Large-Scale Experiment of Co-allocation Strategies for Peer-to-Peer SuperComputing in P2P-MPI

Stéphane Genaud[1], Choopan Rattanapoka[2]

[1] AlGorille Team - LORIA
Campus Scientifique - BP 239,
F-54506 Vandoeuvre-lès-Nancy, France
stephane.genaud@loria.fr

[2]LSIIT-ICPS, UMR 7005 CNRS-ULP
Pôle API, Boulevard Sébastien Brant,
67412 Illkirch, France
choopan@icps.u-strasbg.fr

Abstract

High Performance computing generally involves some parallel applications to be deployed on the multiples resources used for the computation. The problem of scheduling the application across distributed resources is termed as co-allocation. In a grid context, co-allocation is difficult since the grid middleware must face a dynamic environment. Middleware architecture on a Peer-to-Peer (P2P) basis have been proposed to tackle most limitations of centralized systems. Some of the issues addressed by P2P systems are fault tolerance, ease of maintenance, and scalability in resource discovery. However, the lack of global knowledge makes scheduling difficult in P2P systems.

In this paper, we present the new developments concerning locality awareness as well as co-allocation strategies available in the latest release of P2P-MPI. i) The spread *strategy tries to map processes on hosts so as to maximize the total amount of available memory while maintaining locality of processes as a secondary objective. ii) The* concentrate *strategy tries to maximize locality between processes by using as many cores as hosts offer. The co-allocation scheme has been devised to be simple for the user and meets the main high performance computing requirement which is locality. Extensive experiments have been conducted on Grid5000 with up to 600 processes on 6 sites throughout France. Results show that we achieved the targeted goals in these real conditions.*

1 Introduction

Grid computing aims at taking advantage of the many disparate computers interconnected through networks such as the Internet. The idea is to use these machines as a virtual computer architecture to offer distributed resources (processors, memory, disk storage, or even remote instruments) able to solve large-scale applications. Grid computing is therefore becoming a very attractive alternative to parallel machines for many scientific applications. However, the deployment of parallel programs in such heteregeneous, geographically and administratively scattered sets of computing resources is still a challenge. In particular, only a few experiments have demonstrated the feasibility of running message-passing parallel program on hundred of processors in this context. This paper describes the design of the P2P-MPI middleware, so as to achieve this goal.

The paper is structured as follows. Section 2 discusses how co-allocation is handled in related works. In Section 3, we briefly introduce P2P-MPI and we describe its middleware architecture. Section 4 then details how resources are allocated. Experimental results are presented in Section 5. Finally, concluding remarks and future works are presented in Section 6.

2 Related Work

We review here some representative grid software frameworks for task scheduling, and we focus on solutions implemented in real grid middleware. Among these are Legion [7] and Globus [10]. The resource management in Legion is composed of three entities. The reservation service queries and stores remote hosts availabilities. The local scheduler collaborates with the reservation service to compute possible schedules. Finally, the reservation service makes the reservation of time slots at the remote hosts through their resource managers. Legion also offers the possibility to implement specific scheduling strategies in addition to the default scheduler termed *class mechanism* which selects resources based on their characteristics. However, the framework has no information service related to the network infrastructure hence making awkward the design of

schedulers co-allocating "close" resources. Note also that applications are required to conform to the Legion object-oriented programming model. The Globus toolkit has for a long time only provided the basic blocks for resource management, mainly based on the GRAM module. In [8] is first described the resource management infrastructure and the possible strategies for co-allocation. The user had to explicitely specify the resource allocation through a RSL file. Very recently the GridWay [13] metascheduler has been included in the distribution, but it can schedule an MPI application to a single site only.

