon* (MPD) is responsible for self-configuration, as its role is either to search for participating nodes or to act as a gate-keeper of the local resource. The *File Transfer Service* (FT) module handles the data management by transferring executable code, input and output files between nodes. The *Fault Detection Service* (FD) module is necessary for robustness as the application needs to be notified when nodes become unreachable during execution.

In addition, we also rely on external pieces of software. The abstract computing capability is provided by a Java Virtual Machine and the MPD module uses JXTA [1] for self-configuration.

2.2 Discovery for An Execution Platform

In the case of desktop grids, the task of maintaining an up-to-date directory of participating nodes is a so tedious task that it must be automated. We believe one of the best options for this task is *discovery*, which has proved to work well in the many *peer-to-peer* systems developed over the last years for file sharing. P2P-MPI uses the discovery service of JXTA. The discovery is depicted in JXTA as an advertisement publication mechanism. A peer looking for a particular resource posts some public advertisement (to a set of decentralized peers called rendez-vous) and then waits for answers. The peers which discover the advertisement directly contact the requester peer.

In P2P-MPI, we use the discovery service to find the required number of participating nodes at each application execution request. Peers in P2P-MPI are the MPD processes. When a user starts up the middleware it launches a MPD process which publishes its *pipe* advertisement. This pipe can be seen as an open communication channel that will be used to transmit boot-strap information.

When a user requests n processors for its application, the local MPD begins to search for some published pipe advertisements from other MPDs. Once at least n peers have reported their availability, it connects to the remote MPDs via the pipe to ask for their FT and FD services ports. The remote MPD acts as a gate-keeper in this situation and it may not return these service ports if the resource had changed its status to unavailable in the meantime. Once enough hosts have sent their service ports, we have a set of hosts ready to execute a program. We call this set an *execution platform* since the platform lifetime is not longer than the application execution duration.

2.3 Job Submission Scenario

We now describe the steps following a user's job submission to a P2P-MPI grid. The steps listed below are illustrated on Figure 2.

(1) The user must first join the grid. By invoking `mpiboot`, it spawns the MPD process which makes the local node join the P2P-MPI group if it exists, or creates it otherwise.

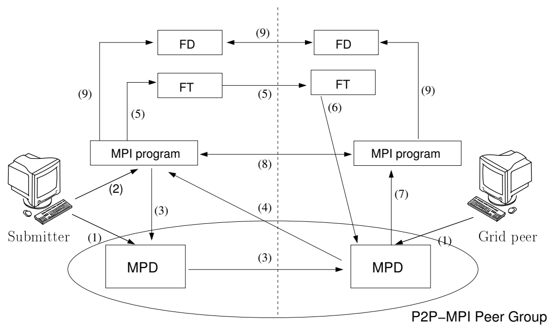


Fig. 2. A submission where the submitter finds one collaborating peer

(2) The job is then submitted by invoking a run command which starts the process rank 0 of the MPI application on local host.
(3) Discovery: the local MPD issues a search request to find other MPDs pipe advertisements. When enough advertisements have been found, the local MPD sends into each discovered pipe, the socket where the MPI program can be contacted.
(4) Hand-shake: the remote peer sends its FT and FD ports directly to the submitter's MPI process.
(5) File transfer: program and data are downloaded from the submitter host via the FT service.
(6) Execution Notification: once transfer is complete the FT service on remote host notifies its MPD to execute the downloaded program.
(7) Remote executable launch: MPD executes the downloaded program to join the execution platform.
(8) Execution preamble: all processes in the execution platform exchange their IP addresses to construct their local communication table.
(9) Fault detection: MPI processes register in their local FD service and starts. Then FD will exchange their heart-beat message and will notify MPI processes if they become aware of a node failure.

3 Replication for Robustness

3.1 Replication

Though absolutely transparent for the programmer, P2P-MPI implements a replication mechanism to increase the robustness of an execution platform. When specifying a desired number of processors, the user can request that the system run for each process[2] an arbitrary number of copies called *replicas*. In practice, it is shorter to request the same number of replicas per process, and we call this

[2] Except for rank 0 process. We assume a failure on the submitter host is critical since the user would lose the control on the application.

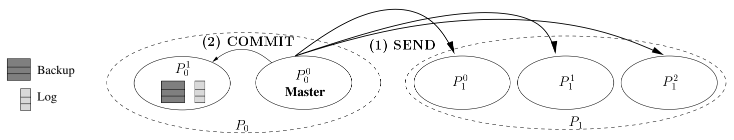


Fig. 3. A message sent from logical process P_0 to P_1

constant the *replication degree*. In the following we name a "usual" MPI process a *logical process*, noted P_i when it has rank i in the application. A logical process P_i is thus implemented by one or several replicas, noted $P_i^0, \ldots, P_i^n$. The replicas are run in parallel on different hosts since the goal is to allow the continuation of the execution even if one host fails.

Of course, replicas behavior must be coordinated to insure that the communication scheme is kept coherent with the semantics of the original MPI program. Ad hoc protocols have been proposed, and our solution follows the *active replication* [6] strategy in which all replicas of the destination group receive the sent message except that we impose coordination on the sender side to limit the number of sent messages.

