# Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration

Gonzalo Martín[a], David E. Singh[a,*], Maria-Cristina Marinescu[b], Jesús Carretero[a]

[a] *Universidad Carlos III de Madrid, Computer Science Department, Leganés, Madrid 28911, Spain*
[b] *Barcelona Supercomputing Center, Computer Applications in Science & Engineering, Barcelona 08034, Spain*

**Abstract**

The work in this paper focuses on providing malleability to MPI applications by using a novel performance-aware dynamic reconfiguration technique. This paper describes the design and implementation of Flex-MPI, an MPI library extension which can automatically monitor and predict the performance of applications, balance and redistribute the workload, and reconfigure the application at runtime by changing the number of processes. Unlike existent approaches, our reconfiguring policy is guided by user-defined performance criteria. We focus on iterative SPMD programs, a class of applications with critical mass within the scientific community. Extensive experiments show that Flex-MPI can improve the performance, parallel efficiency, and cost-efficiency of MPI programs with a minimal effort from the programmer.

*Keywords:*
Malleable MPI applications; performance-aware dynamic reconfiguration; computational prediction model; distributed systems; high performance computing.

*Corresponding author
*Email addresses:* gmcruz@arcos.inf.uc3m.es (Gonzalo Martín), desingh@arcos.inf.uc3m.es (David E. Singh), maria.marinescu@bsc.es (Maria-Cristina Marinescu), jcarrete@arcos.inf.uc3m.es (Jesús Carretero)

## 1. Introduction

The majority of scientific applications that execute on high performance computing (HPC) clusters are implemented using MPI. The focus of this paper is to provide MPI applications with malleable capabilities through performance-aware dynamic reconfiguration. Parallel applications may be *rigid*, *moldable*, *malleable*, or *evolving* based on their capability to vary the number of processors that they execute on at runtime [1]. In rigid and moldable applications the number of allocated processors is fixed for the entire duration of the execution. Evolving and malleable applications, on the other hand, allow changing the number of processors during program execution—an operation called *reconfiguration*. In the case of evolving applications the changes are initiated by the application itself. A malleable application may increase the number of processes when new processors become available in the system and decrease it when currently allocated processors are requested to the resource management system (RMS) by another application with higher execution priority. Malleable applications allow implementing more flexible and efficient scheduling policies [2] that use idle processors to improve resource utilization [3, 4]. While different RMSs support dynamic allocation of resources for malleable applications [5, 6], MPI does not natively provide support for malleability.

The challenge when designing dynamic reconfiguration techniques for malleable MPI applications is not simply to modify the number of processes that the application is running on according to the availability of resources, but to make these decisions based on performance criteria. Reconfiguring actions should only be triggered if process addition or removal may benefit the application performance. For certain classes of problems, increasing the number of processors beyond a certain point does not always result in an improvement in terms of execution time. This is due to larger communication and synchronization overheads, in addition to the overhead incurred by the reconfiguring operations. The support framework must decide how many processors to run on before triggering a reconfiguring action. This number depends on the set of

available processors in the system, the reconfiguring overhead, and the predicted application performance when running on the new processor configuration. Reconfiguring actions involve changing the data distribution of the application (which may lead to load imbalance) and modifying its communication patterns. These lead to changes in the application performance. In addition, this optimization process is considerably more complex when running on architectures with heterogeneous (performance-wise) compute nodes equipped with different types of processors.

In this paper we introduce Flex-MPI, an MPI extension which supports malleability and implements performance-aware dynamic reconfiguration for iterative MPI applications. We have implemented Flex-MPI as a library on top of the MPICH [7] implementation. Our performance-aware dynamic reconfiguration technique allows the user to define the performance objective and constraints of the application. We use the completion time of the application as the performance objective. Flex-MPI automatically reconfigures the application to run on the number of processes that is necessary to increase the performance such that application completes within a specified time interval. Flex-MPI modifies the application performance by adding or removing processes whenever it detects that the performance target is not achieved. The reconfiguration process also depends on the user-given performance constraint which can be either the parallel efficiency or the operational cost of executing the program.

Flex-MPI implements a computational prediction model to decide the number of processes and the process-to-processor mapping that can achieve the required performance objective under a performance constraint. The efficiency constraint results in minimizing the number of dynamically spawned processes to maximize parallel efficiency. The cost constraint focuses on minimizing the operational cost by mapping the newly created dynamic processes to those processors with the smallest cost (expressed in $ per CPU time unit) while satisfying the performance constraint. The operational cost of a program execution has become a key performance factor in the last years [8, 9]. This metric is particularly relevant when we consider heterogeneous systems where each type

of processor may have a different operational cost. These configurations are commonplace when executing HPC applications on the cloud, an approach of increasing interest in the HPC community [10, 11].

To summarize, the main contributions of this work are:

- An **efficient library** which supports malleable applications and provides monitoring, load balancing, data redistribution, and dynamic reconfiguring functionalities for iterative MPI applications.
- A **simple**, **high-level** Application Programming Interface (API) to access Flex-MPI functionalities from any MPI application.
- A **computational prediction model** which can estimate the performance of the parallel application prior to a dynamic reconfiguring action.
- A **novel performance-aware dynamic reconfiguration policy** which automatically reconfigures the MPI application to run on a number and type of processors that satisfy a user-given performance objective under efficiency or cost constraints.
- An **extensive performance evaluation** that demonstrates the capabilities of Flex-MPI to enhance MPI application performance via malleability.

The rest of this paper is organized as follows. Section 2 discusses related work in the area of malleability in MPI applications. Section 3 describes the design of the Flex-MPI library and the programming model of malleable applications. Section 4 discusses the implementation of each software component of Flex-MPI. In Section 5 we present an extensive performance evaluation. Section 6 summarizes the conclusions and discusses future work.

## 2. Related work

Supporting malleability in MPI applications has been a topic of great interest in the last years due to the necessity to maximize resource utilization in

HPC clusters. The vast majority of existing approaches work under the assumption that increasing the number of processors in a parallel application does not increase its execution time. As a result, reconfiguration is done by increasing the number of processes to the maximum available processors in the system. In practice however, parallel applications show a performance threshold beyond which increasing the number of processors does not lead to a significant performance improvement [12] due to increasing communication and synchronization overheads [13].

There are several approaches which provide malleability for MPI applications using offline reconfiguration [14, 15, 16]. Offline reconfiguration is provided by a mechanism which consists in stopping the execution of the application, checkpointing the state in persistent memory, then restarting the application with a different number of processes. This has several important drawbacks, one of the major ones being the overhead introduced by the I/O operations carried out every time the application is reconfigured. Dynamic reconfiguration, on the other hand, provides malleability by allowing the application to change the number of processes while the program is running.

Adaptive MPI (AMPI) [17] is an MPI extension which uses processor virtualization to provide malleability by mapping several virtual MPI processes to the same physical processor. AMPI is built on top of CHARM++, in which virtualized MPI processes are managed as threads encapsulated into CHARM++ objects. The runtime system provides automatic load balancing, virtual process migration, and checkpointing features. Adaptive MPI programs receive information about the availability of processors from an adaptive job scheduler. Based on this information, the runtime system uses object migration to adapt the application to a different number of processes.

Process Checkpointing and Migration (PCM) [18, 19] is a runtime system built in the context of the Internet Operating System (IOS) [20] and uses process migration to provide malleability to MPI applications. The PCM/IOS library allows MPI programs to reconfigure themselves to adapt to the available processors as well as the performance of the application by using either split/merge

operations or process migration. Split and merge actions change the number of running processes and their granularity, while process migration changes the locality of the processes. Processor availability is managed by an IOS agent which monitors the hardware. Although adaptive actions are carried out without user intervention, PCM requires that the programmer instruments the source code with a large number of PCM primitives.

Utrera *et al.* [21] introduces a technique called *Folding by JobType* (FJT) which provides virtual malleability. The FJT technique combines moldability, system-level process folding, and co-scheduling. Parallel jobs are scheduled as moldable jobs, in which the number of processes is decided by the resource manager just before the job is scheduled on the compute nodes. FJT introduces virtual malleability to handle load changes and take advantage of the available processors. This is done by applying a folding technique [22] based on co-scheduling a varying number of processes per processor.

Cera *et al.* [23] introduces an approach called dynamic CPUSETs mapping for supporting malleability in MPI. CPUSETs are lightweight objects which are present in the Linux kernel. They enable users to partition a multiprocessor machine by creating execution areas. CPUSETs features migration and virtualization capabilities, which allows to change the execution area of a set of processes at runtime. Cera's approach uses CPUSETs to effectively expand or fold the number of physical CPUs without modifying the number of MPI processes. While most of the approaches described above are transparent to the programmer, they provide support for malleable applications via operating system-level mechanisms which are not integrated into the MPI library.

The MPI-1 specification [24] requires that the number of application processes remains fixed during its execution. The dynamic process management interface was introduced by the MPI-2 specification and consists of a set of primitives that allow the MPI program to create and communicate with newly spawned processes at runtime. This interface is implemented by several of the existing MPI distributions (e.g. MPICH [7] and OpenMPI [25]) and has been used by several approaches to provide dynamic reconfiguration to malleable MPI

applications.

Cera *et al.* [26, 27] and Leopold *et al.* [28] introduce different approaches that use the dynamic process management interface to enable malleability to iterative MPI applications. However, these approaches do not evaluate the application's performance prior to the reconfiguring action. This may lead to performance degradation and inefficient resource utilization for fine-grained parallel applications that exhibit a small computation to communication ratio.

