UNIVERSIDAD CARLOS III DE MADRID

Optimization Techniques for Adaptability in MPI Applications

Author:
Gonzalo Martín Cruz

Advisors:
David Expósito Singh
Maria-Cristina Marinescu

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

Leganés, 2015

TESIS DOCTORAL

Optimization Techniques for Adaptability in MPI Applications

Autor: Gonzalo Martín Cruz
Director: David Expósito Singh
Co-Director: Maria-Cristina Marinescu

Firma del tribunal calificador:

Presidente:	Firma:
Vocal:	Firma:
Secretario:	Firma:

Calificación:

Leganés, a __ de _____ de _____.

Agradecimientos

Una vez finalizada esta tesis es momento de hacer una recapitulación sobre estos años de trabajo y mostrar mi agradecimiento a todas aquellas personas junto a las que he tenido la oportunidad de crecer tanto profesional como emolcionalmente durante este tiempo. A todos vosotros que me habéis ayudado a lograr este objetivo va dedicada esta tesis.

A Romina y a mi familia, a mis padres Alicia y Jose Antonio y a mi hermano Alexis. Vosotros habéis sido la parte fundamental de la culminación de este trabajo. A Romina, por hacer que cada día merezca la pena y me sienta un completo afortunado por tenerte a mi lado. Nadie más que tú sabe el esfuerzo que me ha costado llegar hasta aquí ni me ha tenido que aguantar tanto en mis malos momentos. Gracias por tu apoyo incondicional cuando más falta me hacía e incluso en aquellas ocasiones en las que no lo merecía, y por tu infinita comprensión a todas mis decisiones. Me siento en deuda contigo; el futuro es nuestro. A mis padres, que desde pequeño me habéis inculcado la idea de que el esfuerzo y el trabajo tienen su recompensa y que debía aprovechar la oportunidad que me estábais ofreciendo, aquella que vosotros no tuvísteis cuando siendo muy jóvenes tuvísteis que empezar a trabajar y dejar atrás vuestra ciudad natal, familia y amigos en busca de un futuro mejor que entregar a vuestros hijos. Sois para mí el mejor ejemplo de lo que quiero ser en la vida y el espejo en el que mirarme cada día. A mi hermano, gracias por ser cómplice en los buenos y malos momentos. Ya sabes dónde tienes a tu hermano para lo que haga falta. No tengo dudas de que llegarás lejos.

También me gustaría dedicar esta tesis al resto de mi familia, tanto a los que están como a los que tristemente se fueron, en especial a mis abuelos, tíos y primos: Gonzalo, Josefa, Sacramento, Enrique, Tomás y Exaltación. Todos vosotros sois importantes en mi vida y un modelo de esfuerzo y sacrificio que tengo presente permanentemente. Mención especial a Huguito, el pequeñajo al que más quiero y que es capaz de llenar de alegría y vitalidad a cada instante.

A mis directores de tesis David y Maria-Cristina, y a Jesús Carretero, por confiar en mí y ofrecerme la oportunidad de realizar mi tesis en ARCOS. Gracias a vosotros me siento orgulloso por las interminables horas de trabajo en despachos y laboratorios de esta universidad que durante 8 años ha sido mi segunda casa. Gracias por vuestro tiempo, esfuerzo y dedicación a nuestra tesis. Me siento muy afortunado de haber podido estar bajo vuestra dirección. A Jesús en especial le agradezco el soporte económico y el esfuerzo por mantenerlo a pesar de las dificultades. También quiero dedicar esta tesis al resto de componentes de ARCOS, de los que he aprendido y me

han ayudado a su finalización desde el punto de vista técnico, académico y humano.

A Pablo, mi compañero de despacho y amigo. Nunca voy a olvidar tantas horas que hemos compartido juntos tanto dentro como fuera de la universidad, al igual que todo lo que he aprendido a tu lado. Hacías que el día a día fuese mucho más divertido e interesante. Sigue demostrando lo crack que eres como persona e investigador. A Cana, nuestro primer compañero de despacho, tus consejos durante el tiempo que compartimos me ayudaron a ver las cosas un poco más claras cuando no todo parecía tan bueno. Gracias por guiarnos con tus charlotes.

A mis amigos de toda la vida: Iván, Víctor, Álvaro y Andrés, algunos de los cuales he mantenido desde la guardería y que después de tantos años aún seguimos viéndonos. A toda la gente que he conocido a lo largo de todo este tiempo en la universidad, en especial a Quique, Julio, Bea, Almudena y Elena, y a todos aquellos que he ido conociendo en estos últimos meses, con los que comparto y aprendo cada día, en especial a: Edu, Alicia, Carol, Diego, Jorge e Ismael.

Esta tesis va dedicada a todos vosotros porque es un poquito tan vuestra como mía.

A todos, muchas gracias.

Abstract

The first version of MPI (Message Passing Interface) was released in 1994. At that time, scientific applications for HPC (High Performance Computing) were characterized by a static execution environment. These applications usually had regular computation and communication patterns, operated on dense data structures accessed with good data locality, and ran on homogeneous computing platforms. For these reasons, MPI has become the *de facto* standard for developing scientific parallel applications for HPC during the last decades.

In recent years scientific applications have evolved in order to cope with several challenges posed by different fields of engineering, economics and medicine among others. These challenges include large amounts of data stored in irregular and sparse data structures with poor data locality to be processed in parallel (*big data*), algorithms with irregular computation and communication patterns, and heterogeneous computing platforms (grid, cloud and heterogeneous cluster).

On the other hand, over the last years MPI has introduced relevant improvements and new features in order to meet the requirements of dynamic execution environments. Some of them include asynchronous non-blocking communications, collective I/O routines and the dynamic process management interface introduced in MPI 2.0. The dynamic process management interface allows the application to spawn new processes at runtime and enable communication with them. However, this feature has some technical limitations that make the implementation of malleable MPI applications still a challenge.

This thesis proposes FLEX-MPI, a runtime system that extends the functionalities of the MPI standard library and features optimization techniques for adaptability of MPI applications to dynamic execution environments. These techniques can significantly improve the performance and scalability of scientific applications and the overall efficiency of the HPC system on which they run. Specifically, FLEX-MPI focuses on dynamic load balancing and performance-aware malleability for parallel applications. The main goal of the design and implementation of the adaptability techniques is to efficiently execute MPI applications on a wide range of HPC platforms ranging from small to large-scale systems.

Dynamic load balancing allows FLEX-MPI to adapt the workload assignments at runtime to the performance of the computing elements that execute the parallel application. On the other hand, performance-aware malleability leverages the dynamic process management interface of MPI to change the number of processes

of the application at runtime. This feature allows to improve the performance of applications that exhibit irregular computation patterns and execute in computing systems with dynamic availability of resources. One of the main features of these techniques is that they do not require user intervention nor prior knowledge of the underlying hardware.

We have validated and evaluated the performance of the adaptability techniques with three parallel MPI benchmarks and different execution environments with homogeneous and heterogeneous cluster configurations. The results show that FLEX-MPI significantly improves the performance of applications when running with the support of dynamic load balancing and malleability, along with a substantial enhancement of their scalability and an improvement of the overall system efficiency.

Resumen

La primera versión de MPI (Message Passing Interface) fue publicada en 1994, cuando la base común de las aplicaciones científicas para HPC (High Performance Computing) se caracterizaba por un entorno de ejecución estático. Dichas aplicaciones presentaban generalmente patrones regulares de cómputo y comunicaciones, accesos a estructuras de datos densas con alta localidad, y ejecución sobre plataformas de computación homogéneas. Esto ha hecho que MPI haya sido la alternativa más adecuada para la implementación de aplicaciones científicas para HPC durante más de 20 años.

Sin embargo, en los últimos años las aplicaciones científicas han evolucionado para adaptarse a diferentes retos propuestos por diferentes campos de la ingeniería, la economía o la medicina entre otros. Estos nuevos retos destacan por características como grandes cantidades de datos almacenados en estructuras de datos irregulares con baja localidad para el análisis en paralelo (*big data*), algoritmos con patrones irregulares de cómputo y comunicaciones, e infraestructuras de computación heterogéneas (cluster heterogéneos, grid y cloud).

Por otra parte, MPI ha evolucionado significativamente en cada una de sus sucesivas versiones, siendo algunas de las mejoras más destacables presentadas hasta la reciente versión 3.0 las operaciones de comuniación asíncronas no bloqueantes, rutinas de E/S colectiva, y la interfaz de procesos dinámicos presentada en MPI 2.0. Esta última proporciona un procedimiento para la creación de procesos en tiempo de ejecución de la aplicación. Sin embargo, la implementación de la interfaz de procesos dinámicos por parte de las diferentes distribuciones de MPI aún presenta numerosas limitaciones que condicionan el desarrollo de aplicaciones maleables en MPI.

Esta tesis propone FLEX-MPI, un sistema que extiende las funcionalidades de la librería MPI y proporciona técnicas de optimización para la adaptación de aplicaciones MPI a entornos de ejecución dinámicos. Las técnicas integradas en FLEX-MPI permiten mejorar el rendimiento y escalabilidad de las aplicaciones científicas y la eficiencia de las plataformas sobre las que se ejecutan. Entre estas técnicas destacan el balanceo de carga dinámico y maleabilidad para aplicaciones MPI. El diseño e implementación de estas técnicas está dirigido a plataformas de cómputo HPC de pequeña a gran escala.

El balanceo de carga dinámico permite a las aplicaciones adaptar de forma eficiente su carga de trabajo a las características y rendimiento de los elementos de procesamiento sobre los que se ejecutan. Por otro lado, la técnica de maleabilidad

aprovecha la interfaz de procesos dinámicos de MPI para modificar el número de procesos de la aplicación en tiempo de ejecución, una funcionalidad que permite mejorar el rendimiento de aplicaciones con patrones irregulares o que se ejecutan sobre plataformas de cómputo con disponibilidad dinámica de recursos. Una de las principales características de estas técnicas es que no requiren intervención del usuario ni conocimiento previo de la arquitectura sobre la que se ejecuta la aplicación.

Hemos llevado a cabo un proceso de validación y evaluación de rendimiento de las técnicas de adaptabilidad con tres diferentes aplicaciones basadas en MPI, bajo diferentes escenarios de computación homogéneos y heterogéneos. Los resultados demuestran que FLEX-MPI permite obtener un significativo incremento del rendimiento de las aplicaciones, unido a una mejora sustancial de la escalabilidad y un aumento de la eficiencia global del sistema.

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Setting the context . . . 2
1.3 Setting the goal . . . 3
1.4 Objectives . . . 4
1.5 Advantages of the approach . . . 5
1.6 Structure and contents . . . 5

2 State of the art **7**

2.1 Introduction . . . 7
2.2 High performance computing . . . 7
2.2.1 Cluster computing . . . 9
2.2.2 Grid computing . . . 11
2.2.3 Cloud computing . . . 13
2.2.4 Performance analysis of parallel applications . . . 15
2.3 Classification of computer architectures . . . 16
2.4 Parallel programming models . . . 18
2.4.1 Shared memory programming . . . 18
2.4.1.1 POSIX threads . . . 19
2.4.1.2 OpenMP . . . 19
2.4.1.3 High Performance Fortran . . . 20
2.4.2 Distributed memory programming . . . 20
2.4.2.1 Message Passing Interface . . . 21
2.4.2.2 Parallel Virtual Machine . . . 22
2.5 Load balancing in parallel applications . . . 22
2.6 Dynamic parallel applications . . . 24
2.6.1 Taxonomy of parallel applications . . . 24
2.6.2 Supporting malleability in MPI applications . . . 25
2.6.2.1 Offline reconfiguration . . . 26

2.6.2.2 Dynamic reconfiguration . . . 27
2.7 Summary and conclusions . . . 29

3 FLEX-MPI: Adaptability Techniques for MPI Applications 31
3.1 Introduction . . . 31
3.2 Runtime system overview . . . 31
3.2.1 Process and data management layer . . . 33
3.2.2 Malleability logic layer . . . 34
3.3 FLEX-MPI execution model . . . 34
3.4 FLEX-MPI programming model . . . 36
3.5 Summary . . . 38

4 Dynamic Process and Data Management in MPI Applications 41
4.1 Introduction . . . 41
4.2 Monitoring . . . 42
4.2.1 Computation monitoring . . . 43
4.2.2 Communication profiling . . . 45
4.3 Dynamic process management . . . 46
4.3.1 MPI communicators . . . 46
4.3.2 Spawn . . . 50
4.3.3 Remove . . . 53
4.3.4 Malleable MPI programs using the FLEX-MPI low-level API 55
4.4 Process scheduler . . . 56
4.4.1 Spawn . . . 57
4.4.2 Remove . . . 58
4.5 Data redistribution . . . 58
4.5.1 Storage formats . . . 59
4.5.2 Domain decomposition . . . 60
4.5.3 Data redistribution functionality . . . 61
4.6 Summary . . . 64

5 Dynamic Load Balancing and Performance-aware Malleability in MPI 67
5.1 Introduction . . . 67
5.2 Dynamic load balancing . . . 69
5.2.1 Dynamic load balancing algorithm . . . 72
5.3 Computational prediction model . . . 77
5.3.1 Modeling the computation cost . . . 78
5.3.2 Modeling the parallel communication cost . . . 79

5.3.3 Modeling process creation and termination costs 82
5.3.4 Modeling data redistribution costs 83
5.4 Malleability policies 85
5.4.1 Strict malleability policy 87
5.4.1.1 SMP algorithm 88
5.4.2 High performance malleability policy 90
5.4.2.1 HPMP algorithm 91
5.4.3 Adaptive malleability policy 93
5.4.3.1 Integrating linear programming methods to find the optimal processor configuration 94
5.4.3.2 AMP algorithm 96
5.4.4 Malleable MPI programs using the FLEX-MPI high-level API 100
5.5 Summary 102

6 Experimental Results 103
6.1 Introduction 103
6.1.1 Benchmarks 104
6.1.1.1 Jacobi 104
6.1.1.2 Conjugate Gradient 105
6.1.1.3 EpiGraph 106
6.1.2 Evaluation platform 108
6.1.2.1 Cost model 109
6.2 Performance evaluation of the dynamic load balancing algorithm . . 111
6.2.1 Homogeneous dedicated system 111
6.2.2 Heterogeneous dedicated system 115
6.2.3 Non-dedicated system 119
6.3 Computational prediction model validation 121
6.3.1 Parameter values 121
6.3.1.1 Creation and termination of MPI processes 121
6.3.1.2 Network parameters 123
6.3.2 Model validation 124
6.4 Performance evaluation of malleable MPI parallel applications 129
6.4.1 Strict malleability policy 129
6.4.2 High performance malleability policy 131
6.4.3 Adaptive malleability policy 134
6.4.3.1 AMP test cases 135
6.4.3.2 Performance analysis of AMP 136
6.4.3.3 Performance evaluation of AMP 139

6.5 Overhead analysis 142
6.6 Summary 144

7 Conclusions **147**
7.1 Main contributions 147
7.2 Publications related to this thesis 149
7.3 Future work 150

Bibliography **153**

List of Figures

2.1 Shared memory and distributed memory system architectures. 8
2.2 Traditional cluster computing architecture. 10
2.3 Typical OpenMP parallel program execution. 19

3.1 The FLEX-MPI runtime system architecture. 32
3.2 Workflow diagram of a malleable FLEX-MPI application. 35
3.3 Comparison of the legacy code (left) and the instrumented FLEX-MPI code (right) of an iterative MPI application. 37

4.1 Process and data management layer inside the FLEX-MPI library. . 41
4.2 Parameters of FLEX-MPI's `XMPI_Monitor_iter_begin`. 43
4.3 Parameters of FLEX-MPI's `XMPI_Monitor_iter_end`. 43
4.4 Performance analysis of the number of $FLOP$ in Jacobi for varying matrix sizes and 8 processes (a), and a varying number of processes and a matrix of 2,000 rows (b). 44
4.5 Illustration of the behavior of PMPI in FLEX-MPI. 45
4.6 MPI communication between processes within a local group performed using an intracommunicator (A) and communication between local and remote processes using an intercommunicator (B). 47
4.7 Parameters of MPI's `MPI_Comm_spawn`. 48
4.8 Example of communication between dynamic processes using the *many communicator* method 50
4.9 Parameters of FLEX-MPI's `XMPI_Spawn`. 51
4.10 Actions of the dynamic process management functionality at process creation. 53
4.11 Parameters of FLEX-MPI's `XMPI_Remove`. 54
4.12 Actions of the dynamic process management functionality at process removal. 55
4.13 Malleable program instrumented with FLEX-MPI low-level interfaces. 56
4.14 Example of process scheduling at process creation with the *balance all* policy. 57

4.15 Example of process scheduling at process creation with the *balance occupied* policy. . . . 57
4.16 Example of process scheduling at process removal with the *balance all* policy. . . . 58
4.17 Example of process scheduling at process removal with the *balance occupied* policy. . . . 58
4.18 Sparse matrix A. . . . 59
4.19 Representation of sparse matrix A in coordinate scheme. . . . 59
4.20 Representation of sparse matrix A in CSR format. . . . 60
4.21 One-dimensional data decomposition with block partitioning of dense (a) and sparse (b) data structures. . . . 61
4.22 Parameters of FLEX-MPI's `XMPI_Get_wsize`. . . . 61
4.23 Parameters of FLEX-MPI's `XMPI_Register_vector`. . . . 62
4.24 Parameters of FLEX-MPI's `XMPI_Register_dense`. . . . 62
4.25 Parameters of FLEX-MPI's `XMPI_Register_sparse`. . . . 63
4.26 Parameters of FLEX-MPI's `XMPI_Redistribute_data`. . . . 63

5.1 Malleability logic layer inside the FLEX-MPI library. . . . 67
5.2 Dynamic load balance of a parallel application running on a heterogeneous dedicated system. . . . 71
5.3 Workload distribution as result of a load balancing operation. . . . 71
5.4 Parameters of FLEX-MPI's high-level `XMPI_Eval_lbalance`. . . . 72
5.5 Comparison of FLEX-MPI's load balancing policies for (a) a data structure depending on (b) the *nnz* and (c) the *weight*. . . . 73
5.6 Dynamic load balance of a parallel application running on a heterogeneous non-dedicated system from sampling interval n (a) to $n+1$ (b). . . . 74
5.7 Load balancing operations performed by FLEX-MPI after process spawning (b) and process removing (c) actions. . . . 77
5.8 Parameters of FLEX-MPI's high-level `XMPI_Monitor_si_init`. . . . 86
5.9 Parameters of FLEX-MPI's high-level `XMPI_Eval_reconfig`. . . . 87
5.10 Parameters of FLEX-MPI's high-level `XMPI_Get_data`. . . . 87
5.11 Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the SMP policy. . . . 90
5.12 Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the HPMP policy. . . . 93
5.13 Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the AMP policy. . . . 99

5.14 MPI malleable program instrumented with FLEX-MPI high-level interfaces. 101

6.1 Performance analysis of the number of $FLOP$ in EpiGraph for varying matrix sizes and 8 processes. 108
6.2 Performance evaluation of Jacobi, CG and EpiGraph on a homogeneous dedicated system. 112
6.3 Load balancing of Jacobi, CG and EpiGraph on a homogeneous dedicated system. 113
6.4 Performance evaluation of Jacobi, CG and EpiGraph on a heterogeneous dedicated system. 116
6.5 Load balancing of Jacobi, CG and EpiGraph on a heterogeneous dedicated system. 117
6.6 Performance analysis of Jacobi (a) and EpiGraph (b) on a heterogeneous dedicated system. 118
6.7 Performance analysis Jacobi on a heterogeneous non-dedicated system for different values of k: (a) $k = 1$, (b) $k = 3$, and (c) $k = 5$. 120
6.8 Measured times for the creation and termination of dynamic MPI processes on 20 computing nodes of class C1. 122
6.9 Comparison between predicted and measured times for MPI Send and Recv (`MPI_Send`/`MPI_Recv`). 125
6.10 Comparison between predicted and measured times for MPI Broadcast (`MPI_Bcast`). 125
6.11 Comparison between predicted and measured times for MPI Gather (`MPI_Gather`). 126
6.12 Comparison between predicted and measured times for MPI All reduce (`MPI_Allreduce`). 126
6.13 Comparison between average predicted and real times for different dynamic reconfiguring actions carried out for Jacobi benchmark running on (a) 8, (b) 16, (c) 32, and (d) 64 processes. 127
6.14 Number of processes and type of processors scheduled for Jacobi and total number processing elements available in the system. 129
6.15 Performance evaluation of EpiGraph that illustrates the number of processes (left Y axis) and execution time (right Y axis) per sampling interval when using static scheduling without FLEX-MPI support. . 130
6.16 Performance evaluation of EpiGraph that illustrates the number of processes (left Y axis) and execution time (right Y axis) per sampling interval when using predefined dynamic scheduling (b). 131
6.17 Performance analysis of Jacobi (a) and Conjugated Gradient (b) that shows speedup (left Y axis) and efficiency (right Y axis). 132
6.18 Number of processes allocated by HPMP, number of PE available (left Y axis) and execution time per iteration (right Y axis) for Jacobi. . . 133

6.19 Number of processes allocated by HPMP, number of PE available (left Y axis) and execution time per iteration (right Y axis) for Conjugate Gradient. 134
6.20 Number of processes and type of processors scheduled by the AMP policy for the execution of J.B.8 under the efficiency (a) and cost (b) constraints. 137
6.21 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. 138
6.22 Performance evaluation of the benchmarks with the support of AMP for satisfying the performance objective. 140
6.23 Number of processes scheduled by AMP for satisfying the performance objective for the benchmarks. 141
6.24 Cost analysis of the benchmarks with AMP support for satisfying the performance objective. 141
6.25 Performance overhead for legacy MPI with static scheduling and FLEX-MPI with static static scheduling and dynamic reconfiguration. 143

List of Tables

2.1 Comparison of different standard parallel programming models. . . . 18

4.1 Flex-MPI low-level interfaces. 42

5.1 FLEX-MPI high-level interfaces. 68
5.2 Algorithms and their cost for MPI routines in MPICH. 81

6.1 Configuration of the heterogeneous cluster *ARCOS* with number of compute nodes and cores for each node class. 108
6.2 Problem classes for the benchmarks evaluated. 109
6.3 Performance evaluation and cost model of the Amazon EC2 platform. 110
6.4 Performance evaluation and cost model of the cluster. 110
6.5 Number of processes (Np) and their mapping to the available node classes (number of nodes and PEs per node) in the heterogeneous configuration. 115
6.6 Average process creation and termination overheads for Jacobi, CG, and EpiGraph in the cluster *ARCOS*. 123
6.7 Hockney model parameters measured on the ARCOS cluster. 123
6.8 Average relative error and relative standard deviation for CPM estimations. 127
6.9 Number of processes initially scheduled (Np) and their mapping to the available class nodes for each test case. 135

Chapter 1

Introduction

This chapter presents the four-step process followed to identify the motivation, scope, and the main goal of this work. It then presents the concrete objectives and the advantages of the approach this thesis is proposing. This chapter concludes with the roadmap of this document.

1.1 Motivation

MPI (Message Passing Interface) [Mes94] is arguably the *industry standard* for developing message passing programs on distributed memory systems. MPI is the specification of a library of functions for interfaces of message-passing routines that include point-to-point and collective communication, as well as synchronization and parallel I/O operations. MPI has become the standard tool for scientific parallel applications running on HPC systems.

MPI was initially designed to run parallel applications on homogeneous, dedicated, static environments such as computing clusters. The execution of an application based on the typical MPI execution model uses a fixed number of processes during the entire execution of the parallel program. However, selecting the most appropriate number of processes for the application may become a challenge. This number depends on the architecture and the program implementation. The most common approaches here are either using a number of processes proportional to the number of processors available or using a number of processes proportional to the degree of parallelism in the application.

The challenge becomes even more complicated when the MPI program executes on a dynamic execution environment, such as one or a combination of the following:

- Heterogeneous (performance-wise) architectures.
- Non-dedicated systems.
- Parallel programs with irregular computation and communication patterns.

Current high-performance computing systems are optimized for applications that operate on dense data structures and regular computation and communication patterns. Dynamic execution environments introduce an additional degree of complexity to the design and implementation of MPI programs and require optimized approaches. Furthermore, the performance of these environments cannot be predicted prior to execution. Dynamic approaches are required in order to change the size of the program, its workload distribution, and the resources allocation at runtime. These approaches have to be independent of the architecture such that the MPI program can execute efficiently in a wide range of environments.

This thesis starts from the premise that the *static* execution model of MPI applications is not the best approach for dynamic execution environments and leads to suboptimal performance, efficiency, and system throughput. The challenge is to propose a dynamic execution model that addresses these issues through optimization techniques that allow the MPI application to adapt to the execution environment (resources available, their performance parameters, as well as the application performance) within user-defined performance requirements.

Adaptability is a feature that enables the application to adjust its workload and granularity to the specific characteristics and availability of the execution environment at runtime. This allows the programmer to focus on the algorithm implementation rather than the inherent platform constraints. The aim of adaptability is to enable a program implementation to execute efficiently regardless of the underlying hardware.

First step: Decide on the target applications: MPI parallel applications which may exhibit regular and irregular computation and communication patterns.

1.2 Setting the context

Most MPI applications executing on high-performance computing clusters are implemented on top of distributed memory systems. Data-driven applications where there is a large synchronization component—such as event-driven simulations—are also better suited to distributed, rather than shared, memory systems. It is in any case difficult to assume a shared memory system—even if distributed in its implementation—if an application wants to take advantage of all the resources available (which may be heterogeneous), whether they are part of a cluster or a cloud configuration. Finally, although using distributed memory does not hide the mechanism of communication, the developer does not have to deal with data protection mechanisms such as locks, which may be tricky and can easily lead to performance penalties.

Second step: Decide on the system requirements: distributed memory systems involving heterogeneous and non-dedicated resources.

1.3 Setting the goal

The problem this thesis is focused to solve is to make MPI applications adaptable when running on HPC architectures. This work focuses on two approaches to improve adaptability: dynamic load balancing and malleability. The proposed system—called FLEX-MPI—includes malleability of three different kinds: Strict malleability policy (SMP), High performance malleability policy (HPMP), and Adaptive malleability policy (AMP). This stands in contrast to the static execution model of standard MPI applications where the application executes on a fixed number of processes. FLEX-MPI monitors the performance metrics of the application, as well as the resources allocated to the program to decide if the application performance satisfies the requirements—or if it needs to be reconfigured.

Dynamic load balancing refers to the practice of adapting workload assignments at runtime to the performance of the computing elements that execute the parallel application. Traditionally, load balancing is achieved by distributing immutable work assignments to the processes at the program start. This requires prior knowledge of the execution platform and the program workload to be uniform during execution. However, in practice this is often unfeasible because the user does not have access to the characteristics of the system and certain classes of problems have unpredictable workloads. Besides that, the challenge when designing dynamic load balancing algorithms is how to capture the application performance at runtime and how to efficiently redistribute the workload between processes.

The challenge when designing dynamic reconfiguration techniques for providing malleability to 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 of 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 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.

Third step: Define the main goal: an adaptable execution model for MPI applications.

Before starting to explain the *how* and *how much* of the techniques we implement to offer malleability, we must first define *what*. What are the performance requirements that the FLEX-MPI implementation must optimize for; that is, how should an optimized application behave like? What should it improve on? The reply

is in the eye of the beholder—the target user, in this case. We may crave speed, optimized resources, energy reduction, or lower costs. These are in general at odds with each other, but generally applications put a lot more focus on one criteria than the rest, or they may do so depending on the context. We need to be able to define what our expectations are to optimize the chances of success.

To perform an effective reconfiguration, FLEX-MPI takes into account the user-given performance constraint, which can be either the parallel efficiency or the operational cost of executing the program. 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 economic expense per CPU time unit) while satisfying the performance constraint. 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.

Fourth step: Define the context for adaptability: application's runtime performance, resources availability and performance, and user-given performance criteria.

1.4 Objectives

The major goal of this thesis is **to propose a set of optimization techniques for adaptability in MPI applications**. To be more specific, this work proposes the following approaches:

- **Dynamic load balancing**. One of the objectives of this thesis is the design and implementation of a dynamic load balancing algorithm for parallel applications which may exhibit regular or irregular execution patterns, running either on homogeneous or heterogeneous (performance-wise) platforms which can be dedicated or non-dedicated. This adaptability technique is aimed to redistribute the application workload by using performance metrics collected at runtime in order to improve the overall performance of the program.

- **Computational prediction model**. This thesis targets the design and implementation of a computational prediction model that will guide FLEX-MPI during the reconfiguration process by computing the performance of the application prior to reconfiguring actions. The computational prediction model must take into account the current application performance, as well as the performance of the currently allocated and available resources.

- **Performance-aware dynamic reconfiguration**. The main approach for adaptability pursued by this thesis is the design and implementation of high-level performance-aware malleability policies. Rather than use the resource availability as the unique criteria to trigger reconfiguring actions, these policies target to adapt the number of processes and type of processors of the application to the current application performance, resources availability, and a set of user-given performance objective and constraints.

- **Novel system architecture that supports adaptability in MPI applications**. This thesis targets the design and implementation of system architecture that encapsulates the adaptability techniques which main design goals are scalability and portability. This work proposes an architectural independent dynamic execution model for MPI programs that allows them to take advantage of the optimization techniques for adaptability.

1.5 Advantages of the approach

The main advantages of our approach when applied to MPI applications are as follows:

- Dynamically adapt applications to take advantage of the available resources at the time.
- Optimize applications based on the criteria defined by the user rather than assuming a unique optimization metric.
- Automatically adapt applications without user intervention and with no prior knowledge about the underlying architecture.

The work presented in this thesis brings to MPI applications high-level functionalities to cope with the breakthrough of dynamic execution environments and allow systems to improve their overall performance and throughput.

1.6 Structure and contents

This remainder of this document is structured as follows:

- Chapter 2 - *State of the art* overviews the current works related with the scope and topics of this thesis. First, we overview the different types of high performance computing architectures: cluster, grid and cloud. Second, we describe the different types of parallel programming models both for shared and distributed memory systems. Then we present several works related with the optimization techniques proposed by this thesis: load balancing and dynamic reconfiguration. Finally, we conclude with the summary and conclusions obtained through the review of related works.
- Chapter 3 - *FLEX-MPI: Adaptability techniques for MPI applications* overviews the design and implementation of FLEX-MPI architecture. Additionally, this chapter introduces the programming and execution models of FLEX-MPI applications.
- Chapter 4 - *Dynamic process and data management in MPI applications* describes the design and implementation of the low-level functionalities of FLEX-MPI: performance monitoring and communication profiling, dynamic process

creation and termination, process scheduling, management of MPI communications, and data redistribution.

- Chapter 5 - *Performance-aware malleability and dynamic load balancing in MPI* introduces the high-level functionalities of FLEX-MPI for adaptability in MPI applications: dynamic load balancing and performance-aware dynamic reconfiguration policies. Additionally, this chapter presents the computational prediction model.
- Chapter 6 - *Experimental results* presents the results of the performance evaluation of MPI applications running FLEX-MPI for adaptability on different cluster configurations.
- Chapter 7 - *Conclusions* discusses the contributions of this thesis, publications, and describes future works.

We conclude with the bibliography used to elaborate this dissertation.

Chapter 2

State of the art

2.1 Introduction

This chapter presents a review of the state of the art related to this thesis. The chapter is organized in five sections: high performance computing, classification of parallel computer architectures, parallel programming models, load balancing in parallel applications, and dynamic MPI parallel applications.

2.2 High performance computing

During the last decades the computer industry have experienced a dramatic transition from sequential to parallel processing. Almasi and Gottlieb [AG89] defined a parallel computer as a "*collection of processing elements that communicate and cooperate to solve large problems fast*". The idea behind it is to split up a large problem into smaller tasks which are then solved concurrently by several processing elements.

Parallel computing systems can be classified depending on their memory model and how processors communicate with each other into shared memory systems and distributed memory systems. Figure 2.1 illustrates the typical architecture of a shared memory system and a distributed memory system, where P stands for a CPU and M stands for a memory module. The shared memory model offers a single address space used by all processors, which means that each memory location is given a unique address within a single range of addresses. The main advantage of the shared memory systems is the high speed for sharing data between processes, while the lack of scalability to a large number of compute nodes and their complexity are their major drawbacks. On the other hand, distributed memory systems represent a more cost-effective solution which consists of commodity stand-alone computers (so-called compute nodes) interconnected by a high speed network. In distributed memory systems each processor manages its own local address space, therefore each processor has its own private data. Nowadays, multi-core processors are present on each commodity computer, which makes each compute node a parallel computer itself.

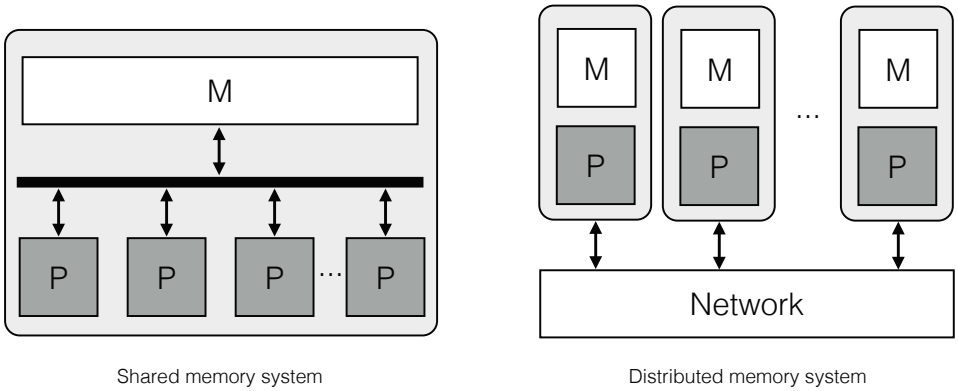


Figure 2.1: Shared memory and distributed memory system architectures.

High performance computing refers to the use of parallel processing for solving data-intensive and compute-intensive applications. Parallel computing enables to solve very large data-intensive applications which data set are large enough that cannot be solved by a single computer. It also enables to save time by minimizing the completion time of an application. Compute-intensive applications can benefit from the computing power of multiple processors to execute faster. Nowadays, high performance computing platforms are used to cover a wide range of parallel applications from science to industry areas such as physics simulations (e.g. weather forecasting, fluids mechanics, seismology, etc.), military simulations, data mining, or rendering of medical images.

Supercomputer performance is commonly measured using as metric the number of floating-point operations per second ($FLOPS$). Since 1993 the TOP500 list [MSDS12] ranks the most powerful supercomputers in the world. The computational power of the supercomputers ranked in the TOP500 list is measured using the HPL (which stands for High-Performance Linpack) benchmark, a portable implementation of the Linpack benchmark suite [DLP03]. The benchmark consists of a set of Fortran subroutines for solving a dense linear system of equations for different problem sizes in double precision arithmetic. The list is published twice a year and has evolved from a simple ranking system to a primary source of information to analyze current and future trends in supercomputer industry.

According to the data collected in the TOP500, distributed systems have become the most popular approach for high performance computing. The most representative distributed computing architectures are cluster, grid and cloud platforms. The last issue of the TOP500 list, published in November 2014, shows that up to 85.8% of the most powerful supercomputers in the world are clusters, while the remaining 14.2% of the share are systems based on the MPP (Massively Parallel Processing) architecture.

Cluster and grid systems are different approaches of the same paradigm. At a glance, cluster and grid architectures can be defined as distributed systems which aim to solve large computational problems, but there are both conceptual and technical differences between them. Cluster architectures generally refers to a set of

tightly coupled systems with centralized job management and scheduling policies, interconnected through a high speed network [B+99]. In most cases, cluster architectures consist of homogeneous computer resources operating as a single computer located in the same physical location. On the other hand, grid architectures are fully decentralized systems, with distributed job management and scheduling policies, interconnected through relatively slow networks, and in most cases consisting of heterogeneous computer resources (both different hardware and operating system) located in different physical locations which can operate independently [Fos02].

Cloud is a new computing paradigm for on-demand distributed computing emerged in the very last years that uses a *pay-as-you-go* model. Cloud computing offers a set of computing services (infrastructure, platform, and software services) usually supported by virtualization technologies [FZRL08]. In the back-end, cloud architectures are traditional interconnected clusters.

2.2.1 Cluster computing

Cluster computing normally refers to tightly coupled systems interconnected through a high speed network, which allow to efficiently exploit locality minimizing the latency in communication operations. Buyya[B+99] defined a cluster as a "*type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource*". Each compute node of a cluster is a COTS (Commercial Off-The-Shelf) computer itself, with its own processor—which may be a single processor or a multiprocessor, memory, I/O facilities, and operating system.

Cluster architectures can be found in many research centers and companies due to their good cost to performance ratio. Figure 2.2 shows the typical architecture of a cluster as it was defined by Buyya [B+99]. The main components of a cluster are a set of multiple, independent computers, high performance networks, and the middleware. Middleware is the piece of software which provides a single access point to the system creating the illusion of a single system image (SSI), an abstraction of the entire distributed system [TVR85].

Cluster architectures offer the following advantages:

- Cost-effectiveness: clusters made of COTS components permits to obtain high computational power at a relatively low cost. That makes high performance clusters a cheaper replacement for the more complex and expensive traditional supercomputers.

- High Performance: cluster systems allow to efficiently exploit parallelism and achieve high performance computing for data-intensive or compute-intensive applications maximizing the throughput.

- Expandability and scalability: cluster computing can scale to very large systems with hundreds or even thousands of interconnected machines working in parallel.

