# Automatic Resource-Centric Process Migration for MPI

Amnon Barak, Alexander Margolin and Amnon Shiloh

Computer Science, The Hebrew University

**Abstract.** Process migration refers to the ability to move a running process from one node and make it continue on another. The MPI standard prescribes support for process migration, but so far it was implemented mostly via checkpoint-restart. This paper presents an automatic and transparent process migration framework that can be used for MPI processes. This framework is advantageous when migration of individual processes for purposes such as load-balancing is more adequate than checkpointing the whole job. The paper describes this framework for process migration in clusters and multi-clusters, how it was tuned for Open MPI and the performance of migrated MPI processes.

Keywords: Cluster, MPI, process migration, load-balancing, checkpoint.

## 1 Introduction

The term "process migration" refers to the ability to stop a running process on one node and make it continue from the same point on another. The main advantage of process migration is run-time flexibility. This includes redistribution of processes for improved performance and resource utilization, e.g., for load-balancing; and flexible cluster configuration, including orderly shutdown, addition of nodes and inclusion in a multi-cluster. In spite of the advantages, the main drawbacks of process migration are the complexity of maintaining migrated processes seamlessly and the need for an adequate policy to decide when, where and which process(es) to migrate.

The MPI standard [1] prescribes support for process migration, but so far direct support was implemented mostly via Checkpoint-Restart (CR) of a whole job, using a package such as the Berkeley Lab BLCR [2]. In the CR approach, all the processes of a job are stopped and their images are saved to persistent storage. The CR approach is reasonable when the demand for resources is relatively stable, but inadequate when resources and/or demand for resources change frequently, such as when running processes with uneven loads; when processes change their demand for resources (cores, GPUs, memory); when temporarily oversubscribing; when nodes are reclaimed by users with higher priority; and when CPUs slow down due to overheating. In such cases, direct migration of individual processes is lighter and more adequate because it does not stop the whole job. This is the main contribution of the current paper.

As an example where process migration can benefit queued MPI jobs, consider a situation where a job is scheduled to start on a given time, but few of its designated nodes are not available at that time, either because they are down or because they are used by overdue jobs. When process migration is available and memory is sufficient, the processes of the scheduled job can be temporarily assigned to those nodes that are already available, despite oversubscribing, and later be migrated to additional nodes as they become available. This avoids situations where jobs are queued while nodes are available but remain idle.

This paper presents a transparent (proactive) process migration framework for MPI based on features of MOSIX [3], a management system targeted for HPC on Linux clusters and multi-clusters. Its relevant features include a decentralized gossip algorithm that provides each node with information about cluster-wide resources [4]; a set of online algorithms that use this information to assign and actively reassign (migrate) processes to nodes, to optimize the performance [5]; and the actual process migration software.

The paper is organized as follows: Sec. 2 describes our process migration framework, including the run-time environment and the algorithms for initiating and managing process migrations in clusters and multi-clusters. Sec. 3 describes how our framework runs migratable Open MPI jobs, including allocation of resources and direct communication between migrated MPI processes. Sec. 4 presents various performance aspects of our process migration using standard benchmarks. Related works are described in Sec. 5 and our conclusion in Sec. 6.

## 2 A Framework for Process Migration

Process migration refers to the ability to stop a running process on one node, preserve all its important elements and then make it continue from the same point on another node. The elements to be preserved depend on the interface between the process and its immediate run-time environment. In a previous project, we developed a process migration framework for general Linux processes [3]. Since MPI processes run on Linux, we used that framework for migration of MPI processes.

This section presents relevant features of our process migration framework, including the run-time environment, the algorithms for initiating and managing migrations and support of multi-clusters.

### 2.1 Our Run-Time Environment

In order to support generic Linux processes, it is important that a process sees the same environment, including files, sockets, process IDs, etc., regardless where it runs or is migrated to. To achieve that, we developed a virtual environment (sandbox) in which each migrated process seem to run as if it is still in its original "home-node", where it was created. This is accomplished by intercepting all the system-calls of the process, then forwarding most of them to the home-node, performing them there and returning the results to the migrated process.