ReSHAPE [29, 30] is the approach that is more closely related to Flex-MPI. ReSHAPE features a runtime framework for malleable, iterative MPI applications that uses performance data collected at runtime to support reconfiguring actions. The ReSHAPE framework increases the number of processes of the application when the there are available processors in the system and the iteration time has improved due to a previous increase or the number of processes has never been expanded before. ReSHAPE then decreases the number of processes when the current set of processors does not provide a performance benefit as a result of a previous increase. Flex-MPI features a much more sophisticated computational prediction model which uses hardware counters and network performance data to estimate the future application performance prior to a reconfiguring action. ReSHAPE assumes that all iterations of a parallel application are identical in terms of computation and communication times, and that they have regular communication patterns. Our approach targets parallel applications with both regular and irregular communication patterns. Additionally, it can handle varying computation times that are due to varying workloads or execution on non-dedicated systems. Flex-MPI uses the computational prediction model to evaluate multiple potential reconfiguration scenarios and choose the one which is predicted to best satisfy the performance objective. ReSHAPE, on the other hand, improves application performance by considerably more costly trial-and-error, i.e. by triggering reconfiguration actions followed by an evaluation of their effect on performance.

## 3. Flex-MPI library overview

Flex-MPI targets iterative *Single Program Multiple Data* (SPMD) applications with both regular and irregular computation and communication patterns. A large proportion of the SPMD parallel applications are iterative, such as linear solvers, particle simulation and fluid dynamics simulators. In the SPMD paradigm each process executes the same code but operates on a different subset of the data. The usual structure of an iterative SPMD application consists of an initializing section in which each process loads its data partition; what follows is an iterative section during which the processes operate in parallel and communicate with each other to reach a global solution. Our approach focuses on applications which use one-dimensional and two-dimensional distributed data structures in which each process stores only its own data partition and does not replicate data managed by other processes.

Flex-MPI is implemented as a library on top of the current stable release of MPICH (v.3.0.4). This makes it fully compatible with all the new features of the MPI-3 standard [31]. Figure 1 shows the execution environment of a Flex-MPI application which consists of: the Flex-MPI library, the MPI user application, the Performance API (PAPI) [32] and MPI library, the user-given performance objective and performance constraints, and the resource management system.

Figure 2 shows the workflow diagram of a malleable MPI application using Flex-MPI. Each box shows in square brackets the Flex-MPI components that provide the corresponding functionality. Initially the MPI application runs on $n$ processes. At every iteration, the MPI program instrumented to use the Flex-MPI API automatically feeds the per-process values of the chosen runtime performance metrics to the monitoring (M) module (label `1.a`). These include hardware performance counters, communication profiling data, and the execution time for each process. Once Flex-MPI has collected these metrics it returns the control to the MPI application (label `1.b`). Additionally, at every *sampling interval*—consisting of a fixed, user-defined number of consecutive iterations—the monitoring module feeds the gathered performance metrics to

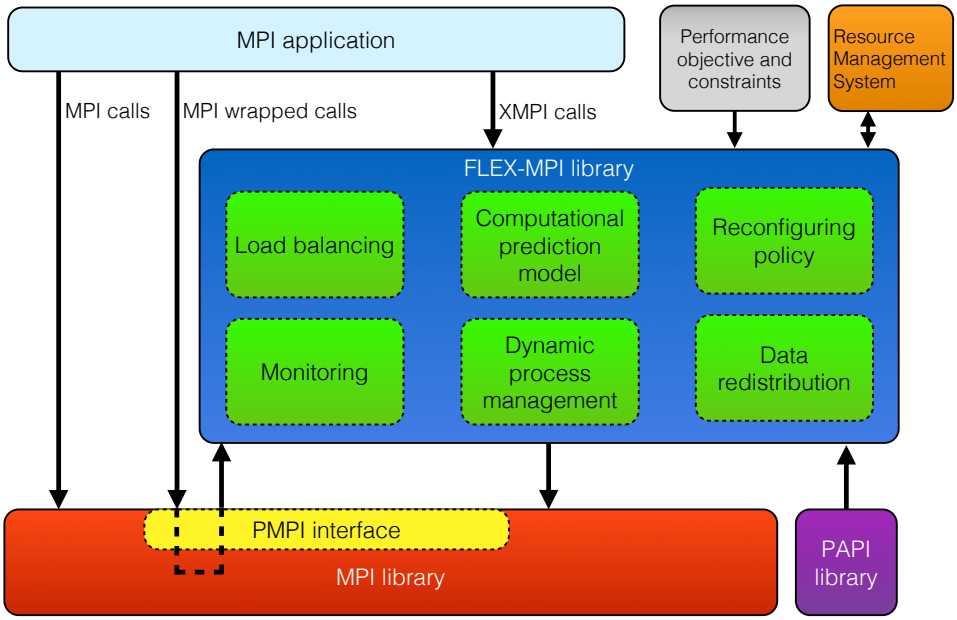


Figure 1: Execution environment of a Flex-MPI application.

the reconfiguring policy (RP) module (label `2`). This allows the RP module to track the current performance of the application and decide whether it needs to reconfigure the application to adjust the performance of the program to the objective. A reconfiguring action involves either the addition (label `3.a`) or removal (label `3.b`) of processes. The computational prediction model (CPM) estimates the number of processes and the computing power (in $FLOPS$) required to satisfy the performance objective. Using this prediction, the RP module computes the new process-to-processor mapping based on the number and type of the processors that are available and the performance constraint (efficiency or cost). The number and type of available processors is provided by the resource management system.

The dynamic process management (DPM) module implements the process spawn and remove functionalities and is responsible for rescheduling the processes according to the mapping. A reconfiguring action changes the data distribution between processes, which may lead to load imbalance. Each time a

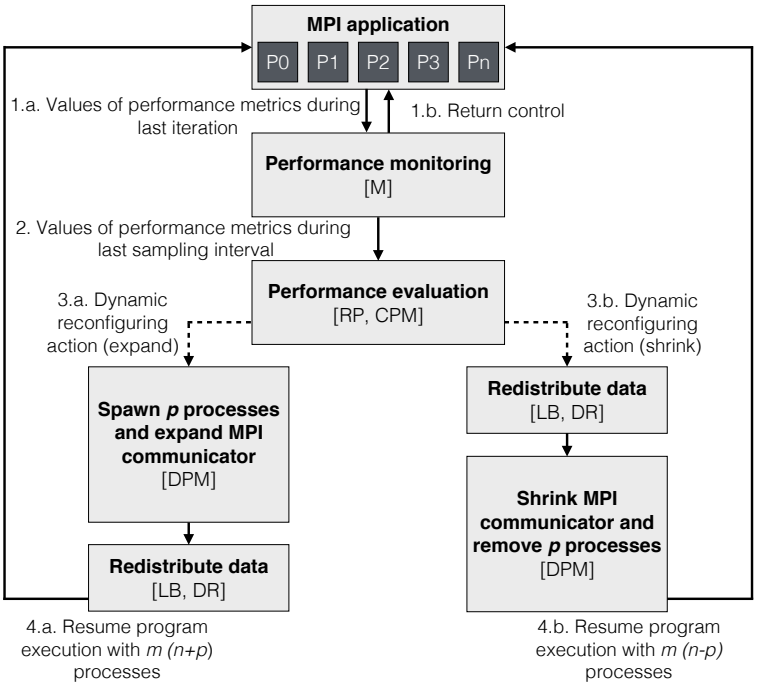


Figure 2: Workflow diagram of a malleable MPI application using Flex-MPI.

reconfiguring action is carried out the load balancing (LB) module computes the new workload distribution based on the computing power of the processing elements allocated to the application. The data redistribution (DR) module is responsible for mapping and redistributing the data between processes according to the new workload distribution. In this work we consider that each of the computing cores of a multi-core processor is a processing element (PE). We also assume that compute nodes are not oversubscribed. Once Flex-MPI has reconfigured the application to the new number of processes ($m$), it resumes its execution (labels `4.a` and `4.b`).

Application developers can access the Flex-MPI library through the API, which consists of a set of high-level interfaces—carrying the `XMPI` prefix—that automatically reconfigure the MPI application. MPI initialize (`MPI_Init`) and finalize (`MPI_Finalize`) functions are wrapped to initialize and finalize the Flex-MPI library functionalities and the MPI environment. MPI point-to-point and

Legacy code (MPI)

```
L1:  MPI_Init(...);

L2:  MPI_Comm_rank(...);
L3:  MPI_Comm_size(...);

L4:  MPI_Scatter (...);

L5:  for (it=0; it < itmax; it++) {

L6:     for (i=displ; i < displ+count; i ++) {
L7:        //Parallel computation
L8:     }

L9:     MPI_Allreduce (...);
L10: }

L11: MPI_Finalize ();
```

Instrumented code (FLEX-MPI)

```
L1:  MPI_init (...);

L2:  MPI_Comm_rank(...);
L3:  MPI_Comm_size(...);

L4:  XMPI_Get_wsize (...);

L5:  XMPI_Register (...);

L6:  XMPI_Get_shared_data (...);

L7:  for (it; it < itmax; it++) {

L8:     XMPI_Monitor_init ();

L9:     for (i=displ; i < displ+count; i ++) {
L10:       //Parallel computation
L11:    }

L12:    MPI_Allreduce (...);

L13:    XMPI_Eval_reconfiguration (...);

L14:    XMPI_Get_process_status (...);

L15:    if (status == EMPI_REMOVED) break;
L16: }

L17: MPI_Finalize ();
```

Figure 3: Comparison of the legacy code (left) and the instrumented Flex-MPI code (right) of an iterative MPI application.

collective communication operations are wrapped to collect performance metrics. Wrapped functions are managed using the MPI profiling interface (PMPI) which redirects the function calls to the Flex-MPI library in a user-transparent way.

Figure 3 shows a comparison between a simplified legacy code sample and the same code instrumented with Flex-MPI functions. The SPMD application uses a data structure (`vector A`) distributed between the processes (`L4`). In the iterative section of the code (`L5-10`) each process operates in parallel on a different subset of the data structure. At the end of every iteration the program performs a collective reduce operation (`L9`). In the legacy code all the MPI specific functions (in red) are managed by the MPI library.

The instrumented code consists of: (1) native MPI functions (in red), (2) wrapped functions (in yellow), (3) Flex-MPI functions which allow the parallel program to get and set some library-specific parameters (in blue), and (4) Flex-MPI functions to access the dynamic reconfiguration library functions (in green). Additionally, all the references to the default communicator `MPI_COMM_WORLD` in

the legacy code are replaced with `XMPI_COMM_WORLD`, a dynamic communicator provided by Flex-MPI. To simplify the presentation, the instrumented code shows the high-level interfaces of the Flex-MPI API without the required function parameters.