In each logical process, one replica is elected as master of the group for sending. Fig. 3 illustrates a send instruction from P_0 to P_1 where replica P_0^0 is assigned the master's role. When a replica reaches a send instruction, two cases arise depending on the replica's status:

- if it is the master, it sends the message to all processes in the destination logical process. Once the message is sent, it notifies the other replicas in its logical process to indicate that the message has been correctly transmitted.
- if the replica is not the master, it first looks up a journal containing the identifiers of messages sent so far (*log* on Fig. 3) to know if the message has already been sent by the master. If it has already been sent, the replica just goes on with subsequent instructions. If not, the message to be sent is stored into a *backup* table and the execution continues. (Execution only stops in a waiting state on a receive instruction.) When a replica receives a commit, it writes the message identifier in its log and if the message has been stored, removes it from its backup table.

3.2 Fault Detection and Recovery

To become effective the replication mechanism needs to be notified of processes failures. The problem of failure detection has received much attention in the literature and we have adopted the *gossip-style* protocol described by [14] for its scalability. In this model, failure detectors are distributed and reside at each host on the network. Each detector maintains a table with one entry per detector known to it. This entry includes a counter called heartbeat counter. During execution, each detector randomly picks a distant detector and sends it its table after incrementing its heartbeat counter. The receiving failure detector will merge its local table with the received table and adopts the maximum heartbeat counter for each entry. If the heartbeat counter for some entry has not increased after a certain time-out, the corresponding host is suspected to be down.

When the local instance of the MPI program is notified of a node failure by its FD service, it marks the node as faulty and no more messages will be sent to it. If the faulty node hosts a master process then a new master is elected in the logical process. Once elected, it sends all messages left in its backup table.

4 Experiments

4.1 Experimental Context

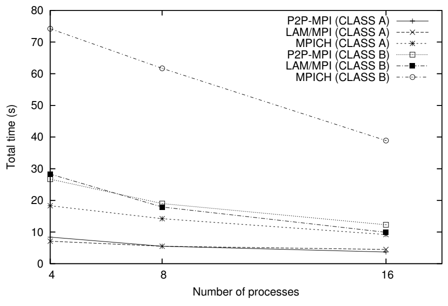
Experiment Setup. Though we claim P2P-MPI is designed for heterogeneous environments, a precise assessment of its behavior in terms of performance is difficult because we would have to define representative configurations for which we can reproduce the experiments. Before that, we measure the gap between P2P-MPI and some reference MPI implementations in an homogeneous environment so as to identify potential weaknesses. The hardware platform used is a student computers room (24 Intel P4 3GHz, 512MB RAM, 100 Mbps Ethernet, Linux kernel 2.6.10). We compare P2P-MPI using java J2SE-5.0, JXTA 2.3.3 to MPICH-1.2.6 (p4 device) and LAM/MPI-7.1.1 (both compiled with gcc/g77-3.4.3). We have chosen two test programs with opposite characteristics from the NAS benchmarks [2] (NPB3.2)[3]. The first one is IS (Integer Sorting) which involves a lot of communications since a sequence of one `MPI_Allreduce`, `MPI_Alltoall` and `MPI_Alltoallv` occurs at each iteration. The second program is EP (Embarrassingly Parallel). It does independent computations with a final collective communication. Thus, this problem is closer to the class of applications usually deployed on computational grids.

Expected Behavior. It is expected that our prototype achieves its goals at the expenses of an overhead incurred by several factors. First the robustness requires extra-communications: regular heart-beats are exchanged, and the number of message copies increase linearly with the replication degree as can be seen on Fig. 3. Secondly, compared to fine-tuned optimizations of communications of MPI implementation (e.g. in MPICH-1.2.6 [16]), P2P-MPI has simpler optimizations (e.g. binomial trees). Last, the use of a virtual machine (java) instead of processor native code leads to slower computations.

4.2 Performances

Benchmarks. Fig. 4 plots results for benchmarks IS (left) and EP (right) with replication degree 1. We have kept the same timers as in the original benchmarks. Values plotted are the average total execution time. For each benchmark, we have chosen two problem sizes (called class A and B) with a varying number of processors. Note that IS requires that the number of processors be a power of two and we could not go beyond 16 PCs.

[3] We have translated IS and EP in java for P2P-MPI from C and Fortran respectively.



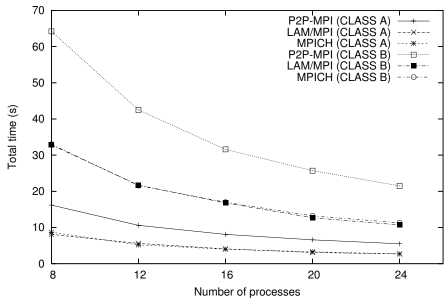


Fig. 4. Comparison of MPI implementations performance for IS (left) and EP (right)

For IS, P2P-MPI shows an almost as good performance as LAM/MPI up to 16 processors. The heart-beat messages seem to have a negligible effect on overall communication times. Surprisingly, MPICH-1.2.6 is significantly slower on this platform despite the sophisticated optimization of collective communications (e.g. uses four different algorithms depending on message size for `MPI_Alltoall`). It appears that the `MPI_Alltoallv` instruction is responsible for most of the communication time because it has not been optimized as well as the other collective operations.