More recently, many projects have designed their architecture on a Peer-to-Peer (P2P) basis. For instance, the long-lived project ProActive [5] has added a P2P infrastructure to ease resource discovery. However, selection of resources for a computation only depends on their CPU load, as the infrastructure has no knowledge about network locality. Vishwa [15] is a framework offering a P2P overlay network in which neighbors are close in terms of network latency. The communication model it provides is based on distributed pipes, and hence applications must be modified to adapt to this model. Zorilla [9] and Vigne [14] are two other middleware systems which also build a P2P overlay network aware of peer locality. For that purpose, Vigne uses algorithms from the Bamboo project [16]. In Vigne, close resources are found using a simple (yet sometimes misleading) heuristic based on DNS name affinity: hosts sharing a common domain name are considered as forming a local group. Zorilla (which also uses Bamboo) proposes *flood scheduling*: the co-allocation request originated at a peer is broadcasted to all its neighbors, which in turn broadcast to their neighbors until the depth of the request has reached a given radius. If not enough peers accepted the job, new flooding steps are successively performed with an increasing radius until the number of peers is reached. The difficulty in this strategy, lies in finding suitable values for the flooding parameters, such as the radius and minimum delays between floods.

3 P2P-MPI overview

P2P-MPI overall objective is to provide a *grid* programming environment for parallel applications. The reader is referred to [12] for a longer description of P2P-MPI. P2P-MPI has two facets. First, it is a middleware which offers appropriate system-level services to the user, such as finding requested resources, transferring files, launching remote jobs, etc. The other facet is the communication library it provides to programmers.

3.1 Communication Library

P2P-MPI is an MPJ implementation. MPJ (Message Passing for Java) [6] is a recommandation issued from the Java Grande Forum which is an adaptation for Java of the MPI specification [1] targeting C, C++ and Fortran. Altough we have chosen Java for portability purpose, the primitives are quite close to the orginal MPI specification. This means a P2P-MPI user benefits from a communication library exposing an MPI-like API. Regarding that aspect, P2P-MPI competes with projects such as MPJ-Express [2] and MPJ/Ibis [3].

3.2 Middleware

P2P infrastructure. We have been using in previous versions of P2P-MPI (until p2pmpi-0.27.0), the JXTA library to implement all P2P operations. This work on co-allocation has lead us to replace the JXTA layer with a new P2P infrastructure, designed more specifically for our needs. The benefits over JXTA in our context are related to completeness and speed of resource discovery, and on the network latencies we capture to help build the P2P overlay. From a user point of view, there is barely no change, except that the RendezVous terminology of JXTA is replaced by the *supernode* concept. A supernode is a necessary entry point for boot-strapping a peer willing to join the overlay. A user simply makes its computer join a P2P-MPI grid by typing `mpiboot` which starts a local background process called MPD. The MPD knows at least one supernode and represents the local resource as a peer in the P2P network. When connecting to a supernode, the MPD registers to the supernode and retrieves a list of peers that it will maintain in its internal cache. The MPD's roles are mainly:

- to maintain the peer membership to the overlay by joining on startup, and by subsequently sending periodic alive signals to supernode,
- to manage the local peer's neighborhood knowledge: each neighbor in the cache is periodically ping'ed to assess network latency to it,
- when an application requests a number of resources, it has the charge of coordinating the discovery of peers, the reservation of resource and to organize the job launch,
- upon a run request from another peer, it acts as a gatekeeper of the resource by controlling how many processes and applications can be run simultaneously.

We have recently introduced an extra helper service for MPD, called reservation service (RS). It has the role of handling the first negotiation regarding requests from and to remote peers. The RS plays an important role in co-allocation and the way job execution requests are handled is detailed in the next section.

http://www.jxta.org

Job execution An application execution is typically invoked from the command line, e.g: `p2pmpirun -n` n `-r` r `-a` $alloc$ `prog`. In this example, the mandatory arguments are the n processes requested to run the `prog` program. The other arguments are optional: r is the replication degree used to request some fault tolerance (explained hereunder), and $alloc$ tells the MPD which strategy must govern the allocation of the n processes on available resources. From this example, we put forward the benefits of a middleware layer supporting application execution, concerning fault tolerance, and resource allocation.