In Flex-MPI the MPI initialize (`L1`), finalize (`L17`), and communication (`L12`) functions are transparently managed by the Flex-MPI library using the PMPI interface. The rest of the MPI specific functions (`L2-3`) are directly managed by the MPI library. The parallel code is instrumented with a set of functions to get the initial partition of the domain assigned to each process (`L4`) and register each of the data structures managed by the application (`L5`). Registering is necessary to know which data structures should be redistributed every time a reconfiguring action is carried out.

The DR module communicates with the newly spawned processes to pass them the corresponding domain partition and the current program iteration number (`it`) before starting the execution of the iterative section (`L6`). Newly spawned processes will compute at most the remaining number of iterations (`L7`). This number is variable and depends on the iterations when the process is created and destroyed. The iterative section of the code is instrumented to monitor each process of the parallel application (`L8`) during every iteration. In addition, at every sampling interval the RP module evaluates whether reconfiguring (`L13`) is required. Then each process checks its execution status (`L14`). In case that the RP module decides to remove a process, this leaves the iterative section (`L15`) and terminates execution.

## 4. Flex-MPI components

This section describes the internal implementation of each architectural component of the Flex-MPI library.

### *4.1. Monitoring*

Flex-MPI uses PAPI and PMPI to dynamically collect performance metrics from the MPI program. We use low-level PAPI interfaces to track the number of

floating point operations $FLOP$, the real time $Treal$ (i.e. the wall-clock time), and the CPU time $Tcpu$ (i.e. the time during which the processor is running in user mode). These metrics are collected for each MPI process and they are preserved during context switch. They allow us to effectively calculate the computing power of each processor as the number of floating point operations per second $FLOPS$. Flex-MPI targets SPMD applications whose computation is based on floating point operations. We use $FLOP$ because it is a good quantitative measure of the workload performed by the class of applications we are considering. In these applications, the number of $FLOP$ is proportional to the workload computed by the process. The performance metrics collected by Flex-MPI can be easily changed to model the performance of a different class of applications (e.g. processor cycles or completed instruction count).

PMPI is an interface provided by the MPI library to profile MPI programs and collect performance data without modifying the source code of the application or accessing the underlying implementation. We use the PMPI interface to collect the type of MPI communication operation, the size of the data transferred between processes, and the time spent in communication operations. In addition, we use PMPI to wrap `MPI_Init` and `MPI_Finalize`. This allows Flex-MPI to initialize and finalize its functionalities transparently.

### *4.2. Dynamic process management*

The dynamic process management module manages the addition and removal of MPI processes, as well as the inter-process communication whenever a reconfiguring action is carried out. This functionality uses the dynamic processes management interface of MPI to spawn dynamic processes at runtime.

MPI provides a default intra-communicator `MPI_COMM_WORLD` which encapsulates the set of all processes initiated by the `mpirun`/`mpiexec` command. From now on we refer to this set of processes as the *initial* set of processes. Those processes which are dynamically spawned and removed at runtime are called *dynamic* processes. Due to the restrictions of the current implementation of MPI, only dynamic processes can be removed at runtime. The members of

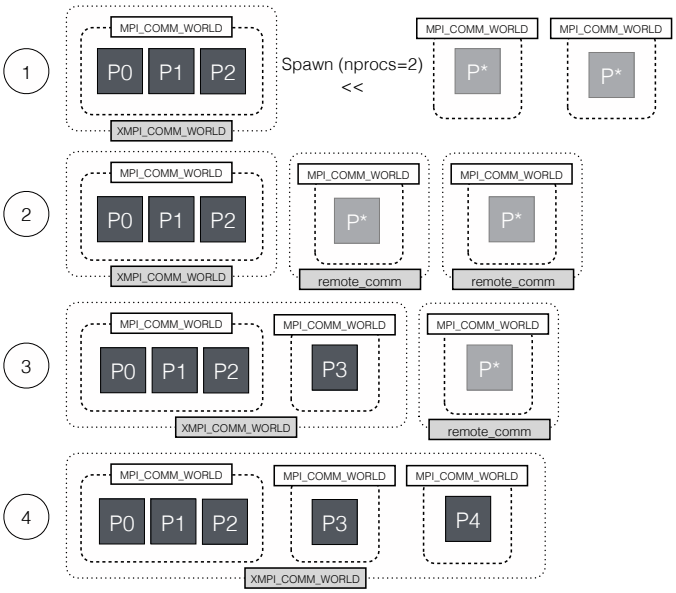


Figure 4: Low-level actions of dynamic process management functionality at process creation.

the initial set of processes (default intra-communicator `MPI_COMM_WORLD`) cannot be terminated until the program completes and dynamic processes can not be added to or removed from this default communicator. For this reason, Flex-MPI introduces its own global intra-communicator called `XMPI_COMM_WORLD` to enable communication between initial and dynamic processes. We implement this communicator using the "merge" method [33].

Flex-MPI decides to spawn a new process if: (1) there are idle processors in the system, (2) the current performance of the application does not satisfy the user-given performance objective, and (3) the computational prediction model estimates a performance improvement that satisfies the performance objective. Figure 4 illustrates the behavior of the dynamic process management module when two dynamic processes (`P3,P4`) are added to an MPI program already running on an initial set of processes (`P0-2`) (step 1). We use `MPI_Comm_spawn` to spawn new processes in Flex-MPI. Although the spawning primitive accepts as input parameter the number of processes to start ($n$), each of the new processes is spawned using an individual call to `MPI_Comm_spawn`. This makes each process

have its own (MPI_COMM_WORLD) intra-communicator and remote_comm remote communicator (step 2). The local and remote communicators are merged by invoking MPI_Intercomm_merge, which returns a new XMPI_COMM_WORLD intra-communicator encapsulating processes P0-3 (step 3). The merge function is invoked once more to merge this intra-communicator and the remote communicator of P4. The result is a global intra-communicator which encapsulates all of the processes (P0-4) (step 4).

This design choice allows fine-grained control over the number of application processes to satisfy the performance constraints. The downside is that process creation time varies linearly with the number of dynamically spawned processes. The current implementation of Flex-MPI supports the creation of $n > 1$ simultaneous processes using MPI_Comm_spawn. However, due to implementation constraints of communicators in MPI, those processes spawned via an individual call to MPI_Comm_spawn cannot be removed individually in subsequent sampling intervals—and group termination may negatively affect the application performance. This is a way to reach a trade off between process creation costs and the granularity of reconfiguring actions (as the number of processes simultaneously created or destroyed) and may be useful for those execution scenarios which involve the dynamic creation of a large number of processes.

The reconfiguring policy dictates both the number of processes and the type of processors on which to spawn them. MPI provides a mechanism to set the *host* key of the MPI_Info argument of MPI_Comm_spawn to the host name of the compute node where the new process needs to be allocated. The dynamic process management module implements a scheduler which uses the mechanism provided by MPI to map processes to compute nodes with processor types corresponding to those dictated by the reconfiguring policy.

Flex-MPI removes a dynamic process if: (1) the current performance of the application does not satisfy the user-given performance objective, and (2) the computational prediction model estimates a performance reduction that satisfies the performance objective as result of this operation. Removing a dynamic MPI process from an application implies disconnecting the process from

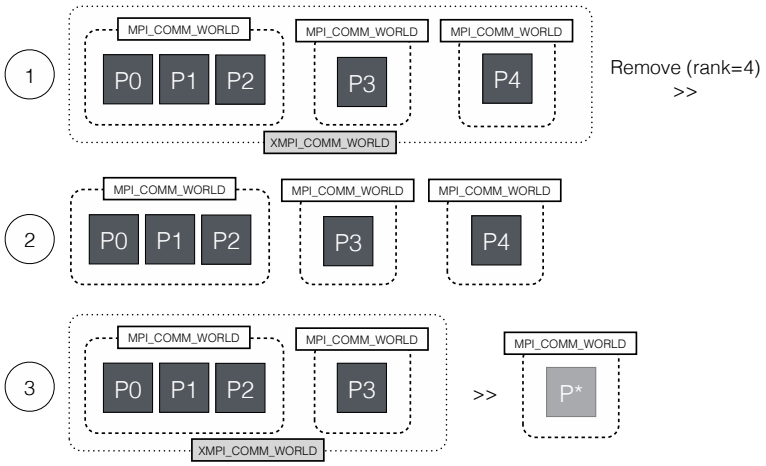


Figure 5: Low-level actions of dynamic process management functionality at process removal.

`XMPI_COMM_WORLD` to allow the process to leave the iterative section and finish execution by invoking `MPI_Finalize`. This operation is implemented by first deallocating the merged intra-communicator and then allocating a new intra-communicator for the remaining processes. Due to the fact that `MPI_Finalize` is blocking and collective for all the processes in `MPI_COMM_WORLD`, each dynamic process must have been spawned via a separated call to allow individual termination.

Figure 5 illustrates the behavior of the dynamic process management module when process `P4` is removed from the previous MPI program (step 1). The current `XMPI_COMM_WORLD` is deallocated. This allows disconnecting `P4` from the rest of the processes (step 2). A new group is then formed via `MPI_Group_incl` to include `P0-P3`, and a new intra-communicator `XMPI_COMM_WORLD` is allocated for this group. `P4` finishes its execution by calling `MPI_Finalize` (step 3).

### *4.3. Load balancing*

Load balancing is a major issue in parallel applications [12] because it can have a huge impact on the overall performance of the program. Flex-MPI integrates a dynamic load balancing technique [34] for SPMD applications that uses the performance metrics collected by the monitoring functionality to make work-

load distribution decisions. One of the main advantages of this approach is that it does not require prior knowledge about the underlying architecture. The dynamic load balancing technique implements a coordination-based method [35].