This approach, which isolates the process from the node in which it is currently running, provides maximal file and data consistency, as well as support of nearly all traditional IPC mechanisms such as messages, semaphores pipes, sockets and signals (with process-IDs kept intact), excluding only shared-memory.

The drawback of this approach is that the maintenance of migrated processes requires increased management and network overheads.

To reduce the network overhead incurred by the use of home-nodes, we developed a peer-to-peer "postal" protocol for direct communication between migrated processes, bypassing their respective home-nodes. This OSI layer 4 protocol guarantees that data always arrives in order and is never lost even when the senders and receivers migrate several times and even while they are in mid-migration, all transparent to the program. This is especially efficient for processes that migrated to the same node.

### 2.2 Initiating and Managing Process Migration

In our framework, process migration can be triggered either automatically or manually, including by the process itself. Automatic migrations are supervised by competitive on-line algorithms that attempt to improve the performance using a gossip-based information collection and process profiling [4]. Process profiling is performed by continuously collecting information about each process' characteristics, such as size, rates of system-calls and volume of I/O. This result in determining the best location for each process, taking into account the respective speed, current load, available memory in the nodes and the migration cost. As resources and the profile of the process change, and subject to threshold values to avoid over migrations, processes may be reassigned and migrated to better locations - which is particularly useful for jobs with unpredictable or changing resource requirements and when several users run simultaneously. The objective are:

- Load balancing.
- Assigning processes to faster nodes.
- Assigning processes to nodes with sufficient memory.
- Sharing the cluster resources among several users.

While a process can be migrated by the framework at any time, processes can also explicitly request to be migrated to any desired node. This way, if a process is expecting to do a significant amount of communication with another process, it can request to be migrated to the location of that other process, or it can "invite" another process (of the same user) to migrate to its current node. Either way, the goal is to reduce the communication latency and the network overhead. Obviously, this option is limited by the fact that only a limited number of processes can run efficiently on each node.

### 2.3 Multi-Clusters

A multi-cluster is a collection of private clusters that are configured to work together. Each cluster may belong to a different group that is willing to share its computing resources so long as it is allowed to disconnect its cluster at any time, especially when they are needed by its local users.

The automatic migration algorithms of the previous section also manage multi-clusters, with only minor adjustments. Note that all migrated processes still run in the environment of their own home-cluster. Thus, from the user's perspective, it does not matter whether their applications run in their own cluster or out on a different cluster.

To allow a flexible use of nodes within and among different groups, we developed a priority scheme, whereby local processes and processes with a higher priority can always move in and push out processes with a lower priority. By proper setting of the priority, private clusters can be shared among users. Public clusters can also be set to be shared among all users.

## 3 Running Migratable Open MPI Jobs

This section describes an adaptation of our framework for Open MPI jobs.

### 3.1 Initial Assignment

The standard practice in Open MPI is to rely for resource discovery on an XML input file provided by each user. This file usually includes the list of nodes and their resources, e.g., number of cores and total memory. Using this information, Open MPI usually assigns processes to nodes in a round-robin fashion, regardless of the current status and availability of these resources.

Our framework provides a dynamic resource discovery system. The simplest approach is to start all the MPI processes in the same node and let our framework migrate them automatically to different available nodes. A variation of this approach is to start an equal number of processes on a fixed (small) subset of nodes and then allow those processes to migrate. Another approach is to find a set of best-available nodes, then launch the MPI processes on that list of nodes. We developed a new Resource Allocation Subsystem (RAS) module which does that. We also added a module under the Open MPI ORTE Daemon's Local Launch Subsystem (ODLS) component, which intercepts the process launch and modifies the arguments so that it uses our framework to launch the Open MPI processes. The launch command line may include three new flags:

- Disallow automatic migration.
- Allow migration to other clusters - in a multi-cluster configuration.
- Start the process on the best available node, not necessarily its home-node.