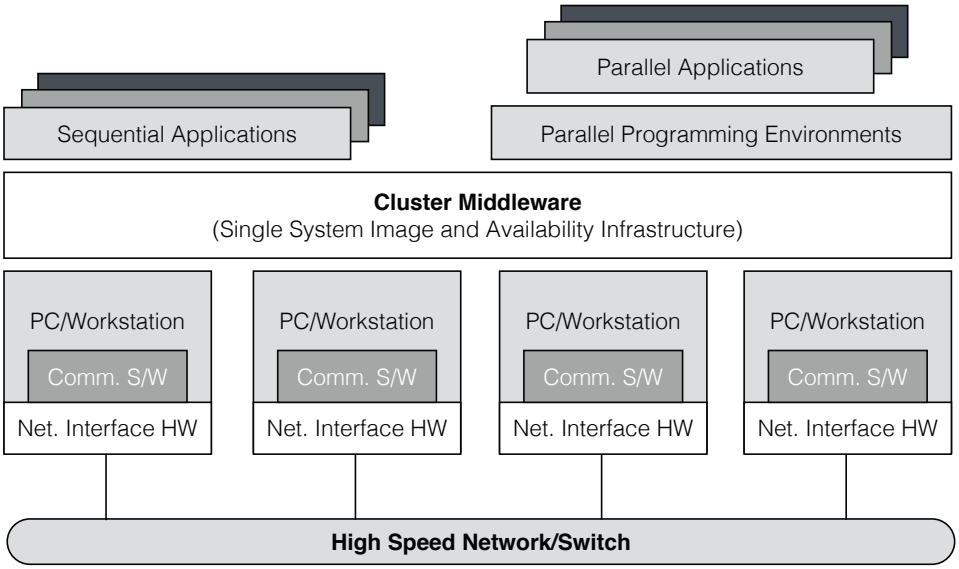


Figure 2.2: Traditional cluster computing architecture.

- High Availability: when a compute node fails its remaining tasks can be computed by other node, ensuring that there is no interruption in service. Moreover, replacing a faulty node is much faster and trivial when compared to replacing a faulty component in an Symmetric Multi-Processing system.

Clusters can be classified [B+99] into different categories based on the following factors:

- **Application target**: computational science or mission-critical applications.
  - High Performance Clusters (HP clusters) are designed to exploit parallel processing and perform computational-intensive operations. The aim of HP clusters is to offer a very high computational power in order to accomplish tasks in the shortest time maximizing the performance.
  - High Availability Clusters (HA clusters) are designed to offer reliability and constant access to both computational resources and service applications, with an aim for integrating performance and availability into a single system.
  - High Throughput Clusters (HT clusters) are designed to run sequential jobs over long periods of time. In opposite of HP clusters, which need large amounts of computational resources to perform parallel jobs in short periods of time, HT clusters also require large amounts of resources for executing sequential jobs over long periods of time (months or years, rather than hours or days).
  - Load Balancing Clusters (LB clusters) are designed to share computational workload in order to achieve an optimal resource utilization avoiding system overload.

- **Node ownership**: cluster owned by an individual or dedicated as a cluster node.
  - Dedicated Clusters: computing resources are shared so that parallel computing can be performed across the entire cluster. Compute nodes are not viewed or used as individual workstations.
  - Non-dedicated Clusters: compute nodes are individual workstations and both parallel and user applications run simultaneously in the system.
- **Node hardware**: PC, Workstation, or SMP.
  - Clusters of PCs (CoPs)
  - Clusters of Workstations (COWs)
  - Clusters of SMPs (CLUMPs)
- **Node configuration**: node architecture and type of OS it is loaded with.
  - Homogeneous Clusters: compute nodes with the same architecture and the same OS.
  - Heterogeneous Clusters: compute nodes with different architectures (e.g. a GPU cluster in which each compute node is equipped with a GPU and a CPU, or a CPU cluster with different, performance-wise, processors across the compute nodes) which may run different OSs.

2.2.2 Grid computing

Ian Foster [Fos02] proposed the definition of grid as "*a system that coordinates resources which are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service*". In his paper Foster suggests a three point list which contains the essence of the definition posted above. First, a grid platform integrates and coordinates resources and users that live within different control domains. Second, a grid is built from multi-purpose open-source protocols and interfaces that address such fundamental issues as authentication, authorizations, resource discovery, and resource access. Finally, a grid allows its constituent resources to be used in a coordinated fashion to deliver non-trivial qualities of service related to response time, throughput, availability, security and co-allocation of multiple resource types.

One of the motivations of grid computing is to provide access to computing resources no matter where the users are located, which is the foundation of the idea of power grid and the origin of the name given to this distributed computing approach. Users have transparent access to heterogeneous (both hardware and software) computing resources which are geographically distributed and interconnected through an heterogeneous network. Although their control is decentralized, grid resources are managed by a software system such as the Globus Toolkit [FK97], an open-source software for building computing grids. Globus Toolkit is the most

widely used software for deploying large-scale grid infrastructures and includes several software services for resource discovery, allocation, and management, security, data management, and process creation.

The vast majority of the projects in which grid systems are involved can be included within e-Science (computationally intensive science) paradigm. Some of the most important research projects which are currently powered by grid systems are the TeraGrid project in the United States, which aggregates the resources of 10 institutions serving 4,000 researchers over 200 universities, and the European Grid Infrastructure (EGI), which provides 13,319 users and more than 300,000 computing cores. Krauter *et al.* [KBM02] proposed the following classification of grid systems depending on their usage:

- **Computational grid**. This category denotes systems that have higher aggregate computational capacity for single applications than any of its machine in the system. Depending on how this capacity is utilized, computational grids can be further subdivided into the following categories:
  - Distributed supercomputing. Such grid type executes the application in parallel on multiple machines in order to reduce the completion time of a job and maximizing performance. Typically, applications that require distributed supercomputing are grand challenge problems such as weather modeling and nuclear simulation.
  - High throughput. This grid subcategory increases the completion rate of a stream of jobs and are well suited for "parameter sweep" type applications such as Monte Carlo simulations.
- **Data grid**. Such grid category is for systems that provides an infrastructure for synthesizing new information from data repositories such as digital libraries or data warehouses that are distributed in a network. Data grids are based in an specialized infrastructure provided to applications for storage management and data access.
- **Service grid**. This category is for systems that provide services that are not provided by any single machine. This category is further subdivided in the following:
  - On demand. This grid subcategory dynamically aggregates different resources to provide new services.
  - Collaborative. It connects users and applications into collaborative workgroups. These systems enable real time interaction between humans and applications via a virtual workspace.
  - Multimedia. Such grids provides an infrastructure for real-time multimedia applications which requires supporting QoS across multiple different machines whereas multimedia application on a single dedicated machine may be deployed without QoS.

2.2.3 Cloud computing

Cloud is a new computing paradigm for on-demand distributed computing emerged in the very last years served in a *pay per use* model. Foster *et al.*[FZRL08] defined cloud as a "*large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet*". According to Foster, clouds and grids share a lot of features in their vision, architecture, and technology, but they also differ in several aspects such as security, programming model, data model, applications, and abstractions.

The first concept in the definition above refers to the fact that cloud computing is a distributed computing paradigm which involves a massive number of computer resources. Second, cloud computing power is served in a *pay-as-you-go* model, in which the user pays only for the requested resources and their usage. Third, cloud provisions a computing platform, an storage platform, and a set of services which are usually supported by virtualization technologies. These resources are abstracted to the user and auto-scalable. Finally, cloud services and infrastructure (hardware and software) are served to the users through the Internet following an on-demand model which allows them to request an "unlimited" number of resources.

Cloud computing represents an opportunity for many individual users and small companies which cannot afford to build and maintain its own infrastructure to start small businesses and increase hardware resources only when there is an increase in their needs [Vog08]. Although cloud computing offers the illusion of "unlimited" computing resources, in the back-end cloud are interconnected clusters generally made of the aggregation of heterogeneous hardware and software resources [VRMCL08].

The most representative features of cloud computing are the following:

- **Elasticity**. It defines the capability of the platform to adapt to variability in the context of the application. This concept implicitly carries itself the notion of scalability. Horizontal scalability in order to scale up or down the number of instances needed to run the application, and vertical scalability to change the type of instances resources in order to meet the requirements of the application.
- **Reliability**. Capability of the platform to accomplish the tasks assigned without fails. Representative fails are service interruption or data loss, which could be mitigated enforcing failure prevention techniques.
- **Quality of Service**. It defines the capability of the platform to accomplish the user requirements, measured in response time, aggregate bandwidth and computational power.
- **Adaptability**. Capability of the platform to adapt to different workload levels ensuring the quality of service demanded by the user. The platform adaptability is related to the middleware.
- **Availability**. It is related to the fault tolerance provided by the platform, which is a critical point in the deployment of cloud infrastructures. In order to

accomplish high availability in the platform, computing and data redundancy techniques are used.

Cloud computing offers different service models to access their services through the Internet:

- **Infrastructure as a Service (IaaS)**. It defines a set of hardware computing resources which can be accessed through the Internet. The access to these resources is supported by virtualization technologies, which offers virtual machines empowered by hypervisor software and running in physical machines.
- **Platform as a Service (PaaS)**. It offers computing resources to host and run client applications with a set of established computing requirements such as number of cloud instances, operating system, programming language environment or database. These software are accessible through APIs.
- **Software as a Service (SaaS)**. It defines a model to access applications or services on-demand, in which both software and associated data are hosted in a cloud server hidden to the users. Users are allowed to online access the software from application clients.

Depending on the deployment model it is possible to categorize cloud computing as follows:

- **Private clouds**. Internal datacenters of a business which owns to a private organization and it is not publicly available. The use and access to the platform are restricted to the users and activities of such organization.
- **Public clouds**. It provisions a publicly available platform which it is usually offered as infrastructure (compute and storage), platform and software resources. Such services are offered by public cloud providers as Amazon, Google or Rackspace.
- **Hybrid clouds**. The combination of both private and public cloud paradigms results in a model in which a set of services remain private and has others which are provided publicly. Hybrid clouds provide the security of a private cloud with the flexibility of a public cloud.
- **Federated clouds**. A federation of clouds is considered as the union of disparate cloud computing infrastructures, including those owned by separate organizations, which take advantage of the interoperability and aggregated capabilities in order to provide a seemingly infinite service computing utility [RBL+09].

Cloud computing systems are subject to performance variability due to several characteristics of their infrastructure: performance degradation induced by virtualization, heterogeneous configuration of both virtualized and physical resources, and shared resources among a large number of users. Several computer

scientists have analyzed the viability of cloud computing infrastructures for high performance computing giving special attention to MPI based applications performance in the cloud[EH08, HZK+10]. Most of them pointed out performance unpredictability as one of the most noticeable flaws of cloud computing platforms [AFG+09, AFG+10, JRM+10].

2.2.4 Performance analysis of parallel applications

In order to analyze the performance of a parallel program, several metrics are taken into account. The first metric is the parallel completion time. The parallel completion time (T) measures the time invested by the system on executing the parallel program. The completion time of a parallel program consists of the sum of the computation time, communication time, synchronization time, and I/O. The computation time usually depends on the size of the problem, the processes to processors mapping, and the performance of the underlying hardware. The communication time is highly dependent on both the performance of the interconnection network and the process mapping. The synchronization time during which the process remains idle waiting for other processes usually depends on the load balance of the parallel tasks between the processes.

Normally, the relative speedup is the most widely used metric for evaluating the performance of a parallel program. It is defined [Kwi06] as the ratio between the completion time of the sequential version of the program executed on one processor to the completion time of the parallel version executed on p processors (Equation 2.1). The relative speedup (S) measures how much faster the parallel algorithm runs than the sequential algorithm. However, sequential execution is not available for very large data-intensive programs which have to be executed in parallel due to memory restrictions. In that cases, the speedup is relative to r processors ($r > 1$) and measures how much faster the program runs on p processors than the program executed on r processors (being $p > r$) (Equation 2.2).

$$S_p = \frac{T_{sequential(1)}}{T_{parallel(p)}} \tag{2.1}$$

$$S_r = \frac{T_{parallel(r)}}{T_{parallel(p)}} \tag{2.2}$$

Nowadays, parallel computing architectures are becoming more complex—multicore processors, several levels in the memory hierarchy, etc.—and therefore it seems important to evaluate how the parallel program exploits the underlying hardware. Exploiting properly the underlying hardware and making an efficient utilization of the available computing power are important to achieve the best performance. Parallel efficiency is a performance metric directly related with the speedup and evaluates how well the parallel program uses the computing resources. The parallel efficiency (E) is defined as the ratio of speedup to the number of processors (Equation 2.3).

$$E = \frac{S_p}{p} \tag{2.3}$$

Ideally, using p processors the parallel program runs p times faster than the sequential version of the program. Scalability is a performance metric which refers to the ability of the program to demonstrate a proportionate performance improvement with the addition of more processors. Two types of scalability are defined: strong scaling refers to the ability of the program to achieve scalability when the problem size of the problem stays fixed as the number of processes increases; weak scaling refers to the ability of the program to achieve scalability when the problem size per processors stays fixed as the number of processes increases. However in practice a parallel program does not achieve such scalability due to the communication and synchronization overheads introduced by parallelization. In fact, at some degree of parallelism adding more processors causes a performance degradation. The *inefficiency factor* [BDMT99] is defined as the ratio between achieved speedup and theoretical speedup and it measures the impact of the communication and synchronization overheads on the performance of the parallel program.

Another useful performance metric is granularity (G) [ST96], defined as the ratio of computation to communication in the parallel program (Equation 2.4). Fine-grained applications are characterized by performing a relatively small amount of computational work between communication phases, which leads to a small granularity ratio. Coarse-grained applications exhibit a high computation to communication ratio due to relatively large amounts of computational work are done between communication phases. The finer the granularity the larger speedup can be achieved by the parallel program. However if the granularity decreases then the overheads of communication and synchronization between processes increase, which degrades the performance of the application. Otherwise, if the granularity is too coarse it may lead to load imbalance.

$$G = \frac{T_{comp}}{T_{comm}} \tag{2.4}$$

During the last years the operational cost of the program execution has become a key performance metric [DYDS$^+$10, DBSDBM13] which measures the cost of running the program in the parallel system. This cost reflects the expenses on hardware acquisition and maintenance, software licenses, electricity, security, etc. The operational cost (C) of the program execution is defined as the cost of running the system per time unit multiplied by the completion time of the application (Equation 2.5).

$$C = cost\ per\ time\ unit\ \times T \tag{2.5}$$

2.3 Classification of computer architectures

The most popular classification of computer architectures was proposed by Michael J. Flynn in 1966. Flynn's taxonomy [Fly72] is based on the notion of a stream of

information which can be either an instruction or data. The instruction stream is defined as the sequence of operations performed by the processor. The data stream is defined as the sequence of data transferred between the memory and the processor. Flynn's classification scheme defines four classifications of computer architectures that are based on the number of concurrent instructions (single or multiple) and data streams (single of multiple) available in the architecture.

- **Single Instruction Single Data (SISD)** refers to a single processor computer which cannot exploit neither instruction or data parallelism. The processing elements performs a single instruction which operates on a single data stream at a time. SISD computers follow the von Neumann architecture.
- **Single Instruction Multiple Data (SIMD)** refers to the classification of a parallel computer architecture which performs a single instruction that operates on different sets of data at a time using multiple arithmetic logic units controlled by a single processing element working synchronously, then exploiting data parallelism. Today's examples of SIMD computers are the Intel Xeon Phi processor and a GPU (Graphics Processing Unit).
- **Multiple Instruction Single Data (MISD)** refers to an uncommon parallel architecture in which multiple instructions operate on the same set of data. Although there are no examples of commercial MISD computers, this classification is normally used for fault tolerance machines which redundantly perform the same instruction to detect and fix errors.
- **Multiple Instructions Multiple Data (MIMD)** refers to a general-purpose parallel computer equipped with multiple processors working asynchronously which execute their own instruction stream and operate on different sets of data. The MIMD classification can be further divided in different subcategories. One the most widely used classifications considers that MIMD systems are subdivided depending on memory organization [Bel89], which can be either shared memory or distributed memory as described in Figure 2.1. In addition to the Flynn's taxonomy, another popular classification considers that parallel MIMD computers can be further subdivided into *Single Program Multiple Data* (SPMD) and *Multiple Program Multiple Data* (MPMD). However, rather than a classification of computer architectures, SPMD and MPMD also represent programming techniques for achieving and exploiting parallelism in parallel computers.
  - **Single Program Multiple Data (SPMD)** [DGNP88] refers to a programming model in which multiple instances of the same program are simultaneously executed on multiple, autonomous processors and operate on different subsets of data. SPMD programs are usually implemented using the message-passing communication model to communicate processes running on different processors. In the message-passing model, processes send and receive messages either in a synchronous or asynchronous mode to communicate data to other processes. Messages also can be used for collective synchronization purposes.

– **Multiple Program Multiple Data (MPMD)** refers to a programming model in which multiple instances of at least two different programs are simultaneously executed on multiple, autonomous processors and operate on different subsets of data. In practice, MPMD applications are not as common as SPMD applications.

2.4 Parallel programming models

A natural classification of programming models for parallel computers uses the memory model of the target architecture to divide the existent approaches. Processes in a parallel program usually communicate to send or receive data to other processes and perform synchronization operations. One of the key properties of shared memory systems is that communication operations are implicitly performed as a result of conventional memory access instructions (i.e. load and stores), while in distributed memory systems communication operations are genuine communications (i.e. send and receives) [CSG99]. Therefore, it makes sense to use a programming model specifically designed for shared memory systems or a programming model specifically designed for distributed memory systems depending on the memory model of the target architecture.

Table 2.1 [DM98] summarizes a comparison between the most widely used standard parallel programming models for shared and distributed memory architectures.

	Pthreads	**OpenMP**	**HPF**	**MPI**
Scalable	sometimes	yes	yes	yes
Incremental parallelization	yes	yes	no	no
Portable	yes	yes	yes	yes
High level	no	yes	yes	yes
Supports data parallelism	no	yes	yes	yes
Performance oriented	no	yes	no	yes

Table 2.1: Comparison of different standard parallel programming models.

2.4.1 Shared memory programming

POSIX threads [NBF96], OpenMP [DM98] and High Performance Fortran (HPF) [Lov93] are the most widely used standard parallel programming models for shared memory systems. Other alternatives are CILK [BJK+95] and Intel Threading Building Blocks (TBB) [Phe08]. Pthreads is the implementation of the standardized threads programming interface for UNIX systems. OpenMP is a portable standard API which supports incremental parallelization in many programming languages, while High Performance Fortran is a set of extensions to Fortran 90 for supporting parallel processing.

Though it is beyond of the scope of this work, in recent years GPGPU (General-Purpose Computing on Graphics Processing Units) has become a trend on HPC.

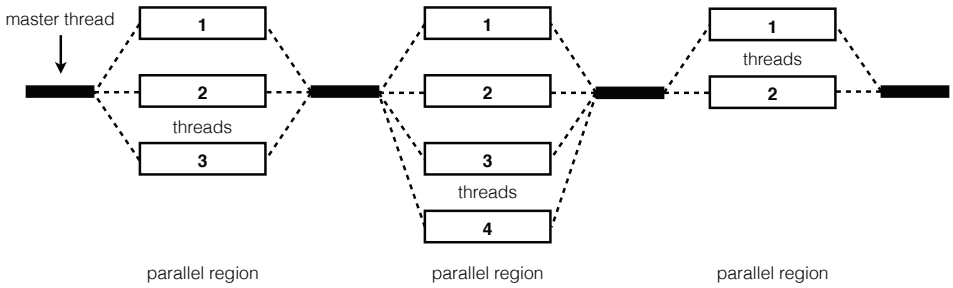


Figure 2.3: Typical OpenMP parallel program execution.

GPGPU refers to the utilization of graphics processing unit to perform scientific computations. A GPU a has a massively parallel set of processors, as well as a high-speed shared memory. CUDA [Cor15], OpenCL [SGS10], OpenACC [WSTaM12], and C++ AMP [GM12] represent the most used alternatives for programming shared memory GPU and heterogeneous CPU/GPU systems.

2.4.1.1 POSIX threads

POSIX threads [NBF96], referred to as Pthreads, is the implementation of the POSIX standard for threads in UNIX systems. The term thread, derived from *thread of execution*, is defined as an independent set of programmed instructions that can be scheduled to run as such by the operating system. A thread is generally contained inside a process, sharing process resources but with its own independent flow of control. Threads model is efficiently used for parallel programming due to significant advantages. Creating a new UNIX process is a heavier operation than threads creation since threads require fewer system resources than processes. Context switching is also a lightweight operation since the thread context switching occurs within the same process. Threads programming model is particularly suitable for multicore architectures, in which each computing core manages a bunch of threads which execute concurrently.

2.4.1.2 OpenMP

OpenMP [DM98] is a portable standard API designed for shared memory architectures. It consists of a set of compiler directives, runtime library routines, and environment variables that extend many programming languages (such as Fortran, C and C++) to exploit parallelism in shared memory computers. The most important feature of OpenMP is that it supports incremental parallelization (through an explicit programming model) in sequential programs. OpenMP is thread-based and it follows a fork-join parallelism model in which the master thread executes sequentially while no parallel regions are detected. The user annotates these regions (e.g. loops) and then the master thread automatically forks the desired number of parallel threads to perform the parallel section. When the threads complete the parallel computation, the results are gathered by the master thread (Figure 2.3).

2.4.1.3 High Performance Fortran

High Performance Fortran (HPF) [Lov93] is a set of extensions to Fortran 90 for supporting parallel processing and accessing to high performance architecture features. The design objectives of HPF were to specify a language which supports efficient data parallel programming, achieve high performance in both SIMD and MIMD systems, and support code tuning. HPF can be successfully used in both shared memory and distributed memory systems. The developer writes its code following the SPMD model annotated with a set of directives and statements which define the data parallelism and then the architecture-specific compiler generates the code for each specific architecture. It works for SIMD, MIMD, RISC and vector systems. HPF implements a data-parallel approach in which the data is distributed at the beginning of the execution and all of the threads are kept alive during the program execution.

2.4.2 Distributed memory programming

Message-passing is the traditional programming model for distributed memory systems. The basics of the model is the communication of concurrent processes using send and receive routines. Transferred data is divided and passed out to other processes using data buffers. Although the message-passing model was designed for programming in distributed memory systems, but it can be used for programming parallel application in shared memory systems. Some of the existent message-passing distributions implement specific optimizations for shared memory systems [BMG06a] in which intra-node communications are performed using shared memory mechanisms with lower latency and overhead, instead of sockets (used for inter-node communications). The main advantages of the message-passing model are: [Com07]:

- Portability: message passing is implemented on most parallel platforms.
- Universality: the model makes minimal assumptions about underlying parallel hardware. Message-passing libraries exist on computers linked by networks and on shared and distributed memory multiprocessors.
- Simplicity: the model supports explicit control of memory references for easier debugging.

The two most widely used message passing environments are Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). Both MPI and PVM can be used for SPMD and—with some restrictions—MPMD models. MPI is a language-independent standard specification for interfaces developed as a library of message passing routines and the several existent implementations cover a wide range of programming languages and architectures. On the other hand, the main idea behind PVM is to assemble a diverse set of interconnected resources into a "virtual machine" which can consists of single processors, shared memory multiprocessors, and scalable multiprocessors.

2.4.2.1 Message Passing Interface

The Message Passing Interface is the specification of a library of functions for interfaces of message-passing routines. The standard was released for the first time in 1994 by the MPI Forum [Mes94], an organization which involves about 60 people from 40 organizations with representatives from the computer industry, government research laboratories and academics. The MPI standard has many language, architecture, and hardware specific implementations which cover a wide range of languages such as C, Fortran, C++, Java or Python.

The execution model of MPI programs consists of a set of processes and a logical communication pattern which allow them to communicate. Although the last version of the MPI-3.0 norm allows MPI processes to access memory from another MPI process, the basic inter-process communication requires a send-receive communication model. The basic communication model of MPI is based in point-to-point communications, which allows a pair of processes to send and receive messages between them. MPI processes are encapsulated within a communicator. The notion of a communicator is a set of processes that can be treated as a group for collective operations and supports an associated topology [DS98]. The basic MPI collective operations include broadcast, reduction, and barriers in addition to another set of more complex operations. Advanced features of MPI support both blocking and non-blocking communications—which allow to overlap computation and communication—and parallel I/O.

The MPI standard has multiple implementations and there are vendor supplied and publicly available implementations. The most popular publicly available implementations of MPI are:

- MPICH [GLDS96] is a high performance and widely portable implementation developed at Argonne National Laboratory. MPICH implementation efficiently supports multiple platforms including commodity clusters, different high-speed networks, and proprietary high-end computing systems.

- OpenMPI [GFB+04] is an all-new open source implementation which combines three MPI implementations: FT-MPI developed by the University of Tennessee, LA-MPI developed at Los Alamos National Laboratory, and LAM/MPI developed by the Indiana University. OpenMPI efficiently supports a wide range of parallel machines and interconnection networks.

In addition to MPICH and OpenMPI, there are a large number of other significant publicly available implementations such as LAM/MPI [BDV94], one of the predecessors of the OpenMPI project, MVAPICH [Pan], a hardware specific implementation for InfiniBand networks, CHIMP [BMS94], a portable, programmable, and efficient MPI implementation, and MPI-LITE [Bha97], which supports multithreaded computation within MPI. On the other hand, the most significant MPI vendor-supplied implementations are HP-MPI [Com07], Intel MPI [Cor11] and IBM-MPI.

Although MPI was designed for programming parallel programs in distributed memory systems, it also can be used in shared memory systems. In a more complex scenario, an hybrid MPI+OpenMP programming model can be efficiently exploited [RHJ09]. An hybrid model shows a significant improvement in performance parallelizing the master task with MPI processes and exploiting parallelism in parallel regions with OpenMP.

2.4.2.2 Parallel Virtual Machine

Parallel Virtual Machine [GBD+94] is a software system that allows a set of heterogeneous interconnected computers to work as a single high performance parallel machine, a large "virtual machine". It was developed by the University of Tennessee, Oak Ridge National Laboratory, and the Emory University in 1989. PVM is message-passing based and handles message routing, data conversion, and task scheduling in a transparent way to the user. The main features of PVM are the translucent access to hardware, process-based computation (each PVM task is responsible for a part of the computation), explicit message passing model, and heterogeneity and multiprocessor support. PVM implements both point-to-point and collective communication operations between processes. Although in the first versions of PVM there were only two collective communication routines (broadcast and barrier) PVM 3.3 implements several new collective routines including global sum, global max, and scatter/gather.

The PVM system consists of two components: a daemon, and a library of PVM interface routines. The daemon needs to be running in the computers which will build the virtual machine and its main functions are process management and performing message-passing operations between processes. The PVM library contains the routines for message passing operations. PVM programs can be written in C, C++, Fortran-77 or Fortran-90 programming languages.

2.5 Load balancing in parallel applications

Load balancing is a major issue in the parallelization of parallel applications [LTW02] because it may have a huge impact on the overall performance of the program. Load balancing refers to the practice of distribute equal amounts of work to each process in a parallel application. In the case of a heterogeneous (performance-wise) system, load balancing is achieved by distributing portions of work that are proportional to the computational power of each processor. The goal of load balancing is to minimize wait and synchronization times between parallel tasks. That is, minimizing the time during which the processes are idle waiting for other processes to complete their computation. Load imbalance occur when the processes finish their computation at different times, forcing others to wait idle. Load balancing is critical to parallel computing due to performance reasons. Efficient load balancing leads to high performance and efficient resources utilization.

Load balance in parallel applications is achieved by using either static or dynamic load balancing techniques [Don98]. **Static load balancing** refers to the prac-

tice of distributing immutable work assignments to the processes at the program start. Static load balancing requires prior knowledge of the execution platform and the program workload to be uniform during execution. However in practice this is often unfeasible because the user does not have access to the characteristics of the system and certain classes of problems have unpredictable workloads. **Dynamic load balancing** techniques, on the other hand, allow to adapt the workload assignments at runtime to handle classes of problems with variable workloads during execution or cope with non-dedicated systems.

Dynamic load balancing techniques can be classified [Don98] as:

- **Dynamic load balancing by pool of tasks**, typically used in parallel programs following the master/slave paradigm. The master process uses a queue of tasks that can be performed in parallel. Initially, the master sends one task to each slave process. As the slaves compute the task assigned they request more tasks to the master until the queue of parallel tasks is empty.

- **Dynamic load balancing by coordination**, typically used in SPMD programs. Every number of fixed iterations all the processes synchronize, evaluate load balance, and, if load imbalance is detected, redistribute the workload. Usually load imbalance is detected by comparing execution time measurements between the processes. When the difference between the fastest and the slowest process surpasses a threshold a load balancing operation is required.

Dynamic load balancing techniques by coordination are the most useful approach to achieve load balance in SPMD, MPI applications [Don98]. The mechanism for load balancing detection can be easily implemented using time measurements of the MPI processes. Then, a data redistribution functionality is required to redistribute the workload between processes each time load imbalance is detected. Much work have been done in supporting dynamic load balancing in SPMD, MPI applications.

Cermele *et al.* [CCN97] address the problem of dynamic load balancing in SPMD applications with regular computations, assuming that each processor may have a different computational capacity. Their technique implements different load balancing policies that use both current and historic performance data to make decisions about the workload distribution.

Charm++ [KK93] is a programming language and runtime system based on C++ for parallel programs, which are decomposed into a set of objects. One of the main characteristics of Charm++ is dynamic load balancing, which provides multiple load balancing strategies ranging from centralized to fully distributed techniques. Charm++ instruments the parallel application to collect performance metrics of every object and the computation and communication pattern of the program, then the runtime system can decide to migrate an object from processor to processor at runtime in order to maximize the performance of the application.

HeteroMPI [LR06] is a library extension of MPI for supporting load balanced computations in heterogeneous systems. HeteroMPI propose a large API for programming on heterogeneous systems including the modification of several of the

standard MPI routines. The load balancing algorithm depends on the estimation of the computational power of each processor. While HeteroMPI does not provide any method to automatically estimate processor speeds at runtime, the API does provide a function specifically designed to benchmark the processors. However, the benchmark code has to be provided by the application programmer.

Galindo *et al.* [GABC08] present an approach to load balance of SPMD applications in heterogeneous systems. Their method is based on simple time measurements within the application source code. The load balancing algorithm uses time measurements and the current number of rows of the dense data structure assigned to the process to change the size of the partitions in the next program iteration, an inaccurate method when it comes to sparse data structures. Their approach does not include a data redistribution method because it is targeted to parallel applications with replicated, non-distributed data, which is unrealistic for the vast majority of parallel applications based on the message-passing model.

Martínez *et al.* [MAG+11] present ALBIC, a dynamic load balancing system based on [GABC08] which focus on load balancing of parallel applications which execute on heterogeneous, non-dedicated systems. They introduce a method for measuring the computational power of processors which uses system load data obtained from the Linux kernel using a specific module that has to be added to the kernel.

ADITHE [MGPG11] is a framework for automatic tuning of iterative algorithms which execute on heterogeneous systems. ADITHE estimates the computational power of each type of processor during the first iterations of the program by measuring the computation times. Then, the data partition attached to each process is reassigned according to the data collected during the benchmarking phase.

Pimienta *et al.* [GABC08] introduce a methodology for automatic tuning of MPI applications as part of the AutoTune project [MCS+13]. Their approach also uses time measurements to detect load imbalance and redistribute the data when it is detected.

2.6 Dynamic parallel applications

2.6.1 Taxonomy of parallel applications

Feitelson and Rudolph [FR96] proposed a classification of parallel applications in four categories based on who (user or system) decides the number of processors used during their execution and when (at the beginning or during execution) it is decided that number. In *rigid* and *moldable* applications the number of processes allocated is fixed during their execution. We refer to these classifications as *static* parallel applications. On the other hand, *dynamic* parallel applications—*malleable* and *evolving*—may change their number of processes at runtime to cope with varying workloads and changes in the availability of resources in the system.

- **Rigid parallel applications** are inflexible applications which run with the specified number of processors before starting the execution until the end of

it. Neither user nor system are allowed to change the resource requirements during execution. This kind of applications required optimal decompositions based on the problem size, in order to avoid inefficient assignments.

- **Moldable parallel applications** let the system to set the number of processors at the beginning of execution and the application initially configures itself to adapt to this number. Once the parallel application is running the number of processes cannot be changed. A moldable application can be made to execute over a wide range of processors. There is often a minimum number of processors on which it can execute and above that number additional processors can be used to improve performance.

- **Malleable parallel applications** can adapt to changes in the number of available processors during execution. As moldable applications, malleable applications can be made to execute over a wide range of processors. However, malleable applications can change their number of processes during execution according to the availability of resources in the system. In malleable applications the number of processes is imposed by the resource management system (RMS), which provides control over parallel jobs and computing resources. An interesting benefit of malleability is it can be used to allow the system to collect information about the application at runtime by trying several configurations and evaluating the resulting performance. This information can later be used to guide reconfiguration decisions.

- **Evolving parallel applications** may change its resource requirements during execution, noting that it is the application itself which triggers the changes. The reason for the interest in this feature is that many parallel applications are composed of multiple phases, and each phase may contain different degrees of parallelism. It is an interesting approach because it is possible for applications to obtain additional resources when needed, and releasing them when they can be used more profitably elsewhere (thereby reducing the cost of running the application).

2.6.2 Supporting malleability in MPI applications

Dynamic parallel applications can be reconfigured to change the number of processes at runtime. Dynamic reconfiguration allows the parallel application to change both the number of processes and their mapping to processors to cope with varying workloads and changes in the availability of resources in the system. Current HPC systems are usually managed by resource management systems with static allocation of resources. That is, the number of resources allocated for a parallel application cannot be remapped once the job is running on the compute nodes. However, dynamic parallel applications allow more flexible and efficient scheduling policies [Utr10], and it is proved that the throughput and resource utilization can be significantly improved when static and dynamic jobs are co-scheduled in the system [Hun04, KKD02, CGR+09].

Enabling malleable features to MPI applications has been an important area of research in the past years. Today's most challenging large-scale applications for distributed memory systems are developed using the message-passing model. MPI has become the *de facto* standard for programming parallel applications for HPC architectures due to its advantages over PVM such as the multiple implementations freely available, the asynchronous communication features, the high performance of the low level implementation of communication buffers, and its portability [GL98, GKP96, Har94, Hem96, Sap94].

Malleability allows the MPI program to increase the number of processes when there are idle processors in the system and then decrease the number of processes when currently allocated processors are degrading the application performance or they are requested by the RMS for another application with higher priority. Malleability is provided either by a technique called **offline reconfiguration** or using **dynamic reconfiguration**.

2.6.2.1 Offline reconfiguration

The basic approach to support reconfiguration of MPI applications is provided via offline reconfiguration, which consists of a mechanism based on stopping the execution of the application, checkpointing the state in persistent memory, and then restarting the application with a different number of processes. Sathish *et al.* [SD03, SD05, VD03] introduce a framework that supports offline reconfiguration of iterative MPI parallel applications. The framework is supported by the *Stop Restart Software* (SRS), a user-level checkpointing library, and the *Runtime Support System* (RSS), that runs in each machine involved in the execution of the MPI program in order to handle data checkpointing. SRS provides a set of routines which should be called from the MPI program to stop and restart the application with a different number of processes. Data redistribution is done via file-based checkpointing.

Raveendran *et al.* [RBA11a, RBA11b] propose a framework which provides malleability to MPI applications which execute on cloud computing platforms using offline reconfiguration and storing the program data in a cloud-based file system with a centralized data redistribution. The decision of changing the number of processes is based on several constraints such as the completion time of the application. Their work assumes that each program iteration executes approximately the same amount of work. Based on this they calculate the required iteration time to meet the user-given completion time. By comparing the current iteration time with the average time per iteration they decide whether to switch to a different number of MPI processes and virtual machine instances. The source code of the program needs to be modified in order to provide to the framework the knowledge about the original dataset distribution, *live* variables, and kernel regions.

2.6.2.2 Dynamic reconfiguration

Dynamic reconfiguration provides malleability by allowing the application to change the number of processes while the program is running. Dynamic reconfiguration is provided either using operating system-level approaches such as processor virtualization or process migration, or using the dynamic process management interface of MPI introduced with MPI-2. The MPI-1 specification [Mes94] requires the number of processes of an MPI application to remain fixed during its execution. While MPI does not natively support malleability, the dynamic process management interface introduced by the MPI-2 specification 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 [Gro02] and OpenMPI [GFB$^+$04]) and has been used by several approaches to provide dynamic reconfiguration to malleable MPI applications.

Adaptive MPI (AMPI) [HLK04] is an MPI implementation 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) [MSV06, MDSV08] is a runtime system built in the context of the Internet Operating System (IOS) [EMDSV06] 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.* [UCL04] introduces a technique called *Folding by JobType* (FJT) which provides virtual malleability to MPI programs. 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 [MZ94] based on co-scheduling a varying number of processes per processor.

Sena *et al.* [SNdS$^+$07] proposes an alternative execution model for MPI applications that targets grid computing environments. They introduce the EasyGrid middleware, an application management system that transforms the MPI application

to take advantage of dynamic resources and provides monitoring and fault tolerance. Their approach focus on MPI master-worker applications in which workers are created dynamically to deal with heterogeneity and resources availability. One of the main avantages of the EasyGrid is that does not require source code modifications.

Cera *et al.* [CGR+10] 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.