Fault tolerance. Fault-tolerance of MPI applications is difficult to handle because during an MPI application execution, a single failure of any of the processes makes the whole application fail. This is particularly important in a grid context, where failures are far more frequent than on supercomputers, the traditional environments for high-performance applications. Solutions commonly proposed to this issue (e.g. [4]) are *checkpoint and restart* mechanisms. However, checkpoint and restart requires the presence of some reliable resources to store system states, which does not fit into our P2P framework.

P2P-MPI proposes another approach for fault-tolerance. We propose replicated processes to increase the application robustness. The replication degree in the above run command means that each MPI process will have r copies running simultaneously on *distinct hosts*. Though replication is not compulsory (and most runs will use r=1 i.e, no replication), it involves constraints on allocation rules in the general case. The simple example `p2pmpirun -n 3 -r 2 prog` requires a minimum of two hosts, say H_0, H_1, to make an allocation that guarantees the previous assertion: processes P_0, P_1 and P_2 of application `prog` could be mapped on H_0 and their replicas P'_0, P'_1 and P'_2 on H_1 (or vice-versa). In this case, a failure of H_0 or H_1 leaves a fully functional set of processes and does not end in a failure of `prog`. Note that the communication library transparently handles all extra-communications needed to keep the system in a coherent state. The advantage of this approach is that the source code of the application does not need any modification. Details can be found in [11].

Allocation. In a grid context, it is not realistic to maintain a static list of resources (such as the `machinefile` of most MPI implementations) and hence we rely on the discovery capabilities of the middleware. Subsequently to the above run request, P2P-MPI dynamically tries (during a limited time) to reserve a suitable set of resources able to host all processes involved. The problem of choosing among the discovered resources which are the most adequate for a specific execution is a difficult problem as several objectives may be followed. Let us list some considerations:

- First, we need co-allocation and hence resource should be available the same time. We have introduced the Reservation Service (RS) for that purpose.
- Second, the grid is a multi-user platform and the allocation must accommodate to the local policies of resources, not known in advance, like e.g, the number of processes that the owner of the resource accept to run simultaneously,
- Third, an MPI application generally benefits from locality of allocated resources since it minimizes the communication costs. For instance, there are today many multicore CPUs and we should favor the allocation of processes on all cores of a CPU if we strictly follow the locality principle. However, it might be more important for the application to access more memory altogether, which is in contradiction with the allocation strategy that chooses all cores on each resource as they share the same memory. We think the user, most of the time, knows these requirements and should advice the middleware of the application's specific needs.

We describe now how the various services of P2P-MPI cooperate to find and reserve a set of resources for one job execution, and which allocation strategies are currently available for co-allocation.

4 Co-allocation Process

4.1 Entities and Notations

Each service maintains a complete or partial knowledge of the P2P network. The supernode maintains the registration of peers through a list called *host list*. Each list element simply is the host IP and its services ports plus a "last seen" time stamp. Each MPD maintains a local cache of the supernode host list, called *cached list*. It periodically contacts its supernode to update its cached list. To each host in the cache list is associated a network latency value. For that, each MPD periodically contacts each host in its cached list and measures the round-trip time (RTT) of an empty message sent to it. Notice that this "ping" test is a standard P2P-MPI communication and does not rely on an ICMP echo measurement, such as ping system command. This approach would involve portability issues and further, ICMP traffic is often blocked or limited by firewalls. Each MPD, as a gatekeeper of the local resource, also manages the resource owner preferences. The owner preferences, expressed in the configuration file, may for instance allow or disallow such or such other peer. The preferences also concern the way the CPU is shared, through two settings:

- the number J of different applications that a node can accept to run simultaneously.
- the number P of processes per MPI application that a node can accept to run.

For instance, J=2 and P=1 would allow two distinct users to run simultaneously one process each for their applications. J=1 and P=2 would allow to simultaneously run two processes of a single application (this setting is often used for dual-core CPUs).