### 3.2 Direct Communication Between Open MPI Processes

The main difference between migrating independent processes and MPI processes is the extensive use of the point-to-point Inter-Process Communication (IPC). MPI usually carries its IPC via TCP/IP sockets, which are critical to the performance of the entire job. This section presents an emulation of TCP/IP sockets using Direct COMmunication (DiCOM) between migrated MPI processes.

Since our postal protocol provides a different API based on unidirectional, per-process mailboxes and not on TCP/IP, translation is therefore required.

The following features of DiCOM are designed to assist this translation:

– A process's mailbox can accumulate data packets from multiple sources, which are then read by the process, usually sequentially. However, an optional feature allows reading mailbox data out-of-order according to specified conditions, such as the PID of the sender process.
– Reading a mailbox can be either blocking or non-blocking.
– Asynchronous notification of message-arrival.

The Byte-Transfer Layer (BTL) component of Open MPI consists of independent modules that provide lower level communication, i.e., sending and receiving raw data for the use of higher layers. Examples of such modules are low-latency interconnections such as OpenIB (Infiniband); communication over shared memory; and the fall-back module - TCP/IP. Note that BTL modules are unaware which MPI function they serve, since this is included in the data itself.

We added a new BTL module for emulating TCP communication using DiCOM. This OSI layer 5 module is responsible to establish a link to the current location of the target MPI process and to transfer data to this process. The module implements a series of functions which are registered and called by the MPI run-time components. The module functions are called in different phases of running each MPI process as follows:

– **Registration phase:** The module receives low-level communication parameters.
– **Initialization phase:** The module detects whether it runs under our framework, and if so, receives the necessary details to contact other processes within that framework (otherwise this module is not used). This is also where DiCOM's asynchronous notification of message arrival is turned on.
– **Progress phase:** The MPI run-time system informs the module that new message(s) have arrived. The module then uses DiCOM's out-of-order and non-blocking message reading functionalities to direct incoming messages to those layers of MPI that are waiting to receive the corresponding messages.
– **Message send phase:** Outgoing messages are converted and sent using the DiCOM API.
– **Message receive phase:** The MPI higher layers register their interest in receiving incoming messages.
– **Finalization phase:** Connections are closed and resources are released.

### 3.3 Oversubscribing

Oversubscribing is an allocation policy that assigns more than one process per core [8]. Traditionally, oversubscribing was discouraged for MPI processes, but process migration provides a greater incentive to use it. Temporary oversubscribing with process migration can be beneficial in certain situations, including the following:

– Allowing a job to start on schedule, even if some designated nodes are not available, making use of other idle nodes.
– When it is necessary to shut down few nodes, e.g., due to overheating or for maintenance, without stopping the whole job.

- When CPU speeds are not uniform, new jobs can begin on slower nodes, then migrate to faster ones as they become available. Also, jobs that are past their allocated time can be migrated to slower nodes rather then killed.
- When running jobs with processes of uneven CPU demands, including when CPU usage is unknown in advance.