Weatherly *et al.* [WLNL03, WLNL06] introduce Dyn-MPI, a runtime system which measures the execution time of the MPI application and the system load and decides whether to drop nodes from computation based on the ratio between computation and communication. This work focuses on MPI based applications which use dense and sparse data structures. The goal of Dyn-MPI is to drop nodes from computation when their participation degrades the overall performance of the application thus reducing the number of processes of the MPI application. However, Dyn-MPI does not provide capability to scaling up the number of MPI processes. Nodes workload and computation timing are monitored by a daemon which should run in each of the compute nodes. Dyn-MPI requires a significant effort from the developer for translating the MPI source code to Dyn-MPI.

Cera *et al.* [CGR+09, Cer12] introduce an approach to support malleability using the dynamic process management interface of MPI-2. Their work focus on adapt the number of processes of the parallel application according to resources availability (malleability) and the internal state of applications with unpredictable needs (evolving). They also integrate their solution with OAR [CDCG+05], a resource management system which supports dynamic allocation of resources. Leopold *et al.* [LSB06, LS06] present the malleable implementation of WaterGAP, a scientific application that simulates global water prognosis availability using MPI-2 features.

ReSHAPE [SR07, SR09, SR10] is a runtime framework for malleable, iterative MPI applications that uses performance data collected at runtime to support reconfiguring actions. The ReSHAPE framework decides whether to expand or shrink the number of processes of the application when the iteration time has been improved due to a previous expansion or the current processor set do not provide any performance benefit as a result of a previous expansion, respectively. The authors assume that all iterations of a parallel application are identical in terms of computation and communication times and have regular communication patterns. Dynamic reconfiguration is provided by a remapping module which uses the dynamic process management interface of MPI-2 to spawn new processes. ReSHAPE also features a process scheduler and an efficient data redistribution module.

Dynamic MPI applications require support from the RMS in terms of adaptive job scheduling strategies in order to maximize the efficiency and overall performance of the system. In recent years several approaches have been presented in the area

of adaptive batch scheduling policies for malleable and evolving parallel applications [BGA14, GKM14, PIR+14, PNR+15]. These works evince the existing gap between dynamic applications and their lack of full support in the current resource management systems.

2.7 Summary and conclusions

This chapter presents the state of the art in the topics that are covered by this thesis. We have started describing the background topics that are necessary to understand the area of research of the dissertation: high-performance computing, parallel computing architectures, and parallel programming models. Specifically, this work focus on optimization techniques for performance and adaptability in MPI applications which execute on computing clusters. The MPI model was appropriate for parallel applications that use the SPMD paradigm. A large proportion of the scientific MPI applications are iterative. We propose an approach which includes a performance-aware dynamic reconfiguration technique and a novel dynamic load balancing technique by coordination for iterative, SPMD MPI-based applications.

Several works focus on enabling malleable capabilities to MPI applications. Some of the existing approaches use offline reconfiguration to provide malleability [SD03, SD05, VD03, RBA11a, RBA11b]. This mechanism 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, which degrades the performance of the parallel application. Dynamic reconfiguration, on the other hand, is a most efficient approach which allows the application to reconfigure at runtime. Several works enable malleability using operating system-level approaches such as processor virtualization and process migration [HLK04, MSV06, MDSV08, UCL04, CGR+10, WLNL03, WLNL06]. This presents several drawbacks, such as the lack of portability of these approaches. One of the major advantages of MPI is their portability across different systems because all the functionalities of MPI are encapsulated on the library itself. However, system-level approaches provide malleability using techniques that are external to the MPI library and it restricts their portatiliby. Indeed, the MPI library features the dynamic process management interface that allows to spawn MPI processes at runtime. Several works use the dynamic process management interface to enable malleability in MPI [CGR+09, Cer12, SNdS+07, LSB06, LS06, SR07, SR09, SR10].

We observe that most of the existing approaches work under the assumption that increasing the number of processors in a parallel application does not increase its execution time. Furthermore, these approaches use reconfiguring actions to expand the number of processes to the maximum number of processors available 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 [LTW02] but increases the operational cost and decreases the parallel efficiency of the program due to increasing communication and synchronization overheads when the number of processors increases [BDMT99]. To perform an

efficient dynamic reconfiguration a malleable action must be driven by performance. The decision to change the number of processors has to be made based on the present and future performance of the application with the new processor configuration. Thus a precise computational model is required to analyze and estimate the application performance.

In dynamic load balancing we observe that time measurements are the widely used approach to detect load imbalance in SPMD, MPI applications. However, this method is not precise to detect and balance the workload in applications which executed on heterogeneous, non-dedicated systems. One of the method proposed consists on access to system load data using a kernel module [MAG+11]. While this method provides promising results, it is very intrusive and data can not be accessed at the process granularity.

Then, after analyzing the existing works and techniques for enabling malleability and dynamic load balancing in MPI applications we conclude that none of them satisfy the main goals of this thesis.

Chapter 3

FLEX-MPI: Adaptability Techniques for MPI Applications

3.1 Introduction

The aim of this thesis is to provide adaptability to MPI applications. To be specific, our work focuses on dynamic load balancing and malleability. In this chapter we introduce the FLEX-MPI architecture, the malleable execution model for MPI applications, and the FLEX-MPI programming model. First we describe the design of the FLEX-MPI runtime system, which implements optimization techniques for enhancing the performance and adaptability of parallel MPI applications. Then, we present the dynamic execution model used by FLEX-MPI to enable malleability and dynamic load balancing in MPI applications. Finally, we introduce the programming model for malleable applications in FLEX-MPI.

3.2 Runtime system overview

This section describes the environment that provides support for malleable applications in MPI. In addition, we summarize each of the functionalities of the FLEX-MPI runtime system. The FLEX-MPI extends the functionalities of the MPICH [Gro02] runtime system, one of the most popular implementations of MPI. This makes any MPI application based on MPICH to be compatible with FLEX-MPI.

FLEX-MPI is provided as a library—implemented in C language—and its functionalities can be accessed via an application programming interface. The highly-optimized, efficient implementation of FLEX-MPI has a negligible overhead in the overall performance of the MPI application and preserves the portability of the MPICH implementation.

FLEX-MPI enables malleability to MPI applications via novel malleability policies that use a performance-aware dynamic reconfiguration technique. The malleability policies take into account the resources available in the system, their performance, and the operational cost of the underlying, heterogeneous hardware. FLEX-MPI uses

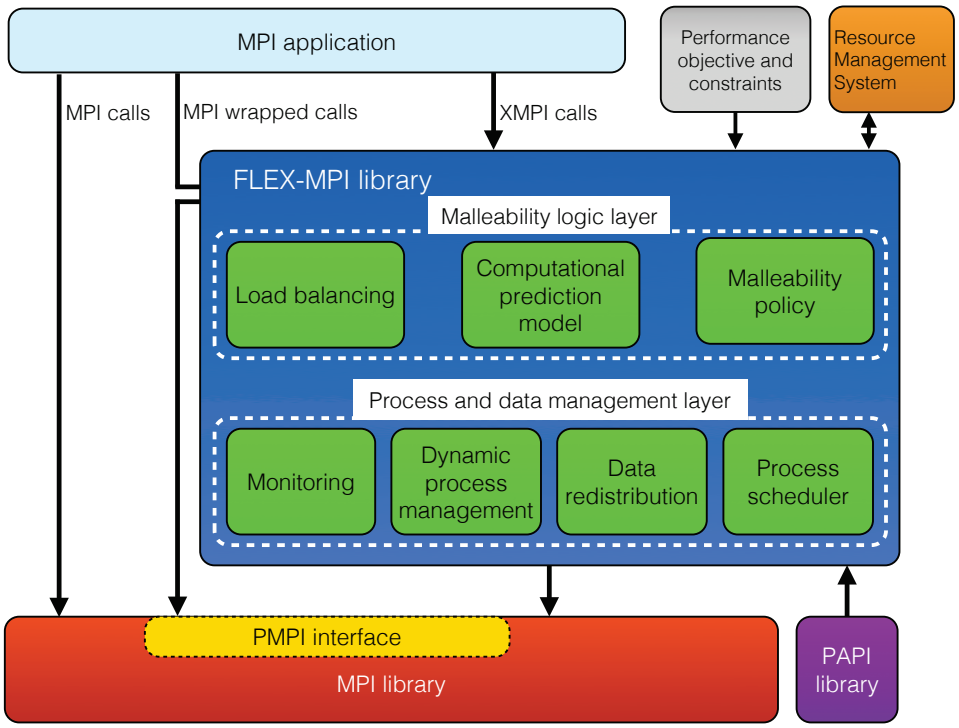


Figure 3.1: The FLEX-MPI runtime system architecture.

this information to increase the application performance, maximize the resource utilization, and minimize the operational cost of the program execution.

Figure 3.1 shows the environment of the FLEX-MPI runtime system which consists of the following components:

- **MPI application**. The user's MPI application instrumented with the FLEX-MPI application programming interface. The instrumented MPI application interacts with the MPI standard library to perform MPI operations and the FLEX-MPI library to perform adaptability actions—such as dynamic load balancing and malleability.

- **MPI library and PMPI interface**. FLEX-MPI is implemented on top of MPICH. That makes FLEX-MPI fully compatible with MPI programs that use all the new features of the MPI-3 standard [MPI]. PMPI is a profiling interface provided by MPI that allows to profile MPI calls and collect performance data in a user-transparent way.

- **Resource Management System (RMS)**. The resource management system which controls the resources availability and schedules the parallel jobs submitted by the users. The RSM feeds FLEX-MPI with availability of resources and system information such as: number of compute nodes available and their operational cost.

- **PAPI library**. The Performance API (PAPI) [MBDH99] is a portable API which provides access to low-level hardware performance counters that enable to measure software and hardware performance. These performance counters are collected by FLEX-MPI via PAPI, and are used to guide performance-aware reconfiguring actions.

- **User-given performance objective and constraints**. Reconfiguring actions in FLEX-MPI are performance-driven. The performance-aware malleability policies of FLEX-MPI allow to reconfigure the application to satisfy a set of user-given performance objective (e.g. completion time or maximum performance) and constraints (e.g. efficiency or cost).

- **FLEX-MPI library**. The FLEX-MPI library includes several components that enable dynamic load balancing and performance-aware malleability functionalities to MPI applications. These functionalities can be accessed from any MPI application using the FLEX-MPI API and are described in the following sections.

3.2.1 Process and data management layer

FLEX-MPI functionalities are divided into low-level functionalities of the process and data management layer and high-level functionalities of the malleability logic layer. The process and data management layer offers basic mechanisms to collect runtime performance metrics of the MPI program, change the number of processes of the application, expand and shrink the MPI communicator, schedule dynamic MPI processes, and move data between processes as a result of workload redistributions. The process and data management layer is directly interconnected with the MPI program and the MPI and PAPI libraries, and it consists of the following components:

- **Monitoring**. The monitoring component gathers performance metrics for each iteration of the parallel application. This includes hardware performance counters via PAPI, communication profiling via PMPI, and the execution time for each process.

- **Dynamic process management**. The dynamic process management component uses the dynamic process management interfaces of MPI to spawn and remove processes at runtime from the parallel application.

- **Process scheduler**. FLEX-MPI implements a dynamic scheduler that handles the scheduling of newly spawned processes in the system satisfying the requirements of the reconfiguring policy.

- **Data redistribution**. FLEX-MPI includes an efficient data redistribution component that uses standard MPI messages to move data between processes as a result of a dynamic reconfiguring action or a load balancing operation.

3.2.2 Malleability logic layer

The malleability logic layer provides malleable capabilities to MPI applications. This layer includes high-level functionalities which implement a computational prediction model, dynamic load balancing techniques, and malleability policies—including performance-aware dynamic reconfiguration. These functionalities use the performance metrics collected by the low-level functionalities to make decisions respect to the number of processes of the application depending on the malleability policy, the user-given performance objective and constraints, and the current and predicted performance of the application. Each time a reconfiguring action or a load balance operation is triggered, the malleability logic layer takes advantage of the low-level functionalities to apply the new processor configuration. A processor configuration describes a set of processing elements allocated to the MPI program.

- **Load balancing**. The load balancing algorithm ensures that the workload assigned to each process of the MPI program is proportional to the computational power of the processor—calculated using runtime performance data collected by the monitoring functionality—which runs the MPI process. The algorithm also considers non-dedicated scenarios in which the MPI program executes simultaneously with other applications.

- **Computational prediction model**. The computational prediction model component uses runtime performance metrics gathered via monitoring to estimate the performance of the parallel application with a new processor configuration prior to the reconfiguring action.

- **Malleability policy**. FLEX-MPI includes performance-aware malleability policies: Strict malleability policy (SMP), High performance malleability policy (HPMP), and Adaptive malleability policy (AMP). These policies automatically reconfigure the application depending on its performance collected at runtime and the user-given performance criteria. The malleability policy analyzes the runtime performance data of the application collected by monitoring and then uses the computational prediction model to make decisions about the most appropriate number and type of processors to satisfy the user-given performance criteria. Changing the number of processors involves using the dynamic process management and scheduling functionalities to spawn or remove processes, and the load balancing and data redistribution functionalities to redistribute the workload each time a reconfiguring action is carried out.

3.3 FLEX-MPI execution model

FLEX-MPI introduces a malleable execution model for MPI applications, rather than the static execution model of MPI standard applications. The malleability model implies that, each time FLEX-MPI is invoked in the iterative section of the program, it acquires the control of the program execution to monitor the application performance and evaluate whether a dynamic reconfiguring action is necessary. FLEX-MPI

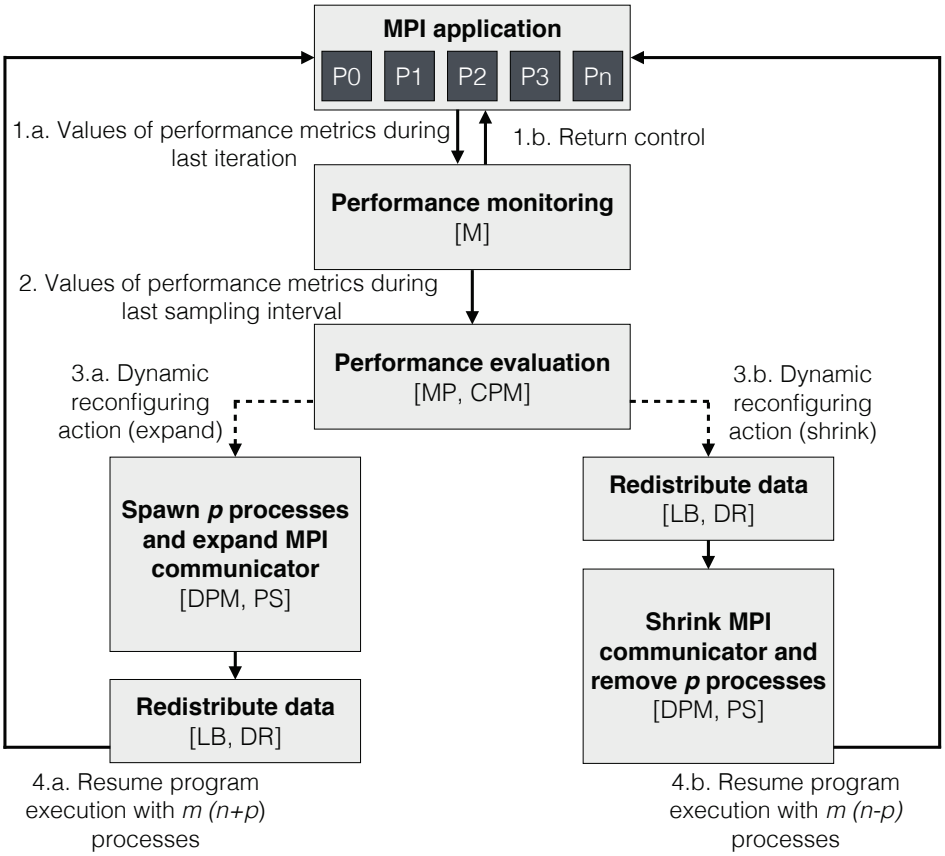


Figure 3.2: Workflow diagram of a malleable FLEX-MPI application.

then returns the control to the application once it has collected performance metrics or at the end of the reconfiguring action.

Figure 3.2 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) component (label `1.a`). These include hardware performance counters, communication profiling data, and the execution time for each process. This allows FLEX-MPI to monitor the performance of each processor core in which an MPI process is running. 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. That is, the maximum number of MPI processes per processor corresponds to the number of PE in the processor. 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—FLEX-MPI evaluates the current performance of the application and decides if it satisfies the user-defined performance requirements or

the application has to be reconfigured. Every sampling interval the monitoring component feeds the gathered performance metrics to the **malleability policy** (MP) component (label `2`). This allows the MP to track the current performance of the application and decide whether it needs to reconfigure the application in order to adjust the performance of the program to the goal imposed by the malleability policy. Otherwise, the application continues its execution with the same number of processes and resources allocated.

A reconfiguring action involves either the addition (label `3.a`) or removal (label `3.b`) of processes and remap the processor allocation. The MP uses the **computational prediction model** (CPM) component to estimate the performance of the application during the next sampling interval based on the runtime performance metrics gathered via monitoring. If the current performance does not satisfy the requirements of the malleability policy, then the CPM informs the MP that the application requires a new reconfiguring action. Otherwise, the application does not need to be reconfigured. In case of a reconfiguring action the CPM estimates the number of processes and the computing power (in $FLOPS$) required to satisfy the performance objective. Using this prediction, the MP computes the new process-to-processor mapping based on the number and type of the processors that are available in the system and the performance criteria (efficiency or cost).

The **dynamic process management** (DPM) component implements the process spawn and remove functionalities—the **process scheduler** (PS) is responsible for rescheduling the processes according to the new mapping. A reconfiguring action changes the data distribution between processes, which may lead to load imbalance. Each time a reconfiguring action is carried out the **load balancing** (LB) component computes the new workload distribution based on the computing power of the processing elements allocated to the application. The **data redistribution** (DR) component is responsible for mapping and redistributing the data between processes according to the new workload distribution.

In case of a process spawning the DPM creates the new processes in the system and expands the MPI communicator to accommodate newly spawned processes. Once all the current processes are encapsulated within the same communicator the load balancing and data redistribution functionalities redistribute the application workload. On the other hand, in case of a process removing action FLEX-MPI firstly calculates the new workload distribution, then disconnects those processes marked by the malleability policy to leave the program, and finally shrinks the MPI communicator with the smaller number of processes. Once Flex-MPI has reconfigured the application to the new number of processes (m), it resumes its execution (labels `4.a` and `4.b`).

3.4 FLEX-MPI programming model

FLEX-MPI introduces a programming model and an application programming interface to provide malleable capabilities to MPI applications. FLEX-MPI is compatible with iterative applications based on MPICH which operates on both dense and sparse data structures with one-dimensional data decomposition and distributed data.

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
(High-level FLEX-MPI API)

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

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

L4:  XMPI_Get_wsize (...);

L5:  XMPI_Register_dense (...);

L6:  XMPI_Get_rdata (...);

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

L8:     XMPI_Monitor_si_init ();

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

L12:    MPI_Allreduce (...);

L13:    XMPI_Eval_reconfig (...);

L14:    XMPI_Get_process_status (...);

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

L17: MPI_Finalize ();
```

Figure 3.3: Comparison of the legacy code (left) and the instrumented FLEX-MPI code (right) of an iterative MPI application.

An MPI application instrumented with FLEX-MPI performs three types of MPI-related calls, as shown in Figure 3.1. Native MPI functions (i.e. `MPI_Comm_rank`) are directly managed by the MPI library. However, some of the calls to native MPI functions are wrapped by FLEX-MPI via the PMPI interface. FLEX-MPI uses the PMPI interface to collect performance data and initialize the library functionalities in a user-transparent way. PMPI allows FLEX-MPI to intercept MPI calls from the application without modifying the source code of the MPI program.

Application developers can access the FLEX-MPI library via the FLEX-MPI API, which consists of a low-level API and a high-level API. The interfaces provided by the FLEX-MPI API carry the `XMPI` prefix. When using the high-level API the number of processes and the process-to-processor mapping are decided by FLEX-MPI depending on the malleability policy used and the available resources in the system. Otherwise, when using the low-level API the developer decides the number of processes, their mapping to the processors in the system, and the data distribution of the application. In some aspects, the low-level API brings to MPI programs a programming model equivalent to use the `fork` system call to create processes in pure C programs.

Figure 3.3 shows a comparison between a simplified legacy code sample and this code instrumented with Flex-MPI functions of the high-level API for automatic, dynamic reconfiguration. The SPMD application (Figure 3.3 left) 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 (Figure 3.3 right) consists of wrapped functions (in orange), native MPI functions (in red), FLEX-MPI functions which allow the parallel program to get and set some library-specific parameters (in blue), and 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 PMPI. 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`). This last operation is necessary to know which data structures should be redistributed every time a reconfiguring action is carried out.

The data redistribution component communicates with the newly spawned processes to pass them the corresponding domain partition before starting the execution of the iterative section (`L6`) and the current iteration of the program (`it`). Newly spawned processes will compute at most the remaining number of program iterationts (`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 malleability policy evaluates whether reconfiguring (`L13`) is required. The malleability policy and the performance objective and constrains are provided by the user to FLEX-MPI as arguments of the `mpiexec/mpirun` command. Then each process checks its execution status (`L14`). In case that the malleability policy decides to remove a process, this leaves the iterative section (`L15`) and terminates execution.

3.5 Summary

In this section we describe the architecture of the FLEX-MPI runtime system that extends MPICH by providing malleable capabilities and dynamic load balancing to iterative MPI applications. FLEX-MPI consists of a process and data management layer and a malleability logic layer. The process and data management layer

provides mechanisms to performance monitoring, change the number of processes, schedule new processes, and data redistribution. The malleability logic layer implements functionalities for automatic dynamic reconfiguration and dynamic load balancing. FLEX-MPI introduces a dynamic execution model that allows MPI applications to take advantage of these adaptability techniques. Finally, we describe the programming model of FLEX-MPI applications and briefly overview the FLEX-MPI API.

Chapter 4

Dynamic Process and Data Management in MPI Applications

4.1 Introduction

In this chapter we describe the design and implementation of the software components of the process and data management layer of FLEX-MPI (Figure 4.1). These components support the low-level functionalities of FLEX-MPI: performance monitoring and communication profiling, dynamic process creation and termination, process scheduling, management of MPI communications, and data redistribution. We also describe the low-level application programming interface which allows MPI applications to access those functionalities of the process and data management layer of FLEX-MPI.

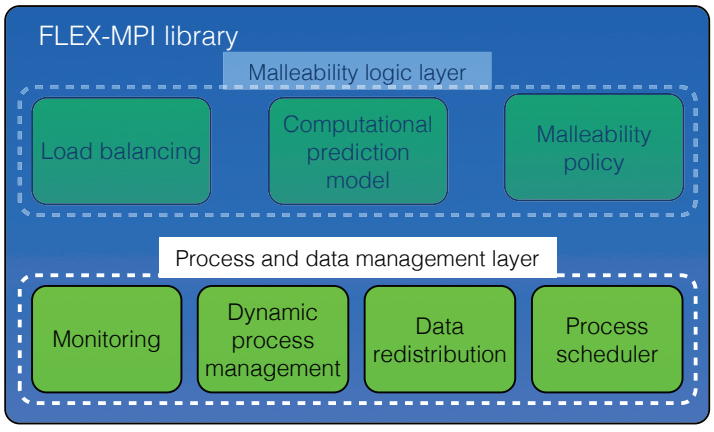


Figure 4.1: Process and data management layer inside the FLEX-MPI library.

Table 4.1 summarizes the low-level interfaces of the FLEX-MPI API that provide the low-level functionalities of the process and data management layer. These interfaces will be covered in greater detail in the following sections.

Table 4.1: Flex-MPI low-level interfaces.

Interface	Description
`XMPI_Monitor_iter_begin`	Starts iteration monitoring
`XMPI_Monitor_iter_end`	Stops iteration monitoring
`XMPI_Spawn`	Spawns dynamic processes
`XMPI_Remove`	Removes dynamic processes
`XMPI_Get_wsize`	Retrieves current data partition
`XMPI_Register_vector`	Registers a vector
`XMPI_Register_dense`	Registers a dense data structure
`XMPI_Register_sparse`	Registers a sparse data structure
`XMPI_Redistribute_data`	Data redistribution

4.2 Monitoring

This section describes the functionalities of the monitoring component of FLEX-MPI. The monitoring component allows to collect performance metrics from the MPI program and the hardware system. These metrics are then later used by the performance-aware malleability policies to evaluate whether a dynamic reconfiguring action is necessary to improve application performance. The monitoring component uses the Performance API (PAPI) [MBDH99] to access computation performance metrics of the MPI program and the hardware system, and the Profiling MPI interface (PMPI) to profile communications of the MPI program.

Figure 4.2 and Figure 4.3 describe the parameters of `XMPI_Monitor_iter_begin` and `XMPI_Monitor_iter_end` interfaces, respectively. These low-level interfaces provide the functionality of program monitoring in FLEX-MPI. When they are placed at the begin and end of the iterative section of the program, these functions collect performance metrics of the application during the current program iteration and return the performance counters at the end of the iteration. `XMPI_Monitor_iter_begin` takes as input an array with the names of PAPI hardware events (i.e. `PAPI_FLOPS`, `PAPI_TOT_CYC`) to be monitored during the program iteration. `XMPI_Monitor_iter_end` returns an array with the hardware performance counters for each hardware event defined in `XMPI_Monitor_iter_begin`.

XMPI_MONITOR_ITER_BEGIN(events)

IN	events	names of hardware events (array of strings)

```
int XMPI_Monitor_iter_begin (char *events[])
```

Figure 4.2: Parameters of FLEX-MPI's `XMPI_Monitor_iter_begin`.

XMPI_MONITOR_ITER_END (counts)

OUT	counts	hardware counters (array of long long integers)

```
int XMPI_Monitor_iter_end (long long *counts)
```

Figure 4.3: Parameters of FLEX-MPI's `XMPI_Monitor_iter_end`.

4.2.1 Computation monitoring

PAPI is a portable API which provides access to a wide variety of hardware performance counters (i.e. floating point operations, CPU completed instructions, etc.). PAPI allows software developers to establish in real time a connection between software performance and hardware events. PAPI offers a high-level API—which provides access to preset events of the CPU—and a low-level API that enables fine-grained measurement of events from the program executable and the hardware. We use low-level PAPI interfaces to track in runtime the number of floating point operations $FLOP$ (`PAPI_FLOPS` hardware event), 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) of the MPI program. These metrics are collected for each MPI process of the parallel application, and they are preserved during context switching. This allows the monitoring component to collect performance metrics for each of the processing elements on which an MPI process is running. FLEX-MPI uses these metrics to calculate the computing power of each processing element as the number of floating point operations per second $FLOPs$ following Equation 4.1.

FLEX-MPI considers both dedicated and non-dedicated system—in which the MPI applications is sharing resources with other user's applications. We use $Treal$ and $Tcpu$ values to detect whether the application is running on a dedicated or a non-dedicated system. The real time is always a little higher than the CPU time because of the OS noise. We assume that when the difference between $Treal$ and $Tcpu$ surpasses a pre-defined threshold—including the OS noise—the application is running on non-dedicated mode. This mechanism allows us to apply more effective load balance strategies that take into account the actual performance of each processing element in the system.

$$FLOPs = \frac{FLOP}{Tcpu(secs.)} \tag{4.1}$$

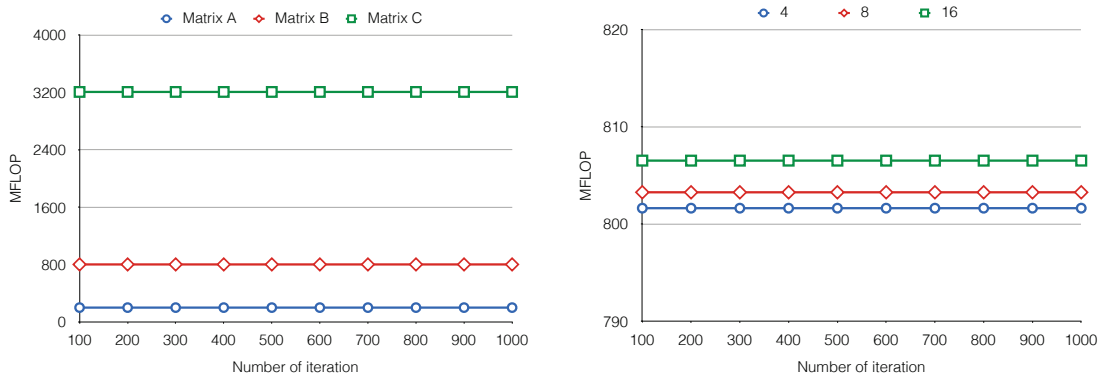


Figure 4.4: Performance analysis of the number of $FLOP$ in Jacobi for varying matrix sizes and 8 processes (a), and a varying number of processes and a matrix of 2,000 rows (b).

Different performance metrics can be used to measure the performance of an application (e.g. FLOP, processor cycles, or CPU completed instructions). We use the number of $FLOP$ because they specifically provide a quantitative value of the workload performed by the process. FLEX-MPI targets SPMD applications whose computation is based on floating point operations, which is reasonable for many MPI-based parallel applications (e.g. linear system solvers, particle simulation, and fluid dynamics). These applications usually exhibit a linear correlation between the $FLOP$ and the workload size per process. That is, the number of $FLOP$ is proportional to the workload computed by the process. Otherwise, for those kinds of applications which CPU usage pattern is correlated with a different performance metric (e.g. integer-based operations), the monitoring component can be easily tuned to collect the specific performance counters that best fit the computation pattern of the application. For instance, the the number of integer instructions can be collected via the `PAPI_TOT_INT` hardware event.

Figure 4.4 illustrates a performance analysis of the MPI implementation of Jacobi—a linear system solver—executed on a homogeneous cluster during 1,000 iterations and a sampling interval of 100 iterations. We measured the number of $FLOP$ performed by the application when computing three matrices (A, B, and C) with different coefficients (1,000, 2,000, and 4,000, respectively) and executing with a varying number of processes. Figure 4.4 (a) shows how the total count of $FLOP$ varies proportionally as the matrix size increases. Figure 4.4 (b) shows that the total count of $FLOP$ when operating on the same matrix slightly varies as the number of processes increases. This short experiment demonstrates the linear correlation that exists between program computation and the $FLOP$ performance metric in the class of parallel applications that FLEX-MPI targets. Regardless, the monitoring component of FLEX-MPI is very flexible and the performance metrics collected can be easily changed to model the performance of a different class of applications.

The events counted by hardware performance counters are processor specific. In heterogeneous systems, different processors may count different values for the same hardware event. FLEX-MPI performs an initial calibration of the counters by

performing a measurement of the per-processor counts for a sample of hardware events on the different types of processors. This calibration is carried out by running a microbenchmark with a negligible overhead in performance before starting the computation of the application in the system.

4.2.2 Communication profiling

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. The monitoring component of FLEX-MPI uses PMPI to profile MPI communications of the parallel application. Specifically, we collect the type of MPI communication operation (i.e. `MPI_Gather`), the number of processes involved in the operation and their ranks in the MPI communicator, the size of the data transferred between processes, and the time spent in the operation.

The PMPI interface allows every single standard MPI function to be called either with the MPI or the PMPI prefix, both of them with identical syntax. In fact, MPI interfaces are weak symbols for PMPI interfaces, which provide the corresponding functionality. Figure 4.5 illustrates an example of the PMPI functionality when an MPI function is invoked within a FLEX-MPI application. The FLEX-MPI library intercepts the function call (label `1`), invokes `MPI_Wtime` to save the time in which the MPI function was called, and then invokes the corresponding `PMPI_Gather` interface of the MPI library which actually implements the functionality of the communication operation (label `2`). Once the MPI library has performed the operation it returns the control to the FLEX-MPI library (label `3`) that uses the `Collect_Perf_data` routine to collect performance data of the communication operation. Finally, the FLEX-MPI library returns the control to the MPI application (label `4`).

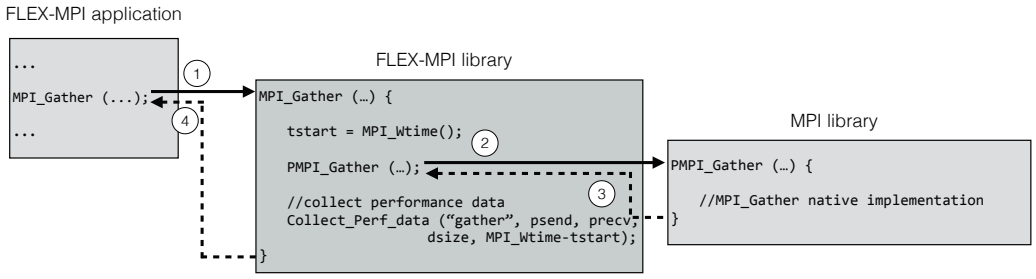


Figure 4.5: Illustration of the behavior of PMPI in FLEX-MPI.

Besides profiling communication operations in MPI, we use PMPI to wrapper the MPI initialize—`MPI_Init`—and finalize—`MPI_Finalize`—functions. This allows FLEX-MPI to initialize and finalize its functionalities in a user-transparent way. The behavior of the wrapper procedure is quite similar to the communication profiling. First we proceed with the FLEX-MPI functionality encapsulated within `MPI_Init` and `MPI_Finalize`; then we invoke the corresponding functions of the MPI library. Basically, we wrapper `MPI_Init` to initialize global variables and monitoring. We wrapper `MPI_Finalize` to release resources and deallocate library variables.

4.3 Dynamic process management

This section describes the dynamic process management component of FLEX-MPI. The functionality of the DPM is to support the creation and termination of dynamic MPI processes in the FLEX-MPI application. This component is also responsible for the management of communications between processes each time a reconfiguring action is carried out.

4.3.1 MPI communicators

MPI is the *de facto* industry standard for programming high performance computing applications in distributed memory systems. The MPI specification [Mes94] defines the standard implementation of the message-passing libraries. The MPI Forum, the group that define and maintain the MPI standard, has released up to date three major issues of the MPI specification: MPI-1, issued in 1994; MPI-2, issued in 1997; and MPI-3, issued in 2012.

The MPI standard defines a *communicator* as a context for a communication operation between a set of processes which share that communication context. A *group* of processes is an ordered set of process identifiers and each process belongs to one or more groups of processes. Each process is identified within a group by a unique *rank* which is used to send and receive messages. The MPI standard defines two types of communicators:

- **Intracommunicator** for communication operations within a single, local group of processes.
- **Intercommunicator** for communication operations between two (local and remote) non-overlapping groups of processes.

MPI features point-to-point and collective communication operations. However, at the lowest level collective operations are built upon point-to-point communication functions. Point-to-point operations (`MPI_Send` and `MPI_Recv`) involve participation of two processes, a sender and a receiver. A collective operation, on the other hand, involves participation of all processes in a communicator. Both point-to-point and collective operations can be performed using both types of communicators and MPI provides blocking and non-blocking versions of the communication functions. Non-blocking versions of point-to-point routines are present in MPI since MPI-1. However, non-blocking collectives are one of the major features introduced in the MPI-3 specification. A blocking communication function does not returns until the communication has finished. That is, all data have been received by the receiver process or processes. In contrast, non-blocking communication functions return immediately, even if the communication has not completed yet. Both the sender and receiver must verify through a specific MPI function that the data has successfully been sent and received by the processes. MPI provides a wide set of collective operations which can be classified as:

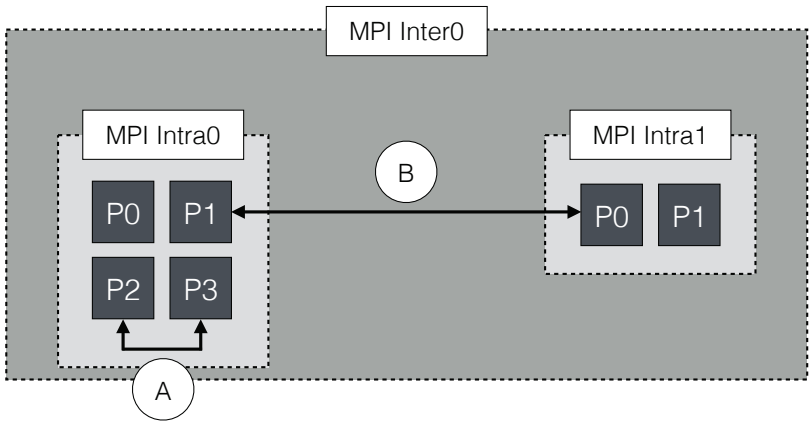


Figure 4.6: MPI communication between processes within a local group performed using an intracommunicator (A) and communication between local and remote processes using an intercommunicator (B).

- **One-to-many**. A single process sends data to many receivers. For instance, the broadcast `MPI_Bcast` operation is an example of one-to-many operation in MPI.
- **Many-to-one**. A subset of processes send data to one receiver. For instance, the gather `MPI_Gather` operation employs the many-to-one communication pattern.
- **Many-to-many**. Every process participates both in send and receive data. For instance, the all-to-all `MPI_Alltoall` operation follows the many-to-many pattern.

Figure 4.6 illustrates an example of an MPI point-to-point communication between processes. Processes within the same group can communicate using the local intracommunicator MPI_Intra0 or MPI_Intra1. For instance, processes with ranks `P2` and `P3` communicate using MPI_Intra0 (label `A`). They belong to the same local group of processes which are encapsulated within the intracommunicator. Processes from different groups, on the other hand, communicate using the global intercommunicator MPI_Inter0. Communication between process with rank `P1` from the local group and process with rank `P0` from the remote group is performed using the intercommunicator (label `B`). MPI point-to-point functions require both the sender and the receiver to know the rank of destination and source processes, respectively. When two processes communicate using an intercommunicator, `MPI_Send` and `MPI_Recv` take as parameter the rank of the destination or source process in the remote group.

MPI provides by default a global intracommunicator called `MPI_COMM_WORLD` that encapsulates every process of the parallel application started with the `mpirun` or `mpiexec` command.

The MPI-1 specification of the standard defined MPI applications as SPMD (Single Program Multiple Data) programs. The first specification only allowed to

MPI_COMM_SPAWN (command, argv, maxprocs, info, root,
comm, intercomm, error)

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to command (array of strings, significant only at root)
IN	maxprocs	maximum number of processes to start (integer, significant only at root)
IN	info	set of key-value pairs telling the runtime system where and how to start the processs (handle, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	intracommunicator containing group of spawning processes (handle)
OUT	intercom	intercommunicator between original group and the newly spawned group (handle)
OUT	error	one code per process (array of integer)