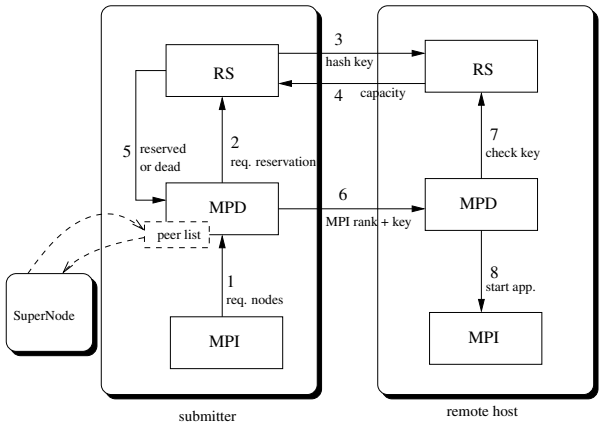

Figure 1. The job submission procedure.

4.2 Reservation Schema

We now describe step by step the reservation procedure, as depicted in Figure 1.

1. **Submission:** Recall a user submits a job with `p2pmpirun -n` n `-r` r `-a` *alloc* `prog`. This starts the MPI application, which in the initialization phase (`MPI_Init`) assigns the local MPD the task of finding the set of hosts able to executes $n \times r$ processes.

2. **Booking:** First, the local MPD verifies if it knows enough (i.e, at least $n \times r$) nodes in its cached list. If not, it triggers a cached list update request to supernode to try to acquire recently registered peers. The list is then sorted by ascending latency values. The MPD asks the local RS to book a number of hosts, starting from the beginning of its cached list (hence starting with hosts having the lowest network latencies). Actually, when possible, the request is an *overbooking* to anticipate unavailable hosts.

3. **RS-RS Brokering:** Local RS sends reservation request messages to others RS with a unique hash key.

4. The RS that receives the reservation request messages verifies whether it can accept this request by checking if the current number of applications being run does not exceed J. It also checks at this stage if the requester belongs to the denied IP list. If the request is acceptable, it replies back to the requester by sending an OK message with the value P. If not, it replies back to the requester with a NOK message.

5. **RS-MPD Response:** The local RS gathers answers from remote RS to form the list $rlist$ of reserved hosts. This list is then passed back to MPD. Nodes that have not responded before a given timeout are also marked as dead at this step. The MPD receives the $rlist$, and updates its cached list regarding peers marked dead.

6. **Allocation**: Then, the MPD allocates the processes to all or a subset of the hosts in $rlist$. Because of overbooking, the number of reserved hosts is often larger than necessary: we call $slist$ the selected subset chosen to map the application processes. It is the same as $rlist$ except that it is limited to $n \times r$ hosts (what we need at most). Formally $slist = rlist[1, \ldots, \min(|rlist|, n \times r)]$. The implication is that all reservations for hosts in $rlist$ but not in $slist$ are cancelled since they will not be used. Once $slist$ has been extracted, and before the MPI ranks distribution can take place, the MPD must decide whether the allocation is feasible. It is feasible if the two following conditions are met:

   (a) $|slist| \geq r$

   (b) $\sum_{i=0}^{|slist|} c_i \geq n \times r$, where $c_i = \min(P_i, n)$.

   First constraint says we have at least hosts r selected hosts to insure that no two replicas would have to reside on a same host. Second constraint is about the number of processes that can be hosted on the whole: we call c_i the capacity of host i, which is basically P except for marginal cases (we must not allocate more than n processes to a single host even if $P > n$ since two copies would be on that host). We therefore check that the sum of individual host capacities is large enough to execute all processes. Finally, MPD sends a request to start the MPI application. The request includes the rank and a unique hash key. The other MPDs are chosen accordingly to one of the allocation strategy selected (see section 4.3).

7. The remote MPD verifies that the unique key matches the one its RS holds for current reservation.

8. The remote MPD launches the MPI application.

4.3 Allocation Strategies

In our context, an allocation strategy must meet two criteria. (a) First, it must assign the $n \times r$ processes to the $|slist|$ reserved hosts in a sensible and understandable way regarding the user's concerns. An example of "bad" distribution would be for example, one that allocates as many processes as possible on the last host of $slist$ that is the host with higher network latency. (b) Second, in case some processes are replicated, the rank assigned to mapped processes must guarantee that no two copies of a process are on the same processor. For the first criterion, we propose two simple strategies called *spread* and *concentrate*. Below are the algorithms for each strategy, in which we use the extra notations: d is the number of distributed processes so far, and u_i the number of processes mapped onto host i.