The load balancing module computes the new workload distribution using the values for the computing power of each processor on which the application is running the MPI processes. The idea is to assign to each process a data partition that is proportional to the *relative computing power* ($RCP$) of the processor on which it is running. The relative computing power of processor $i$ ($RCP_i$) [36, 37] is computed in Equation 1 as the computing power of the processor (in $FLOPS$) divided by the sum of the computing power of all of the $p$ processors on which the MPI program is running. Rather than balance the workload using the number of rows or columns as reference, our balancing algorithm uses the number of nonzero elements. This allows us to balance the workload of parallel applications that use sparse data structures.

$$RCP_i = \frac{FLOPS_i}{\sum_{i=0}^{p} FLOPS_i} \tag{1}$$

More importantly, our approach can balance the workload of applications running both in dedicated and in shared modes [38]. When having exclusive access to the resources it can still be the case that the workload is not distributed according to the computing power of each processor—either from the start or during the execution. In a non-dedicated system [38] the computing resources are shared between different user applications which may come and go, and can have irregular execution patterns. Our previous work [34] provides more details and a practical evaluation of the dynamic load balancing algorithm.

### *4.4. Data redistribution*

Flex-MPI provides a user-transparent data redistribution mechanism which is triggered as a result of load balancing when a reconfiguring action is carried out. The data redistribution module uses MPI standard messages to efficiently move data between MPI processes at runtime. The data structures that it can

handle must be one-dimensional (e.g. vectors) or two-dimensional (e.g. matrices), and they may be dense or sparse with block-based one-dimensional (row or column) domain decomposition. Once the load balancing module decides the new workload distribution, the data redistribution module maps it to a set of data partitions—one per process—and moves this data from the previous to the new owners.

When the dynamic process management module spawns a new process this will receive a portion of the data which is proportional to the computing power of the processor mapped to the process. When a process is terminated its data is transferred to the remaining processes according to the computing power of the processors mapped to each of these processes.

### *4.5. Computational prediction model*

At the end of every sampling interval the computational prediction model calculates the performance of the application when executing on different possible new processor configurations. The execution time of a parallel application ($T_{si}$) during a sampling interval ($si$) depends on the computation time $T_{computation}$ and the communication time $T_{communication}$. In this work we assume that the MPI application uses synchronous MPI operations. In this case the synchronization overhead counts as part of the communication time of the application. We account separately for the process creation and termination operations involved in reconfiguring ($T_{overhead\_process}$), as well as for the data redistribution operations ($T_{overhead\_data}$). Equation 2 defines the execution time of a parallel application during a sampling interval as computed by the computational prediction model, and it takes as input the application performance data collected by monitoring and the system network performance data collected via benchmarking prior to the MPI program execution.

$$T_{si} = T_{computation} + T_{communication} + T_{overhead\_process} + T_{overhead\_data} \quad (2)$$

*4.5.1. Modeling the computation cost*

To estimate the computation time of the application during the next sampling interval Flex-MPI needs to first obtain the number of $FLOP$ in the current sampling interval. This value is provided by the monitoring module. For SPMD applications with regular computation patterns Flex-MPI uses linear extrapolation to predict the number of $FLOP$ in the next sampling interval based on the values in the past intervals. For those applications with irregular computation patterns Flex-MPI uses a profiling of the parallel application prior to the malleable execution. The reason to use profiled data is that there is no reliable method to predict the computation pattern of irregular applications [5]. For these applications, the computational prediction model of Flex-MPI uses the number of $FLOP$ collected at runtime to correct differences between the profiled and measured performance.

Equation 3 calculates the predicted computation time for the next sampling interval. It takes as inputs the estimated $FLOP$ and the computational power ($FLOPS$) of each processor ($p$).

$$T_{computation} = \frac{FLOP}{\sum_{p=1}^{np} FLOPS_p} \tag{3}$$

*4.5.2. Modeling the parallel communication cost*

There are several parallel communicational models to predict the performance of MPI communication operations, the most known of which are LogGP [39] and PLogP [40]—based on LogP [41] and the Hockney model [42]. These models use a set of standardized system parameters to approximate the performance of the algorithms which implement the collective MPI operations. The cost models for the synchronous MPI operations provided by MPICH [7] are all based on the Hockney model [43, 44]. We use these to model the performance of the communication operations in Flex-MPI.

The Hockney model assumes that the time to communicate between two nodes is $\alpha + n\beta$, where $\alpha$ is the network latency, $n$ is the size of the message in bytes, and $\beta$ is the transfer time per byte. In addition to these, the MPICH

cost models use two more parameters: $p$—the number of processes involved in the communication—, and $\gamma$—used for reduction operations.

The communication model uses the profiling data gathered by the monitoring module (via PMPI) and the MPICH cost functions to predict the cost of communications for the current sampling interval ($T_{communication\_model\_i}$). Flex-MPI requires that $\alpha$, $\beta$, and $\gamma$ are previously defined based on the network performance. These values are provided to Flex-MPI via a configuration file. To obtain precise estimations we introduce $\lambda$, a correction parameter that accounts for the difference between the estimation for the previous sampling interval $(i-1)$ and the real value as measured by the monitoring module (Equation 4). This value is then used to correct for the estimation made by the Hockney model for the current sampling interval $(i)$ (Equation 5).

$$\lambda_i = \frac{T_{communication\_estimated\_i-1}}{T_{communication\_real\_i-1}} \tag{4}$$

$$T_{communication\_estimated\_i} = T_{communication\_model\_i} \times \lambda_i \tag{5}$$

*4.5.3. Modeling process creation and termination costs*

The costs of creating and destroying a dynamic process depend on various factors such as the operating system, the MPI implementation, and the size of the program binary. This implies performing offline tests to obtain these values and pass them as input to the Flex-MPI library. By default Flex-MPI creates and destroys processes individually; the overall process creation and destruction costs will therefore grow linearly with the number of dynamic processes involved in reconfiguring actions. Equation 6 and Equation 7 model the costs associated with creating *nprocs_spawn* and removing *nprocs_remove* dynamic processes executing an MPI application, where *process_spawning_cost* and *process_removing_cost* are the creation and termination cost per process.

$$T_{overhead\_process(spawn)} = nprocs\_spawn \times process\_spawning\_cost \tag{6}$$

$$T_{overhead\_process(remove)} = nprocs\_remove \times process\_removing\_cost \quad (7)$$

*4.5.4. Modeling data redistribution costs*

Data redistribution operations use the collective communication primitive `MPI_Alltoallv` to efficiently distribute data between processes. Flex-MPI can redistribute dense vectors and matrices, as well as sparse matrices stored in $CSC$ (*Compressed Sparse Column*) or $CSR$ (*Compressed Sparse Row*) format. The load balancing functionality calculates the future distribution depending on the number of rows, columns, or nonzero elements of each structure that need to be assigned to every process based on their type (e.g. `int`, `double`) and the $RCP$ of the processors.

The cost of data redistribution is computed using Equation 8, where $nd$ and $ns$ are the number of dense and sparse data structures, $data_k$ is the total size of data structure $k$ that is redistributed in the current step, and $p$ is the number of processes involved in redistribution. The estimation takes into account the storage format for the dense (*Cost_rdata_dense*) and sparse (*Cost_rdata_sparse*) data structures. The redistribution costs are calculated using the cost function for `MPI_Alltoallv` based on the Hockney model.

$$T_{overhead\_data} = \sum_{i=1}^{nd} Cost\_rdata\_dense(data_i, p) + \sum_{j=1}^{ns} Cost\_rdata\_sparse(data_j, p) \quad (8)$$

*4.6. Reconfiguring policy*

Algorithm 1 shows the pseudocode of the reconfiguring algorithm. This functionality checks whether the application satisfies the user-given performance objective at every sampling interval. If so, it continues executing on the same processor configuration. Otherwise it performs a reconfiguring action by adding or removing processes. The algorithm consists of three phases: the first phase for performance evaluation; the second phase for analysis of malleable reconfiguring actions; and the third phase for process reconfiguring.

**Algorithm 1** Reconfiguring algorithm for malleable applications with performance constraints.

```
1: ———— First phase ————
2: Texe_accum ← Texe_accum + Treal_n
3: Tgoal_{n+1} ← calculateGoal(Texe_accum, Goal)
4: Test_{n+1} ← CPM(alloc_procs_set)
5: if |Test_{n+1} − Tgoal_{n+1}| < tol then
6:    ———— Second phase ————
7:    if Test_{n+1} > (Tgoal_{n+1} + tol) then
8:       avail_procs_set ← requestAvailableProcessorsRMS()
9:       for s = 0 to s = |avail_procs_set| do
10:         (ΔFLOPS, T_comm, T_spawn, T_rdata) ← CPMreconfig(s, T_goal, cFLOPS)
11:         (procs_set, T_comp, cost) ← mappingSimplex(constraint, ΔFLOPS, s, avail_procs_set)
12:         Test_S_{n+1} = T_comp + T_comm + T_spawn + T_rdata
13:         if |Test_S_{n+1} − Tgoal_{n+1}| < tol then
14:            suitable_procs_sets ← push(procs_set)
15:         end if
16:      end for
17:   else if Tgoal_{n+1} > (Test_{n+1} + tol) then
18:      for r = 0 to r = MAX_PROCS_REMOVE do
19:         (ΔFLOPS, T_comm, T_remove, T_rdata) ← CPMreconfig(r, T_goal, cFLOPS)
20:         (procs_set, T_comp, cost) ← mappingSimplex(constraint, ΔFLOPS, r, alloc_procs_set)
21:         Test_R_{n+1} = T_comp + T_comm + T_remove + T_rdata
22:         if |Test_R_{n+1} − Tgoal_{n+1}| < tol then
23:            suitable_procs_sets ← push(procs_set)
24:         end if
25:      end for
26:   end if
27: end if
28: new_procs_set ← selectBestProcsSet(suitable_procs_sets, alloc_procs_set)
29: ———— Third phase ————
30: submitAllocationRMS(new_procs_set)
31: reconfiguring(new_procs_set)
32: new_workload_distribution ← loadBalance(new_procs_set)
33: dataRedistribution(new_workload_distribution)
```

**First Phase** (`lines 1-5`): The first step (`line 2`) consists of capturing (via monitoring) the execution time $Treal_n$ of the current sampling interval $n$ to update the application execution time ($T_{accum}$). This value is used by *calculateGoal* (`line 3`) to compute the execution time $Tgoal_{n+1}$ that is necessary to satisfy the user-given performance objective during the next sampling interval. $CPM$ (`line 4`) uses the computational prediction model to predict the execution time for the next sampling interval ($Test_{n+1}$) under the current processor configuration. When the difference between the required and predicted

execution times is bigger than a given threshold *tol* (`line 5`) the algorithm performs a reconfiguring action.