```
int MPI_Comm_spawn (const char *command, char *argv[],
          int maxprocs, MPI_Info info, int root, MPI_Comm comm,
          MPI_Comm *intercomm, int array_of_errcodes[])
```

Figure 4.7: Parameters of MPI's `MPI_Comm_spawn`.

start parallel applications using the `mpirun/mpiexec` command and specifying the number of processes of the application as a command line argument. MPI-2 introduced, among other advanced features such as parallel I/O and RMA (Remote Memory Access), the dynamic process management interface that allows MPI programs to spawn processes dynamically at runtime. The dynamic process management interface provides two functions to dynamically spawn MPI processes at runtime: `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. The first routine allows to spawn processes of the same binary with the same arguments, while the second one allows to spawn processes of multiple binaries with different sets of arguments. Figure 4.7 shows the list of parameters of the spawning function `MPI_Comm_spawn`. This function takes as input the intracommunicator of the spawning group of processes and returns an intercommunicator which encapsulates parent and child groups of processes. The spawning function is blocking and collective over all processes in the parent intracommunicator and one of the processes in this group participates in the routine as root.

MPI-2 and MPI-3 programs may have several instances of the global intracommunicator `MPI_COMM_WORLD`. All processes started with `mpirun/mpiexec` are encapsulated within an instance of `MPI_COMM_WORLD`. Furthermore, those processes created within the same call to the spawning function are encapsulated within a separate instance of `MPI_COMM_WORLD`. Separate calls to the spawning function result in multiple instances of the global intracommunicator. The current implementation of `MPI_COMM_WORLD` has two major drawbacks that collide with the dynamic process management interface: (1) those processes encapsulated within the same instance of `MPI_COMM_WORLD` cannot release resources and finalize their execution until every process in the intracommunicator has made a call to the finalize routine `MPI_Finalize`, and (2) an instance of the global intracommunicator cannot be deallocated at runtime. These represent a problem for process removal in MPI dynamic applications:

- Those processes started with the `mpirun` or `mpiexec` command cannot be removed individually at runtime. They only can be removed from the program when all of them invoke `MPI_Finalize`.
- Multiple processes spawned in the same call to `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` cannot be removed individually at runtime. They only can be removed from the program when all of them invoke `MPI_Finalize`.

While the first issue is a hard constraint imposed by the MPI standard and its implementations, the second issue can be solved by spawning new processes in separate calls to the spawning functions. This will allow later to remove each process separately when the malleability policy or the programmer required it.

Communication with spawned processes in MPI also presents several issues. `MPI_COMM_WORLD` is static and newly spawned processes cannot be added or removed. The static singularity of MPI communicators is a good match to rigid and moldable applications in which the number of processes and resources allocated remain fixed during the program execution. However, malleable and evolving applications are not currently fully-supported by the MPI standard. Theses applications require an advanced approach to communicate with dynamic processes at runtime. Graham *et al.* [GK09] proposed an approach to add support for fully-dynamic communicators to the MPI standard. In their proposal, an MPI communicator can change its size dynamically—grow and shrink. However, the MPI standard does not consider dynamic communicators.

The programmer has to explicitly manage communications with dynamic processes using the available alternatives offered by the MPI standard. MPI introduced a set of routines to dynamically allocate (`MPI_Comm_create`) and deallocate (`MPI_Comm_free/MPI_Comm_disconnect`) new communicators in the program. These routines allow to create new communicators that enable communication between spawning and spawned processes. There are two alternative methods for communicating with dynamic processes in MPI [RWS11]:

- **Many communicator method**. This method relies on an array of communicators, each of them point-to-point communicators between each pair of processes in the program. These point-to-point communicators are created using `MPI_Comm_connect` and `MPI_Comm_accept` routines, that enable client/server communication between two groups of processes. Both groups may be composed of one or more processes. Each time `MPI_Comm_spawn` is invoked, a set of point-to-point communicators are created between each of the spawning and spawned processes. The major drawbacks of this method are that it does not scale well when the number of processes is very large and, more importantly, it does not allow global collective operations between all processes. Figure 4.8 illustrates the many communicator method in a program with three processes. Each process p uses a dynamic array that stores in each position i a point-to-point communicator $p > i$ that communicates processes with ranks p and i. Each time a reconfiguring action is carried out every process reallocates the array to adapt it to the new number of processes.

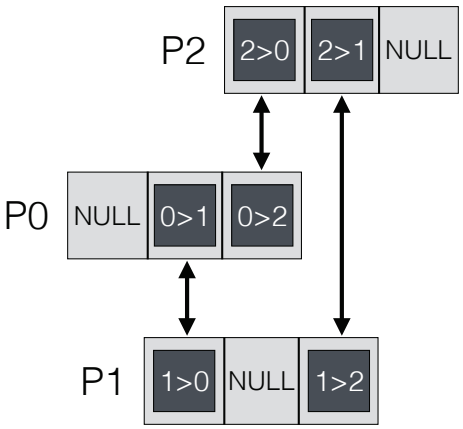


Figure 4.8: Example of communication between dynamic processes using the *many communicator* method

- **Merge method**. This method uses the merge `MPI_Intercomm_merge` routine to merge parent and child intercommunicator into an intracommunicator. The merged communicator can be successively used to spawn and communicate new processes in multiple calls to the spawning routine, thus enabling collective communications with dynamic processes. A drawback of this method is that `MPI_Comm_spawn` and `MPI_Intercomm_merge` are blocking and collective over all process in the communicators, which may degrade the performance of the reconfiguration. When using the merge method all processes in the parent intracommunicator act as parent processes.

Although the work in [RWS11] demonstrates that both methods performed similarly in terms of performance, the merge method enables collective communication operations between all processes. These are essential to the majority of MPI-based applications. In FLEX-MPI we use the merge method to enable global communications in malleable MPI applications. Due to the fact that global, static intracommunicator `MPI_COMM_WORLD` cannot be used in the merge operation, we use a global intracommunicator called `XMPI_COMM_WORLD` that enables communication between processes in FLEX-MPI programs using the merge method. This communicator is internally managed by the dynamic process management component of FLEX-MPI. However, it requries that every occurrence of `MPI_COMM_WORLD` in the program to be replaced by `XMPI_COMM_WORLD`. FLEX-MPI rebuilds the `XMPI_COMM_WORLD` intracommunicator in a user-transparent way each time a reconfiguring action is carried out.

4.3.2 Spawn

FLEX-MPI uses the dynamic process management interface of MPI to spawn new processes in malleable programs. Figure 4.9 shows the parameters of the FLEX-MPI's low-level interface `XMPI_Spawn`. This routine allows to spawn dynamic processes at runtime and enables communication between processes of the FLEX-MPI application through the `XMPI_COMM_WORLD` communicator. `XMPI_Spawn` takes as parameters the

XMPI_SPAWN (command, argv, nprocs, info, node_class)

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to command (array of strings, significant only at root)
IN	nprocs	number of processes to start (integer, significant only at root)
IN	info	set of key-value pairs (handle, significant only at root)
IN	node_class	set of identifiers telling the scheduler where to spawn the processes (array of strings, significant only at root)

```
int XMPI_Spawn (const char *command, char *argv[], int nprocs,
MPI_Info info, char *node_class[])
```

Figure 4.9: Parameters of FLEX-MPI's `XMPI_Spawn`.

name of the program and the number of new processes to be started, the arguments to program, and a set of key-value pairs. In addition, the spawning routine takes as optional input the *node_ class* parameter. The scheduler component of FLEX-MPI uses this parameter to decide where to spawn new processes. Resource management systems—i.e. Torque resource manager [Sta06]—use identifiers that allow cluster administrators to create classes of hosts within the cluster. Node identifiers allow cluster users to request resources by class instead of using the name of a particular host, which allows to apply more efficient scheduling policies in the cluster. The *node_ class* parameter allows the programmer to specify a node class for each new spawned process. Each node class value must match at least one of the node identifiers in the cluster. The FLEX-MPI scheduler then uses the node class to allocate the new process into a particular host which node class matches the host identifier used by the RMS.

The spawning function creates each of the new processes through a separate call to `MPI_Comm_spawn`, thus allowing FLEX-MPI to remove later each dynamic process individually. Those processes which are dynamically spawned and removed at runtime are called *dynamic* processes. However, those processes started with the `mpirun/mpiexec` command are static and cannot be removed at runtime. From now on we refer to this set of processes as the *initial* set of processes.

Algorithm 1 shows the implementation of the FLEX-MPI's spawning function. The algorithm is iterative and the same procedure is repeated for each of the spawned *nprocs* processes (line `1`). First, the algorithm decides which nodes of the cluster will allocate the new processes depending on the *node_ class* parameter. If provided (lines `2-5`), the algorithm uses this parameter to schedule the new process to a particular host. Otherwise, the algorithm delegates the scheduling task on the MPICH process manager. The FLEX-MPI's scheduler will be covered in detail in Section 4.4. The FLEX-MPI's scheduler uses the node class argument to allocate the new process to a cluster node whose class corresponds to the class provided by the programmer (line `3`). `MPI_Info_set` allows to set the info key to the value of the host returned by the FLEX-MPI scheduler (line `4`). The info key is provided to the MPI's spawning function `MPI_Comm_spawn` (line `6`) which creates the new process into the host *new_ host* and returns an intercommunicator (*intercomm*) between the spawning pro-

Algorithm 1 Implementation of FLEX-MPI's `XMPI_Spawn`.

```
1: for n = 1 to nprocs do
2:   if node_class then
3:     new_host = Schedule_spawn (node_class[n]);
4:     MPI_Info_set (&info, "host", new_host);
5:   end if
6:   MPI_Comm_spawn (bin, argv, 1, info, 0, XMPI_COMM_WORLD, &in-
     tercomm, &mpierr);
7:   MPI_Intercomm_merge (intercomm, 0, &XMPI_COMM_WORLD);
8:   MPI_Disconnect (&intercomm);
9: end for
```

Algorithm 2 Implementation of FLEX-MPI's `XMPI_Spawn` for the creation of n simultaneous processes.

```
1: if node_class then
2:   new_hosts = Schedule_spawn_multiple (node_class, nprocs);
3:   for n = 1 to nprocs do
4:     MPI_Info_set (info[n], "host", new_hosts[n]);
5:   end for
6: end if
7: MPI_Comm_spawn (bin, argv, nprocs, info, 0, XMPI_COMM_WORLD,
   &intercomm, mpierr);
8: MPI_Intercomm_merge (intercomm, 0, &XMPI_COMM_WORLD);
9: MPI_Disconnect (&intercomm);
```

cesses encapsulated within the currently allocated instance of `XMPI_COMM_WORLD` and the spawned processes. Then we use the merging function `MPI_Intercomm_merge` to create a new, `XMPI_COMM_WORLD` intracommunicator that encapsulates both spawning and spawned processes (line `7`). Finally we disconnect the intercommunicator since it will not be used anymore in the execution (line `8`).

The design choice described above 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. For this reason, the current implementation of FLEX-MPI also supports the creation of $n > 1$ simultaneous processes. 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. Algorithm 2 shows the implementation of the FLEX-MPI's spawning function for the creation of n simultaneous processes via an individual call to `MPI_Comm_spawn`.

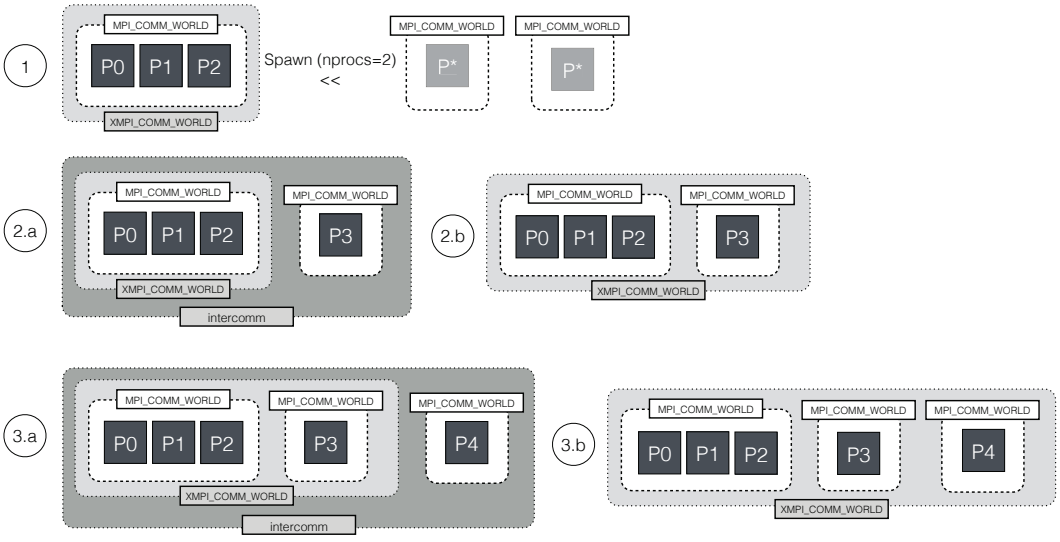


Figure 4.10: Actions of the dynamic process management functionality at process creation.

Figure 4.10 illustrates the behavior of the dynamic process management component when two dynamic processes (`P3,P4`) are added to an FLEX-MPI program already running on an initial set of processes (`P0-2`) (step `1`). Each of the new processes is spawned individually. This makes each process have its own (`MPI_COMM_WORLD`) intracommunicator. FLEX-MPI spawns the first new process `P3` (step `2.a`), then merges the intercommunicator `intercomm` into a new instance of `XMPI_COMM_WORLD`, and deallocates the intercommunicator (step `2.b`). Likewise, the dynamic process management spawns process `P4` (step `3.a`) and enables communication using the global intracommunicator `XMPI_COMM_WORLD` (step `3.b`).

4.3.3 Remove

An MPI process finalizes execution by invoking `MPI_Finalize`. The finalize function terminates the MPI execution environment and allows the process to release resources. `MPI_Finalize` is collective and blocking over all connected processes. This implies that the process has to be previously disconnected from every communicator (i.e. `XMPI_COMM_WORLD`) before invoking `MPI_Finalize`. Nevertheless, the `MPI_COMM_WORLD` is automatically deallocated by the finalize routine when all processes in the communicator reach `MPI_Finalize`.

Due to these limitations, (1) any of the processes in the initial set can not be removed at runtime, and (2) only those processes spawned through a separate call to `MPI_Comm_spawn` can be removed at runtime. The process removal operation implies the dynamic process to be disconnected from `XMPI_COMM_WORLD`. This allows the process to leave the iterative section of the program and finish execution by invoking `MPI_Finalize`.

Figure 4.11 shows the parameters of the FLEX-MPI's low-level interface `XMPI_Remove`. This routine allows the programmer to disconnect dynamic processes that

XMPI_REMOVE (nprocs, node_class)

IN	nprocs	number of processes to remove (integer, significant only at root)
IN	node_class	set of identifiers telling the scheduler from where remove the processes (array of strings, significant only at root)

```
int XMPI_Remove (int nprocs, char *node_class[])
```

Figure 4.11: Parameters of FLEX-MPI's `XMPI_Remove`.

can be removed from the MPI program at runtime. `XMPI_Remove` takes as input parameters the number of processes to remove and the node class for each removed process. The number of processes to remove must be greater than or equal to the number of dynamic processes running in the application. The scheduler component uses the node class to decide the host from where the process will be removed.

Algorithm 3 shows the implementation of the FLEX-MPI's removing function. The algorithm uses the FLEX-MPI scheduler to obtain the rank of each process that will be removed (lines 1-7) depending on the *node_class* parameter. If provided (line 3), the scheduler retrieves the rank of a process running on a cluster node whose class corresponds to the value of the class in the parameter. Otherwise (line 5), FLEX-MPI randomly chooses the rank of a running process (except root). These ranks are into an array (*remv_rank*). Each process uses a flag word that stores a value that indicates the status of the process in the program—running or removed. `XMPI_Sched_remove` indicates the removed processes to change the flag value when they are chosen to leave the program execution. FLEX-MPI provides a routine called `XMPI_Get_process_status` that allows processes to access the status flag. Using this functionality each process can evaluate its status when the removing function returns and, in the proper case, leave the iterative section of the code and finish execution. The next step in the algorithm consists of creating a group with the processes encapsulated within `XMPI_COMM_WORLD` (line 8), duplicating the intracommunicator (line 9), then deallocating it (line 10). This allows FLEX-MPI to create a new instance of `XMPI_COMM_WORLD` that encapsulates only those processes that continue in the program execution. The group *running_group* is a group of processes that excludes those processes that have been scheduled to remove (line 11). Finally, FLEX-MPI allocates the new intracommunicator (line 12).

Figure 4.12 illustrates the behavior of the dynamic process management functionality when process `P4` is removed from the FLEX-MPI application (step 1). First, the current instance of `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_excl` to exclude `P4`, and a new intracommunicator `XMPI_COMM_WORLD` is allocated for this group. The status flag of `P4` is set to XMPI_REMOVE and the process finishes its execution by calling `MPI_Finalize` (step 3).

Algorithm 3 Implementation of FLEX-MPI's `XMPI_Remove`.

```
1: for n = 1 to nprocs do
2:    if node_class then
3:       remv_rank[n] = Schedule_remove (node_class[n]);
4:    else
5:       remv_rank[n] = PickRandomRank (nprocs-n);
6:    end if
7: end for
8: MPI_Comm_group (XMPI_COMM_WORLD, &group);
9: MPI_Comm_dup (XMPI_COMM_WORLD, &dupcomm);
10: MPI_Comm_disconnect (XMPI_COMM_WORLD);
11: MPI_Group_excl (group, 1, remv_rank, &running_group);
12: MPI_Comm_create (dupcomm,running_group,&XMPI_COMM_WORLD);
```

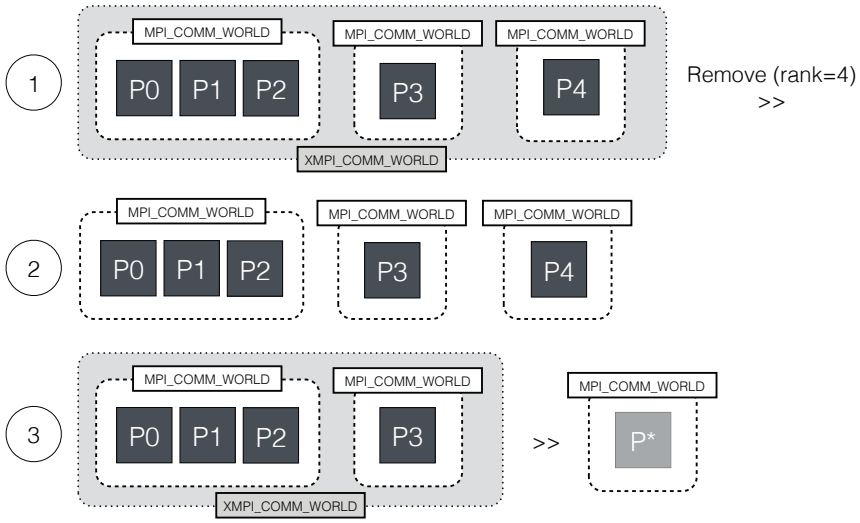


Figure 4.12: Actions of the dynamic process management functionality at process removal.

4.3.4 Malleable MPI programs using the FLEX-MPI low-level API

This section shows how to use the FLEX-MPI's low-level API to create malleable programs. Figure 4.13 shows the parallel code of a non-iterative SPMD program instrumented with the FLEX-MPI low-level API to spawn and remove dynamic processes. The figure on the left corresponds to the code executed by the initial set of processes, the figure on the right shows the code executed by the processes that are dynamically spawned and removed. We assume that the programmer initially launches the program to run with three processes via the command line (`mpirun/mpiexec`), spawns three additional processes (`XMPI_Spawn`), then removes one of the dynamic processes running on a dual class node (`XMPI_Remove`). Note that only those processes in the initial set participate in the spawning action. However, the removing action involves every process in the execution. After the removing action every process in

Program: flex_static.c

```
MPI_Init (&argv, &argc);

MPI_Comm_rank (XMPI_COMM_WORLD, &rank);
MPI_Comm_size (XMPI_COMM_WORLD, &size);

XMPI_Spawn ("flex_dynamic", argv, 3, NULL, NULL);

MPI_Allreduce (&sendbuf, &recvbuf, count, MPI_INT,
               MPI_MAX, 0, XMPI_COMM_WORLD);

XMPI_Remove (1, NULL);

MPI_Allgather (&sendbuf, scount, MPI_INT, &recvbuf, rcount,
               MPI_INT, XMPI_COMM_WORLD);

MPI_Finalize ();
```

Program: flex_dynamic.c

```
MPI_Init (&argv, &argc);

MPI_Comm_rank (XMPI_COMM_WORLD, &rank);
MPI_Comm_size (XMPI_COMM_WORLD, &size);

MPI_Allreduce (&sendbuf, &recvbuf, count, MPI_INT,
               MPI_MAX, 0, XMPI_COMM_WORLD);

XMPI_Remove (1, NULL);

status = XMPI_Get_process_status ();

if (status != XMPI_REMOVED)
    MPI_Allgather (&sendbuf, scount, MPI_INT,
                   &recvbuf, rcount, MPI_INT,
                   XMPI_COMM_WORLD);

MPI_Finalize ();
```

Figure 4.13: Malleable program instrumented with FLEX-MPI low-level interfaces.

the dynamic side evaluates its status (`XMPI_Get_process_status`) and those processes flagged as removed reach the finalize function (`MPI_Finalize`). On the other hand, the remaining processes perform a collective operation (`MPI_Allgather`), then finish their execution.

4.4 Process scheduler

The process scheduler component of FLEX-MPI is responsible for scheduling the creation and removal of dynamic processes in the system. The MPI standard delegates on the implementation the task of process scheduling and mapping of processes to processors. For instance, the MPICH implementation features the *Hydra* process manager that allows the programmer to decide the process scheduling by using a set of command-line options when the programmer launches the MPI program via `mpirun/mpiexec`. Hydra features several scheduling policies as round-robin, or process-core binding.

However, the Hydra process manager does not provide such flexibility when dealing with dynamic processes. Hydra uses a round-robin policy for the spawned processes via `MPI_Comm_spawn`. It distributes an equal number of newly spawned processes among the available hosts, but the process manager does not track information about the current number of running processes in each host or the last host in which a dynamic process was spawned in a previous call to `MPI_Comm_spawn`. This results in oversubscribed nodes and performance degradation when dealing with dynamic processes.

The MPI standard does define a user-level mechanism that allows the programmer to specify the host that will allocate the spawned process [CPM+06, CPP+07]. This mechanism the programmer to have control over the scheduling of dynamic processes and their mapping to processors by setting the `MPI_Info` argument of `MPI_Comm_spawn` to the host name of the compute node where the new process needs to be allocated. Our scheduler (1) uses the described mechanism to schedule new

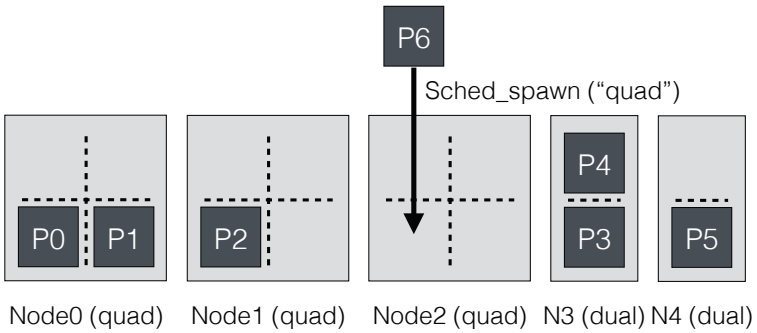


Figure 4.14: Example of process scheduling at process creation with the *balance all* policy.

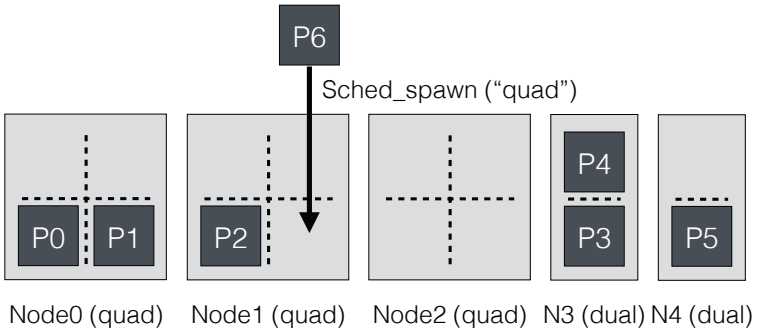


Figure 4.15: Example of process scheduling at process creation with the *balance occupied* policy.

processes, (2) tracks the number of MPI running processes on each compute node, and (3) avoids node oversubscription by balancing the processes between nodes of the same class.

FLEX-MPI's process scheduler features two scheduling policies for dynamic processes: *balance all*, and *balance occupied*. The first one balances the number of processes between the available compute nodes of the same node class. The second policy balances the number of processes between those nodes of the same class where there is already running at least one MPI process of the malleable application. The first policy has as a primary goal to minimize the impact of resource sharing—memory, buses, network devices, etc.—in the application performance. The second one has as objective to save resources by minimizing the number of compute nodes used by the application. The programmer can set the scheduling policy as a command-line argument to FLEX-MPI.

4.4.1 Spawn

Figures 4.14 and 4.15 illustrate the behavior of the scheduling policies at process creation in FLEX-MPI. The compute cluster consists of five nodes of two different classes, *quad*—equipped with quad-core processors—and *dual*—equipped with

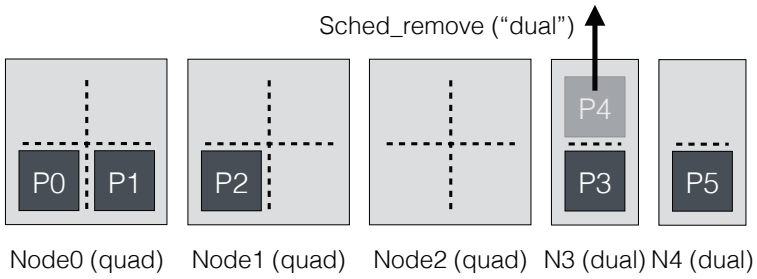


Figure 4.16: Example of process scheduling at process removal with the *balance all* policy.

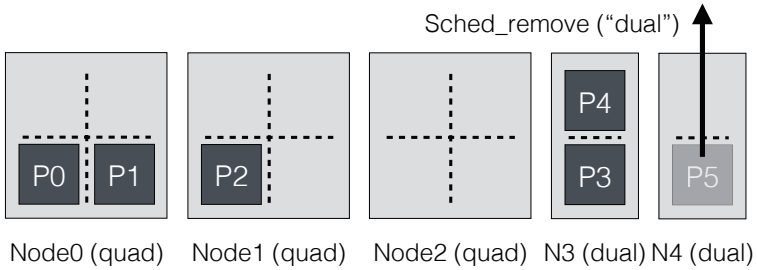


Figure 4.17: Example of process scheduling at process removal with the *balance occupied* policy.

dual-core processors. The malleable application is running with five processes and it spawns a new dynamic process that needs to be allocated into a quad node. Following the *balance all* policy (Figure 4.14), *Node2* allocates the new processes in order to balance the number of running processes between all nodes of the quad class. Otherwise, under the *balance occupied* policy (Figure 4.15), *Node1* allocates the process in order to balance the number of running processes between those quad class nodes that have at least one MPI process running (*Node0* and *Node1*).

4.4.2 Remove

Figures 4.16 and 4.17 illustrate the behavior of the scheduling policies at process removal in FLEX-MPI. In this example a dynamic process is removed from a dual class node. Following the *balance all* policy, the process will be removed from *Node3*, thus balancing the number of processes running on dual nodes (Figure 4.16). Otherwise, under the *balance occupied* policy (Figure 4.17), the dynamic process is removed from *Node4* in order to minimize the number of compute nodes used by the application.

4.5 Data redistribution

The data redistribution component provides a simple mechanism to automatically move data between processes that operate on multiple data structures through a

$$A = \begin{bmatrix} 4 & 3 & 0 & 0 \\ 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 8 \end{bmatrix}$$

Figure 4.18: Sparse matrix A.

```
IA = [0 0 1 2 3 3]
JA = [0 1 2 3 0 3]
 A = [4 2 3 9 1 8]
```

Figure 4.19: Representation of sparse matrix A in coordinate scheme.

single operation. FLEX-MPI allows automatic redistribution of dense and sparse data structures with one-dimensional decomposition and block partitioning.

4.5.1 Storage formats

The data redistribution component handles both one-dimensional (e.g. vectors) and two-dimensional (e.g. matrices) data structures, that can be either dense or sparse. FLEX-MPI assumes that vectors and dense matrices are stored in one-dimensional arrays, while sparse matrices use a compressed storage format. FLEX-MPI can redistribute vectors and dense matrices stored in arrays, and sparse matrices stored in CSC (*Compressed Sparse Column*) or CSR (*Compressed Sparse Row*) format. These compressed formats take advantage of the sparse structure of the matrix to reduce the footprint of the matrix in main memory and support efficient matrix operations.

Figure 4.18 shows a sparse matrix populated primarily with zeros. The basic approach to represent sparse matrices in memory is the *coordinate scheme*. The coordinate scheme uses three one-dimensional arrays of length number of non-zero entries (nnz) of the matrix. Every nnz element a_{ij} is represented by a triple (i, j, a_{ij}), where i corresponds to the row index, j is the column index, and a_{ij} is the numerical value enclosed in the matrix entry. The arrays are usually sorted by row index, then column index to improve access time. However, row-wise and column-wise operations on data structures stored in coordinate scheme require very large searching operations. The first array IA stores the values of i, the second array JA stores the values of j, and the third array A stores the values of a_{ij}. Figure 4.19 shows the representation of matrix A in coordinate scheme. For instance, the triple $(2, 3, 9)$ represents that the value of $a_{2,3}$ is 9.

Compressed formats preserve the high-performance for accessing operations and also improve the performance of row-wise and column-wise operations by reducing searching times. These formats are highly efficient for arithmetic operations—i.e. matrix-vector product—and column slicing. Compressed formats use a representation

```
IA = [0 2 3 4 6]
JA = [0 1 2 3 0 3]
 A = [4 2 3 9 1 8]
```

Figure 4.20: Representation of sparse matrix A in CSR format.

structure in three arrays IA, JA, and A. A and JA arrays are equivalent to the coordinate scheme and their length is nnz. In contrast, in CSR IA is an array of length number of rows of the matrix plus one. $IA[i]$ contains a pointer to the index of JA that stores the occurrence of the first nnz element of row i. The number of nnz of row i is calculated by $IA[i+1] - IA[i]$. CSC format is similar to CSR except IA stores row indexes and JA contains column pointers. Figure 4.20 shows the representation of matrix A in CSR format.

4.5.2 Domain decomposition

In parallel computing, decomposition or partitioning refers to the process of breaking down the problem into smaller partitions that can be distributed to parallel tasks. The basic approach to data decomposition among parallel tasks is domain decomposition. The most common domain decomposition is the one-dimensional domain decomposition using a block partitioning scheme [KGGK94]. Block-based, one-dimensional domain decomposition corresponds to divide the data structure into sets of rows or columns. Then, each portion of the domain is assigned to a parallel task that operate on the data partition. Figure 4.21 illustrates the one-dimensional, column-wise domain decomposition of dense (a) and sparse (b) data structures. One-dimensional decomposition is performed by dividing the number of columns in the matrix (N) by the number of processing elements (p).

A large proportion of SPMD applications usually operate on multiple data structures. However, the vast majority of these applications use the same domain decomposition for all of their data structures. That is, every process computes the same partition in each data structure. FLEX-MPI handles parallel applications that operate on multiple data structure with a global domain decomposition.

In parallel applications, the portion of the domain assigned to each process is usually expressed as a combination of a count—the number of rows or columns assigned to the process—and a displacement. The displacement is the number of rows or columns that represents the offset of the data partition relative to the starting partition of the structure. FLEX-MPI provides a routine that allows processes to retrieve the data partition assigned to the process, that changes as result of a data redistribution operation. Figure 4.22 shows the parameters of FLEX-MPI's `XMPI_Get_wsize`. This routine takes as input the process rank, number of processes, and number of rows or columns of the data structure and returns the count and displacement of the portion of the domain that is currently assigned to the process.

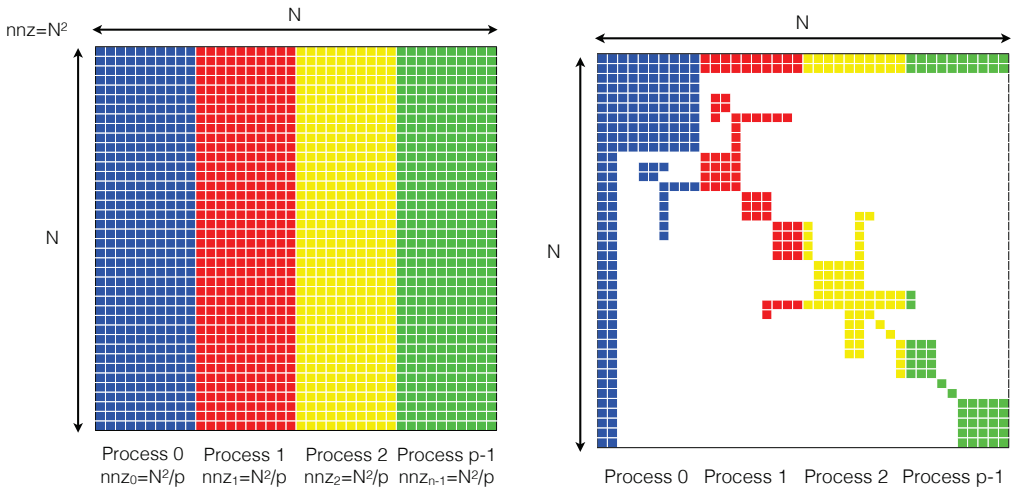


Figure 4.21: One-dimensional data decomposition with block partitioning of dense (a) and sparse (b) data structures.

XMPI_GET_WSIZE (rank, nprocs, count, displ)

IN	rank	rank of process (integer)
IN	nprocs	number of processes (integer)
OUT	count	number of rows/cols (integer)
OUT	displ	displacement of data (integer)

```
int XMPI_Get_wsize (int rank, int nprocs, int *count, int *displ)
```

Figure 4.22: Parameters of FLEX-MPI's `XMPI_Get_wsize`.

4.5.3 Data redistribution functionality

The data redistribution functionality provides a set of low-level interfaces to register data structures and redistribute data. Every data structure susceptible to be redistributed as a result of a load balancing operation or a dynamic reconfiguring action must be registered in the source code of the program.

Registering routines allow FLEX-MPI to store a pointer to the memory address of the data structure and the total size of the data, among other properties. This pointer is then used by the data redistribution routine to access and redistribute data in a user-transparent way. FLEX-MPI uses a *struct* type—or in other programming languages, a record—that holds the properties of the redistributable data structure. The data redistribution component uses MPI standard messages to efficiently move data between processes. Registering routines must be also used to register data structures in FLEX-MPI programs that use the high-level API.

FLEX-MPI focus on MPI applications with distributed data structures in which data partitions are not replicated among processes. However, certain classes of par-

XMPI_REGISTER_VECTOR (address, size, datatype, scheme)

IN	address	pointer to the data structure (void)
IN	size	size of data (integer)
IN	datatype	datatype of elements (handle)
IN	scheme	data distribution scheme (handle)

```
int XMPI_Register_vector (void *address, int size,
            MPI_Datatype datatype, XMPI_Scheme scheme)
```

Figure 4.23: Parameters of FLEX-MPI's `XMPI_Register_vector`.

XMPI_REGISTER_DENSE (address, size, datatype, scheme)

IN	address	pointer to the data structure (void)
IN	size	size of data (integer)
IN	datatype	datatype of elements (handle)
IN	scheme	distribution scheme (handle)

```
int XMPI_Register_dense (void *address, int size,
            MPI_Datatype datatype, XMPI_Scheme scheme)
```

Figure 4.24: Parameters of FLEX-MPI's `XMPI_Register_dense`.

allel programs commonly operate on data structures that are replicated among processes. Data replication implies that every process stores a whole copy of the data structure. For instance, in parallel linear system solvers that compute the solution of $Ax = b$, vector x stores the approximation to the solution calculated in each program iteration. Each process iteratively updates the partition of the vector that corresponds to the portion of the domain assigned to the process. However, every process needs the results calculated by all processes to compute the next program iteration. For this reason, FLEX-MPI supports redistribution of fully distributed—disjoint partitions—and replicated data structures.

FLEX-MPI's `XMPI_Register_vector` allows the programmer to register a vector data structure. Figure 4.23 shows the parameters of this routine. This function takes as input a user-defined string that identifies the data structure, a pointer to the memory address of the array that stores the data structure, the data type, the size of data in number of rows or columns, and the data distribution scheme. The data distribution scheme takes as possible values either `XMPI_DISTRIBUTED` for distributed data structures or `XMPI_REPLICATED` for replicated data distribution. This function assumes that the nnz of the data structure corresponds to the value of the size parameter.

FLEX-MPI's low-level interface `XMPI_Register_dense` allows the programmer to register dense, 2D data structures—square matrices. Figure 4.24 shows the parameters of the registering routine, which interface is similar to `XMPI_Register_vector`. `XMPI_Register_dense` assumes that the nnz of the dense data structure corresponds to the squared value of the size parameter.

XMPI_REGISTER_SPARSE (address_ia, address_ja, address_a, size, datatype, scheme)

IN	address_ia	pointer to the IA array (integer)
IN	address_ja	pointer to the JA array (integer)
IN	address_a	pointer to the A array (void)
IN	size	size of data (integer)
IN	datatype	datatype of elements (MPI_Datatype)
IN	scheme	distribution scheme (handle)

```
int XMPI_Register_sparse (int *address_ia, int *address_ja, int
*address_a, int size, MPI_Datatype datatype, XMPI_Scheme scheme)
```

Figure 4.25: Parameters of FLEX-MPI's `XMPI_Register_sparse`.

XMPI_REDISTRIBUTE_DATA (scount, sdispl, rcount, rdispl)

IN	scount	number of rows/cols of current data (integer)
IN	sdispl	displacement of current data (integer)
IN	rcount	number of rows/cols of new data (integer)
IN	rdispl	displacement of new data (integer)

```
int XMPI_Redistribute_data (int scount, int sdispl, int rcount,
                            int rdispl)
```

Figure 4.26: Parameters of FLEX-MPI's `XMPI_Redistribute_data`.

`XMPI_Register_sparse` is the low-level interface provided by FLEX-MPI to register sparse data structures. Figure 4.24 shows the parameters of this routine. In addition to the user-defined identifier, this function receives as input the addresses of the three arrays (IA, JA, A) that store the sparse matrix, the data type of the matrix values stored in A, the coefficient represented as the number of rows or columns, the number of nnz of the matrix, and the data distribution scheme.

Registering functions must be placed in the code after the memory allocation of each data structure susceptible to be redistributed, thus allowing FLEX-MPI to access the address of each redistributable data structure. Once the data structures have been registered, FLEX-MPI allows automatic redistribution of multiple data structures. FLEX-MPI's `XMPI_Redistribute_data` provides the data redistribution functionality in FLEX-MPI applications. This function allows the programmer to indicate the current domain partition assigned to each process and the new domain partition that each process will receive as result of the redistribution operation. Then, FLEX-MPI uses MPI messages to move data partitions from old to new owners in a user-transparent way. Figure 4.26 shows the parameters of `XMPI_Redistribute_data`. This routine takes as input arguments the count ($scount$) and displacement ($sdispl$) of the currently assigned domain partition and the count ($rcount$) and displacement ($rdispl$) that indicate the new partition of the domain assigned to the process.

Algorithm 4 shows the implementation of FLEX-MPI's `XMPI_Redistribute_data`. This function performs data redistribution of every

Algorithm 4 Implementation of FLEX-MPI's `XMPI_Redistribute_data`.