Spread tends to map processes on hosts so as to maximize the total amount of available memory while maintaining locality as a secondary objective. The strategy is to assign the MPI processes to all selected hosts (the $|slist|$ closest hosts regarding latency) in a round-robin fashion.

```
1: d := 0
2: ∀_i, u_i := 0
3: cont := true
4: while cont do
5:    i := 0
6:    while (i < |slist|) and cont do
7:       if (u_i < c_i) then
8:          u_i := u_i + 1
9:          d := d + 1
10:      end if
11:      if (d = n × r) then
12:         cont := false{all processes are allocated}
13:      end if
14:      i := i + 1
15:   end while
16: end while
```

Concentrate tends to maximize locality between processes by using as many cores as hosts offer. The strategy is to assign the maximum MPI processes to the capacity of each host (c_i).

```
1: d := 0
2: ∀_i, u_i := 0
3: cont := true
4: while cont do
5:    i := 0
6:    while (i < |slist|) and cont do
7:       u_i := min(c_i, (n × r) − d)
8:       d := d + u_i
9:       if (d = n × r) then
10:         cont := false{all processes are allocated}
11:      end if
12:      i := i + 1
13:   end while
14: end while
```

Once either strategy has reserved enough processes place-holders, we must meet the criterion (b) when numbering the processes, i.e, assigning MPI ranks to processes. The assignment algorithm host is straight-forward: we assign the MPI rank from rank 0 to $n-1$ according to the u_i and continue along with the host i in $slist$. If some $u_i = 0$, it means no process has been mapped to host i and we simply cancel the reservation. The algorithm is as follows:

```
1: rank := 0
2: for host i in slist do
3:    if u_i = 0 then
4:       cancel reservation on host i.
5:    end if
6:    l := 0 {temporary variable}
7:    while l < u_i do
8:       assign rank rank to host i.
9:       rank := rank + 1
10:      l := l + 1
11:      if rank ≥ n then
12:         rank := 0
13:      end if
14:   end while
15: end for
```

5 Experiments on Grid'5000

The experimental grid testbed we use, Grid'5000, is a federation of clusters at distant geographical sites at the scale of France. There are nine sites, and each site hosts a couple of heterogeneous clusters. The resources in our experiment are taken from five different sites in addition to the local site nancy, where job requests are originated. Available resources are summarized in Table 1. The distant sites are, sorted by RTT to the local site (ICMP echo between frontal hosts at each site): lyon (10.5ms), rennes (11.6ms), bordeaux (12.6ms), grenoble (13.2ms), sophia (17.1ms). We can see that latencies between nancy and distant sites are very close for most of them. The bandwidth between sites is 10Gbps everywhere except the link to bordeaux which is at 1Gbps.

The main objective is to assess the allocation mechanism effects at the scale of applications composed of hundreds of processes. A secondary objective is to observe the impact of the two strategies on parallel program executions. (This last point would obviously deserve a larger study, but these preliminary tests sketch important tendencies).

Site	Cluster name	CPU	#Nodes	#CPUs	#Cores
nancy	grelon	Intel Xeon 5110	60	120	240
lyon	capricorn	AMD Opteron 246	50	100	100
rennes	paravent	AMD Opteron 246	90	180	180
bordeaux	bordereau	AMD Opteron 2218	60	120	240
grenoble	idpot	Intel Xeon IA32	8	16	16
grenoble	idcalc	Intel Itanium 2	12	24	48
sophia	azur	AMD Opteron 246	32	64	64
sophia	sol	AMD Opteron 2218	38	76	152