The **Second Phase** (`lines 6-28`) analyzes different process reconfiguring scenarios and selects the best ones under user-given performance objective and performance constraints. The algorithm first decides—in `lines 7 and 17`—whether to increase or decrease the number of PEs depending on which of the predicted and required times is bigger. To add new processors (`line 8`) Flex-MPI sends a request to the RMS. The RMS sends back a list of the additional PEs that can be used by the malleable application. Each iteration starting in `line 9` evaluates a different *execution scenario*, which is associated to a number $s$ of additional PEs ranging from 0—no reconfiguration—to the maximum number available. Function $CPMreconfig$ in `line 10` uses the computational prediction model to calculate the number of FLOPS ($\Delta FLOPS$) necessary to achieve the performance objective $Tgoal$. The computational prediction model uses as a parameter the number of currently allocated PEs ($p$) and it takes into account the predicted times for communication and the predicted reconfiguring overheads (for both process creation and data redistribution) when $p + s$ processors are used.

Function *mappingSimplex* (`line 11`) uses the Simplex algorithm [45] to find a set *procs_set* of $s$ PEs that satisfies the performance objective according to the user-given constraint. This is necessary because we consider heterogeneous architectures consisting of PEs with different performance characteristics. In the case of imposing the efficiency constraint the function returns the PE set whose computational power is closer to $\Delta FLOPS$. In the case of the cost constraint it returns the PE set with the smallest operational cost and computational power closest to $\Delta FLOPS$. `Line 12` calculates the predicted execution time during the sampling interval $Test\_S_{n+1}$. Due to the reconfiguring overheads it is possible that this time does not satisfy the performance objective. For this reason, in `line 13` the algorithm evaluates if the deviation of the execution time is below a predefined tolerance. If true, *proc_set* is stored in a list of suitable scenarios (`line 14`). This procedure is repeated for each value of $s$.

To remove processes we are limited to the number of dynamic processes which have been previously spawned by the application ($MAX\_PROCS\_REMOVE$). For each configuration scenario $r$ we calculate the computational power (`line 19`) that satisfies the performance objective. Function *mappingSimplex* (`line 20`) now returns the specific subset of PEs that needs to be removed to obtain the required computational power—maximizing the number of processes that will be removed to improve efficiency and save operational costs.

Function *selectBestProcsSet* (`line 28`) selects from the list of suitable execution scenarios the processor configuration which satisfies both the performance objective and the performance constraint. For the efficiency constraint the algorithm selects the scenario which leads to the smallest number of processes. For the cost constraint it selects the one which leads to the minimum operational cost.

Finally, in the **Third Phase** (`lines 29-33`) the algorithm reconfigures the application to run on the newly selected processor configuration. In (`line 30`) Flex-MPI notifies the RMS of the new processor allocation. The following steps consist in performing the process reconfiguring through the dynamic process management functionality (`line 31`), computing the load balance for the new processor configuration (`line 32`), and redistributing the workload (`line 33`).

## 5. Experimental results

The platform we used for evaluation is a heterogeneous cluster consisting of 23 nodes of 4 different types connected via a flat Gigabit Ethernet network—i.e. all nodes are connected to the same switch and therefore the latency and bandwidth are equal for all node pairs. The nodes run Linux Ubuntu Server 10.10 with the 2.6.35-32 kernel and the MPICH 3.0.4 distribution. The cluster is managed by the TORQUE resource manager [46]. Table 1 shows the characteristics of each node class.

To perform a realistic evaluation, we assigned an operational cost to each computing core based on the economic costs incurred when using the equivalent Amazon EC2 instances in terms of performance. Table 2 summarizes the

Table 1: Configuration of the heterogeneous cluster with number of compute nodes and cores for each node class.

| Class | Nodes | Cores | Processor | Frequency | RAM |
|---|---|---|---|---|---|
| C1 | 20 | 80 | Intel Xeon E5405 | 2.00 GHz | 4 GB |
| C7 | 1 | 24 | Intel Xeon E7-4807 | 1.87 GHz | 128 GB |
| C6 | 1 | 12 | Intel Xeon E5645 | 2.40 GHz | 24 GB |
| C8 | 1 | 24 | Intel Xeon E5-2620 | 2.00 GHz | 64 GB |

Table 2: Performance evaluation and cost model of the Amazon EC2 platform.

| Instance type | Performance per core (GFLOPS) | Cost per core ($/hour) | Economic efficiency per core (GFLOPS/$1) |
|---|---|---|---|
| m1.small | 2.00 | 0.100 | 20.00 |
| c1.medium | 1.95 | 0.100 | 19.50 |
| m1.xlarge | 2.85 | 0.200 | 14.25 |

operational cost of Amazon EC2 that we obtained from [47]. Table 3 shows the actual costs for each node class of our cluster. We evaluated the performance of each node class using the HPL benchmark [48] for values of $N$ (order of the coefficient matrix A) between 18,000 and 110,000, depending on the RAM capacity of each node. We can see that these costs are proportional to those in Table 2 in terms of their performance per processor core. We then associated each class with an Amazon EC2 instance of similar performance (column *Related Amazon EC2 instance type* in Table 3). Based on this association we assigned the same economic efficiency to the classes as that of the corresponding Amazon EC2 instances. For C8 nodes the performance is not similar to any of the Amazon EC2 instances. We assigned them a smaller economic efficiency which allows us to evaluate the effectiveness of using Flex-MPI with three node categories: a powerful, expensive, economically inefficient class C8, two not highly powerful, but cheap and highly cost-efficient classes C1 and C7, and a class C6 of intermediate performance and efficiency.

Table 3: Performance evaluation and cost model of the cluster.

| Class | Related Amazon EC2 instance type | Performance per core (GFLOPS) | Cost per core ($/hour) | Economic efficiency per core (GFLOPS/$1) |
|---|---|---|---|---|
| C1 | m1.small | 1.90 | 0.095 | 20.00 |
| C7 | c1.medium | 2.25 | 0.115 | 19.50 |
| C6 | m1.xlarge | 2.90 | 0.204 | 14.25 |
| C8 | - | 4.62 | 0.462 | 10.00 |

Table 4: Problem sizes for the benchmarks evaluated.

| | Jacobi | | | Conjugate Gradient | | | EpiGraph | | |
|---|---|---|---|---|---|---|---|---|---|
| **Problem size** | **Order** | **NNZ** | **Size (MB)** | **Order** | **NNZ** | **Size (MB)** | **Order** | **NNZ** | **Size (MB)** |
| A | 10,000 | $1.0\times10^8$ | 381 | 18,000 | 6,897,316 | 99 | 1,000,000 | 145,861,857 | 1,308 |
| B | 20,000 | $4.0\times10^8$ | 1,523 | 36,000 | 14,220,946 | 210 | 2,000,000 | 180,841,317 | 1,803 |
| C | 30,000 | $9.0\times10^8$ | 3,454 | 72,000 | 28,715,634 | 440 | 3,000,000 | 241,486,871 | 2,462 |

Our benchmark suite consists of three parallel applications: Jacobi, Conjugate Gradient, and EpiGraph. They are written in C and modified to integrate high-level Flex-MPI interfaces. Jacobi is an application which implements the iterative Jacobi method for solving systems of linear equations that uses dense matrices. Conjugate Gradient implements an iterative algorithm for solving systems of linear equations that use sparse, symmetrical, and definite positive matrices. EpiGraph [49] is an epidemic simulator for urban environments and operates on a sparse matrix that represents the interconnection network between the individuals in the population. Jacobi and Conjugate Gradient have a regular communication pattern and exhibit an approximately constant execution time per iteration. EpiGraph's workload, on the other hand, varies over time depending on the number of infected individuals during each iteration. The communication pattern in EpiGraph is irregular, and the number of communications and the size of the data sent between processes varies over time.

Table 4 shows the different problem sizes that we used for our benchmarks.

The dense matrices we use for Jacobi were randomly generated using MATLAB [50]. The sparse matrices in Conjugate Gradient are a subset of the University of Florida sparse matrix collection [51]. The sparse matrices used by EpiGraph are generated by the application based on actual data from real social networks [49]. Each benchmark application was executed for a different number of iterations. We executed Jacobi for 2,500 iterations, CG for 20,000 iterations, and EpiGraph for 86,400 iterations (which corresponds to 60 days of simulation). We used a sampling interval of 100 iterations for Jacobi and CG, and of 8,640 iterations for EpiGraph.

### *5.1. Validating the computational prediction model*

This section describes the experiments conducted in order to collect the system parameters required by the computational prediction model as well as the process of validation.