The EP benchmark clearly shows that P2P-MPI is slower for computations because it uses Java. In this test, we are always twice as slow as EP programs using Fortran. EP does independent computations with a final set of three `MPI_Allreduce` communications to exchange results in short messages of constant size. When the number of processors increases, the share of computations assigned to each processor decreases, which makes the P2P-MPI performance curve tends to approach LAM and MPICH ones.

Replication Overhead. Since replication multiplies communications, the EP test shows very little difference with or without replication, and we only report measures for IS. Figure 5 shows the performances of P2P-MPI for IS when each

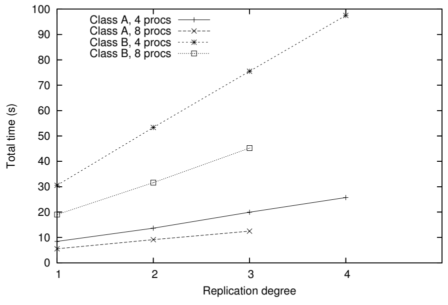


Fig. 5. Performance (seconds) for IS depending on replication degree

logical process has one to four replicas. For example, for curve "Class B, 8 processors", 3 replicas per logical process means 24 processors were involved. We have limited the number of logical processes so that we have at most one replica per processor to avoid load-imbalance or communications bottlenecks. As expected, the figure shows a linear increase of execution time with the replication degree, with a slope depending on the number of processors and messages sizes.

5 Related Work

Since the deployment of message passing applications in unstable environments is challenging, fault-tolerance of MPI has been well studied. Most works are devoted to check-point and restart methods (e.g. [3, 7, 12]) in which the application is restarted from a given recorded state. The replication approach is an alternative which does not require any specific reliable resource to store system states.

The work closest to ours is the P3 project [15], which share common characteristics with P2P-MPI. First, JXTA discovery is also used for self-configuration: *hosts* entities automatically register in a peer group of workers and accept work requests according to the resource owner policy. Secondly, both a master-worker and message passing paradigm are proposed. Unlike P2P-MPI, P3 also uses JXTA for its communications. This allows to communicate without consideration of the underlying network constraints (e.g. firewalls) but incurs performance overhead when the logical route established goes through several peers. In addition, P3 has no integrated fault-tolerance mechanism for message passing programs.

6 Conclusion and Future Work

We have described in this paper the design of a grid middleware offering a message-passing programming model. The middleware integrates fault-detection and replication mechanisms in order to increase robustness of applications execution. Two NAS parallel benchmarks with opposite behavior have been run on a small configuration and compared with performances obtained with LAM/MPI and MPICH. The results show good performance and are encouraging for experiments at large scale on the opening Grid5000 testbed[4] to study the scalability of the system. In-depth study of replication and robustness is also under work. Next developments should also concern strategies for mapping processes onto resources. Though the peer-to-peer model abstracts the network topology, we could use some network metrics (e.g. ping time) to choose among available resources. Also, the mapping of replicas could be based on information about resources capability and reliability.

[4] `http://www.grid5000.org`

References

[1] JXTA. `http://www.jxta.org`.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The Intl. Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[3] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPIch-V2: a fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing 2003*, Phoenix USA, Nov. 2003.

[4] E. Caron, F. Deprez, F. Frédéric Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A scalable approach to network enabled servers. In *8th EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910. Springer-Verlag, Aug. 2002.

[5] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. Mpj: Mpi-like message passing for java. *Concurrency: Practice and Experience*, 12(11), Sept. 2000.

[6] F. Schneider. *Replication Management Using State-Machine Approach. In S. Mullender, Distributed Systems*, chapter 7, pages 169–198. Addison Wesley, 1993.

[7] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *EuroPVM/MPI 2000*. Springer, 2000.

[8] G. Fedak, C. Germain, V. Néri, and F. Cappello. XtremWeb : A generic global computing system. In *CCGRID*, pages 582–587. IEEE Computer Society, 2001.

[9] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in an Heterogeneous Computing Environment. In *EuroPVM/MPI*. Springer, 1998.

[10] N. T. Karonis, B. T. Toonen, and I. Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing, special issue on Computational Grids*, 63(5):551–563, May 2003.

[11] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, Aug. 1999.

[12] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. MPI-FT: Portable fault tolerenace scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.

[13] MPI Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[14] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical report, Ithaca, NY, USA, 1998.

[15] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2P-based middleware enabling transfer and aggregation of computational resource. In *5th Intl. Workshop on Global and Peer-to-Peer Computing, in conjunc. with CCGrid05*. IEEE, May 2005.

[16] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operation in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, Feb. 2005.

[17] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *GRID 2002*, volume 2536 of *LNCS*, pages 1–12. Springer, Nov. 2002.