Table 1. Characteristics of available computing resources at the different sites

To tackle the first objective, we run a program whose each process simply echoes the name of the host it runs on. Through this experiment, we observe where processes are mapped depending on the chosen strategy and processes requested by counting hosts and cores allocated at each site. For all peers, their P parameter in the configuration is set to the number of cores in the host's CPU. For *concentrate* we consider the closer the processes are from nancy, the better are the results. For *spread*, a good allocation should map only one process per host as much as possible, and hosts selected should be the closest from nancy. The effectiveness of the strategies essentially depends on the accuracy of the latency measurement, which may differ from the RTT given by an ICMP echo command (ping) as explained in Section 4.1. The latency we measure with P2P-MPI must not necessarily be very close to the ICMP RTT, but should preserve the ranking between hosts relatively to RTT.

For the second objective, we have chosen to test two programs from the NAS benchmarks (NPB3.2) with opposite characteristics, namely IS (Integer Sorting) and EP (Embarrassingly Parallel). IS involves a lot of communications since a sequence of one `MPI_Allreduce`, `MPI_Alltoall` and `MPI_Alltoallv` occurs at each iteration. EP (Embarrassingly Parallel) does independent computations with a final collective communication.

5.1 Results for Co-allocation

Figures 2 and 3 plot the repartition of processes throughout the sites for the two strategies. The legends in top-left corners give the RTT to site nancy and the overall number of hosts and cores available at each site. The experiment consists in running the *hostname* program, requesting from 100 to 600 processes by steps of 50.

For *concentrate*, the processes are allocated on the 60 hosts available at nancy only, up to 200 processes. Next, when the capacity of 240 cores at nancy is exceeded by the request, further hosts are first allocated at lyon (5 for -n 250), as expected with respect to the RTT ranking. Subsequent requests (from -n 300) reveal that hosts from lyon, rennes and bordeaux fiercely compete for the latency ranking. We observe that the latency ranking for these hosts is interleaved with respect to sites. This is easily explained by the fact that the latencies to nancy for the three sites are within 0.6ms (RTT 1.1ms), while the latency measurements made by peers is subject to CPU and TCP load variations. This mapping thus seems adapted to applications involving many communications because of the nearness of processes.

With *spread*, hosts are chosen from the four closest sites up to 250 processes, but contrarily to *concentrate* more hosts are allocated in each site. From 300 processes, the strategy leads to take hosts from all sites to keep the load on each peer to only one process. We can clearly see on the right figure, the round-robin allocation of processes once the host list is exhausted: the number of cores allocated at nancy makes a stair at 400 processes since there are not enough hosts (350) to map one process per hosts and the closest peers are first chosen to host a second process as they have extra available cores. On the whole, we observe that all peers have been discovered and the strategy tends to use them all. So, this is a good strategy to use for application demanding much memory, as only one application process will be mapped per host provided there are enough hosts.

5.2 Results for Applications

As a concrete example of allocation strategy impact, we run the benchmark EP from 32 to 512 processes. As mentioned above EP only makes four final collective communication (`MPI_Allreduce` of one double) so that the computing to communication ratio is very high. The graph on the left of Figure 5.2 shows that EP using 32 to 256 processes is slightly faster when allocation strategy *spread* than with *concentrate*. This is probably due to the intensive memory accesses that may represent a bottleneck with *concentrate*, not compensated by locality in the collective communication. With 512 processes, the problem size per process becomes smaller and the overheads related to memory and communications seem to reach an equilibrium at this point.

IS and EP have been translated in Java for P2P-MPI from C and Fortran respectively.