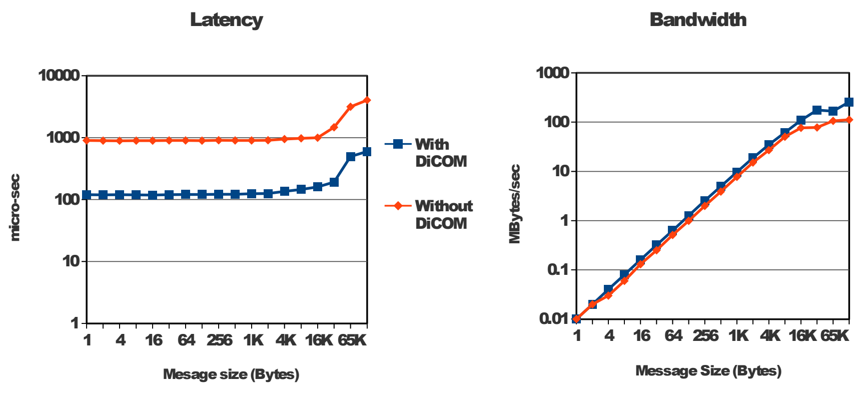
## 4 Performance of Process Migration

This section presents various performance aspects of our process migration using standard benchmarks. The tests were done on a symmetric cluster of Intel's Quad-core i7 (2.67 GHz) and 6GB memory nodes, connected by QDR Infiniband. Each test was performed 5 times and the average of the results is shown.

### 4.1 IPC Between Migrated MPI Processes

To evaluate the performance of our BTL module, described in Sec.3.2, we used the OSU benchmark [6]. Two MPI processes were migrated to different nodes and communicated with and without the module. The left and right sides of Fig. 1 show the respective (log scale) latency and bandwidth for message sizes up to 131KB. The measurements show that the latency with DiCOM was up to 7.5 times less than without DiCOM while the bandwidth was up to 2.3 times higher. This is due to DiCOM's success in avoiding the overhead of communicating via home-nodes.

**Fig. 1.** Latency and bandwidth between migrated MPI processes



### 4.2 Overhead of Migrated Processes

We used the NPB benchmark suite [7] to compare the run-time of class C applications, initially starting all the processes in different idle nodes, then immediately migrating away some processes to other idle nodes, from none to all 4.

The results are shown in Table 1. For each application, the "None" column shows the run-time (in Sec.) of the applications without migrating, while the remaining columns show the corresponding run-times with 1–4 migrations. Accordingly, both applications with large memory and applications with higher IPC volumes incur higher overhead, consisting of migration plus communication.

**Table 1.** Run-times of NPB applications with different number of migrated processes

| **Application name** | **Number of migrated processes** | | | | |
|---|---|---|---|---|---|
| | **None** | **1** | **2** | **3** | **4** |
| MG | 39.3 | 39.9 | 39.9 | 40.1 | 40.1 |
| LU | 313.3 | 317.1 | 317.5 | 315.2 | 315.1 |
| BT | 336.5 | 342.2 | 342.7 | 342.2 | 343.3 |
| CG | 72.8 | 76.1 | 75.8 | 75.9 | 77.3 |
| SP | 352.9 | 365.1 | 364.8 | 365.7 | 365.9 |
| EP | 93.4 | 93.3 | 93.1 | 92.4 | 92.8 |

### 4.3 Migration Speeds

We repeated the previous test, this time migrating only one process 10 times back and forth, resulting in Table 2. Column 2 shows the run-time (in Sec.) without migrating, Columns 3, the run-time with 10x2 migrations, then Column 4 shows the average time per migration. Column 5 gives the size of migrated processes, then the last column shows the migration speeds. It can be seen that larger processes migrate relatively faster for their size. This is explained by the fixed cost of initiating and managing a migration.

**Table 2.** Migration time and speed of NPB applications

| **Application name** | **Run-time without migration** | **With 10 x 2 migrations** | **Average migration time** | **Process size MB** | **Migration speed MB/Sec.** |
|---|---|---|---|---|---|
| MG | 39.2 | 59.4 | 1.01 | 618.2 | 612.1 |
| LU | 313.3 | 320.4 | 0.36 | 196.6 | 546.1 |
| BT | 338.3 | 355.9 | 0.88 | 413.1 | 469.4 |
| CG | 72.8 | 83.8 | 0.59 | 260.6 | 441.7 |
| SP | 353.0 | 374.9 | 1.10 | 352.3 | 320.3 |
| EP | 92.2 | 93.4 | 0.06 | 17.0 | 283.3 |

### 4.4 Benefiting From Oversubscribing

To show how jobs with non-uniform workload distribution can benefit from oversubscribing, we ran an application that simulated processes with uneven run-time. A fixed problem was divided into a variable number of processes, some short and some long, thus leaving room for improvement by combining oversubscribing with load balancing. Processes that migrate from a node where all other processes are active to a node where some processes are idle can get more CPU cycles, both for themselves and for the processes on their former node, thus reducing the overall run-time of the job.