We first present an analysis of the overhead of process creation and termination operations during the execution of our benchmark applications. We measure the overhead of process creation and termination using predefined reconfiguring points during the execution. These values are used to effectively predict the overhead of reconfiguring actions in Flex-MPI. Figures 6 (a) and (b) show these overheads for our benchmarks. The size of the binaries are 28KB each for Jacobi and CG, and 184KB for EpiGraph. In these figures the x-axis represents the number of dynamic processes spawned or removed. For the measurement of the overhead of process creation all applications start with 8 initial processes. For instance in Figure 6 (a) the $x$ value 8 means that the application goes from executing on 8 to 16 processes. For the measurement of the overhead of process termination we trigger the process removing action when the application is running on 80 processes. The measurement of process termination is slightly more complex due to the fact that only those processes which have been previously spawned dynamically can be later removed. For instance, we measure the overhead of removing 32 processes—$x$ value 32 in Figure 6 (b)—by starting the application with 48 processes, spawning 32 dynamic processes, then

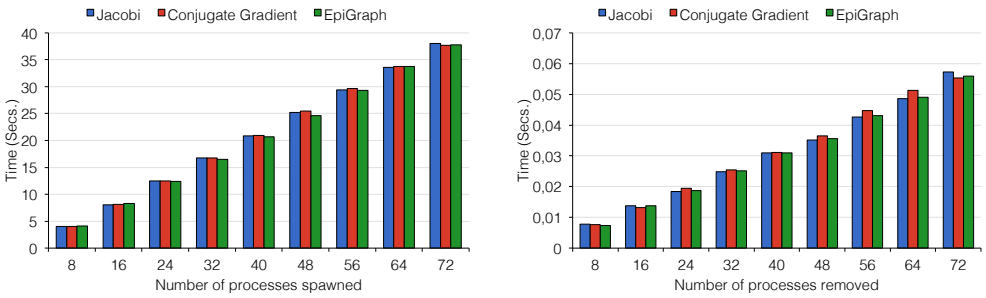


Figure 6: Measured times for the creation and termination of dynamic MPI processes on 20 computing nodes of class C1.

Table 5: Hockney model parameters measured on the ARCOS cluster.

| **Parameter** | **Description** | **Measured value** |
|---|---|---|
| $\alpha$ | Latency | 50 $\mu$secs. |
| $\beta$ | Transfer time | 0.008483 $\mu$secs. per byte |
| $\gamma$ | Computation cost of reduction operation | 0.016000 $\mu$secs. per byte |

removing them and measuring the time spent in this last operation. For our benchmarks the average creation and destruction times in the ARCOS cluster are 520.1 $ms$ and 0.8 $ms$. These values show that the creation and destruction costs do not depend on the binary size in the case of our benchmarks.

To predict the performance of communication operations in Flex-MPI we use MPICH's communication models, which are based on the Hockney model. Table 5 shows the parameter values for the Hockney model. These parameters were measured directly on the ARCOS cluster using point-to-point tests.

To validate the computational prediction model we execute a modified Jacobi code in which the reconfiguring actions are predefined to occur at a particular iteration. Figure 7 shows a comparison between the predicted and real times for executions starting from (a) 8, (b) 16, (c) 32, and (c) 64 initial processes when adding and removing different numbers of dynamic processes. Real times correspond to the times measured during the sampling interval following the sampling interval in which the reconfiguring action is carried out. Results

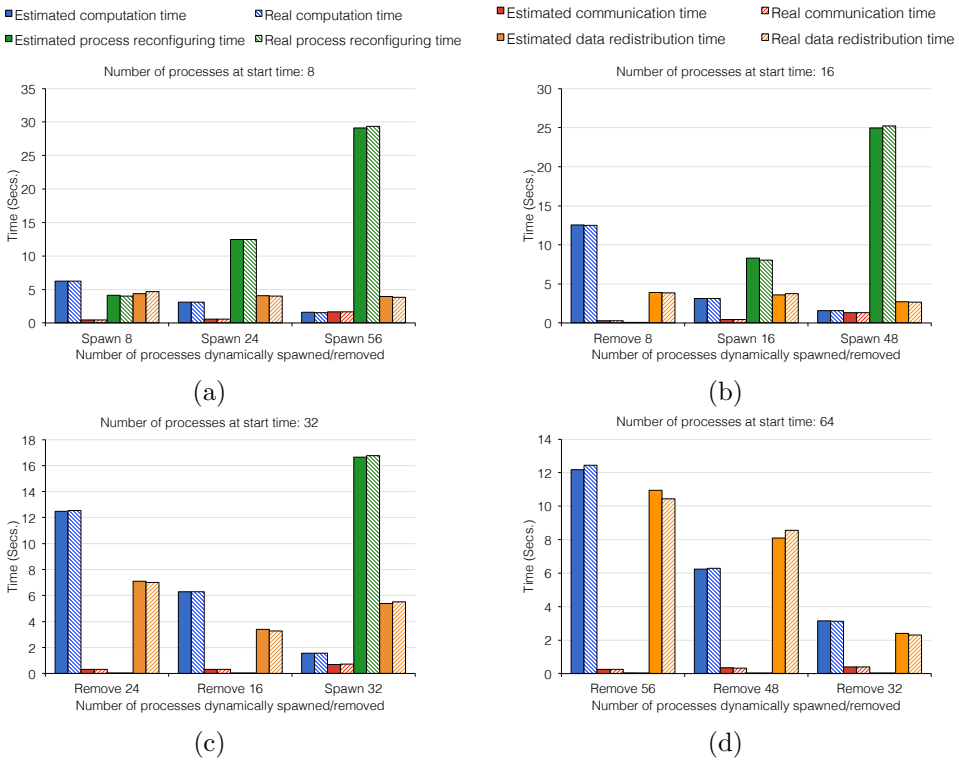


Figure 7: Comparison between predicted and real times for different dynamic reconfiguring actions.

indicate that CPM can predict with great accuracy the performance of parallel applications prior to a reconfiguring action, such as the relative error of estimation times show: [-0.79%, 2.15%] for computation, [-4.79%, 4.44%] for communication, [-4.92%, 5.55%] for process reconfiguring, and [-4.76%, 6.45%] for data redistribution.

### *5.2. Overhead analysis*

This section presents the performance evaluation of the overhead of Flex-MPI. To do this we compare the execution times for J.B.8 for the following cases: (1) the program executes legacy MPI code (compiled with MPICH v3.0.4) with static process allocation, (2) the program executes Flex-MPI code with static scheduling, (3) the program executes Flex-MPI code with dynamic reconfiguration under efficiency constraints, and (4) the program executes Flex-MPI

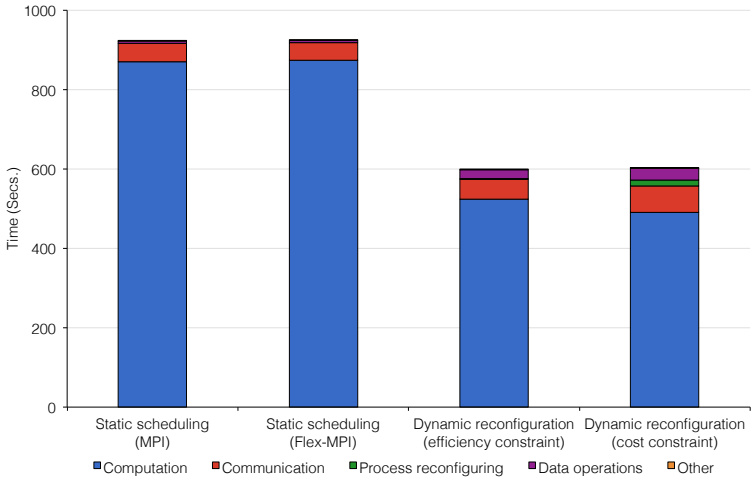


Figure 8: Performance overhead for legacy MPI with static scheduling and Flex-MPI with static static scheduling and dynamic reconfiguration, for J.B.8.

code with dynamic reconfiguration under cost constraints. The performance objective in (3, 4) is to reduce the completion time of the application by 35% compared to the completion time of the application with static scheduling (1, 2). Figure 8 reflects the time allocated to the different phases—computation, communication, process reconfiguring, data operations, and other—for each of these four cases. For scenarios (1) and (2) the data operations time accounts for the time it takes to read the matrix data from disk; for scenarios (3) and (4) it additionally accounts for the data redistribution time. For scenario (1) *other* summarizes the overhead of the MPICH library; for scenarios (2), (3), and (4) it summarizes the overhead of Flex-MPI library initialization, monitoring, communication profiling, and evaluation of load balancing and reconfiguration algorithms.

When comparing the results for (1) and (2) we see that the Flex-MPI overhead is negligible and has no impact on the final application performance. The results for dynamic reconfiguration (3, 4) show that the Flex-MPI overhead (including process reconfiguring, data operations, and other) takes up to 13.81% of the execution time of the dynamic application. These results reflect the trade off between performance improvement and the overhead of the Flex-MPI library.

Table 6: Number of processes initially scheduled ($Np$) and their mapping to the available class nodes for each malleability test case.

| | | | | Process mapping | | | |
|---|---|---|---|---|---|---|---|
| **Test case** | **Problem** | **Problem size** | **Np** | **C1** | **C7** | **C6** | **C8** |
| J.A.8 | Jacobi | A | 8 | 2 | 2 | 2 | 2 |
| J.B.8 | Jacobi | B | 8 | 2 | 2 | 2 | 2 |
| J.C.8 | Jacobi | C | 8 | 2 | 2 | 2 | 2 |
| J.C.24 | Jacobi | C | 24 | 6 | 6 | 6 | 6 |
| CG.A.4 | Conjugate Gradient | A | 4 | 1 | 1 | 1 | 1 |
| CG.B.4 | Conjugate Gradient | B | 4 | 1 | 1 | 1 | 1 |
| CG.C.4 | Conjugate Gradient | C | 4 | 1 | 1 | 1 | 1 |
| CG.C.8 | Conjugate Gradient | C | 8 | 2 | 2 | 2 | 2 |
| E.A.8 | EpiGraph | A | 8 | 2 | 2 | 2 | 2 |
| E.B.8 | EpiGraph | B | 8 | 2 | 2 | 2 | 2 |
| E.C.8 | EpiGraph | C | 8 | 2 | 2 | 2 | 2 |

### *5.3. Performance evaluation of malleable MPI applications*

This section presents the performance evaluation of Flex-MPI capabilities. Table 6 summarizes the test cases that we considered and the number of initial processes and types of processors for each one of them. In our experiments the performance objective is to reduce the completion time of the malleable applications by 25%, 30%, and 35% compared to the completion time for static scheduling—the completion time of the application using $Np$ processes with a static processor allocation. For each of these objectives we evaluate the execution under both constraint types—efficiency and cost. The completion time for static scheduling is the sum of computation and communication times, as well as load balance and data redistribution overheads in case that the application is unbalanced. Dynamic reconfiguration incurs overheads of process creation and termination, in addition to load balance and data redistribution overheads associated with the reconfiguring actions. The maximum number of processors available for each benchmark application corresponds to the number of resources

of the ARCOS cluster as shown in Table 1. To provide a fair comparison we apply load balance in both static and dynamic scenarios.