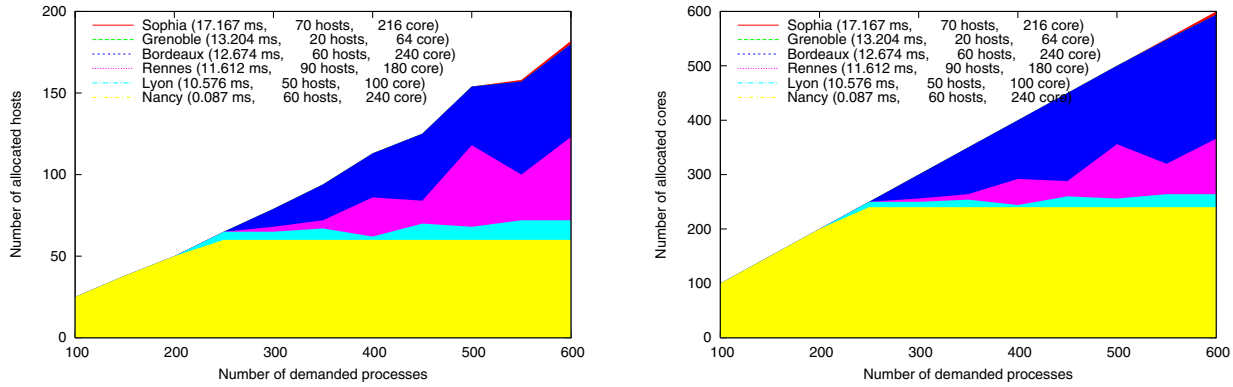

Figure 2. *concentrate* **results: locations of allocated hosts (left) and allocated cores (right)**

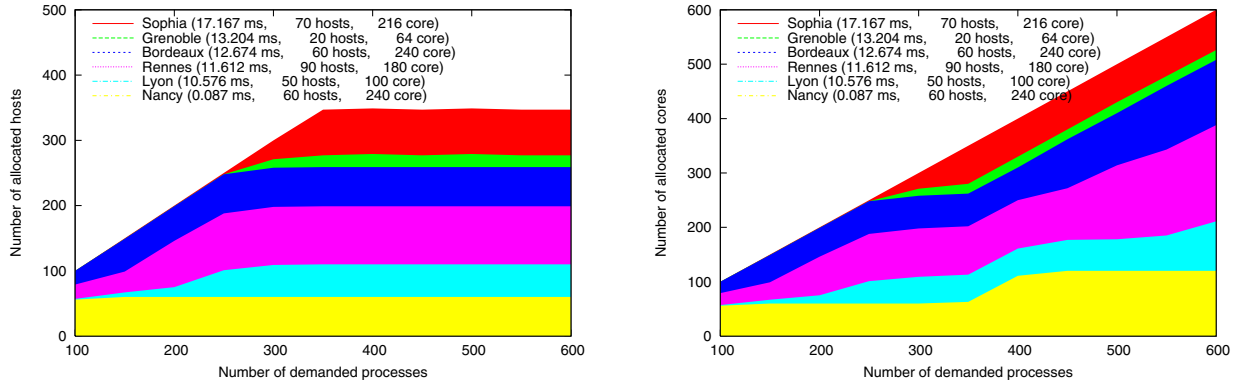

Figure 3. *spread* **results: locations of allocated hosts (left) and allocated cores (right)**

The performance curves for IS are due to the low computations to communications ratio. With 32 processes, *spread* leads to better performances than *concentrate*: with *spread* all processes are in the same cluster so that communications pay a low latency while there is no overhead due to concurrent memory accesses. This appears to be the case with *concentrate*. Using 64 processes with *spread* means that four processes are allocated outside the local cluster and the communication overhead leads to a slowdown. Keeping the processes inside the cluster with *concentrate* gives a roughly constant execution time. Figures for 128 processes and above show the same phenomena.

6 Conclusion

In this article, we describe the deep changes brought to the P2P-MPI infrastructure, and we discuss how well these developments tackle the problems targeted by P2P-MPI. Recall that our goal is to address the deployment of large-scale parallel message-passing programs. In the present case, we have to deal with applications involving hundreds of processes scattered over a nationwide set of computers. The new infrastructure which now accounts for network locality of peers, has allowed us to devise two allocation strategies. We propose the simple and understandable paradigms *spread*, which maps only one process on the closest peers, and *concentrate*, which use computing resources of closest peers as much as possible. Users can easily decide, depending on the execution environment and on their application which strategy is best suited. On one hand, *spread* involves more network communications but let each computer memory accessed by only one process. On the other hand, *concentrate* increases locality of processes but may lead to memory contention or exhaustion. This paper contributes to show, through real experiments, that such strategies can be implemented effectively to tackle the goal of allocating up to 600 processes. Further, the allocation strategy effects on program executions have also been veri-