We initially ran 4 long (∼1800 Sec.) processes in one 4-core node and 4 short (∼360 Sec.) processes in the remaining 4-core nodes. We then applied oversubscribing by dividing each process into 2 processes of equal time (1800 Sec. to 2x900 Sec. processes and 360 Sec. to 2x180 Sec. processes). We repeated the test, further dividing the 1800 Sec. processes into 3, 4 and 6 processes of equal time. Table 3 shows the average run-time (in Sec.) of the application with and without process migration. For reference, Column 4 shows the optimal (theoretical) run-time with migration (excluding migration overheads).

**Table 3.** Run-times of a job with non-uniform workload with and without migration

| **Number of processes per core** | **Without migration** | **With migration** | **Optimal run-time** |
|---|---|---|---|
| 1 | 1798 | 1799 | 1798 |
| 2 | 1797 | 1079 | 1079 |
| 3 | 1796 | 839 | 839 |
| 4 | 1796 | 722 | 719 |
| 6 | 1797 | 608 | 599 |

From the table it can be seen that process migration improves performance and that nearly-optimal run-times were achieved.

### 4.5 Resolving Memory Pressure

This test demonstrates how process migration can improve the performance by better utilization of memory resources. In a cluster of multi-cores, although only one process at most is assigned per core, it is quite possible to arrive at "memory-oversubscribing" resulting in memory thrashing.

We enforced memory-oversubscribing in an application that simulated processes with unpredictable memory demands, initially with one process per core. As a result, the memory of some 4-core nodes was exhausted, causing them to thrash. We ran the application twice. Without process migration the average run-time was 590 Sec. and with process migration it was 369 Sec., an improvement of 37.5%.

## 5 Related Work

The migration of processes has been shown to improve the run-time of MPI jobs [9]. So far, the majority of work was based on Checkpoint-Restart (CR) of a whole job [10, 11, 12], which was accomplished by user-level or kernel-level libraries [13], such as BLCR [2]. Using RDMA over Infiniband was shown to improve the performance of the above [14, 15].

An alternative form of CR is available in Java-MPI [16] using JVMs. In this approach the process state is captured in one JVM and can then be restored on another.

Adaptive MPI (AMPI) supports transparent process migration [17], based on the CHARM++ framework that provides load-balancing through user-level migratable threads. AMPI requires the applications to be rewritten in the CHARM++ object oriented language.

A feature that allows an Open MPI process to restart communication over a different network after a checkpoint was presented in [18]. This feature can improve the communication between MPI processes that were moved to a common node.

For better load-balancing and other optimizations of MPI process placement, it was shown that processes can hint the underlying MPI implementation about their expected load, thus allowing the implementation to achieve a better placement [19].

## 6 Conclusions

Traditionally, process migration has been associated both with improved performance, by load-balancing and with flexibility, by dynamic reallocation of resources. These two properties seem even more important for the next generation's medium/large (Peta/Exa) scale systems that will include thousands of nodes with diverse resources, where node failures are expected to be quite common. In large systems, gossip algorithms about the state of the nodes are necessary to support process migration. This paper presented a proactive process migration framework for running Open MPI processes in clusters and multi-clusters. The performance penalty of our framework is reasonable.

One way to reduce the dependency of MPI jobs on our "Archimedes stand"[1] home-node, is to start all the processes of a job on a few home-nodes, then distribute processes to the remaining nodes. A more challenging project would be to develop a process migration scheme that does not use home-nodes at all.

## Acknowledgments

The authors wish to thank E. Levy for his help. This research was supported in part by grants from the Council for Higher Education and from Dr. and Mrs. Silverston, Cambridge, UK.