```
1: for n = 1 to ndata_structures do
2:    (scounts, sdispls, rcounts, rdispls) ← calculatePartitions (scount, sdispl,
      rcount, rdispl);
3:    retrieveAddress (id[n], &old_partition[n], &scheme[n]);
4:    if scheme[n] == XMPI_DISTRIBUTED then
5:       MPI_Alltoallv (scounts, sdispls, &old_partition[n], rcounts, rdispls,
         &new_partition[n]);
6:    else if scheme[n] == XMPI_REPLICATED then
7:       MPI_Allgatherv (scounts, sdispls, &old_partition[n], rcounts, rdispls,
         &new_partition[n]);
8:    end if
9:    reassignPointer (&old_partition[n], &new_partition[n]);
10: end for
```

data structure that has been previously registered (line 1). Since this function receives as input the definition of the data partitions as a combination of the count and displacement, *calculatePartitions* (1) computes the number of elements of the data partition, and (2) returns four arrays that contain the number of elements (*scounts*) and displacements (*sdispls*) of every data partition assigned to every process and the number of elements (*rcounts*) and displacements (*rdispls*) of every new data partition requested by every process (line 2). Then, *retrieveAddress* uses the user-defined identifier (*id*[*n*]) to retrieve the memory address (*old_partition*) and the data distribution scheme (*scheme*[*n*]) of the data structure (line 3). Depending on the distribution scheme (line 4) the algorithm performs either `MPI_Alltoallv` (line 5) or `MPI_Allgatherv` (line 7). `MPI_Alltoallv` allows to move data from all to all processes, which is useful for redistributing distributed data structures. `MPI_Allgatherv` gathers data from all processes and then delivers the combined data to all processes, which is useful for data redistribution of replicated data structures. The algorithm shows a simplified version of the actual implementation of `XMPI_Redistribute_data`. Since sparse data structures are stored in three arrays, redistributing sparse data implies to perform three MPI communication operations. `XMPI_Redistribute_data` considers the type of the data structure—dense or sparse—in order to perform the appropriate redistribution. Finally, we reassign the old partition pointer to point to the address of the new partition, thus enabling the process to access the new data partition (line 9).

4.6 Summary

In this section we describe the low-level functionalities of FLEX-MPI. These functionalities are encapsulated in the process and data management layer of the FLEX-MPI library. The monitoring component provides the performance monitoring and communication profiling functionalities by means of hardware performance counters and PMPI. The dynamic process management component of FLEX-MPI supports

the creation and removal of dynamic processes in malleable applications and also handles communications between processes. FLEX-MPI uses a global communicator that enables communication between static and dynamic processes in malleable programs. The process scheduler is responsible for supporting the scheduling of dynamic processes in the system taking into account the current availability of resources. The data redistribution component provides a mechanism that redistributes the data between processes. The major feature of the data redistribution functionality is that it redistributes multiple data structures through a single function call, thus minimizing modification of the source code of the application. We also describe the low-level API that provides access to low-level functionalities of the FLEX-MPI API.

Chapter 5

Dynamic Load Balancing and Performance-aware Malleability in MPI

5.1 Introduction

In this chapter we describe the design and implementation of the software components of the malleability logic layer of FLEX-MPI (Figure 5.1). These components provide the high-level functionalities of FLEX-MPI: dynamic load balancing, performance prediction, and performance-aware dynamic reconfiguration. This chapter also describes the high-level application programming interface that provides access to those functionalities of the malleability logic layer of FLEX-MPI.

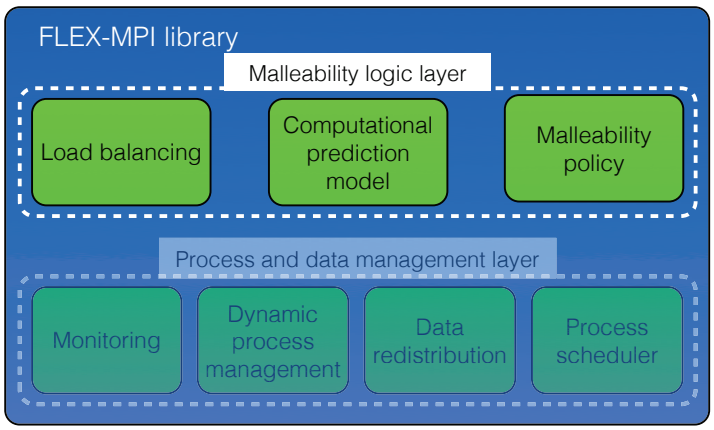


Figure 5.1: Malleability logic layer inside the FLEX-MPI library.

Table 5.1: FLEX-MPI high-level interfaces.

Interface	Description
`XMPI_Monitor_si_init`	Starts sampling interval monitoring
`XMPI_Eval_lbalance`	Evaluates dynamic load balancing
`XMPI_Get_data`	Retrieves data (significant to newly spawned processes)
`XMPI_Eval_reconfig`	Evaluates dynamic reconfiguration (including dynamic load balancing)

The functionalities of the malleability logic layer are high-level because adaptive actions are carried out without user intervention. The user does impose the performance objective of the application and the performance constraints that the parallel application must satisfy. Base on the performance criteria, FLEX-MPI's high-level functionalities decide to reconfigure the number of processes and rebalance the workload guided by the current performance data provided by the low-level monitoring component, the future application performance predicted by the computational prediction model, the malleability policy, and the availability of resources in the system. The malleability policy uses low-level dynamic process management and process scheduler components to change the number of processes at runtime. The dynamic load balancing component uses the low-level data redistribution component to redistribute the workload as result of a load balance operation.

Table 5.1 summarizes the high-level interfaces of the FLEX-MPI API that provide automatic, dynamic load balancing and performance-aware dynamic reconfiguration. Adaptive actions are performed at the end of every sampling interval—consisting of a fixed, user-defined number of consecutive program iterations that can be set as a command line argument to the FLEX-MPI program. At the end of every sampling interval FLEX-MPI evaluates the performance of the application based on the performance metrics gathered by the monitoring component during the sampling interval. By default, FLEX-MPI uses a sampling interval of 100 iterations. However, a different value can be set by the user as an option-argument to the FLEX-MPI program.

FLEX-MPI's high-level API provides a routine to monitor the performance of the application at the granularity of sampling interval. `XMPI_Monitor_si_init` function does not takes input parameters. This routine feeds the routines that evaluate adaptive actions—`XMPI_Eval_lbalance` and `XMPI_Eval_reconfig`—with the performance metrics collected during a given sampling interval. In addition, `XMPI_Get_data` allows dynamically spawned processes to retrieve their data partition from the current data owners using the data redistribution functionality. This function returns the current program iteration so dynamic processes can compute only the remaining iterations. For instance, a dynamic process spawned at iteration i will compute only $n - i$ iterations, where n is the total number of iterations of the program.

5.2 Dynamic load balancing

This section describes the high-level functionalities provided by the dynamic load balancing (DLB) component of FLEX-MPI. Load balancing is a major issue in parallel applications [LTW02] because it can have a huge impact on the overall performance of the program. In parallel applications, the slowest process determines the global performance of the program. Load balancing aims to maximize times while the processor is performing work and minimize processor idle times, thus maximizing application throughput and efficiency of resource usage.

The basic approach to load balance is to use an static approach and distribute equal amounts of work to each process in the parallel application—the workload distribution is fixed during program execution. Static load balancing algorithms do not rebalance the workload at runtime. However, today's parallel applications may face complex execution scenarios that require adaptive load balancing approaches. We identify five issues that lead to load imbalance in parallel applications under a static load balancing approach.

- **Irregular applications**. Irregular applications exhibit variable workloads and irregular communication patterns during execution. In these applications the amount of work each process will perform cannot be predicted, therefore it cannot be assigned statically at program start. Current high-performance computing systems are optimized for data locality and applications that exhibit regular computation and communication patterns [TF15]. In recent years, the emergence of applications with irregular patterns and unpredictable data accesses require optimized approaches for these issues [FTM+14].

- **Sparse data structures**. Static load balancing algorithms evenly distribute the same amount of data among processes using column-wise or row-wise decomposition. However, on sparse data structures non-zero elements are unevenly distributed. Sparse data structures result on unpredictable data accesses to memory, which incur on large overheads and degraded performance. This leads to load imbalance in parallel applications which workload depends on the number of non-zero elements of the data partition. Furthermore, for certain classes of problems their workload depends not only in the nnz but in the values of the elements of the data structure.

- **Dynamic parallel applications**. Reconfiguring actions involve changing the workload distribution of the application. Dynamic parallel applications require a mechanism to change the data distribution of the application and achieve load balance each time a reconfiguring action is performed. However, static approaches do not support parallel applications with dynamic creation and termination of processes.

- **Heterogeneous systems**. Parallel applications running on computing platforms which consist of non-identical machines require non-uniform workload distribution. The idea consists of assigning to each processing element (PE)

a workload partition that is proportional to its computing power. Static approaches require prior knowledge about the underlying architecture, which is not always feasible.

- **Non-dedicated systems**. In non-dedicated systems computing resources such as CPU, memory, or buses are shared between multiple user programs. This source of overhead can have a huge impact on the performance of the processor that cannot be predicted. Parallel applications that run on non-dedicated system require adaptive approaches that handle workload distribution to adapt it to variable overheads.

These issues highlight that static load balancing algorithms cannot cope with parallel applications that run on complex scenarios. Dynamic load balancing algorithms, on the other hand, allow to change the workload assignments at runtime, thus supporting dynamic parallel applications with irregular computation and communication patterns, applications that operate on sparse data structures, and complex execution scenarios such as heterogeneous and non-dedicated systems. The mechanism that detects load imbalance can be implemented using time measurements, then comparing execution times between processes.

FLEX-MPI implements a dynamic load balancing technique for SPMD applications that uses performance metrics collected by the monitoring functionality to detect load imbalance and make workload distribution decisions. One of the main advantages of this approach is that it does not require prior knowledge about the underlying architecture. The DLB technique implements a coordination-based method [Don98]. This means that all the processes synchronize at pre-defined times and evaluate the load balancing algorithm. We consider to balance the application workload (1) at the end of every sampling interval when the algorithm detects load imbalance and (2) after a reconfiguring action—either to spawn or remove processes.

Figure 5.2 illustrates the dynamic load balance of a FLEX-MPI application running on a heterogeneous dedicated system. Initially, the programmer distributes equal amounts of data to each process. This results inefficient because the slowest process (`P2`) forces other processes to wait idle until it finishes computation (a). At the end of the sampling interval, the DLB algorithm of FLEX-MPI detects load imbalance and computes a new workload distribution. The load balancing mechanism detects load imbalance when the difference between the execution times of fastest and slowest processes surpasses a pre-defined threshold. The new workload distribution results in minimizing wait times and leads to an overall performance improvement (b).

Figure 5.3 illustrates the data redistribution operation as result of the load balancing operation in the FLEX-MPI application. Initially, the data structure is distributed evenly (a), which leads to load imbalance because processes `P0` and `P2` are running on processing elements more powerful than those processing elements allocated to `P2` and `P3`. The new workload distribution computed by the load balancing algorithm implies a data redistribution operation between processes (b). After data redistribution the new data partitions are available to the processes (c).

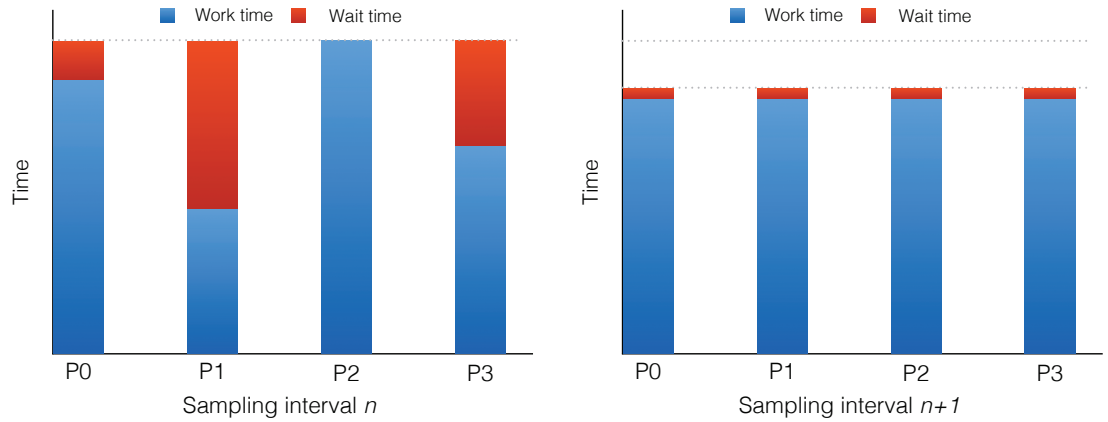


Figure 5.2: Dynamic load balance of a parallel application running on a heterogeneous dedicated system.

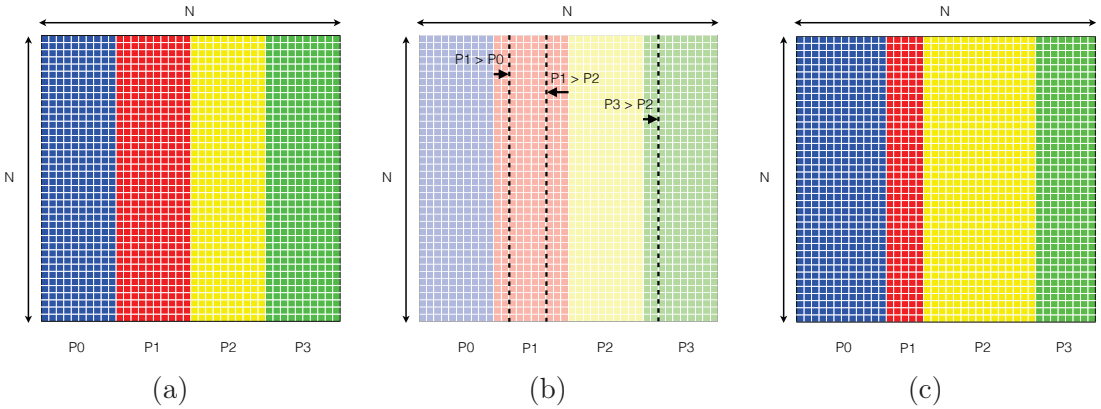


Figure 5.3: Workload distribution as result of a load balancing operation.

FLEX-MPI provides a high-level interface that allows the programmer to access to the dynamic load balancing functionality. Figure 5.4 shows the parameters of FLEX-MPI's high-level `XMPI_Eval_lbalance`. This routine takes as input the process rank, the number of running processes, the current iteration, the maximum number of iterations allowed to perform, and the count and displacement of the currently assigned data partition—these will be automatically updated by the routine with the count and displacement of the new data partition as result of the load balancing operation. Additionally, the load balancing routine takes as input an optional parameter (*weight*)—a double array of size the number of rows/columns of the data structure. This optional parameter is used by the load balancing policies of FLEX-MPI to make workload decisions, which are explained below.

FLEX-MPI implements two load balancing policies that optimize workload balance depending on the type of parallel application. The load balancing policy is decided by the user, who provides the policy as a command line argument to the

XMPI_EVAL_LBALANCE (rank, nprocs, it, maxit, count, displ, weight)

IN	rank	rank of the process (integer)
IN	nprocs	number of processes (integer)
IN	it	current program iteration (integer)
IN	maxit	maximum number of program iterations (integer)
INOUT	count	number of rows/cols of current data partition (integer)
INOUT	displ	displacement of current data partition (integer)
IN	weight	array indicating the column-wise or row-wise workload (double)

```
int XMPI_Eval_lbalance (int rank, int nprocs, int it, int maxit,
            int *count, int *displ, double *weight)
```

Figure 5.4: Parameters of FLEX-MPI's high-level `XMPI_Eval_lbalance`.

FLEX-MPI application. The first load balancing policy—`XMPI_LB_NNZ`—is suited to parallel applications in which their workload depends on the number of non-zero elements of the data partition. This policy balances the workload by assigning to every PE a data partition that contains the same number of non-zero elements. In this case the *weight* array is unused. The second policy—`XMPI_LB_WEIGHT`—is suited to parallel applications that operate on data structures in which their workload depends on the values of the elements of the data structure. In this case the user must provide the *weight* array. Each element of the array indicates the workload associated to the row-wise or column-wise entries of the data structure.

Figure 5.5 illustrates an example of the load balancing of a data structure (a) using different load balancing policies to balance the workload depending on (b) the *nnz* and (c) the row-size or column-wise weights of the data structure. The column-wise weights are calculated as follows: the array stores the number of positive column elements (Figure 5.5 (a)). If the process workload depends on the number of non-zero elements the load balancing algorithm distributes to each process a data partition that contains the same number of *nnz* (Figure 5.5 (b)). Otherwise, if the process workload depends on the values of the column-wise entries (i.e. the application computes a function with the positive entries) the load balancing algorithm uses the *weight* array. In this case the load balancing algorithm distributes to each process a data partition of the same *weight* (Figure 5.5 (c)). This behavior provides high flexibility since the array is provided by the user to the balancing operation. The user can adapt the values of the array to the workload pattern and data structures of each application.

5.2.1 Dynamic load balancing algorithm

FLEX-MPI's dynamic load balancing algorithm balances the workload of both regular and irregular parallel applications, that operate on dense and sparse data structures, running on both homogeneous and heterogeneous systems. These systems can be either dedicated or non-dedicated.

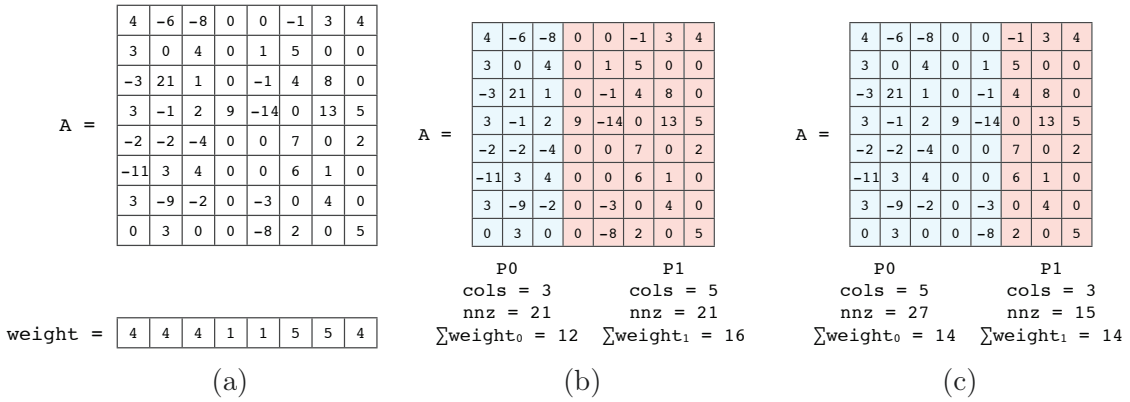


Figure 5.5: Comparison of FLEX-MPI's load balancing policies for (a) a data structure depending on (b) the *nnz* and (c) the *weight*.

In non-dedicated systems multiple user applications run concurrently and share the computing resources of the compute nodes. Sharing resources means that applications have interferences which degrade their performance. Balance the workload of parallel applications that run on non-dedicated systems represents a challenge because resources such as the CPU time allocated to the processes fluctuates over time. Furthermore, the frequency and magnitude of the interference is unpredictable.

Our approach considers two types of external interferences: *burst* and *long − term* interferences. Burst loads correspond to short-duration interferences which do not significantly affect the application performance. Long-term interferences reduce the CPU time allocated to the application thus affecting its performance. FLEX-MPI is able to discriminate between these two kinds of loads and effectively balance the application workload depending on the magnitude of the interference. FLEX-MPI takes into account the overhead of the data redistribution operation associated to the load balancing operation. The most useful approach is to tolerate short interferences as to avoid the cost of rebalancing too eagerly. Otherwise, the algorithm rebalances the workload when it detects a long-term interference present in the PEs.

The DLB algorithm implements a mechanism to discriminate between burst and long-term interferences. We introduce a user-defined parameter k that represents the sensibility of the algorithm to identify interferences as long-term loads. When a PE has been running in non-dedicated mode during k consecutive sampling intervals it is considered that long-term interference is present on that PE. In that case, the workload should be considered to be redistributed because the interference is responsible for the load imbalance.

Figure 5.6 illustrates the dynamic load balance of a parallel application running on a heterogeneous, non-dedicated system. Besides the workload imbalance, a long-term interference is present in two of the processing elements (a). The interference increases load imbalance, thus degrading significantly the overall performance. The load balancing algorithm takes into account both the interferences and the heterogeneous performance of the PEs to balance the workload of the parallel application, thus minimizing the execution time of the program (b).

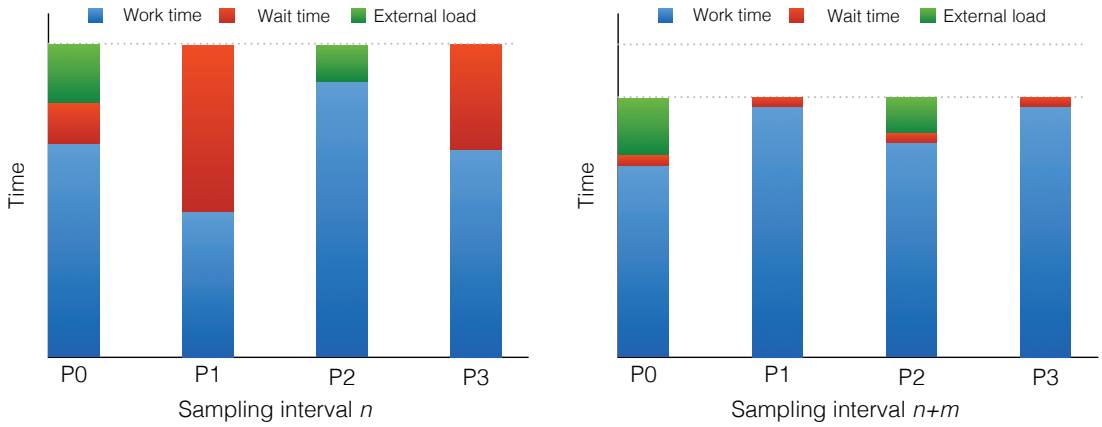


Figure 5.6: Dynamic load balance of a parallel application running on a heterogeneous non-dedicated system from sampling interval n (a) to $n+1$ (b).

Algorithm 5 shows the implementation of the dynamic load balancing algorithm of FLEX-MPI. This algorithm is evaluated at the end of every sampling interval n. The first step (`lines 1-6`) evaluates the load balance of the application. Function *evaluateLoadBalance* (`line 1`) detects load imbalance when the difference between the execution times ($Treal$) of fastest ($min(Treal)$) and slowest ($max(Treal)$) processes is larger than a user-defined threshold value (TH_1). If so, the application workload is more unbalanced than what the user can tolerate.

The second step (`lines 8-12`) evaluates which of the processing elements involved in executing the parallel application are dedicated and which not. When the difference between the CPU time ($Tcpu$) and the real time ($Treal$) of a processing element is small we can safely assume that it executes only the MPI process. Otherwise, when the real time is significantly higher than the CPU time then the PE is being shared between multiple processes of different user applications. The real time counts the time during the processing elements is performing computation and the overhead of the interference. The CPU time, on the other hand, only counts the dime during the PE is performing computation. The real time is always a little higher than the CPU time because of OS noise and interrupts. Function *evaluateLocalDedicated* (`line 8`) uses a threshold parameter TH_2 to account for this overhead and mark the difference between dedicated and non-dedicated processing elements. We consider that values of the real time that surpass the CPU time by 5% are reasonable for setting the tolerance threshold TH_2 between OS noise and interference.

Next step (`lines 14-19`) evaluates if any PE allocated to the parallel application has been running non-dedicated during the current sampling interval n. Function *evaluateGlobalDedicated* (`line 14`) gathers the dedicated status of all processing elements and returns true in case all of them have been running dedicated during the sampling interval. Otherwise, it returns false in case of a non-dedicated system—that is, any of the PEs has been running non-dedicated. The algorithm stores the result in a circular buffer ($dedicated\_system$) of length K.

In case the system has been running non-dedicated during the current sampling interval n (`line 21`) the algorithm evaluates (`line 26`) whether the interference is burst or a long-term interference. Function $evaluateInterference$ (`line 22`) evaluates and returns the type of interference. If the system has been running non-dedicated during the last k sampling intervals it is considered that long-term interference is present on the system and the workload should be considered to be redistributed. Otherwise, the interference is considered as an isolated burst.

When a bursty interference is detected, the algorithm tolerates it without performing load balancing for $(k-1)$ consecutive sampling intervals. In the k^{th} sampling interval one of two things will happen: (1) either there will be another burst, in which case it leads to the conclusion that rather than a series of bursts, a long-term load is present and the workload will be rebalanced taking into account the interference, or (2) the processing elements will run in dedicated mode, in which case it will also be a candidate for load balancing evaluation.

The algorithm evaluates (`line 29`) whether it should redistribute the workload if the workload is unbalanced and either (1) the system has been dedicated during the current sampling interval but the application is unbalanced or (2) long-term interference is detected on any of the PEs and it is leading to load imbalance. To effectively balance the workload, the algorithm distributes to each process a data partition which workload is proportional to the relative computing power of the PE on which it is running. The relative computing power of processing element i (RCP_i) [BGB06, MAG+11] is computed as the computing power of the PE (in $FLOPS$) divided by the sum of the computing power of the p PEs on which the MPI program is running.

When the application is evaluated for load balancing, function $computeFLOPs$ (`lines 31-34`) computes the computing power of each PE i as the number of $FLOPS$ performed during the sampling interval. In order to effectively compute the current computing power of the PE we use the real time, because it takes into account, if exists, the magnitude of the interference.

Function $computeRCP$ (`lines 36-40`) computes the relative computing power of each processing element i using hardware performance counters. The RCP is used by the algorithm to compute the new workload distribution (`lines 42-48`). Function $computeDataDistribution$ (`line 42`) computes the new workload distribution—count and displacement of each new data partition—depending on the RCP of each processing element. The data redistribution component uses the new workload distribution to make available the new data partitions to the processes.

FLEX-MPI also balances the application workload after a reconfiguring action is carried out by applying the kernel section of the DLB algorithm (`lines 31-48`). In case of a process spawning action, FLEX-MPI distributes to new processes a data partition that is proportional to their computing power. FLEX-MPI estimates the computing power of newly spawned processes assuming that their computing power is the same as processes currently allocated to PEs of the same processor class.

Algorithm 5 Implementation of the FLEX-MPI's dynamic load balancing algorithm.