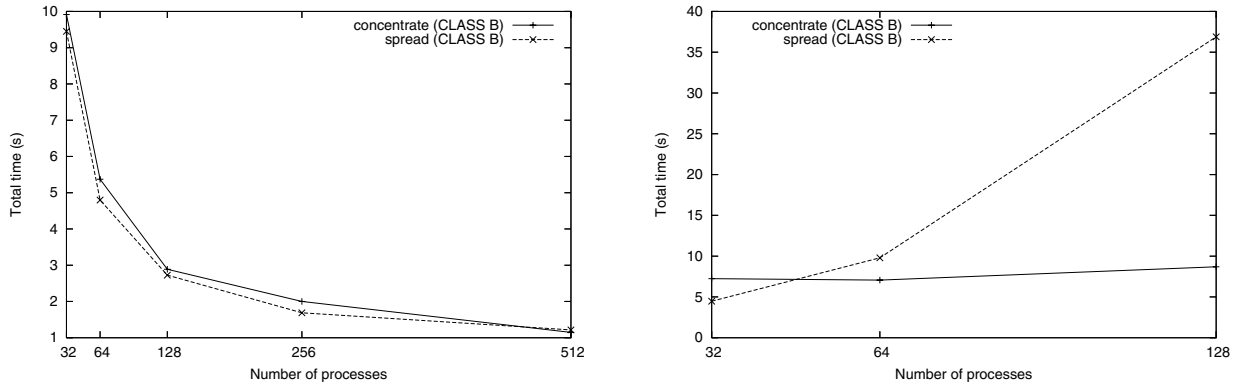

Figure 4. Execution times for EP (left) and IS (right) depending on allocation strategies

fied on two NAS benchmarks. As a future work, we should work at improving the accuracy of our latency measurement so that it becomes closer to ICMP values and less sensitive to external load. Also, we should work at the design of mixed strategies, or more complex strategies which still do not require the user to be knowledgeable about the platform characteristics. Last, a broad study may be carried out to better understand the impacts of such allocation strategies on a wider range of applications.

References

[1] MPI: A message passing interface standard, version 1.1. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[2] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *CLUSTER*. IEEE, 2006.

[3] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: a flexible and efficient message passing platform for java. In *Euro PVM/MPI 2005*, volume 3666 of *LNCS*, Sept. 2005.

[4] 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.

[5] D. Caromel, A. di Costanzo, and C. Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, 33(4-5):275–288, May 2007.

[6] 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.

[7] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion resource management system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

[8] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 37, Washington, DC, USA, 1999. IEEE Computer Society.

[9] N. Drost, R. V. van Nieuwpoort, and H. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRID'06)*, page 14. IEEE, 2006.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[11] S. Genaud and C. Rattanapoka. Fault management in P2P-MPI. In *In proceedings of International Conference on Grid and Pervasive Computing, GPC'07*, LNCS. Springer, May 2007.

[12] S. Genaud and C. Rattanapoka. P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs. *Journal of Grid Computing*, 5:27–42, 2007.

[13] E. Huedo, R. S. Montero, and I. M. Llorente. A modular meta-scheduling architecture for interfacing with pre-ws and ws grid resource management services. *Future Generation Comp. Syst.*, 23(2):252–261, 2007.

[14] E. Jeanvoine, C. Morin, and D. Leprince. Vigne: Executing easily and efficiently a wide range of distributed applications in grids. In *Proceedings of Euro-Par 2007*, pages 394–403, Rennes, France, 2007.

[15] M. V. Reddy, A. V. Srinivas, T. Gopinath, and D. Janakiram. Vishwa: A reconfigurable p2p middleware for grid computations. In *ICPP*, pages 381–390. IEEE Computer Society, 2006.

[16] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.