[1] Give me a place to stand and I will move the whole world, Archimedes 287 BC.

## References

[1] The Message Passing Interface (MPI) standard. `http://www.mcs.anl.gov/mpi/`

[2] Berkeley Lab Checkpoint/Restart. `http://ftg.lbl.gov/checkpoint`

[3] Barak, A., Shiloh, A.: The MOSIX cluster operating system for high-performance computing on Linux cluster, multi-clusters and clouds. `http://www.MOSIX.org/pub/MOSIX_wp.pdf` (2012)

[4] Amar, L., Barak, A., Drezner, Z., Okun, M.: Randomized gossip algorithms for maintaining a distributed bulletin board with guaranteed age properties. Concurrency and Computation: Practice and Experience 21 (2009) 1907-1927

[5] Amir, Y., Awerbuch, B., Barak, A., Borgstrom, R.S., Keren, A.: An opportunity cost approach for job assignment in a scalable computing cluster. IEEE Tran. Parallel and Dist. Systems 11.7 (2000) 760-768

[6] Liu, J., Chandrasekaran, B., Yu, W., Wu, J., Buntinas, D., Kini, S.P., Wyckoff, P., Panda, D.K.: Micro-benchmark level performance comparison of high-speed cluster interconnects. `http://nowlab.cse.ohio-state.edu/publications/conf-papers/2003/liuj-hoti03.pdf`. Hot Interconnect 11 (2003)

[7] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks. Tech. Report, RNR-94-007, NASA (1994)

[8] Iancu, C., Hofmeyr, S., Blagojevic, F., Zheng, Y.: Oversubscription on multicore processors. Proc. 2010 IEEE Int'l Sym. on Parallel and Dist. Processing (2010)

[9] Corbal, J., Duran, A., Labarta, J.: Dynamic load balancing of MPI+OpenMP applications. Proc. Int'l Conf. on Parallel Processing (ICPP) (2004) 195-202

[10] Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. Proc. 21st IEEE Int'l Parallel and Dist. Processing Sym. (IPDPS) (2007) 1-8

[11] Liu, T., Ma, Z., Ou, Z.: A novel process migration method for MPI applications. Proc. 15th IEEE Pacific Rim Int'l Sym. on Dependable Computing (2009) 247-251

[12] Wang, C., Mueller, F., Engelmann, C., Scott S.: Proactive process-level live migration in HPC environments. Proc. 2008 ACM/IEEE conf. on Supercomputing (SC '08) (2008)

[13] Roman, E.: A Survey of Checkpoint/Restart implementations. Tech. Report LBNL-54942C, Berkeley Lab (2002)

[14] Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for MPI programs over Infiniband. Proc. 35th Int'l Conf. on Parallel Processing (ICPP) (2006) 471-478

[15] Ouyang, X., Rajachandrasekar, R., Besseron, X., Panda, D.K.: RDMA-based job migration framework for MPI over Infiniband. Proc. 2010 IEEE Int'l Conf. on Cluster Computing (CLUSTER) (2010) 116-125

[16] Ma, R.K.K., Wang, C., Lau, F.C.M.: M-JavaMPI: A Java-MPI binding with process migration support. Proc. 2nd IEEE Int'l Sym. on Cluster Computing and the Grid (CCGRID) (2002) 255

[17] Huang, C., Zheng, G., Kale, L., Kumar, S.: Performance evaluation of Adaptive MPI. Proc. 11th ACM SIGPLAN Sym. on Principles and Practice of Parallel Programming (PPoPP) (2006) 12-21

[18] Hursey, J., Mattox, T.I., Lumsdaine, A.: Interconnect agnostic checkpoint/restart in Open MPI. Proc. 18th ACM Int'l Sym. on High Performance Dist. Computing (HPDC '09), Garching (2009) 49-58

[19] Keller, J., Majeed, M., Kessler, C.W.: Balancing CPU load for irregular MPI applications. Proc. Int'l Conf. on Parallel Computing (ParCo) (2011)