```
1: /* evaluateLoadBalance:
2:  * MPI_Allgather (Treal_i, Treal)
3:  * return true if (max(Treal) − min(Treal))/max(Treal) > TH_1
4:  * else false
5:  */
6: load_imbalance ← evaluateLoadBalance (Treal_i, TH_1)
7:
8: /* evaluateLocalDedicated:
9:  * return true if (Treal_i − Tcpu_i)/Treal_i > TH_2
10: * else false
11: */
12: dedicated_PE_i ← evaluateLocalDedicated (Treal_i, Tcpu_i)
13:
14: /* evaluateGlobalDedicated:
15:  * MPI_Allgather (dedicated_PE_i, dedicated_PE)
16:  * return true if all(dedicated_PE[ ]) == true
17:  * else false
18:  */
19: dedicated_system[n%k] ← evaluateGlobalDedicated (dedicated_PE_i)
20:
21: if (dedicated_system[n%k] == false) then
22:    /* evaluateInterference:
23:     * return long_term if during_last_k(dedicated_system[ ]) == false
24:     * else burst
25:     */
26:    interference ← evaluateInterference (dedicated_system, k)
27: end if
28:
29: if ( (load_imbalance == true) and
    ( (dedicated_system[n%k] == true) or (interference == long_term) ) then
30:
31:    /* computeFLOPS:
32:     * return FLOPS_i ← FLOPs_i/Treal_i
33:     */
34:    FLOPS_i ← computeFLOPS (FLOPs_i, Treal_i)
35:
36:    /* computeRCP:
37:     * MPI_Allreduce (FLOPS_i, FLOPS, MPI_SUM)
38:     * return RCP_i ← FLOPS_i/FLOPS
39:     */
40:    RCP_i ← computeRCP (FLOPS_i)
41:
42:    /* computeDataDistribution:
43:     * data_distribution[i].count ← data_structure.size * RCP_i
44:     * data_distribution[i].displ ←
45:     * data_distribution[i − 1].count + data_distribution[i − 1].displ
46:     * return data_distribution
47:     */
48:    data_distribution ← computeDataDistribution (RCP_i)
49: end if
```

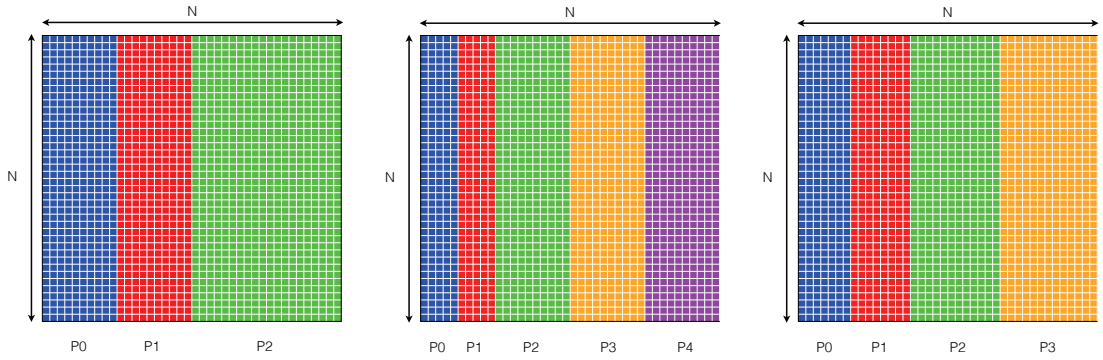


Figure 5.7: Load balancing operations performed by FLEX-MPI after process spawning (b) and process removing (c) actions.

This first estimation is then revised in subsequent sampling intervals using performance metrics collected at runtime. Otherwise, in case of a process removing action, the data partition of removed processes is distributed to those processes that remain in the execution. The DLB algorithm computes a new workload distribution in which each remaining process increases its data partition by a portion of the data partition of the removed processes that is proportional to their computing power.

Figure 5.7 illustrates an example of the load balancing operations performed by FLEX-MPI to balance the workload of a malleable application running on three processes (a). The parallel application is running on a heterogeneous dedicated system which consists of a PE allocated to process `P2` that is twice as powerful the processing elements allocated to processes `P0` and `P1`. FLEX-MPI spawns two processes and schedules them to processing elements of the same type of the PE allocated to `P0` (b). This results in a new workload distribution that assigns to processes `P2-4` a data partition of twice the size of the data partitions assigned to processes `P0` and `P1`. In the third step, FLEX-MPI removes the dynamic process `P4` and the load balancing component rebalance the workload among the remaining processes (c).

5.3 Computational prediction model

This section describes the functionalities provided by the computational prediction model (CPM) component of FLEX-MPI. The CPM implements a mathematical model that allows the malleability policies to estimate the application performance during the next sampling interval. The FLEX-MPI's malleability policies evaluate at the end of every sampling interval whether a reconfiguring action—either to expand or shrink the number of processes—might represent a benefit to the application performance or not. Therefore, after that evaluation one of the two following things will happen: (1) the policy decides to reconfigure the program to run on a different processor configuration or (2) the policy decides that the current processor configuration is the best suited to the application performance. A processor configuration consists of a set of processing elements allocated to the MPI program.

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 ($T_{communication}$) (Equation 5.1). In this work we assume that the MPI application uses synchronous MPI communication operations. We assume that the synchronization overhead counts as part of the communication time of the application.

$$T_{si} = T_{computation} + T_{communication} \tag{5.1}$$

Besides the number of processes and the algorithm, the execution time of a parallel application during a sampling interval depends on: the processor configuration performance and the system network performance. The values that model both type of components are provided as input to the CPM. The monitoring component feeds the runtime performance metrics (collected via PAPI) to the CPM that allow it to estimate the computation cost of the application. To effectively estimate the communication cost of the application the CPM uses profiling data (collected by monitoring via PMPI) and the system network performance. The network performance depends both on the network topology and the network technology—that are specific to the system environment.

However, a reconfiguring action involves changing the number of processes and the workload distribution of the applications. These implies the overheads for the process creation and termination operations ($T_{overhead\_process\_reconfig}$), as well as for the data redistribution operations ($T_{overhead\_data\_redist}$). Therefore, the execution time of a malleable application during a sampling interval that follows the reconfiguring action (T_{sir}) is computed as Equation 5.2.

$$T_{sir} = T_{computation} + T_{communication} + T_{overhead\_process\_reconfig} + T_{overhead\_data\_redist} \tag{5.2}$$

The overhead of data redistribution also depends on the system network performance and size of data transferred, that is calculated at runtime. The overhead of process creation and termination depends on the number of process spawned or removed, the operating system, and the size of the program binary. Therefore, these overheads are specific to the underlying hardware, the system environment, and the application characteristics.

5.3.1 Modeling the computation cost

To estimate the computation time of the application during the next sampling interval, FLEX-MPI implements different approaches depending on the type of parallel application. We consider regular and irregular parallel applications. Regular parallel applications are those with regular computation patterns. That is, all iterations of the application are identical on terms of execution times. Irregular applications, on the other hand, exhibit irregular computation and communication patters. In irregular applications computation times fluctuate, and the number of communications and the size of data transferred vary over time.

For parallel applications with regular computation patterns FLEX-MPI uses linear extrapolation to predict the number of $FLOPs$ in the next sampling interval based on the values of $FLOPs$ gathered in the past intervals. Instead of assuming that all iterations of the application are identical, we use extrapolation to account for slightly variations that can be detected at runtime.

For those applications with irregular computation patterns FLEX-MPI uses a profiling of the parallel application prior to the malleable execution. This profiling collects the number of $FLOPs$ computed by the application in a previous execution that is representative of the usual computation pattern of the application. The reason to use profiled data is that there is no reliable method to predict the computation pattern of irregular applications [KP11]. To provide an accurate estimation, the computational prediction model of FLEX-MPI uses the number of $FLOPs$ collected at runtime to correct differences between the profiled and measured performance. When the CPM detects a significant difference between profiled data and performance data collected during the last sampling interval, we calculate a correction factor that is applied to compute the estimation for the next sampling interval.

Once the CPM has estimated the number of $FLOPs$, it uses Equation 5.3 to calculate the predicted computation time for the next sampling interval. It takes as inputs the estimated $FLOPs$ and the computational power (in $FLOPS$) of each processing element (p) allocated to the program.

$$T_{computation} = \frac{FLOPs}{\sum_{p=1}^{np} FLOPS_p} \tag{5.3}$$

5.3.2 Modeling the parallel communication cost

As explained in Section 5.3 the performance of communications in a parallel application depends on the network topology and the network technology used. The network topology describes the arrangement of compute nodes on the system. There are two types of network topologies: physical and logical. Physical topology refers to the physical placement of nodes and the way they are actually connected to the network through switches and cables that transmit the data. Logical topology in contrast defines how the data passes through the network without regard to the physical interconnection of the nodes. Indeed a network's logical topology usually does not correspond to its physical topology. The most common yet basic logical topologies are bus, star, ring, mesh, and tree [Hal09].

Network technology refers to the technologies used to interconnect nodes on the system. The most popular high-performance network technologies for HPC systems are Ethernet [MB76], Myrinet [BKS$^+$95], and Infiniband [A$^+$00]. In the latest TOP500 [MSDS12] list (November 2013), the number of supercomputer using Ethernet interconnect is 42.40%, while 41.20% use Infiniband, and only 0.2% of the supercomputers use Myrinet.

High-performance networks aim to minimize latency and maximize bandwidth thus increasing communication performance. Network topology and technology affect the latency and bandwidth of the system network. For instance, while Infiniband

and Ethernet—in their most advanced versions—offer high bandwidths of up to 100 Gigabits per second, Infiniband offers lower latency but at higher costs.

Furthermore, MPI communication performance also depends on the MPI implementation used. Several MPI implementations implement some different optimization techniques to improve the performance of communication. For instance, MPICH implements Nemesis [BMG06a, BMG06b]—a communication subsystem that features optimization mechanisms for intranode communication in shared memory systems. Other MPI implementations are specifically designed for a particular network technology—such as MVAPICH [KJP08], an implementation designed for MPI on Infiniband.

There are several parallel communication models (PCMs) to model the performance and predict the execution time of MPI point-to-point and collective communication operations, the most known of which are LogGP [AISS95] and PLogP [KBV00]—which are based on LogP [CKP+93]—and the Hockney model [Hoc94]. These models use a set of standardized system parameters to approximate the performance of the algorithms which implement the MPI operations.

PCMs based on LogP describe a communication network in terms of latency (L), overhead (o), gap per message (g), and number of process involved in the communication operation (P). LogP models the time to send a constant-size, small message between two nodes as $L + 2o$ assuming that a sender's delay g between consecutive communication. LogGP extends LogP model to consider transmission of large messages. It introduces a new parameter (G) that represents the gap per byte and models the time to send a message of size m between two nodes as $L+2o+(m-1)G$. PLogP is another extension to LogP model that considers that latency, gap, and overheads are dependent on message size. It introduces sender (o_s) and receiver (o_r) overheads and models the time to send a message between two nodes as $L + g(m)$. Gap parameter in PLogP is defined as the minimum delay between consecutive communication, implying that $g(m) \geq o_s(m)$ and $g(m) \geq o_r(m)$.

The Hockney model, on the other hand, characterizes a communication network in terms of latency and bandwidth. Hockney model assumes that the time to send a message of size n—in bytes—between two nodes is $\alpha + n\beta$, where α is the network latency, and β is the transfer time per byte. The Hockney model assumes the same latency and bandwidth between every pair of nodes of the network. A flat network—in which all nodes are connected to a central switch—provides that connectivity properties. In this work we use the Hockney model to predict the execution times of MPI communication operations in FLEX-MPI because the specification of the MPICH implementation provides the cost models based on the Hockney model for all algorithms of synchronous, point-to-point and collective operations [TRG05, TG03].

In addition to α and β parameters, the MPICH cost models use two more parameters: p—the number of processes involved in the communication, and γ—used for reduction operations. γ characterizes the computation cost per byte for performing the reduction operation locally on any process [TRG05]. Table 5.2 summarizes the cost models based on the Hockney model for the algorithms of several MPI routines as provided by MPICH.

Table 5.2: Algorithms and their cost for MPI routines in MPICH.

MPI routine	**Algorithm**	**Duration**
`MPI_Send`/`MPI_Recv`		$T = \alpha + n\beta$
`MPI_Bcast`	Binomial tree	$T = \lceil \log p \rceil (\alpha + n\beta)$, for short messages ($<$ 12KB) or $p < 8$
	Van de Geijn's	$T = (\log p + p - 1)\alpha + 2\frac{p-1}{p} n\beta$, otherwise
`MPI_Scatter`	Binomial tree	$T = \log p\ \alpha + \frac{p-1}{p} n\beta$
`MPI_Gather`	Binomial tree	$T = \log p\ \alpha + \frac{p-1}{p} n\beta$
`MPI_Allgather`	Recursive-doubling	$T = \log p\ \alpha + \frac{p-1}{p} n\beta$, for short- and medium-size messages ($<$ 512KB)
	Ring	$T = (p-1)\alpha + \frac{p-1}{p} n\beta$, otherwise
`MPI_Reduce`	Binomial tree	$T = \lceil \log p \rceil (\alpha + n\beta + n\gamma)$, for short messages ($\leq$ 2KB)
	Rabenseifner's	$T = 2\ \log p\ \alpha + 2\frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$, otherwise
`MPI_Allreduce`	Recursive-doubling	$T = \log p\ \alpha + n \log p\ \beta + n \log p\ \gamma$, for short messages ($\leq$ 2KB)
	Rabenseifner's	$T = 2\ \log p\ \alpha + 2\frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$, otherwise
`MPI_Alltoall`	Bruck's	$T = \log p\ \alpha + \frac{n}{2} \log p\ \beta$, for short messages ($\leq$ 256 bytes)
	Pairwise-exchange	$T = (p-1)\alpha + n\beta$, otherwise

To predict the execution times of MPI communication for the next sampling interval, FLEX-MPI's computational prediction model uses the profiling data gathered by the monitoring component during the last sampling interval, the system network performance parameters—latency and bandwidth, and the MPICH cost functions for point-to-point and collective operations.

FLEX-MPI requires that α, β, and γ are previously defined based on the actual network performance. These values are specific to the particular system network configuration and are provided to FLEX-MPI via a configuration file. We use a performance benchmark that measures the values of α, β, and γ for the system network. There are several network benchmarks that measure the performance of MPI routines such as MPIbench [GC01], SKaMPI [RSPM98], and OMB (OSU Micro-benchmark suite) [Net09], among others. However, these test specifically measure network performance metrics using MPI routines to communicate between nodes. For this reason, we wrote our own network benchmark that uses Linux performance tools to measure latency and bandwidth using TCP/IP packets. The network benchmark performs several point-to-point tests and returns the latency and bandwidth between every pair of nodes in the cluster network. It uses *tcpping* [Ste04] to measure the network latency and $Iperf$ [TQD+05] to measure bandwidth performance.

Our benchmark measures network performance metrics assuming a small volume of data traffic present in the system network while performing the benchmark. However in practice the network traffic in a cluster fluctuates over time due to the data exchanged between different user applications—which is unpredictable and affects the performance of communication. To obtain precise estimations we introduce λ, a correction parameter that accounts for the difference between the estimation and the real value as measured by the monitoring component for the last sampling interval (Equation 5.4). This value is then used to correct for the estimation of the current sampling interval (n) (Equation 5.5).

$$\lambda_n = \frac{T_{communication\_estimated\_n-1}}{T_{communication\_real\_n-1}} \tag{5.4}$$

$$T_{communication\_estimated\_n} = T_{communication\_estimated\_n} \times \lambda_n \tag{5.5}$$

5.3.3 Modeling process creation and termination costs

In case the malleability policy decides to spawn or remove processes, the CPM takes into account the overhead of process creation and termination operations to predict the execution time of the next sampling interval. The time spent by FLEX-MPI on creating a dynamic process is different to the time spent on removing it. These overheads depends on various factors such as the operating system, the MPI implementation, and the size of the program binary. The CPM uses an offline benchmark that tests the costs of process reconfiguring actions in a particular environment.

A process spawning action in a Linux environment implies the OS's process manager to create the process through a fork-exec call, then allocate the address space to the process, and enable communication with other MPI processes through

a SSH connection. The associated overheads depend on the particular execution environment of the FLEX-MPI application. Equation 5.6 models the cost associated with the spawning of *nprocs_spawn* dynamic processes in FLEX-MPI applications. *process_spawning_cost* represents the cost associated with the spawning of a single dynamic process in the application.

$$T_{overhead\_process(spawn)} = nprocs\_spawn \times process\_spawning\_cost \tag{5.6}$$

Equation 5.7 models the cost associated with removing *nprocs_remove* dynamic processes from a FLEX-MPI application. *process_removing_cost* represents the cost associates with removing a single dynamic process from the malleable application. A process removing action implies at system level to deallocate the process resources, then close all active SSH connections with other MPI processes, finally safely finalize the process. It makes sense that the overhead of spawning a dynamic process is significantly higher than the overhead of removing a single process.

$$T_{overhead\_process(remove)} = nprocs\_remove \times process\_removing\_cost \tag{5.7}$$

Since the overheads of process creation and termination depend on the particular execution environment of the FLEX-MPI application, we wrote a benchmark that measures these costs in the system. Our benchmark uses FLEX-MPI's low-level interfaces to test the time spent on reconfiguring actions and returns the average time to create and terminate a dynamic application process in the system. We run this benchmark offline for every considered application that we want to provide malleable capabilities. The measured times provided by the benchmark are then provided as input to the CPM. Section 6.3.1.1 discuses a practical evaluation of this benchmark.

5.3.4 Modeling data redistribution costs

To model the cost of data redistribution the CPM takes into account the system network performance, the data size of the registered data structures, and the cost models for the MPI operations involved in the redistribution operation—`MPI_Alltoallv` and `MPI_Allgatherv`. Once the load balancing component has computed the new workload distribution it computes the total size of data that will be transferred between processes in the redistribution operation. That size depends on the number of registered data structures, their size in bytes, and their data type (e.g. integer, double, etc.). Then the CPM uses the network performance metrics to predict the cost of redistribution using the MPICH's cost models based on the Hockney model for `MPI_Alltoallv` and `MPI_Allgatherv`, which are the collective operations used in the data redistribution functionality.

The cost of data redistribution is computed using Equation 5.8, where *nd* and *ns* are the number of dense and sparse data structures handled by the application, *data_i* and *data_j* are pointers to the FLEX-MPI's struct type that holds the properties (i.e. memory address, size) of the data structure, p is the number of processes

involved in redistribution, α, β, γ are the network performance parameters, and λ is the FLEX-MPI's correction parameter for network performance estimation.

$$T_{overhead\_data\_redist} = \sum_{i=1}^{nd} Cost\_rdata\_dense(data_i, p, \alpha, \beta, \gamma, \lambda) + \sum_{j=1}^{ns} Cost\_rdata\_sparse(data_j, p, \alpha, \beta, \gamma, \lambda) \tag{5.8}$$

The estimation takes into account the storage format and the distribution scheme of the data structure. Equation 5.9 and Equation 5.10 model the costs associated to the redistribution of a single dense data structure and sparse matrix in CSR or CSC format. Redistribution costs of a dense data structure (Equation 5.9) are computed by the CPM using the cost model based on the Hockney model for the algorithm of the corresponding MPI collective operation used to redistribute the data structure. CPM uses the number of elements that will be redistributed ($rdata(data_i)$) and their data type to calculate the size of data in bytes that will be moved between processes. FLEX-MPI is aware of the current data partitioning, which allows the CPM to effectively calculate the total size of data that will be redistributed among processes. The specific MPI operation used depends on the user-defined data distribution scheme for the data structure, which can be distributed or replicated. As explained in Section 4.5, data structures with distributed data scheme are redistributed using `MPI_Alltoallv`, while data structures with replicated data scheme are redistributed using `MPI_Allgatherv`.

$$\begin{aligned} Cost\_rdata\_dense(data_i, p) = \textbf{if } & (data_i.scheme == XMPI\_DISTRIBUTED) \\ & Cost\_MPI\_Alltoallv\ (rdata(data_i), p, \alpha, \beta, \gamma, \lambda) \\ \textbf{else if } & (data_i.scheme == XMPI\_REPLICATED) \\ & Cost\_MPI\_Allgatherv\ (rdata(data_i), p, \alpha, \beta, \gamma, \lambda) \end{aligned} \tag{5.9}$$

Redistribution costs of a sparse data structure (Equation 5.10) are computed by the CPM as the sum of the costs for redistributing IA, JA, and A arrays that store the sparse matrix in CSR or CSC format. Redistribution costs for the array of size number of rows/colums plus one that stores the pointers—IA in CSR and JA in CSC—are computed using the cost model for `MPI_Alltoallv`, since this array is distributed among processes regardless of the user-defined data distribution scheme. Otherwise, the CPM computes the redistribution costs for the arrays of size nnz that store the rows or columns indexes and the nnz values depending on the distribution scheme.

$$Cost\_rdata\_sparse(data_j, p) = Cost\_MPI\_Alltoallv\ (data_j.rows, p, \alpha, \beta, \gamma, \lambda) + \\ \textbf{if}\ (data_j.scheme == XMPI\_DISTRIBUTED) \\ 2 * Cost\_MPI\_Alltoallv\ (rdata(data_j), p, \alpha, \beta, \gamma, \lambda) \\ \textbf{else if}\ (data_j.scheme == XMPI\_REPLICATED) \\ 2 * Cost\_MPI\_Allgatherv\ (rdata(data_j), p, \alpha, \beta, \gamma, \lambda) \tag{5.10}$$

5.4 Malleability policies

This section describes the implementation of the malleability policies of FLEX-MPI that allow to reconfigure the number of processes of the application at runtime in order to satisfy the performance requirements of each policy. FLEX-MPI provides high-level malleability because reconfiguring actions are performed according to the specifications of the policy without user intervention. The reconfiguring algorithm of these policies is evaluated at every sampling interval. As a result, FLEX-MPI decides whether or not reconfiguring the application taking into account (1) the current application performance metrics provided by the monitoring functionality, (2) the future application performance estimation provided by the CPM, (3) the idle resources in the system, and (4) the performance objective and constraints provided by the user.

FLEX-MPI features three high-level malleability *policies*: **Strict malleability policy** (SMP), **High performance malleability policy** (HPMP), and **Adaptive malleability policy** (AMP). The first one, SMP, implements the basic approach of malleability that reconfigures the number of processes of the application depending solely on the availability of resources of the system. That is, expanding the number of processes when resources become available, then removing processes when they are allocated to resources requested by the RMS in benefit of another application with highest priority in the system. However, increasing the number of processes of the parallel application may degrade its performance but can increase the operational cost and decrease the parallel efficiency of the program [LTW02]. To perform an efficient dynamic reconfiguration, the decision to change the number of processors has to be made based on the present and future performance of the application with the new processor configuration.

FLEX-MPI introduces two novel, performance-aware reconfiguring policies that take into account the performance of the application to guide the process reconfiguring actions. The first of these policies, HPMP, aims to reconfigure the application to the processor configuration that provides the highest performance to the program. That is, reconfiguring actions to expand the number of processes are carried out when resources become available in the system and the computational prediction model estimates a performance improvement as a result of the spawning action to allocate those resources. Otherwise, the current processor configuration remains until the RMS notifies the availability of additional resources or it requests some of the currently allocated processors in benefit of an application with highest priority.

XMPI_MONITOR_SI_INIT ()

```
int XMPI_Monitor_si_init (void)
```

Figure 5.8: Parameters of FLEX-MPI's high-level `XMPI_Monitor_si_init`.

The second of the performance-aware policies, AMP, introduces a novel execution model for MPI applications that allows the user to define the *performance objective* and *constraints* of the program. 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. This technique improves the performance of the parallel application while increases the resource utilization and the cost-efficiency of program executions. This policy uses the *Simplex* algorithm [Dan98] for linear programming to reconfigure the application to the optimal processor configuration that satisfies the performance objective and constraints.

FLEX-MPI introduces a set of high-level interfaces that allow the programmer to enable dynamic reconfiguration in iterative SPMD applications. Figure 5.8 shows the prototype of FLEX-MPI's high-level `XMPI_Monitor_si_init`, which does not take input parameters. This function enables the program to indicate the FLEX-MPI library the start of an iteration. FLEX-MPI then starts collecting performance metrics via the low-level monitoring functionality and accounts them for the current sampling interval. This routine takes no input parameters and returns true in case the initialization of the monitoring functionality for the current iteration succeeds.

Figure 5.9 describes the parameters of `XMPI_Eval_reconfig`. This function evaluates the algorithm of the malleability policy, then reconfigures the application to the new processor configuration in case of a reconfiguring action to satisfy the malleability policy. This routine takes as input the process rank, the number of running processes, the current iteration, the maximum number of iterations allowed to perform, the count and displacement of the currently assigned data partition, and the optional weight array. Additionally, this routine takes as input the arguments (*argv*) to the new processes and the name of the program binary (*command*) to be spawned during the reconfiguring action. As a result of executing this routine FLEX-MPI updates the status of every process in the application, so those processes which current status is removed can safely leave the iterative section of the program and finish their execution. The user can set the malleability policy for the program via a command line flag.

Figure 5.10 shows the parameters of `XMPI_Get_data`. This function allows dynamically spawned processes to transparently retrieve their data partition from the current data owners using the data redistribution functionality. Additionally, this function returns the current program iteration so dynamic processes can compute

XMPI_EVAL_RECONFIG (rank, nprocs, it, maxit, count, displ, weight, argv, command)

IN	rank	rank of the process (integer)
IN	nprocs	number of processes (integer)
IN	it	current program iteration (integer)
IN	maxit	maximum number of program iterations (integer)
INOUT	count	number of rows/cols of current data partition (integer)
INOUT	displ	displacement of current data partition (integer)
IN	weight	array indicating the column-wise or row-wise workload (double)
IN	argv	arguments to command (array of strings, significant only at root)
IN	command	name of program to be spawned (string, significant only at root)

```
int XMPI_Eval_reconfig (int rank, int nprocs, int it, int maxit,
            int *count, int *displ, double *weight, char *argv[],
            char *command)
```

Figure 5.9: Parameters of FLEX-MPI's high-level `XMPI_Eval_reconfig`.

XMPI_GET_DATA (it)

OUT	it	current program iteration (integer)

```
int XMPI_Get_data (int *it)
```

Figure 5.10: Parameters of FLEX-MPI's high-level `XMPI_Get_data`.

only the remaining iterations. Note that this routine is significant only for newly spawned processes.

Section 5.4.4 shows a detailed example of a SPMD parallel application instrumented with the high-level FLEX-MPI API to enable dynamic, performance-aware reconfiguration. Following sections describe the algorithm and implementation of the malleability policies featured in FLEX-MPI.

5.4.1 Strict malleability policy

The Strict malleability policy (SMP) is based on the assumption that increasing the number of processes of a parallel application decreases the execution time per iteration as a result of adding more resources to the program. Most of the existent approaches in the literature do not take into account the application performance and implement malleability techniques that depend solely on the availability of resources in the system. The goal of this policy is to enable dynamic reconfiguration to parallel applications that execute on systems with varying resource availability over time.

The SMP policy of FLEX-MPI decides to increase the number of processes of the parallel application via dynamic reconfiguration if the following conditions take place:

- There are idle processing elements in the system, and
- The number of processes of the new processor configuration does not surpasses a user-given, maximum number of processes.

On the other hand, the SMP policy decides to shrink the number of processes of the application if the following condition takes place:

- The RMS requests allocated processing elements on behalf of another application with highest priority.

Another powerful feature of the SMP policy allows the user to describe via configuration file the process scheduling for the parallel application. This file must contain the type of reconfiguring action, the number of processes to spawn or terminate, the type of processors that allocate these processes, and the number of iteration in which the reconfiguring action must be carried out. The SMP policy will automatically adjust the number and type of processes of the application when resources are available in the system.

5.4.1.1 SMP algorithm

Algorithm 6 Pseudocode for the implementation of the Strict malleability policy.

```
1: action, sys_procs_set ← getSystemStatus ()
2: if action == null then
3:    return
4: end if
5: if action == SPAWN_ACTION then
6:    max_procs_spawn← numProcs (sys_procs_set)
7:    new_procs_set ← mapping (alloc_procs_set, sys_procs_set, action, max_procs_spawn)
8: else if action == SHRINK_ACTION then
9:    max_procs_remove ← numProcs (alloc_procs_set - initial_procs_set)
10:   new_procs_set ← mapping (alloc_procs_set, sys_procs_set, action, min_procs_remove)
11: end if
12: if new_procs_set != alloc_procs_set then
13:   submitAllocationRMS (new_procs_set)
14:   dynamicReconfiguration (new_procs_set)
15:   updateProcessStatus ()
16:   new_workload_distribution ← loadBalance (new_procs_set)
17:   dataRedistribution (new_workload_distribution)
18: end if
```

Algorithm 6 shows the pseudocode for the implementation of the Strict malleability policy algorithm. Every sampling interval FLEX-MPI evaluates the algorithm, that may lead to a reconfiguring action. The first step (`line 1`) retrieves the current status of the system. Function *getSystemStatus* returns the type of reconfiguring action required and the processor configuration associated to the action ($Q = sys\_procs\_set$). The action type can take three possible values: (1) `SPAWN_ACTION` in case there are idle processing elements in the system, (2) `SHRINK_ACTION` in case the RMS requests allocated resources on behalf of another application, or (3) `null` in case no reconfiguring action is required to the program. The processor configuration describes either the layout of available PEs—in case of a spawning action—or the layout of PEs that must be released by the FLEX-MPI

application—in case of a shrinking action. FLEX-MPI uses the following syntax to describe the processor configuration:

hostname:procType:numPE

This syntax is quite similar to the syntax of the MPICH's *machinefile*, where *hostname* stands for the nodename, *procType* stands for the processor type, and *numPE* stands for the number of PEs of the node. If the action type is null (`lines 2-4`) the algorithm then ends function execution without applying any change in the application configuration. In case additional processes become available the algorithm computes (`line 6`) the maximum number of processes that can be spawned (max_procs_spawn). This number corresponds to the number of available PEs of the processor configuration associated to the reconfiguring event ($max\_procs\_spawn = |Q|$). Once the algorithm calculates the maximum number of processes, function *mapping* in `line 7` returns the new processor configuration (new_procs_set). It describes the layout of currently allocated PEs ($P = alloc\_procs\_set$) plus the layout of available resources (Q), where $new\_procs\_set = P \cup Q$. Note that if $MAX\_USER\_PROCS > |P| + |Q|$ the algorithm will discard some processing elements of Q to satisfy the aforementioned condition.

On the other hand in case of a shrink action the algorithm is limited to the number of dynamic processes which have been previously spawned by FLEX-MPI (max_procs_remove). This number is computed (`line 9`) as the number of processes of the current processor configuration P minus the number of processes of the initial set of processes, thus $max\_procs\_remove = |P|$ -$|initial\_procs\_set|$. In `line 10` function *mapping* returns the new processor configuration new_procs_set, that is a subset of the currently allocated processing elements. The new processor configuration consists of the current processor configuration minus the set of PEs associated to the shrink action. Therefore $new\_procs\_set = P \setminus Q$, where $Q \subset P$. If $|Q| > |P|$ - $|initial\_procs\_set|$ then *mapping* will return the initial set of processes because those processes can not be dynamically removed at runtime, thus $new\_procs\_set = initial\_procs\_set$.

Once the new processor configuration has been computed, in `line 12` the algorithm evaluates if the new processor configuration is different from the current one ($Q \setminus P \neq \emptyset$). If it is different a reconfiguring action is required, and so in `line 13` FLEX-MPI notifies the RMS the processing elements that will be allocated for the application. Then in `line 14` FLEX-MPI reconfigures the application to the new processor configuration through the dynamic process management functionality, whether to spawn new processes or remove processes from the program. Function *updateProcessStatus* in `line 15` updates the status of the processes. Removed processes will notice their updated status and leave the iteration section of the program and therefore the application execution. Function *loadBalance* in `line 16` computes the load balance for the new processor configuration using the load balancing component and returns the new workload distribution. Finally, function *dataRedistribution* (`line 17`) redistributes the workload using the data redistribution component of FLEX-MPI to efficiently move the data between processes.

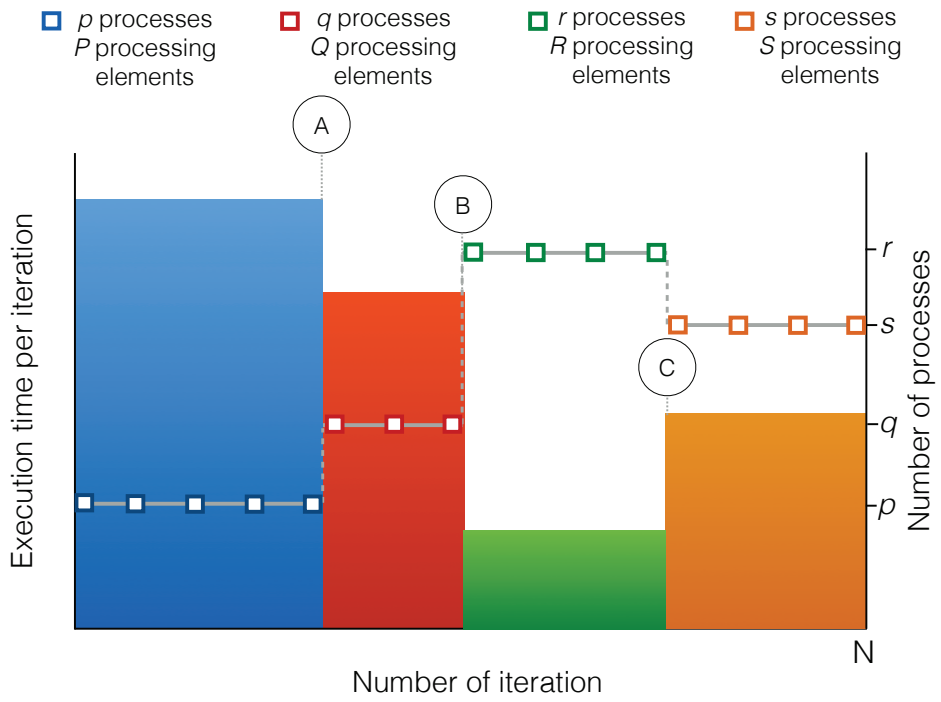


Figure 5.11: Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the SMP policy.

Figure 5.11 illustrates an example of a FLEX-MPI application using the SMP policy. Initially the FLEX-MPI application runs on a set P processes and an execution time per iteration of ti_p. At iterations A and B additional PEs become available, so FLEX-MPI reconfigures the application to run on q and then r processes, where $r > q > p$. This decreases the execution time per iteration to ti_q and ti_r respectively as a result of using more resources. However, at iteration C the RMS requests some resources so FLEX-MPI performs a reconfiguring action to remove those processes from the application thus the execution time per iteration increases up to ti_s.

5.4.2 High performance malleability policy

High performance malleability policy (HPMP) is a novel performance-aware malleability policy introduced by FLEX-MPI. The goal of this policy is to expand the number of processes to the processor configuration that provides the highest performance to the program in terms of completion time. To achieve this goal the HPMP policy takes into account the current application performance collected via monitoring and the future performance as computed by the CPM to guide reconfiguring actions to expand the number of processes. Otherwise if the CPM does not expect an immediate performance improvement as a result of adding more resources to the program, the number of processes remains until the next sampling interval.

Therefore, the HPMP policy decides to expand the number of processes of the application via dynamic reconfiguration if the following conditions take place:

- There are idle processing elements in the system, and
- The computational prediction model estimates a performance improvement as a result of adding resources to the program.

On the other hand, the HPMP policy decides to shrink the number of processes of the application if the following condition takes place:

- The RMS requests allocated processing elements on behalf of another application with highest priority.

5.4.2.1 HPMP algorithm

Algorithm 7 Pseudocode for the implementation of the High performance malleability policy.

```
1: action, sys_procs_set ← getSystemStatus ()
2: if action == null then
3:    return
4: end if
5: if action == SPAWN_ACTION then
6:    max_procs_spawn ← numProcs (sys_procs_set)
7:    Test_P ← CPM (alloc_procs_set)
8:    for s = 0 to s = max_procs_spawn do
9:       Test_S ← CPM (alloc_procs_set + (sys_procs_set, s))
10:      if Test_S < Test_P then
11:         procs_set ← mapping (alloc_procs_set, sys_procs_set, action, s)
12:         suitable_procs_sets ← push (procs_set)
13:      end if
14:   end for
15:   new_procs_set ← selectBestProcsSet (suitable_procs_sets, HPMP)
16: else if action == SHRINK_ACTION then
17:   max_procs_remove ← numProcs (alloc_procs_set - initial_procs_set)
18:   new_procs_set ← mapping (alloc_procs_set, sys_procs_set, action, max_procs_remove)
19: end if
20: if new_procs_set != alloc_procs_set then
21:   submitAllocationRMS (new_procs_set)
22:   dynamicReconfiguration (new_procs_set)
23:   updateProcessStatus ()
24:   new_workload_distribution ← loadBalance (new_procs_set)
25:   dataRedistribution (new_workload_distribution)
26: end if
```

Algorithm 7 shows the pseudocode for the implementation of the High performance malleability policy algorithm. The algorithm is evaluated at the end of every sampling interval i. The first step (`line 1`) consists of retrieving the action type

and the processor configuration associated to the action ($Q = sys\_procs\_set$). If the RMS notifies FLEX-MPI of the availability of additional resources in the system, the algorithm computes first (`line 6`) the maximum number of processes that can be spawned in the current sampling interval (max_procs_spawn). Then the algorithm uses the computational prediction model to predict the execution time for the next sampling interval ($i + 1$) under the current processor configuration ($P = alloc\_procs\_set$). Function CPM in `line 7` returns the execution time estimated for the next sampling interval ($Test\_P$). Then the algorithm evaluates (`lines 8-13`) the performance of every possible *execution scenario*. Every execution scenario is associated to a processor configuration S which consists of the set of currently allocated processes (P) plus s additional PEs ranging from 0—no reconfiguration—to the maximum number available (max_procs_spawn). Function CPM in `line 9` then returns the estimated execution time for the next sampling interval ($Test\_S$) under the corresponding processor configuration S. The idea behind this algorithm is to evaluate all the possible execution scenarios (with different processor configurations) and take the one that predictably will provide the highest performance.

In case the estimated execution time for $Test\_S$ is lower than $Test\_P$ (`line 10`), the algorithm uses function *mapping* (`line 11`) to compute the processor configuration for S as $S = P \cup Q'$, where $Q' \subseteq Q$ and $s = |Q'|$. Next the algorithm stores the processor configuration in a list of suitable scenarios (`line 12`). Once every possible execution scenario has been evaluated, function *selectBestProcsSet* (`line 15`) selects from the list of suitable execution scenarios the processor configuration (new_procs_set) which leads to the highest performance according to the goal of the HPMP policy. That is, the processor configuration with the minimum predicted completion time for the next sampling interval.

On the other hand, a removing action (`lines 16-19`) involves computing the new processor configuration taking into account the maximum number of processes that can be removed from the application (max_procs_remove). Function *mapping* (`line 18`) computes the new processor configuration (new_procs_set) which consists of the currently allocated PEs minus those PEs associated to the shrink action $new\_procs\_set = P \setminus Q$.

Once the algorithm has generated the new processor configuration, it computes the difference between the new and the current processor configuration (`line 20`) and performs the necessary actions to reconfigure the number of processes of the program. These include notifying RMS of the current availability of resources in `line 21`, performing the actual reconfiguring action to expand or shrink the number of processes in `line 22`, updating the processes status in `line 23`, and finally redistributing the application workload by computing the new data distribution (`line 24`) and finally moving the data between processes (`line 25`).

Figure 5.12 illustrates an example of a FLEX-MPI application using the HPMP policy. Initially, the application runs on a set of P processes and the execution time per iteration is ti_p. At iterations A and B additional resources become available and the CPM estimates an immediate performance improvement, thus FLEX-MPI reconfigures the application to expand the number of processes to q and then r processes respectively, where $r > q > p$. The new execution scenario improves the performance

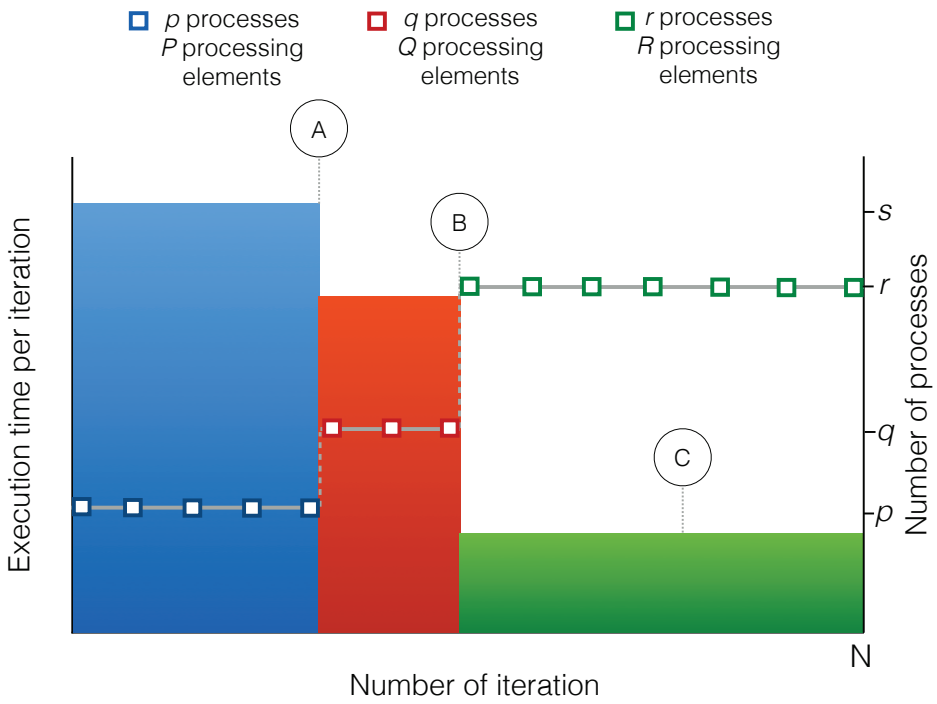


Figure 5.12: Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the HPMP policy.

of the parallel application while decreases the execution time per iteration to ti_r. At iteration C additional resources (S) become available. However, the algorithm of the HPMP policy evaluates every possible execution scenario but the CPM does not estimate a performance improvement, thus FLEX-MPI decides to keep the current processor configuration which provides the highest performance to the program.

5.4.3 Adaptive malleability policy

Adaptive malleability policy (AMP) is the most sophisticated performance-aware reconfiguring policy introduced by FLEX-MPI. The goal of the AMP policy is to dynamically reconfigure the number of processes of the application to satisfy a user-given performance objective. This policy allows the user to define the performance objective and constraints of the application, using 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. The AMP policy evaluates the performance of the application every sampling interval and modifies it by adding or removing processes whenever the reconfiguring policy detects that the performance target is not achieved.

The reconfiguration process in AMP also depends on the user-given performance

constraint which can be either the parallel efficiency or the operational cost of executing the program. The efficiency constraint results in minimizing the number of dynamically spawned processes and maximizing the number of dynamically removed processes in order 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 cost unit per CPU time unit) and removing processes from those processes with the largest cost while satisfying the performance constraint. This cost may represent whether the economic cost or the energetic cost, or any other unit that represents a cost associated to the usage of the resources. For this reason we introduce a new syntax used to describe the layout of resources in FLEX-MPI that includes the *cost* of the resources per processing element per time unit:

hostname:procType:numPE:cost

To achieve its performance goal the AMP policy takes into account the current application performance collected via monitoring and the future performance computed by the CPM to guide reconfiguring actions to expand and shrink the number of processes. The AMP policy decides to expand the number of processes of the application via dynamic reconfiguration if all the following conditions take place:

- There are idle processing elements in the system, and
- The current performance of the parallel application does not satisfy the user-given performance objective, and
- The CPM estimates a performance improvement that satisfies the performance objective under the user-given performance constraint as a result of expanding the number of processes.

On the other hand, the AMP policy decides to shrink the number of processes of the application under the following circumstances:

- The current performance of the parallel application does not satisfy the user-given performance objective, and
- The CPM estimates that shrinking the number of processes to a new processor configuration will satisfy the performance objective under the user-given performance constraint.

5.4.3.1 Integrating linear programming methods to find the optimal processor configuration

One of the main features of the AMP policy is that it uses algorithms for linear programming to find the most appropriate process configuration to satisfy the user-given performance objective under the performance constraints. Linear programming, sometimes known as linear optimization, is a mathematical method for the optimization of a linear function subject to a set of linear constraints. That is, maximizing

or minimizing the linear objective function taking into account the constraints that may be equalities or inequalities.

One of the best-known methods for linear programming problems is the *Simplex* algorithm [Dan98] introduced by George Dantzig in 1947, which became the basis of the mathematical optimization area. The Simplex method iteratively operates on linear programs in *standard form*. Specifically, FLEX-MPI uses the implementation of the Simplex algorithm included in the *GNU Linear Programming Kit* [GLP03]. A linear programming problem in standard form consists of a linear function, also called objective function of the form:

$$f(x_1, x_2) = c_1x_1 + c_2x_2 + ... + c_nx_n \tag{5.11}$$

The objective function is subject to one or more linear constraints of the form:

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n \leq b_i \tag{5.12}$$

And which variables are non-negative:

$$x_i \geq 0 \tag{5.13}$$

Standard maximization problem and *standard minimization problem* describe two classes of special linear programming problems in standard form which seek to maximize and minimize the objective function, respectively. The maximum or minimum value of the objective function is called the *optimal value*, while the collection of values of the variables $(x_1, x_2, ..., x_n)$ constitutes the *optimal solution* of the problem. A linear programming problem is usually expressed in matrix form, so the standard maximization problem becomes:

$$max\{c^T x | Ax \leq b | x \geq 0\} \tag{5.14}$$

While the standard minimization problem becomes:

$$min\{c^T x | Ax \geq b | x \geq 0\} \tag{5.15}$$

The AMP policy introduces $mappingSimplex$, a variant of the function *mapping* described in Section 5.4.2 that uses the Simplex method to find the optimal processor configuration that satisfies the performance objective and constraints of the FLEX-MPI application. This is necessary because we consider heterogeneous architectures consisting of processing elements with different performance characteristics and different cost. Basically, the idea is to use the number of processes and the cost of their PEs allocated as the variables and coefficients of the objective function, and the performance of the PEs as the constraints of the linear programming problem. Then we use the Simplex method to find the optimal solution (i.e. processor configuration) depending on the type of reconfiguring action and the user-given performance constraints.

The standard minimization problem shows up for spawning actions:

- Under the parallel efficiency constraint, the problem is to minimize the number of processes that need to be spawned to satisfy the user-given performance objective, thus allocating to the extent possible the most powerful processing elements.
- Under the cost constraint, the problem is to minimize the cost associated to the processes that need to be spawned to satisfy the user-given performance objective, thus allocating to the extent possible the processing elements with the smallest operational cost.

On the other hand, the standard maximization problem shows up for shrink actions:

- Under the parallel efficiency constraint, the problem is to maximize the number of processes that need to be removed to satisfy the user-given performance objective, thus removing to the extend possible the less powerful processing elements.
- Under the cost constraint, the problem is to maximize the cost associated to the processes that need to be removed to satisfy the user-given performance objective, thus removing to the extend possible the processing elements with the largest operational cost.

5.4.3.2 AMP algorithm

Algorithm 8 shows the pseudocode for the implementation of the Adaptive malleability policy algorithm. 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.

The **First Phase** (`lines 1-5`) of the algorithm evaluates the current and future performance of the application under the current processor configuration. The first step (`line 2`) consists of capturing via monitoring the execution time $Treal$ of the current sampling interval i to update the accumulated execution time of the application ($Texe_{accum}$). This value is used by function *calculateGoal* (`line 3`) to compute the execution time $Tgoal$ that is necessary to satisfy the user-given performance objective during the next sampling interval. This function computes $Tgoal$ by dividing the execution time remaining to the user-given performance objective by the number of sampling intervals remaining. CPM (`line 4`) then returns the predicted execution time for the next sampling interval ($Test\_P$) under the current processor configuration $P = alloc\_procs\_set$, where $p = |P|$.

When the difference between the required and predicted execution times is bigger than a given threshold *tol* (`line 5`) the algorithm goes to the second phase in order to perform a reconfiguring action. AMP uses a default value of 1% for *tol*, though the user can override this default and provide its own desired value.

The **Second Phase** (`lines 6-30`) analyzes different process reconfiguring scenarios and selects the best one under user-given performance objective and performance constraints. The algorithm first decides whether to increase (`line 7`) or decrease (`line 18`) the number of processing elements depending on which of the predicted and required times is bigger.

Algorithm 8 Pseudocode for the implementation of the Adaptive malleability policy.