Figure 9 shows a comparison between the behavior of the reconfiguring policy module under efficiency (a) and cost (b) constraints when executing J.B.8 with a performance improvement objective of 35%. The statically scheduled application takes 923 seconds and 1,535 cost units to complete on 8 processors. When we impose the efficiency constraint—Figure 9 (a)—Flex-MPI triggers two reconfiguring actions at iterations 300 and 2,200 to increase the computing power of the application. Flex-MPI optimizes resource provisioning by minimizing the number of dynamically spawned processes. The maximum number of simultaneously executing processes using the efficiency constraint is 13 with a total operational cost of 1,928 units. Dynamic processes execute on the least cost-efficient yet most powerful processors—of class C8. When we impose the cost constraint—Figure 9 (b)—Flex-MPI schedules new processes on the most cost-efficient processors of our cluster—nodes of class C1. Flex-MPI triggers several reconfiguring actions to satisfy the performance objective and the cost constraint guided by the performance-aware reconfiguring policy. We can see that in iteration 1,100 the reconfiguring policy concludes that the current performance is below what is needed to reach the objective. As a result it increases the computing power by adding three additional processors. In iteration 1,300 the same module concludes that these processors lead to a performance above what is needed and eliminates two of them. The maximum number of simultaneously executing processes using the cost constraint is 20 with a total operational cost of 1,543 units. The dynamic application running under the efficiency constraint takes 597 seconds to execute—which is 35.33% faster than static scheduling. The dynamic application with the cost constraint takes 601 seconds– 34.89% faster than static scheduling. Both dynamic executions satisfy the performance objective.

Figure 10 shows the workload (in $GFLOP$) of Jacobi in every iteration and the computing power (in $GFLOPS$) of the processor configuration in every sampling interval for J.B.8. The workload stays by and large constant in both

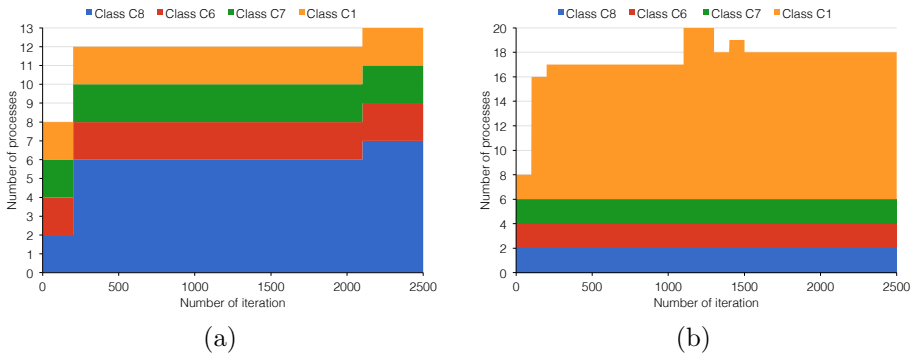


Figure 9: Number of processes and type of processors scheduled by Flex-MPI for the execution of J.B.8 under the efficiency (a) and cost (b) constraints.

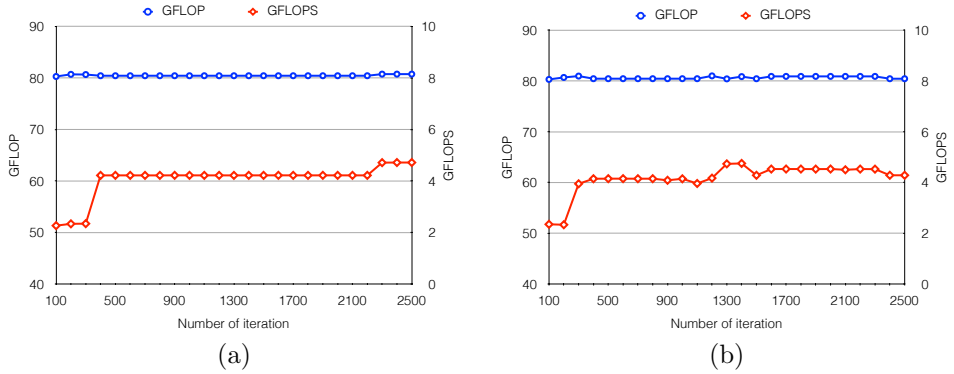


Figure 10: Application workload (in blue, left y-axis) and computing power (in red, right y-axis) for the execution of J.B.8 under the efficiency (a) and cost (b) constraints.

cases, regardless of the number of simultaneously executing processes. Figure 9 shows that the computing power varies with the number and type of processes that are added or removed in every sampling interval. This affects the execution time per iteration and therefore the completion time of the application.

Figure 11 summarizes the performance improvement of dynamic reconfiguring compared with static scheduling for the our test cases (Table 6) and performance objectives of 25%, 30%, and 35% performance improvement with both efficiency and cost constraints. Results show that Flex-MPI dynamically adapts the number of processors to satisfy the user-given performance requirements. Note that the number and type of processors allocated to achieve the

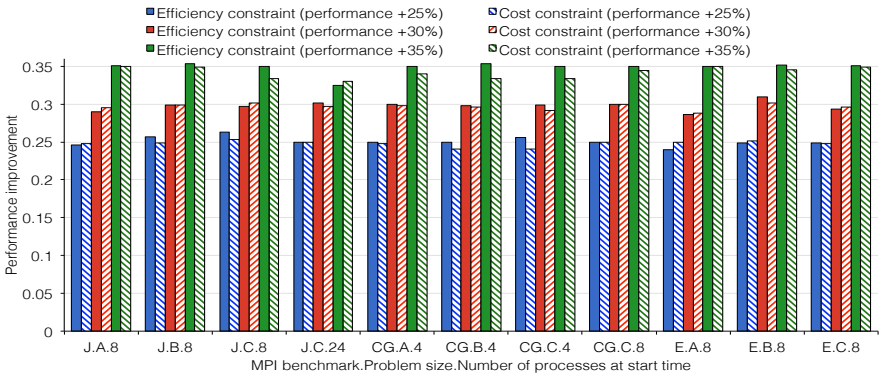


Figure 11: Performance evaluation of the malleable MPI applications with Flex-MPI support for satisfying the performance objective.

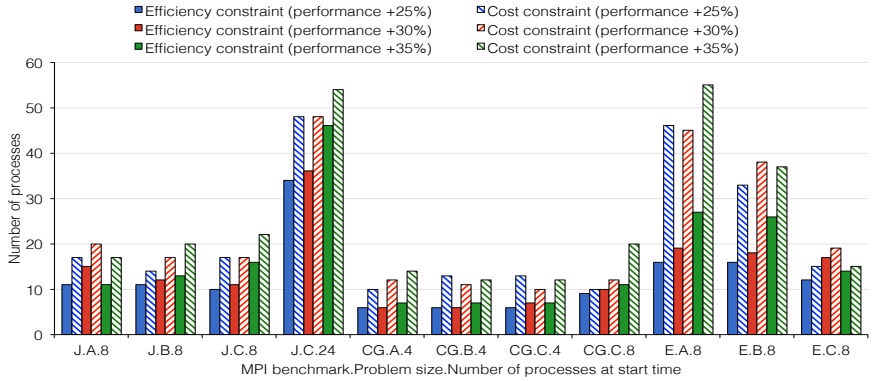


Figure 12: Number of dynamic processes scheduled by Flex-MPI for satisfying the performance objective for the malleable MPI applications.

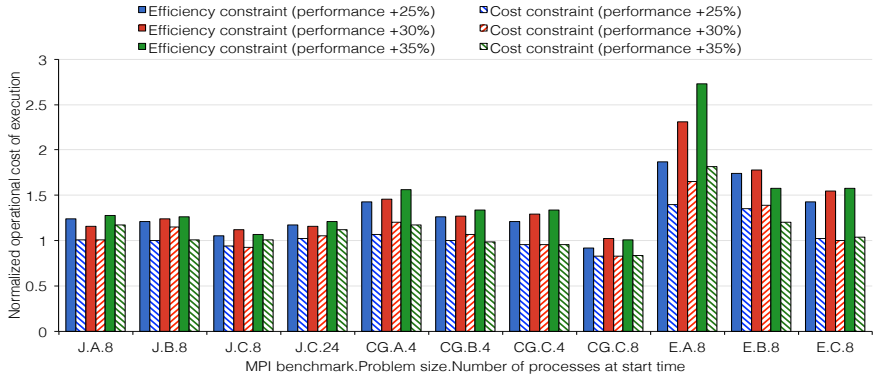


Figure 13: Cost analysis of the malleable MPI applications with Flex-MPI support for satisfying the performance objective.

performance improvement depend on the user-given performance constraint. This can be seen in greater detail in Figures 12 and 13. Figure 12 shows the maximum number of simultaneously executing processes for each performance objective, for each test case. The number of processes is always bigger when imposing the cost, rather than the efficiency, constraint. This is because the most cost-efficient nodes are the less powerful and Flex-MPI requires a greater number of them to increase the performance of the application to the point where the applications complete their execution within the required time interval. The effect of this election can be seen in Figure 13, which shows the normalized operational cost of each test case relative to static scheduling. We can observe that the operational cost when imposing the cost constraint is always smaller than that obtained for the efficiency constraint.

## 6. Conclusions and Future work

This paper describes the design and implementation of Flex-MPI, a library that confers malleability to iterative SPMD MPI applications with minimal user effort. Flex-MPI enables MPI applications to spawn and remove dynamic processes at runtime to optimize their performance, parallel efficiency, and cost-efficiency. Some of the main technical contributions captured by this library are: automatic monitoring via hardware performance counters, prediction of the future performance of (regular and irregular) parallel applications, and performance-aware dynamic reconfiguration guided by user-defined cost and efficiency constraints. Our techniques work equally well for applications operating on dense and sparse data structures, as well as for heterogeneous and homogeneous, shared or dedicated, architectures.