```
1: ———— First phase ————
2: Texe_accum ← Texe_accum + Treal
3: Tgoal ← calculateGoal (Texe_accum, Goal)
4: Test_P ← CPM (alloc_procs_set)
5: if |Test_P − Tgoal| < tol then
6:    ———— Second phase ————
7:    if Test_P > (Tgoal + tol) then
8:       sys_procs_set ← getSystemStatus(only_available)
9:       max_procs_spawn ← numProcs (sys_procs_set)
10:      for s = 0 to s = max_procs_spawn do
11:         (ΔFLOPS, T_comm, T_spawn, T_rdata) ← CPMreconfig(p+s, Tgoal, cFLOPS)
12:         (procs_set, T_comp, cost)←mappingSimplex(constraint, ΔFLOPS, sys_procs_set, s)
13:         Test_S = T_comp + T_comm + T_spawn + T_rdata
14:         if |Test_S − Tgoal| < tol then
15:            suitable_procs_sets ← push (procs_set)
16:         end if
17:      end for
18:   else if Tgoal > (Test_P + tol) then
19:      max_procs_remove ← numProcs (alloc_procs_set - initial_procs_set)
20:      for r = 0 to r = max_procs_remove do
21:         (ΔFLOPS, T_comm, T_remove, T_rdata) ← CPMreconfig(p−r, Tgoal, cFLOPS)
22:         (procs_set, T_comp, cost)←mappingSimplex(constraint, ΔFLOPS, alloc_procs_set, r)
23:         Test_R = T_comp + T_comm + T_remove + T_rdata
24:         if |Test_R − Tgoal| < tol then
25:            suitable_procs_sets ← push (procs_set)
26:         end if
27:      end for
28:   end if
29: end if
30: new_procs_set ← selectBestProcsSet(suitable_procs_sets+alloc_procs_set,AMP)
31: ———— Third phase ————
32: if new_procs_set != alloc_procs_set then
33:    submitAllocationRMS (new_procs_set)
34:    dynamicReconfiguration (new_procs_set)
35:    updateProcessStatus ()
36:    new_workload_distribution ← loadBalance (new_procs_set)
37:    dataRedistribution (new_workload_distribution)
38: end if
```

In case the algorithm decides to spawn new processes, it first retrieves (`line 8`) the processor configuration that corresponds to the layout of available processing elements in the system ($Q = sys\_procs\_set$). When used, the $only\_available$ flag tells the function $getSystemStatus$ to return only the processor configuration of available PEs. Once the processor configuration has been retrieved, the algorithm computes (`line 9`) the maximum number of processes that can be spawned in the current sampling interval (max_procs_spawn).

The next step (`lines 10-17`) consists of evaluating every possible execution scenario for the application. Every execution scenario (`line 10`) is associated to a processor configuration S which consists of the current processor configuration (P) plus a subset of s additional processes, where $s \in (0, max\_procs\_spawn)$. Function $CPMreconfig$ in `line 11` is a variant of CPM that uses the computational prediction model to calculate the number of FLOPS ($\Delta FLOPS$) necessary to achieve the performance objective $Tgoal$ of the application for the next sampling interval. The CPM uses as a parameter the number of currently allocated PEs (p) and the number of additional PEs (s) associated to the execution scenario evaluated and returns the predicted reconfiguring overheads (both for process creation and data redistribution) as well as the predicted communication time for the next sampling interval when $p + s$ processing elements are used.

Function $mappingSimplex$ (`line 12`) uses the Simplex algorithm to find a set $procs\_set$ of s PEs that satisfies the performance objective according to the user-given constraint. 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 13` calculates the predicted execution time during the sampling interval $Test\_S$. Due to the reconfiguring overheads it is possible that this time does not satisfy the performance objective. For this reason, in `line 14` 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 15`). This procedure is repeated for each value of s.

On the other hand, to remove processes (`lines 18-28`) the algorithm first computes (`line 19`) the maximum number of processes that can be removed from the application (max_procs_remove). For each execution scenario (`line 20`) the algorithm computes the computational power (`line 21`) that satisfies the performance objective. Note that in case of a shrink action, this value represents the number of FLOPS that has to be removed in order to degrade the performance of the application to a point that satisfies the user-given performance objective. Every execution scenario is associated to a R processor configuration which consists of the current processor configuration P minus a subset of r PEs, where $r \in (0, max\_procs\_remove)$. Function $CPMreconfig$ also returns the predicted reconfiguring overheads for data redistribution and process removing associated to remove r processes, as well as the predicted communication time when $p - r$ processing elements are used. Function $mappingSimplex$ (`line 22`) then 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.

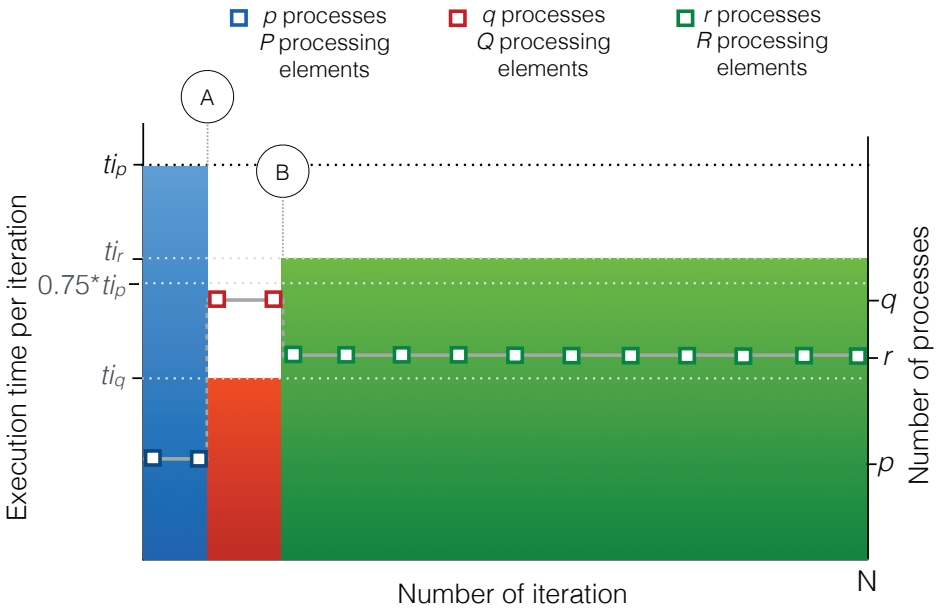


Figure 5.13: Execution time (left y-axis) and number of processes (right y-axis) per sampling interval of an example execution of a FLEX-MPI application using the AMP policy.

The last step of the second phase (`line 30`) consists of selecting 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 32-38`) the algorithm calculates if the current processor configuration and the configuration computed by the algorithm are different. In that case, the algorithm reconfigures the application to run on the newly selected processor configuration. In (`line 33`) FLEX-MPI notifies the RMS of the new processor allocation, whether to allocate new processors or release currently allocated PEs. The following steps consist in performing the process reconfiguring through the dynamic process management functionality (`line 34`), updating the status of the application processes (`line 35`), computing the load balance for the new processor configuration (`line 36`), and redistributing the workload (`line 37`) between the processes of the new processor configuration.

Figure 5.13 illustrates an example of a FLEX-MPI application with regular computation and communication patters using the AMP policy. Initially the application runs on a set of P processes with an execution time per iteration of ti_p, so the AMP computes an estimated completion time for the static version of the application (t_{static}) of:

$$t_{static} = ti_p \times N$$

However, the user-given performance objective is set to a performance improvement of 25% by reducing the completion time of the program under P by 25%, and the performance constraint is set to maximize the parallel efficiency. At iteration A—which corresponds to the first sampling interval—the AMP policy expands the number of processes to q in order to improve the performance of the application. The CPM estimates that the processor configuration Q represents the execution scenario with the minimum number of processes that satisfies the performance objective of the application. However, at iteration B the AMP policy detects via monitoring that the current performance is higher than imposed by the user, so it decides to remove some processes in order to satisfy the performance objective. Note that, since the AMP policy is performance-aware, it is able to correct inaccurate estimations made in previous sampling intervals. FLEX-MPI then reconfigures the application to run on a new R processor configuration, which is a subset of the previous Q processor configuration. The number of processes removed from R is the maximum number of processes that satisfies that the R processor configuration will met the performance objective. The parallel application runs on the R processor configuration until the end of its execution with a completion time of:

$$t_{dynamic} = ti_p \times A + ti_q \times (B - A) + ti_r \times (N - B) + T_{overhead\_process\_reconfig} + T_{overhead\_data\_redist}$$

The completion time of the dynamic version of the application ($t_{dynamic}$) computes the execution time of the application itself plus the overheads of processes reconfiguring actions and data redistribution. To summarize, the dynamic version presents a performance improvement of 25% when comparing to the static version of the application:

$$t_{dynamic} = t_{static} \times 0.75$$

5.4.4 Malleable MPI programs using the FLEX-MPI high-level API

This section shows how to use the FLEX-MPI's high-level API to create malleable MPI programs. Figure 5.14 shows the parallel code of an iterative SPMD program instrumented with the FLEX-MPI high-level API. This program operates on several dense data structures that are shared between processes without replication. First, each process retrieves the count and displacement of the domain partition assigned to the process depending on the current number of processes and the total size of the data structures (`XMPI_Get_wsize`). The program declares the data structures and uses the registering function (`XMPI_Register_vector`, `XMPI_Register_dense`) to indicate the size, type, and the data distribution scheme of each data structure. Then, each process gets its status (`XMPI_Get_data`) so newly spawned processes can retrieve the current program iteration and their data partition from the current processes using the data redistribution functionality. On the other hand, those processes into the initial set of processes will load their data partition from the file system.

Program: spmd.c

```
MPI_Init (&argv, &argc);

MPI_Comm_rank (XMPI_COMM_WORLD, &rank);
MPI_Comm_size (XMPI_COMM_WORLD, &size);

XMPI_Get_wsize (rank, nprocs, dsize, &displ, &count);

double *A = (double*) malloc (dsize * count * sizeof(double));
double *x = (double*) malloc (dsize * sizeof(double));
double *x_recv = (double*) malloc (dsize * sizeof(double));

XMPI_Register_vector (&x, dsize, MPI_DOUBLE, XMPI_REPLICATED);
XMPI_Register_dense (&A, dsize, MPI_DOUBLE, XMPI_DISTRIBUTED);

status = XMPI_Get_process_status ();

if (status == XMPI_NEWLY_SPAWNED)
    XMPI_Get_data (&it);
else if (status == XMPI_RUNNING) {
    load_data_fs (&A, &x);
    it = 0;
}

for (it; it < maxit; it ++) {

    XMPI_Monitor_si_init ();

    for (i = 0; i < count; i ++) {
        A[i] = A[i] + func(A[i], x[i+displ]);
    }

    MPI_Allreduce (&x, &x_recv, dsize, MPI_INT, MPI_MAX, 0, XMPI_COMM_WORLD);

    XMPI_Eval_reconfig (rank, size, it, &count, &displ, NULL, argv, "spmd");

    if (XMPI_Get_process_status () == XMPI_REMOVED) break;

    memcpy (x_recv, x, dim*sizeof(double));
}

free(A)
free(x)
free(x_recv)

MPI_Finalize ();
```

Figure 5.14: MPI malleable program instrumented with FLEX-MPI high-level interfaces.

We assume that the programmer initially launches the program to run with n processes via the command line (`mpirun/mpiexec`) and uses the HPMP reconfiguring policy of FLEX-MPI to achieve the maximum performance being the maximum number of processes ($n + m$). The iterative section of the code includes the `XMPI_Monitor_si_init` function to indicate the start of monitoring. Then, each process operates in parallel on a different subset of the data structure, performs a collective MPI operation (`MPI_Allreduce`), and the program allows FLEX-MPI to reconfigure the number of processes to increase the performance. After the performance evaluation and reconfiguring action, each process evaluates its status and those processes flagged as removed jump out of the iterative section and finish their execution. On the other hand, the remaining processes perform a memory copy operation to update their current solution, then finish their execution either when the maximum number of iterations is reached or their are flagged as removed in a following sampling interval.

5.5 Summary

In this chapter we describe the high-level functionalities of FLEX-MPI that are encapsulated in the malleability logic layer of the library. The malleability logic layer introduces high-level components for dynamic load balancing, performance prediction, and performance-aware dynamic reconfiguration. These functionalities are high-level because adaptive actions are performed without user intervention. FLEX-MPI introduces a dynamic load balancing technique for SPMD applications that uses performance metrics to detect load imbalance and makes workload distribution decisions. Furthermore, it does not require prior knowledge about the underlying architecture and works either for regular and irregular parallel applications.

The computational prediction model component implements a mathematical model that allows to estimate the application performance. It allows FLEX-MPI to estimate if a reconfiguring action may represent a benefit for the application in terms of performance improvement.

Finally, FLEX-MPI features three high-level malleability policies: Strict malleability policy (SMP), High performance malleability policy (HPMP), and Adaptive malleability policy (AMP). The SMP policy allows a FLEX-MPI program to automatically change the number of processes at runtime depending on the availability of resources in the system. HPMP and AMP are performance-aware because, in addition to the availability of resources, they take into account the performance of the application to guide the process reconfiguring actions. HPMP automatically reconfigures the application to run on the processor configuration which provides the highest performance to the application. Meanwhile, the AMP policy uses a user-given performance objective and constraints to guide reconfiguring actions such the application completes within a specified time interval. This chapter also describes the high-level API that provides access to those functionalities of the malleability logic layer of FLEX-MPI.

Chapter 6

Experimental Results

6.1 Introduction

In the previous chapters we have provided a formal description of the different components of FLEX-MPI. In this one we analyze and discuss the practical evaluation of FLEX-MPI for different applications on homogeneous and heterogeneous platforms. The main goal of this chapter is to demonstrate that FLEX-MPI is able to dynamically adjust the number of processes of the parallel application at runtime to cope with the availability of resources in the system and the user-defined performance criteria. This chapter consists of five main sections:

- The first section **summarizes the experimental conditions** including a description of the applications used as benchmarks as well as the main characteristics of compute resources used for running the collection of experiments.
- The second section presents the **performance analysis of the dynamic load balancing algorithm** through the evaluation of parallel benchmarks which exhibit regular and irregular computation patterns, running on homogeneous and heterogeneous cluster configurations, in dedicated and non-dedicated scenarios. The goal of this section is to evaluate the capabilities of the dynamic load balancing algorithm to improve the performance, efficiency, and scalability of parallel applications.
- The third section presents an **empirical evaluation of the computational prediction model**, which consists of different components that are evaluated separately including the performance models of CPU and network communications, and the model that estimates the process creation and termination overheads. All these models have been tuned for our considered architectures in order to provide an accurate prediction of the application performance. The goal of this set of experiments is to validate the capabilities of the computational prediction model to effectively predict the performance of applications.
- The fourth section presents a **performance analysis of malleable MPI applications** running with FLEX-MPI for dynamic reconfiguration. Several

features are considered here: (1) the capability of FLEX-MPI for adjusting dynamically the number of process of the MPI application at runtime, and (2) the efficiency of the different malleability policies for achieving different performance objectives, and (3) the capability of the policies for satisfying the performance objective taking into account the performance constraints (i.e. efficiency and cost). The goal of this section is to evaluate the performance of the SMP, HPMP, and AMP malleability policies to satisfy the performance objective and constraints of parallel applications running on dynamic execution environments.

- Finally, the fifth section presents an **analysis of the overhead of FLEX-MPI** and its impact on the application performance. The goal of this section is to evaluate the impact the FLEX-MPI library on the performance of parallel programs.

This chapter concludes with a summary of the results derived from this work.

6.1.1 Benchmarks

Our benchmark suite consists of three parallel applications used as benchmarks: Jacobi, Conjugate Gradient (CG), and EpiGraph. All of them are written in C language and modified to integrate high-level FLEX-MPI interfaces to enable dynamic reconfiguration. The idea behind the election of these benchmarks was to consider relevant applications with different characteristics. Jacobi and Conjugate Gradient are compute-intensive applications broadly employed as computational kernels. Both of them have a regular communication pattern and exhibit an approximately constant execution time per iteration. The differences between them reside in the matrix format that they use (dense matrices for Jacobi and sparse matrices for CG) and the data locality degree of the memory references (which is smaller for CG). These differences lead to different performance behaviors when they are executed with FLEX-MPI. Finally, EpiGraph is a communication-intensive epidemic simulator with poor data locality and a variable workload over time. EpiGraph represents an example of a complex application with varying performance during its execution. The following sections provide a more detailed description of each benchmark.

6.1.1.1 Jacobi

Jacobi is an application which implements the iterative Jacobi method—named after Carl Gustav Jacob Jacobi—for solving diagonally dominant systems of linear equations. A system of n linear equations can be represented as Equation 6.1, where A represents a dense square matrix of order n consisting of the coefficients of the variables of the system, x are n unknowns (associated to the solution), and b are n constant terms.

Algorithm 9 shows the pseudocode of the sequential implementation of Jacobi from [BDGR95]. The main computation involves a dense matrix-vector product and several divisions. x_{old} stores the current solution to the system for every iteration.

Algorithm 9 Jacobi iterative method pseudocode.

```
1:  Generate a initial value to the solution x
2:  for k = 0 to k = |maximum_number_iterations| do
3:     for i = 0 to i = n do
4:        x_new[i] = 0
5:        for j = 0 to j = nwithj ≠ i do
6:           x_new[i] = x_new[i] + A[i][j] * x_old[j]
7:        end for
8:        x_new[i] = (b[i] − x_new[i])/A[i][j]
9:     end for
10:    x_old = x_new
11:    Check convergence; continue if necessary
12: end for
```

Based on it, a new solution (x_{new}) is computed for each iteration which is subsequently used as input for the following iteration. The algorithm finishes when the convergence criteria is met (producing a solution under a given tolerance error) or when the maximum number of iterations is reached.

$$Ax = b \tag{6.1}$$

The parallel implementation of Jacobi performs a block distribution of i-loop. Data dependencies are only related to x_{new} and x_{old}, so the parallel model involves two communication calls. The first one to perform the initial distribution of A among the existing processes (usually following a block partitioning strategy), the second one to update x_{old} values for all the processes. The initial data distribution is performed before starting the iterative section of Jacobi, while updating the x_{old} vector is performed as the end of every iteration of the outer loop (k-loop).

6.1.1.2 Conjugate Gradient

Conjugate Gradient implements an iterative algorithm for determining the solution of systems of linear equations (that follow the same principle as in Equation 6.1) that require of symmetric, positive-definite matrices. CG improves the gradient descent minimization problem by making every gradient direction conjugate to the previous ones.

Algorithm 10 shows the sequential pseudocode of CG from [BDGR95], where functions `prod` and `mult` compute respectively the dot product and the matrix dot product. The solution (x) is updated in each iteration by a multiple of the search direction (v). In the same way, the search direction v is updated using the residuals r. We can notice that CG involves one matrix dot product and several vector dot products per iteration.

Algorithm 10 Conjugate Gradient pseudocode.

```
1: Generate a initial value to the solution r = b − Ax_0
2: v = r
3: c=prod(r,r)
4: for k = 0 to k = |maximum_number_iterations| do
5:    z=mult(A,v)
6:    α = c/prod(v, z)
7:    for i = 0 to i = n do
8:       x[i] = x[i] + α ∗ v[i]
9:       r[i] = r[i] − α ∗ z[i]
10:   end for
11:   d = prod(r, r)
12:   β = d/c
13:   for i = 0 to i = n do
14:      v[i] = r[i] + β ∗ v[i]
15:   end for
16:   c = d
17:   Check convergence; continue if necessary
18: end for
```

The conjugate gradient method is often based on sparse matrices that are too large to be handled as dense matrices by a direct implementation. The implementation used to benchmark the performance of FLEX-MPI uses a sparse matrix storage format which provides an irregular memory access pattern. This leads to a less predictive performance behavior, given that the computation cost is related to the sparse matrix structure. The parallel implementation of GC involves communications for exchanging the results of the search directions and residuals.

6.1.1.3 EpiGraph

EpiGraph [MMSC11b, MSMC15b] is an epidemiological simulator for urban environments that operates on a sparse matrix that represents the interconnection network between the individuals in the population. EpiGraph is designed as a deterministic, scalable tool which simulates the propagation of influenza in scenarios that cover extended geographic areas. It consists of three main components: the social model, the epidemic model, and the transportation model.

The social interconnection model is represented via an undirected connection graph that captures heterogeneity features at the level of both the individual and each of his interactions. The epidemic model is specific to the infectious agent under study and extends the classic SIR epidemic model [KM27] to include additional states. Finally, the transportation model reflects the movement of people between cities for work, study, or vacation, and it is based on the gravity model proposed by Viboud *et al.* [VBS+06].

Algorithm 11 shows the pseudocode of EpiGraph's simulation algorithm. The iterative algorithm has four phases which execute every time step and for each one

Algorithm 11 EpiGraph's spatial transmission algorithm.

Require: $(regions, social, status, distance, parameters)$ where *regions* are the urban regions considered in the simulation, *social* is the set of graphs describing the social network of each urban region; *status* contains characteristics and health status of each individual for each urban region; *distance* stores the distance for every pair of urban regions; and *parameters* are parameters of the epidemic model for each individual)

Ensure: $(status)$ where *status* is the updated status of individuals.