We present an extensive validation of the computational prediction model and a detailed performance analysis of a set of benchmarks that are representative for the class of applications we target. Our results show that the computational prediction model effectively estimates the performance of parallel applications prior to a reconfiguring action. This allows the performance-aware

reconfiguring policy to provision a number and type of processors that satisfies the user-defined performance criteria. The performance analysis of a set of well-known SPMD MPI programs shows that Flex-MPI can significantly improve the performance of parallel applications. This increases the efficiency of resource utilization and the cost-efficiency of program executions.

There are several interesting directions for future work that are allowed by the Flex-MPI framework and which can considerably widen the scope of our approach. Some of them aim to extend the capabilities of our approach to cover a wider set of parallel applications. For instance, by extending the current model to support applications with asynchronous communications, which may overlap communication and computation. This is a realistic extension given that the Flex-MPI implementation allows control over both the application and the MPI library. The second is an extension of the data redistribution component to support parallel applications with three-dimensional domain decomposition or cyclic data distribution.

Another research direction is the design and implementation of optimization techniques for adaptability of Flex-MPI applications to Cloud environments. This involves extending the capabilities of monitoring and dynamic process management components of Flex-MPI to take into account the overhead of virtualization and the variable performance of the interconnection network between instances, and evaluate their impact on the performance of HPC applications.

Finally, we are currently working on the design and implementation of a centralized dynamic load balancing algorithm that can consider multiple Flex-MPI applications together to optimize the overall system performance. Additionally, we are considering more performance metrics than just FLOPS, and we are currently evaluating the introduction of more hardware counters such as cache misses in multicore processors and power consumption. These new metrics will improve both the load balancing algorithm as well as the application performance monitoring at execution time.

## References


[1] D. Feitelson, L. Rudolph, Toward convergence in job schedulers for parallel supercomputers, in: D. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, volume 1162 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1996, pp. 1–26.

[2] G. Utrera, " Virtual Malleability" Applied to MPI Jobs to Improve Their Execution in a Multiprogrammed Environment, Universitat Politècnica de Catalunya, 2010.

[3] J. Hungershofer, On the combined scheduling of malleable and rigid jobs, in: Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on, IEEE, 2004, pp. 206–213.

[4] L. V. Kalé, S. Kumar, J. DeSouza, A malleable-job system for timeshared parallel machines, in: Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on, IEEE, 2002, pp. 230–230.

[5] C. Klein, C. Pérez, An RMS for non-predictably evolving applications, in: Cluster Computing (CLUSTER), 2011 IEEE International Conference on, IEEE, 2011, pp. 326–334.

[6] O. Sonmez, B. Grundeken, H. Mohamed, A. Iosup, D. Epema, Scheduling strategies for cycle scavenging in multicluster grid systems, in: Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on, IEEE, 2009, pp. 12–19.

[7] W. Gropp, MPICH2: A new start for MPI implementations, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2002, pp. 7–7.

[8] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, et al., The HiPEAC vision, Report, European Network of Excellence on High Performance and Embedded Architecture and Compilation 12 (2010).

[9] M. Duranton, D. Black-Schaffer, K. De Bosschere, J. Maebe, The HIPEAC vision for advanced computing in horizon 2020 (2013).

[10] M. Ben Belgacem, B. Chopard, A hybrid hpc/cloud distributed infrastructure: Coupling ec2 cloud resources with hpc clusters to run large tightly coupled multiscale applications, Future Generation Computer Systems (2014).

[11] S. Benedict, Performance issues and performance analysis tools for hpc cloud applications: a survey, Computing 95 (2013) 89–108.

[12] R. Lepère, D. Trystram, G. J. Woeginger, Approximation algorithms for scheduling malleable tasks under precedence constraints, International Journal of Foundations of Computer Science 13 (2002) 613–627.

[13] E. Blayo, L. Debreu, G. Mounie, D. Trystram, Dynamic load balancing for ocean circulation model with adaptive meshing, in: Euro-Par99 Parallel Processing, Springer, 1999, pp. 303–312.

[14] S. S. Vadhiyar, J. J. Dongarra, SRS: A framework for developing malleable and migratable parallel applications for distributed systems, Parallel Processing Letters 13 (2003) 291–312.

[15] K. R. Mayes, M. Luján, G. D. Riley, J. Chin, P. V. Coveney, J. R. Gurd, Towards performance control on the Grid, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363 (2005) 1793–1805.

[16] L. A. Rao, J. Weissman, MPI-Based Adaptive Parallel Grid Services, 2003.

[17] C. Huang, O. Lawlor, L. V. Kale, Adaptive MPI, in: Languages and Compilers for Parallel Computing, Springer, 2004, pp. 306–322.

[18] K. El Maghraoui, B. K. Szymanski, C. Varela, An architecture for reconfigurable iterative MPI applications in dynamic environments, in: Parallel Processing and Applied Mathematics, Springer, 2006, pp. 258–271.

[19] K. El Maghraoui, T. J. Desell, B. K. Szymanski, C. A. Varela, Dynamic malleability in iterative MPI applications, in: 7th Int. Symposium on Cluster Computing and the Grid, 2008, pp. 591–598.

[20] K. El Maghraoui, T. J. Desell, B. K. Szymanski, C. A. Varela, The Internet Operating System: Middleware for adaptive distributed computing, International Journal of High Performance Computing Applications 20 (2006) 467–480.

[21] G. Utrera, J. Corbalan, J. Labarta, Implementing malleability on MPI jobs, in: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, 2004, pp. 215–224.

[22] C. McCann, J. Zahorjan, Processor allocation policies for message-passing parallel computers, volume 22, ACM, 1994.

[23] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, P. O. Navaux, Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI, in: Distributed Computing and Networking, Springer, 2010, pp. 242–257.

[24] M. P. I. Forum, MPI: a message passing interface standard, volume 8, 1994, pp. 165–414.

[25] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2004, pp. 97–104.

[26] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, P. O. Navaux, et al., Supporting MPI malleable applications upon the OAR resource manager, in: Colibri: Colloque d'Informatique: Brésil/INRIA, Coopérations, Avancées et Défis, 2009.

[27] M. C. Cera, Providing adaptability to MPI applications on current parallel architectures, Ph.D. thesis, Universidade Federal do Rio Grande do Sul, 2012.

[28] C. Leopold, M. Süß, Observations on MPI-2 support for hybrid master/slave applications in dynamic and heterogeneous environments, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2006, pp. 285–292.

[29] R. Sudarsan, C. J. Ribbens, ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment, in: Parallel Processing, 2007. ICPP 2007. International Conference on, IEEE, 2007, pp. 44–44.

[30] R. Sudarsan, C. J. Ribbens, Design and performance of a scheduling framework for resizable parallel applications, Parallel Computing 36 (2010) 48–64.

[31] MPI Forum, Message Passing Interface (MPI) Forum Home Page, 2009. URL: `http://www.mpi-forum.org/`.

[32] P. Mucci, S. Browne, C. Deane, G. Ho, PAPI: A portable interface to hardware performance counters, in: Proceedings of the Department of Defense HPCMP Users Group Conference, 1999, pp. 7–10.

[33] N. Radcliffe, L. Watson, M. Sosonkina, A comparison of alternatives for communicating with spawned processes, in: Proceedings of the 49th Annual Southeast Regional Conference, ACM, 2011, pp. 132–137.

[34] G. Martin, M.-C. Marinescu, D. E. Singh, J. Carretero, FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems, in: International European Conference on Parallel and Distributed Computing, EuroPar, 2013.

[35] J. Dongarra, Overview of PVM and MPI, http://www.netlib.org/utk/people/JackDongarra/pdf/pvm-mpi.pdf (1998).

[36] M. Beltran, A. Guzman, J. Bosque, Dealing with heterogeneity in load balancing algorithms, in: ISPDC'06, IEEE, 2006, pp. 123–132.

[37] J. Martínez, F. Almeida, E. Garzón, A. Acosta, V. Blanco, Adaptive load balancing of iterative computation on heterogeneous nondedicated systems, The Journal of Supercomputing 58 (2011) 385–393.

[38] R. Buyya, et al., High Performance Cluster Computing: Architectures and Systems (Volume 1), Prentice Hall, Upper SaddleRiver, NJ, USA 1 (1999) 999.

[39] A. Alexandrov, M. F. Ionescu, K. E. Schauser, C. Scheiman, LogGP: incorporating long messages into the LogP modelone step closer towards a realistic model for parallel computation, in: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, ACM, 1995, pp. 95–105.

[40] T. Kielmann, H. E. Bal, K. Verstoep, Fast measurement of LogP parameters for message passing platforms, in: Parallel and Distributed Processing, Springer, 2000, pp. 1176–1183.

[41] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. Von Eicken, LogP: Towards a realistic model of parallel computation, volume 28, ACM, 1993.

[42] R. W. Hockney, The communication challenge for MPP: Intel Paragon and Meiko CS-2, Parallel computing 20 (1994) 389–398.

[43] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, International Journal of High Performance Computing Applications 19 (2005) 49–66.

[44] R. Thakur, W. D. Gropp, Improving the performance of collective operations in MPICH, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2003, pp. 257–267.

[45] G. B. Dantzig, Linear programming and extensions, Princeton university press, 1998.

[46] G. Staples, TORQUE resource manager, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, 2006, p. 8.

[47] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, D. H. Epema, Performance analysis of cloud computing services for many-tasks scientific computing, Parallel and Distributed Systems, IEEE Transactions on 22 (2011) 931–945.

[48] A. Petitet, R. Whaley, J. Dongarra, A. Cleary, HPL–a portable implementation of the high–performance Linpack benchmark for distributed–memory computers, http://www.netlib.org/benchmark/hpl (2005).

[49] G. Martín, M.-C. Marinescu, D. E. Singh, J. Carretero, Leveraging social networks for understanding the evolution of epidemics, BMC Syst Biol 5 (2011).

[50] M. U. Guide, The mathworks, Inc., Natick, MA 5 (1998).

[51] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (2011) 1:1–1:25.