```
1: for timestep = 1 → simulation_time do
2:   for all region n ∈ regions do
3:     for all individual l ∈ social_n do
4:       UpdateStatus(l, status_n(l), parameters(l))
5:       if status_n(l) is infectious then
6:         ComputeSpread(l, social_n, status_n, parameters(l))
7:       end if
8:     end for
9:     Interventions(status_n)
10:    for all region m ∈ urban_regions, (m ≠ n) do
11:      Transportation(social_m, social_n, distance_{m,n})
12:    end for
13:  end for
14: end for
```

of the simulated urban regions. The first phase (`line 4`) consists in updating the status of every local individual l based on the epidemic model. The second phase (`line 6`) consists in computing the dissemination of the infectious agent using the social model. The third phase (`line 9`) consists in evaluating both pharmaceutical and non-pharmaceutical interventions in order to mitigate the propagation of the infectious disease. Finally, in the fourth phase, the propagation of the infection via the transportation model (`line 11`) is computed once a day for each pair of urban regions. Each subset of processes corresponding to a region compute the number of individuals which move from this region to another region depending on the size of the populations and the geographical distance between them.

EpiGraph uses a large portion of memory to store each status and connections of each individual. For example, a simulation of an area with 92 cities with an overall population of 21,320,965 inhabitants requires 31.3 GB of memory. This amount of data requires parallel data distribution and processing. EpiGraph's workload, on the other hand, varies over time depending on the number of infected individuals during each iteration. Figure 6.1 shows a performance analysis of EpiGraph running on 8 processes for simulations of different populations: 1,000,000, 2,000,000 and 3,000,000 million inhabitants and a simulated time of 60 days, which corresponds to 86,400 iterations. We can observe the irregular computation pattern performed by the application on each execution phase. In addition, the communication pattern is irregular, and the number of communications and the size of the data sent between processes varies over time. All these features make EpiGraph a challenging application for testing FLEX-MPI.

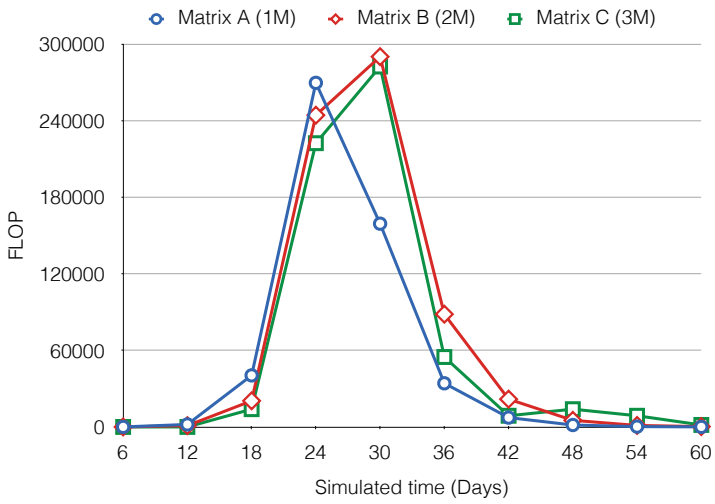


Figure 6.1: Performance analysis of the number of *FLOP* in EpiGraph for varying matrix sizes and 8 processes.

6.1.2 Evaluation platform

The platform we used for evaluation is a heterogeneous cluster which nodes are interconnected via Gigabit Ethernet using a flat network topology. All compute 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 [Sta06].

Table 6.1 summarizes the specifications of each node class that integrate the cluster *ARCOS*. The heterogeneous cluster consists of 26 nodes of four different Intel-based processor types with different performance and hardware characteristics. The amount of available resources varies drastically among the classes, ranging from a single but extremely powerful C8 class node to 20 less powerful C1 class nodes.

Table 6.1: Configuration of the heterogeneous cluster *ARCOS* 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	3	36	Intel Xeon E7-4807	1.87 GHz	128 GB
C6	2	24	Intel Xeon E5645	2.40 GHz	24 GB
C8	1	24	Intel Xeon E5-2620	2.00 GHz	64 GB

Table 6.2: Problem classes for the benchmarks evaluated.

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

In order to perform an elaborated evaluation, we have used different problem classes for our benchmarks. Table 6.2 summarizes the problem classes (A, B, C) used for Jacobi, CG, and EpiGraph in terms of matrix order and number of Megabytes. The dense matrices we use for Jacobi were randomly generated using MATLAB Software [Gui15], a programming environment for numerical computing that includes a wide number of software tools for generating and managing random matrices. The sparse matrices in Conjugate Gradient are a subset of the University of Florida *Sparse Matrix Collection* [DH11], from Timothy Davis and Yifan Hu. We collected these matrices from their publicly available repository of matrices. The sparse matrices used by EpiGraph are generated by the application based on actual data from real social networks [MMSC11b].

6.1.2.1 Cost model

The cost model summarizes the operational cost associated to each computing core of the cluster *ARCOS*. The high-level Adaptive malleability policy of FLEX-MPI uses this cost model to guide performance-aware dynamic reconfiguring actions under the cost constraint.

To perform a realistic evaluation of the high-level malleability policies, 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. We obtained the performance evaluation of different EC2 instances and their economic cost from [IOY+11]. The authors used the High Performance Computer LINPACK benchmark [PWDC05] to report the performance of different instance types in terms of GFLOPS. Table 6.3 summarizes the performance (in GFLOPS), economic cost (in \$/hour), and economic efficiency (in GFLOPS per \$1) per computing core of each instance type.

Table 6.3: 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

Table 6.4: 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 6.4 summarizes the actual performance, costs, and economic efficiencies for each node class of the cluster *ARCOS*. We evaluated the performance of each node class in the cluster *ARCOS* using the HPL benchmark for values of N (order of the coefficient matrix A) between 18,000 and 110,000, depending on the RAM capacity of each node. This approach allows us to achieve the maximum performance in each compute node and consequently, obtain realistic performance values. We then assigned each node class an economic cost that is proportional to those in Table 6.3 in terms of their performance per processor core. We associated each class with an Amazon EC2 instance of similar performance (column *Related Amazon EC2 instance type* in Table 6.4). 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.

6.2 Performance evaluation of the dynamic load balancing algorithm

This section presents a performance analysis of the Dynamic Load Balancing (DLB) algorithm of FLEX-MPI and summarizes the results of the evaluation of this component. The goal of this section is to demonstrate the performance of the algorithm to improve the performance of both regular and irregular parallel applications running on homogeneous and heterogeneous cluster configurations, in dedicated and non-dedicated systems.

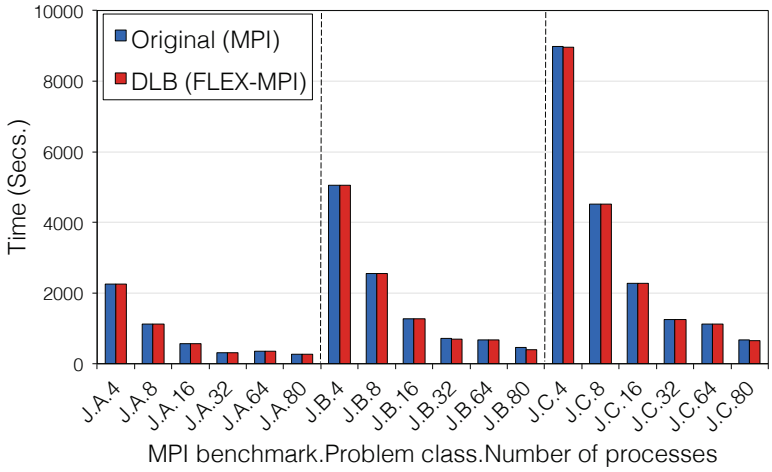
This section is organized as follows: first, we analyzed and evaluated the performance of the algorithm to improve the performance of applications running on a homogeneous, dedicated cluster configuration. Second, we evaluated the capabilities of the algorithm on a performance-wise heterogeneous, dedicated system. Finally, we analyzed the behavior of the load balancing algorithm when external applications with workload that vary over time are sharing the underlying architecture for execution.

To provide a comprehensive evaluation of the DLB algorithm we have evaluated Jacobi, Conjugate Gradient and EpiGraph benchmarks and their respective problem classes described in Table 6.2, running on a wide number of processes ranging from 4 to 80 processes depending on the cluster configuration of the experiment. Every test case is defined using the *X.Y.Z* formula, where X stands for the benchmark application, Y represents the problem matrix, and Z stands for the number of processes of the experiment. In all test cases we compared the performance and load balancing of the application running the unbalanced, native MPI version of the program to the FLEX-MPI version using DLB. Jacobi, Conjugate Gradient and EpiGraph use a block partitioning of the matrix such that every process operates on a data partition that corresponds to the matrix order divided by the number of processes, therefore creating workload imbalance across PEs. For that reason the considered benchmarks provide a good case study for the DLB algorithm and its capabilities.

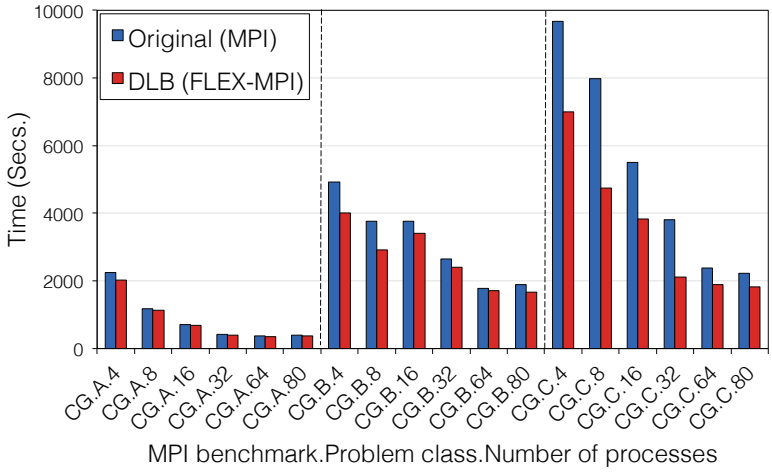
Additionally, we present a performance comparison of the different load balancing policies (`XMPI_LB_NNZ` and `XMPI_LB_WEIGHT`) for those experiments that involve the EpiGraph benchmark. Since the workload of EpiGraph exhibits a correlation to the evolution of the epidemic over time, this benchmark takes advantage of the *weight* policy to balance the workload depending on the number of infected individuals per processing element.

6.2.1 Homogeneous dedicated system

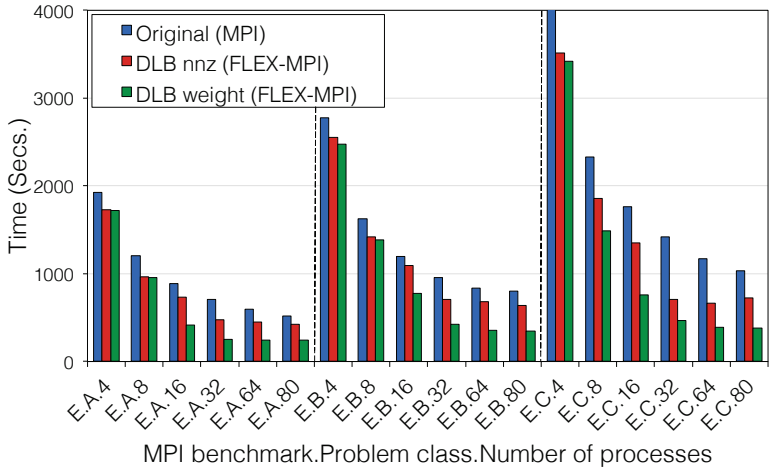
This section evaluates the performance of the dynamic load balancing algorithm for our benchmarks running on an homogeneous dedicated system. The homogeneous cluster configuration consists of C1 class nodes of the *ARCOS* cluster. The number of processes ranges from 4 processes allocated to four C1 nodes to 80 processes allocated to twenty C1 nodes running 4 processes per node.



(a) Jacobi

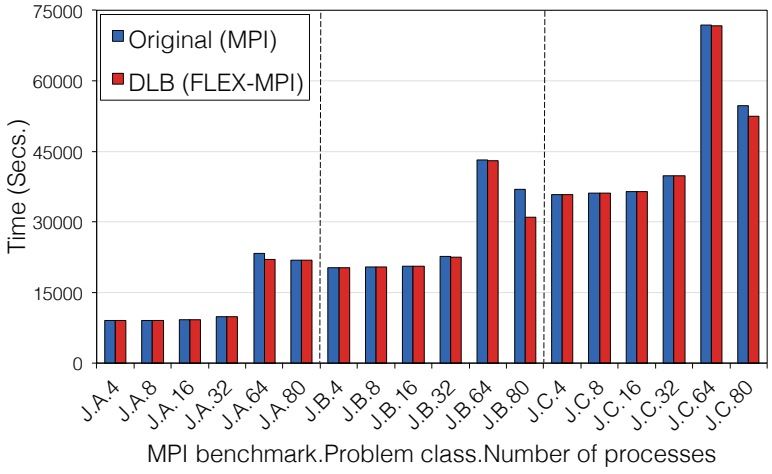


(b) CG

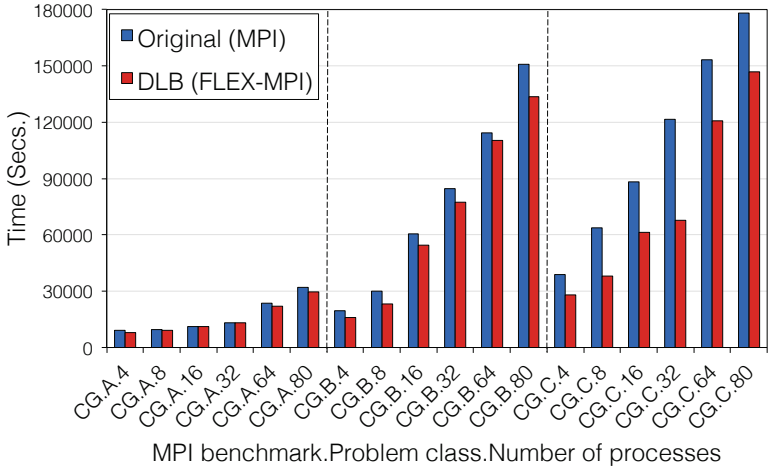


(c) EpiGraph

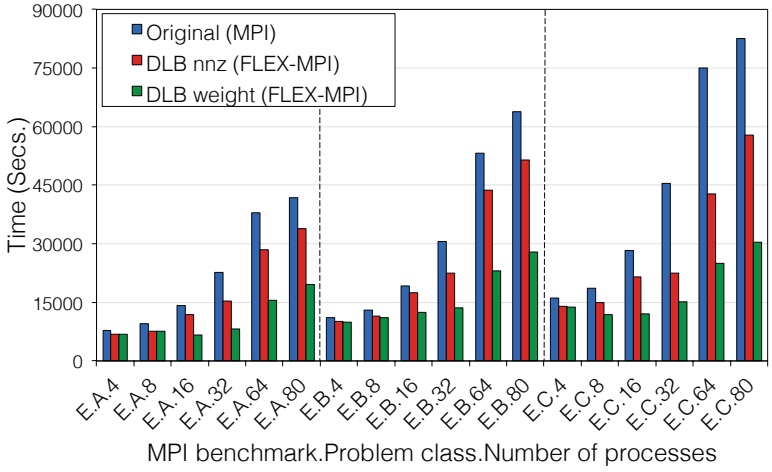
Figure 6.2: Performance evaluation of Jacobi, CG and EpiGraph on a homogeneous dedicated system.



(a) Jacobi



(b) CG



(c) EpiGraph

Figure 6.3: Load balancing of Jacobi, CG and EpiGraph on a homogeneous dedicated system.

Figures 6.2 and 6.3 show the experimental results of the DLB algorithm on the homogeneous cluster configuration for Jacobi, CG and EpiGraph. The performance evaluation takes into account the I/O, computation and communication times of the processes, as well as the FLEX-MPI overhead. Figure 6.2 illustrates the execution time of the original benchmarks running native MPI, and FLEX-MPI using dynamic load balancing. Additionally, for the EpiGraph benchmark these figures show the execution time of the benchmark running FLEX-MPI using the *weight* policy. The X axis displays the test case: the name of the benchmark, the problem class, and the number of processes of the execution. The Jacobi benchmark does not take much advantage of DLB when running on a homogeneous dedicated system because it is a regular application that operates on a dense data structure. However, these results demonstrate that FLEX-MPI does not introduce any overhead on the execution of the program while improving its performance up to 16.05% for J.B.80. The CG benchmark does exhibit a significant performance improvement using DLB that ranges from 0.73% for CG.A.32 to 44.21% for CG.C.32. Likewise, the EpiGraph benchmark exhibits a performance improvement of up to a dramatic 66.64% for E.C.64 when using the *weight* DLB policy. These results demonstrate that the DLB algorithm is a good approach for applications that exhibit dynamic conditions such as CG—operates on a sparse data structure—and EpiGraph—exhibits an irregular computation and communication pattern. These applications are very prone to load imbalance.

Figure 6.3 illustrates the total resource consumption of native MPI and FLEX-MPI benchmarks, computed as the execution time of the application multiplied by the number of PEs used. We can observe that FLEX-MPI enables the parallel application to significantly reduce its resource consumption when using the DLB algorithm for load balancing at runtime. For instance, the resource consumption for EpiGraph in E.B.80 is reduced from 63840 secs. to 51360 secs. when using the *nnz* policy and up to 27840 secs. when using the *weight* policy, which represents a reduced resource consumption of 56.40% regarding the unbalanced execution. The benchmarks exhibit a reduction of their resource consumption of up to 16.05% for Jacobi, between 0.73% and 44.20% for CG, and between 10.53% and 66.63% for EpiGraph. These results demonstrate the capabilities of the dynamic load balancing to improve the efficiency of resources consumption. Besides the performance improvement in terms of speedup and scalability, the DLB algorithm reduces the processor time which may have a huge impact on the economic cost and energy consumption of the computing facilities.

These results reflect the capability of the DLB algorithm to balance the workload of the application at runtime, thus reducing synchronization times between processes and improving their performance by adapting the workload to the computing power of the PE. CG and EpiGraph exhibit a large performance improvement even when running on homogeneous systems because the initial block partitioning leads to unevenly distributed nonzero entries across processes, therefore creating workload imbalance.

6.2.2 Heterogeneous dedicated system

This section evaluates the performance of the DLB algorithm for our benchmarks running on an heterogeneous dedicated system. The heterogeneous cluster configuration consists of nodes of different classes of the *ARCOS* cluster, ranging from 4 to 48 processes. Table 6.5 summarizes the description of the heterogeneous cluster configuration for the test cases considered in the performance evaluation.

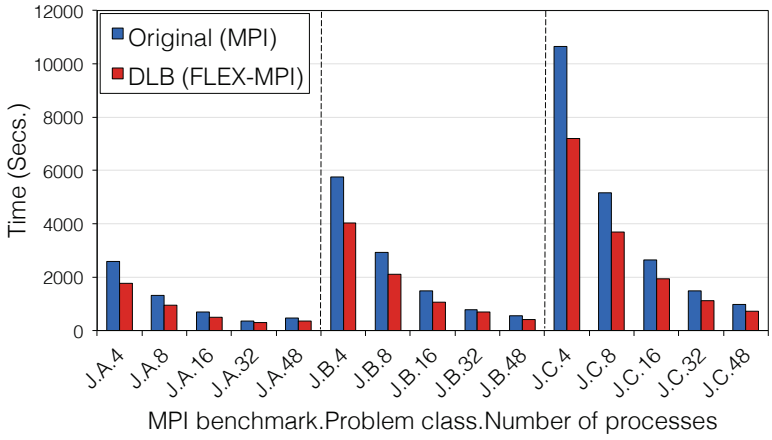
Table 6.5: Number of processes (Np) and their mapping to the available node classes (number of nodes and PEs per node) in the heterogeneous configuration.

	Nodes (PE)			
Np	**C1**	**C7**	**C6**	**C8**
4	1 (1)	1 (1)	1 (1)	1 (1)
8	1 (2)	1 (2)	1 (2)	1 (2)
16	1 (4)	1 (4)	1 (4)	1 (4)
32	2 (4)	1 (8)	1 (8)	1 (8)
48	4 (4)	2 (8)	2 (8)	1 (16)

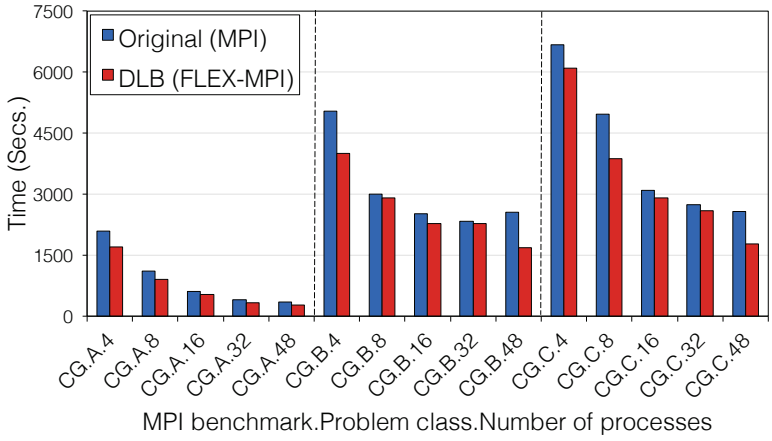
Figures 6.4 and 6.5 show the experimental results of the DLB algorithm on the heterogeneous cluster configuration for our considered benchmarks. Figure 6.4 illustrate the performance evaluation of the benchmarks by means of the comparison of the execution time of the native MPI version to the FLEX-MPI version that uses DLB. The performance improvement ranges from 11.85% (J.B.32) to 32.33% (J.C.4) for Jacobi, 2.02% (CG.B.32) - 30.66% (CG.C.48) for Conjugate Gradient and 18.25% (E.A.4) - 69.21% (E.C.32) for EpiGraph. On the other hand, Figure 6.5 illustrate the total resource consumption of the benchmarks when running native MPI and FLEX-MPI codes on the heterogeneous cluster. These experiments show that DLB can effectively reduce the resource consumption of the application up to 32.45% for Jacobi, 30.71% for CG, and 69.20% for EpiGraph.

These results demonstrate that both regular and irregular applications can take advantage of DLB when running on heterogeneous systems. Besides the uneven distribution of sparse data structures, the initial block partitioning strategy used for most of the applications does not consider the performance difference among PEs. This is because mostly it is difficult to obtain a fair estimation of the mapping of the performance to the data structure—which may be sparse. Additionally, the user often does not have knowledge about the characteristics of the underlying hardware because he does not have privileged access to the system.

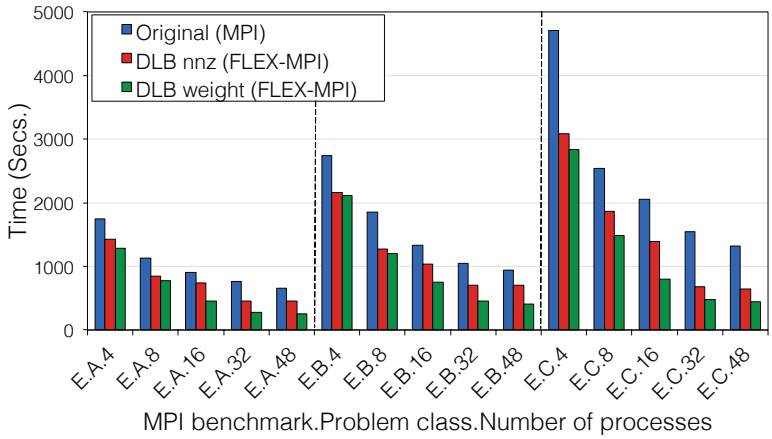
These results reflect the capabilities of the DLB algorithm to balance the workload of the application taking into account the specific computing power of each type of PE, which may vary significantly, and without requiring prior knowledge of the underlying architecture.



(a) Jacobi

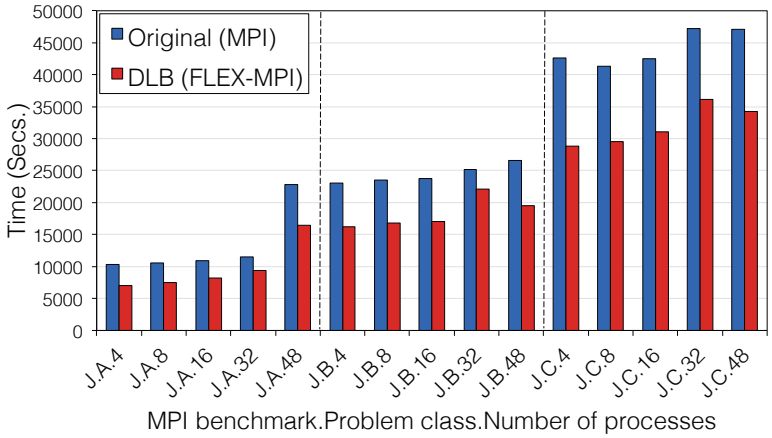


(b) CG

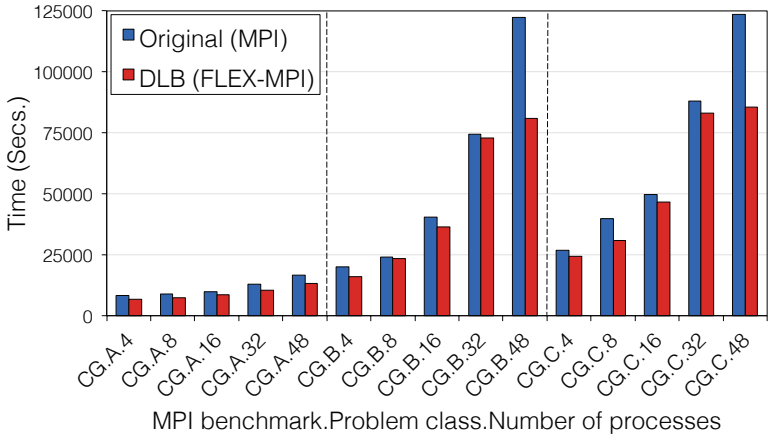


(c) EpiGraph

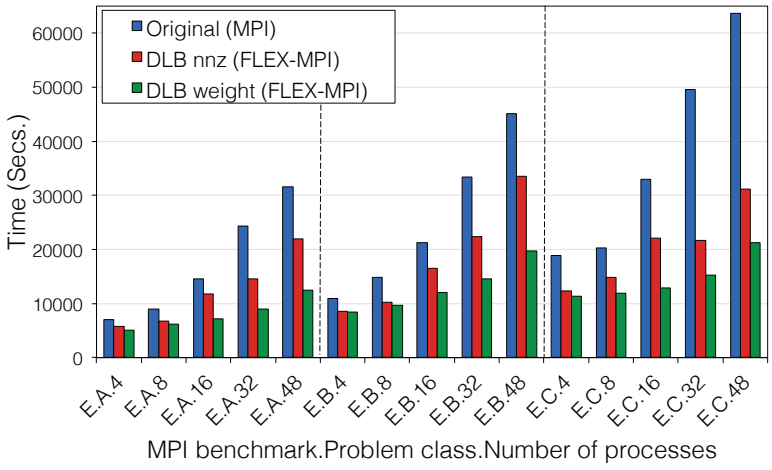
Figure 6.4: Performance evaluation of Jacobi, CG and EpiGraph on a heterogeneous dedicated system.



(a) Jacobi

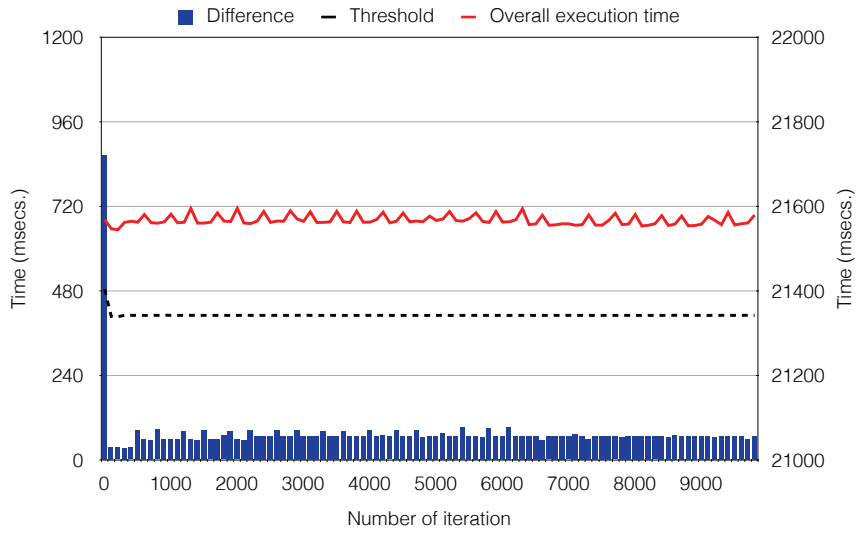


(b) CG

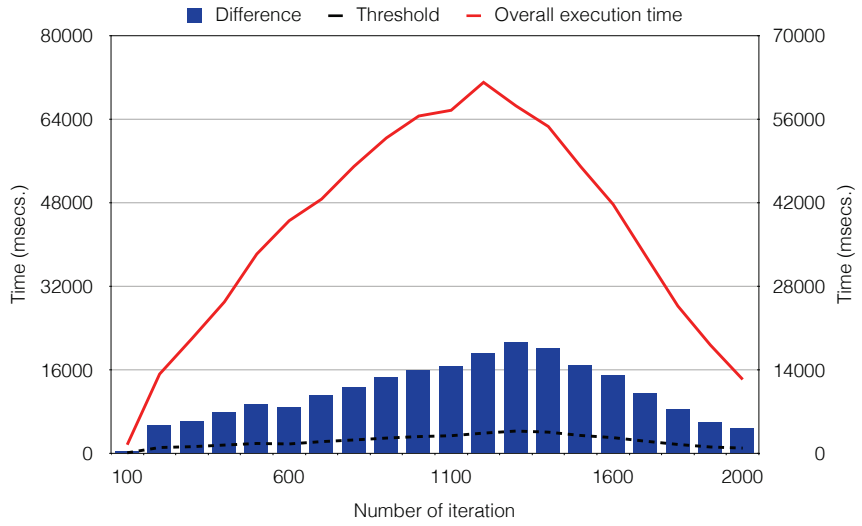


(c) EpiGraph

Figure 6.5: Load balancing of Jacobi, CG and EpiGraph on a heterogeneous dedicated system.



(a) Jacobi



(b) EpiGraph

Figure 6.6: Performance analysis of Jacobi (a) and EpiGraph (b) on a heterogeneous dedicated system.

In order to provide a comprehensive analysis of the DLB algorithm when dealing with irregular applications and heterogeneous systems, Figure 6.6 illustrates a typical execution of Jacobi (a) and EpiGraph (b) running with the support of FLEX-MPI for load balancing. These figures show the overall execution time (red line) of the application, the difference (blue bars) between the fastest and slowest processes, and the threshold value (black dotted line). The threshold value is computed at runtime by the DLB algorithm. When the time difference between processes surpasses this threshold value, the algorithm detects load imbalance and then triggers a load balancing operation in order to rebalance the workload of the application.

Due to the regular computation pattern of Jacobi the amount of work done in each iteration is the same. This leads to very small variations over time in the execution time per iteration. When executing on a heterogeneous system, Jacobi requires a single data redistribution operation to balance the workload. It is carried out during the first sampling interval, in which the workload imbalance is larger than the imbalance tolerated by the algorithm. From that moment on the application is balanced and no further data redistribution operations are necessary. In contrast, EpiGraph is an irregular application which exhibits a highly variable workload per iteration. This kind of applications may require several data redistribution operations to balance the workload because the variations over time create performance differences that surpass the threshold value of the DLB algorithm in each sampling interval.

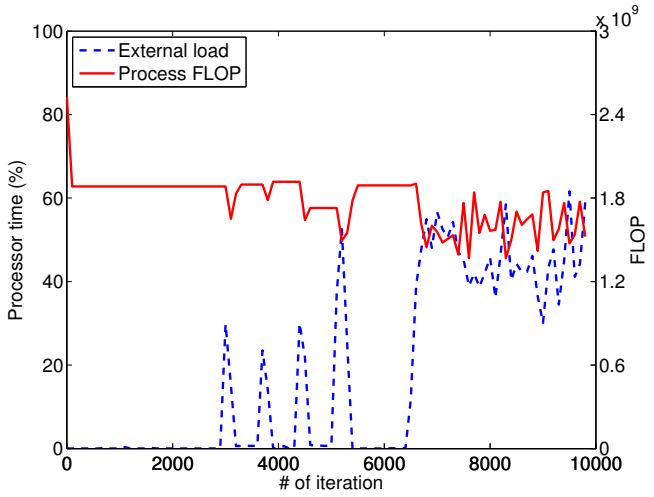
6.2.3 Non-dedicated system

This section evaluates the performance of the DLB algorithm for our benchmarks running on a non-dedicated system. The primary goal of the evaluation is to analyze the capability of the DLB algorithm to adapt the workload of the application to the dynamic execution environment at runtime.

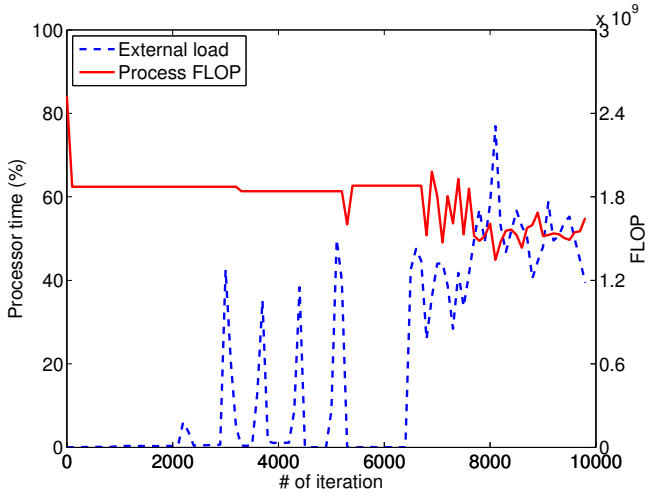
The following experiment evaluates how well the load balancing algorithm performs when external applications with workload that vary over time are sharing the underlying architecture for execution. We run Jacobi for a heterogeneous configuration with one C1 node and one C6 node, each running 4 processes per node. We artificially introduce an external load which simulates an irregular computing pattern, creating an interference in our application. This interference is created by two processes which are simultaneously executed on the C1 node together with the target application. The interference consists of a burst of short computing intervals followed by a single long computing interval which lasts until the end of the execution.

Figure 6.7 shows what happens on one of the PEs when we run Jacobi using FLEX-MPI with DLB and we introduce an interference on a subset of the processing elements. The workload redistribution triggered by the load balancing algorithm leads to a different number of $FLOP$ performed by the process (in red). The amount of data which needs to be redistributed depends on the magnitude of the interference (in blue) and the value of the k parameter—described in Algorithm 5. The k parameter allows the user to modify the behavior of the DLB algorithm in response to the changes in the execution environment such as it represents the number of consecutive sampling intervals during which an interference is considered as long term, thus triggering a load balancing operation. Small values of k lead to load balancing operations each time an interference is detected by the algorithm. Otherwise, large values of k can tolerate external loads that last several sampling intervals.

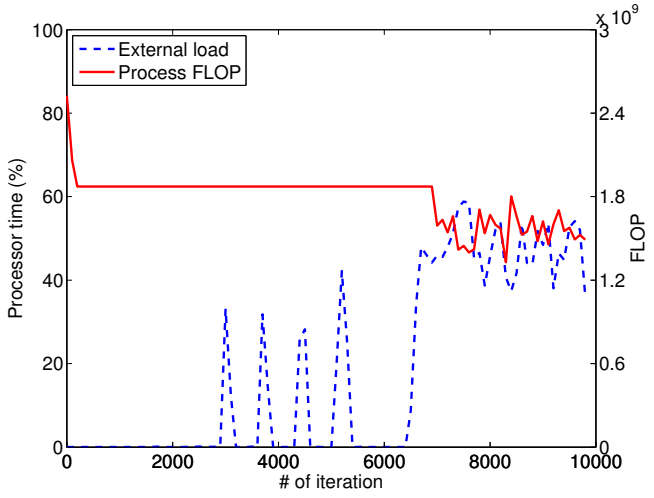
Returning to Figure 6.7, we can observe that for the smallest value $k = 1$ the application adapts immediately to changes in the performance of the processing element, carrying out load balance for every external load burst. With $k = 3$ the first smaller interference bursts are discarded, then the application adapts to the larger one. Finally, larger values of k lead to discarding all the interference bursts but considering the long-term load.



(a) $k = 1$



(b) $k = 3$



(c) $k = 5$

Figure 6.7: Performance analysis Jacobi on a heterogeneous non-dedicated system for different values of k: (a) $k = 1$, (b) $k = 3$, and (c) $k = 5$.

6.3 Computational prediction model validation

FLEX-MPI uses a computational prediction model (CPM) that increases the accuracy in the search of best execution scenario (by spawning or removing processes) that better meets the performance requirements. The model consists of several components for predicting computation and communication times, overhead of creation and termination of MPI processes and overhead of data redistribution operations. Each one of these components has to be tuned according to the application and hardware characteristics.

This section summarizes the performance evaluation of the CPM to predict, for a given reconfiguration scenario (change the number of processes to run on a different processor configuration), both the computation and communication times as well as the overhead associated to the application reconfiguration (process creation and removal, and data redistribution). Prior to the model evaluation, we describe the experiments conducted in order to collect the system parameters required by the computational prediction model for our considered benchmarks.

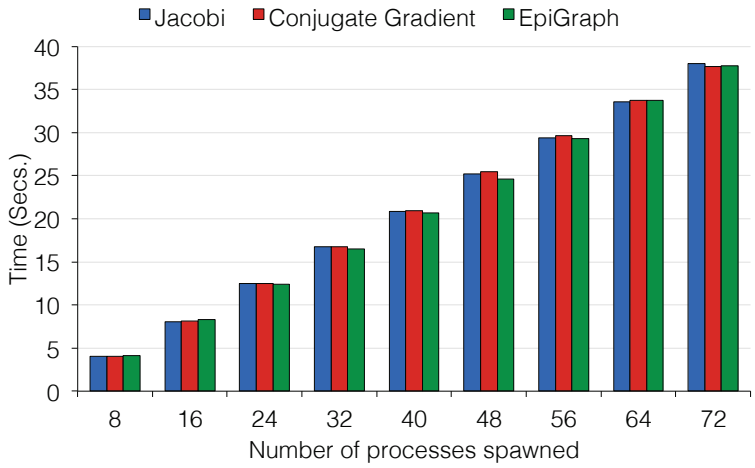
6.3.1 Parameter values

The CPM requires as input a set of hardware-specific parameters that are necessary to compute the predicted values for creation and termination of MPI processes, as well as the network parameters to predict the performance of communication operations in the parallel application and the data redistribution overhead. This section summarizes the experiments carried out to collect the values of the parameters that correspond to the overhead of MPI process creation and termination for Jacobi, CG, and EpiGraph, and network performance for the cluster *ARCOS*.

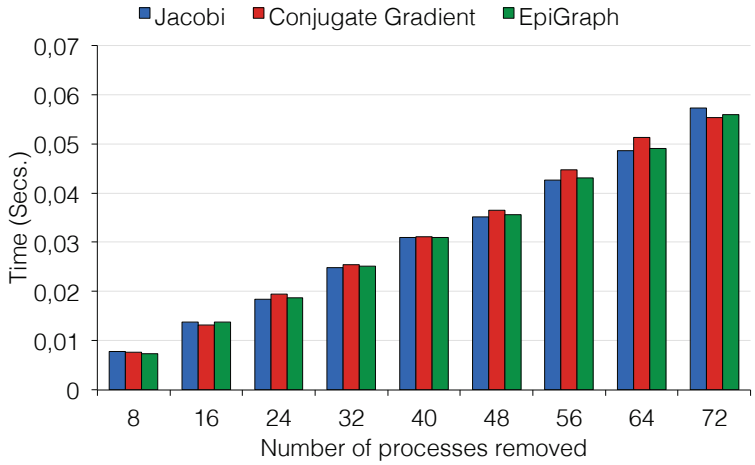
6.3.1.1 Creation and termination of MPI processes

The overhead of process creation and termination depends on the number of process spawned or removed, the operating system, the MPI implementation, and the size of the program binary. Therefore, these overheads are specific to the system environment and the application. We measure the overhead of process creation and termination in the cluster *ARCOS* using predefined reconfiguring points during the execution of the benchmarks. These values are used to effectively predict the overhead of reconfiguring actions in FLEX-MPI.

Figures 6.8 (a) and (b) show the overheads of process creation and termination for Jacobi, CG, and EpiGraph in the cluster. These results are the average values of multiple repetitions of the experiments. The size of the binaries are 28KB each for Jacobi and Conjugate Gradient, 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 6.8 (a) the x value 8 means that the application goes from executing on 8 to 16 processes.



(a)



(b)

Figure 6.8: Measured times for the creation and termination of dynamic MPI processes on 20 computing nodes of class C1.

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 6.8 (b)—by starting the application with 48 processes, spawning 32 dynamic processes, then removing them and measuring the time spent in this last operation.

Table 6.6 shows the average creation and destruction times in the cluster *ARCOS* for our benchmarks. These values show that in the case of our benchmarks the creation and destruction costs do not depend on the binary size but of the number of processes that are created or destroyed, and the execution environment (operating system and MPI implementation).

Table 6.6: Average process creation and termination overheads for Jacobi, CG, and EpiGraph in the cluster *ARCOS*.

Action	Average time (ms)	Standard deviation (ms)
Creation	519.700	8.000
Termination	0.801	0.060

Table 6.7: Hockney model parameters measured on the ARCOS cluster.

Parameter	Description	Measured value
α	Latency	100 μsecs.
β	Transfer time	0.008483 μsecs. per byte
γ	Computation cost of reduction operation	0.016000 μsecs. per byte

6.3.1.2 Network parameters

The CPM uses the communication models of MPICH implementation to predict the performance of communication operations in FLEX-MPI. This allows FLEX-MPI to model the performance of communications of the parallel application as well as the performance of data redistribution operations.

The communication models of MPICH are based on the Hockney model [Hoc94], which characterizes a communication network in terms of latency and bandwidth. Every communication model based on the Hockey model requires the following parameters: α is the network latency, and β is the transfer time per byte. Additionally, the communication models of MPICH use a γ parameter—used for reduction operations—that represents the computation cost per byte for performing the reduction operation locally on any process.

Given the fact that the network topology of the cluster *ARCOS* is flat, we assume that latency and bandwidth are equal for all the node pairs. Therefore we conducted several point-to-point experiments in order to obtain the values of the parameters of the Hockey model in the cluster. Our network benchmark measures the network latency and bandwidth between all the node pairs of the system, for data streams of different sizes using UNIX-based network performance tools. Specifically, our benchmark uses `tcpping` [Ste04] to measure the network latency and `Iperf` [TQD+05] to measure bandwidth performance. We fine-tuned specific command options in order to adapt the tools to the network performance of the cluster and obtain the most accurate results.

Table 6.7 summarizes the average parameter values for the Hockney model measured for multiple repetitions by our network benchmark on the cluster *ARCOS*. The α parameter was directly reported by the network benchmark, while β was computed using the network bandwidth and the size of data transferred between nodes. Furthermore, the γ parameter was computed by performing several MPI reduction

operations (i.e. `MPI_Reduce` and `MPI_Allreduce`) and measuring the average time per byte spent by each process to compute the reduction operation.

6.3.2 Model validation

Once we collected all the parameters for the benchmarks and the evaluation platform, we feed the computational prediction model with those values and validate its capabilities to predict the performance of the parallel application prior to reconfiguring actions. First we validate the capabilities of the CPM to predict the performance of communication operations in FLEX-MPI applications. We evaluated the accuracy of the estimations of the CPM by comparing predicted times versus measured times for a wide range of MPI-based communication operations. To be specific, we evaluated the `MPI_Send` and `MPI_Recv` point-to-point operations, as well as the collective operations `MPI_Bcast` (one-to-many), `MPI_Gather` (many-to-one), and `MPI_Allreduce` (many-to-many). These operations represent the whole diversity of MPI communication types: one-to-one, one-to-many, many-to-one, and many-to-many.

The MPICH distribution we are using implements a set of optimizations for intra-node communication. It is important to evaluate whether our evaluations are correct in the case that there is more than one process mapped to the same node. For these tests we used nodes of class C1 which have a multi-core processor with 4 cores.

For the experiments consisting of 4, 8, and 16 processes we used 4, 8, and 16 nodes respectively with one process per node. For the experiments with 32 processes we used 16 nodes with two processes per node; for those with 64 processes we used 16 nodes with 4 processes per node. This allows us to measure the performance of collective operations for both inter-node and intra-node scenarios. Figures 6.9, 6.10, 6.11 and 6.12 show that the estimated time (using Hockney model) is a good approximation for the measured time, for each of the MPI operations that we evaluated: `MPI_Send` and `MPI_Recv`, `MPI_Bcast`, `MPI_Gather`, and `MPI_Allreduce`. These results demonstrate the capabilities of the CPM to predict the performance of MPI operations in a FLEX-MPI application.

Once we have validated the network performance estimation of the CPM, we validate its functionalities to predict the execution time of a parallel application for all its execution phases: computation, communication, process reconfiguring overheads, and data redistribution overheads. To validate the CPM we execute a modified Jacobi code in which the reconfiguring actions are predefined to occur at a particular iteration. Then we validate the model by comparing predicted times—computed by the CPM—and real times—measured in the program.

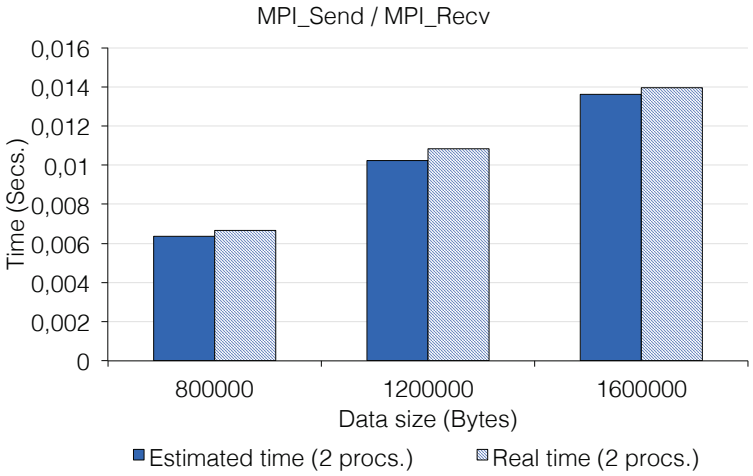


Figure 6.9: Comparison between predicted and measured times for MPI Send and Recv (`MPI_Send`/`MPI_Recv`).

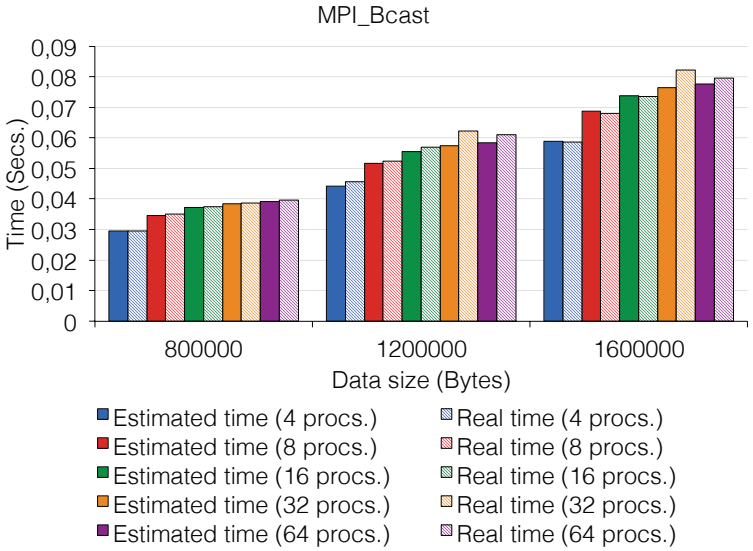


Figure 6.10: Comparison between predicted and measured times for MPI Broadcast (`MPI_Bcast`).

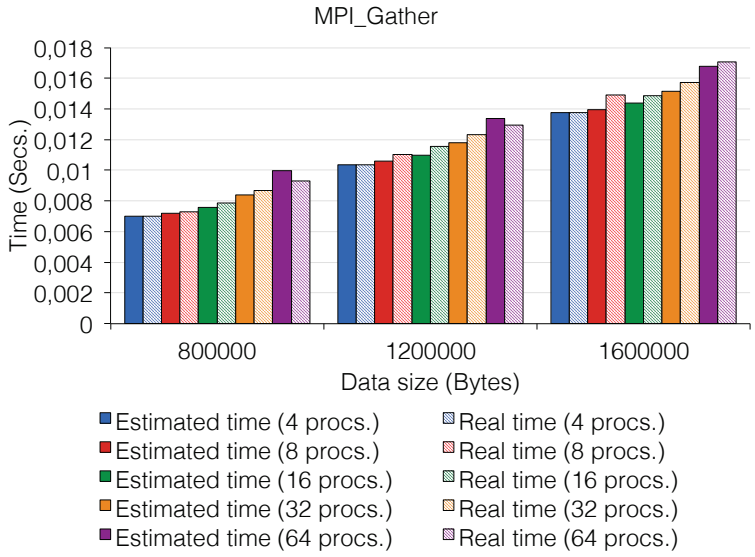


Figure 6.11: Comparison between predicted and measured times for MPI Gather (`MPI_Gather`).

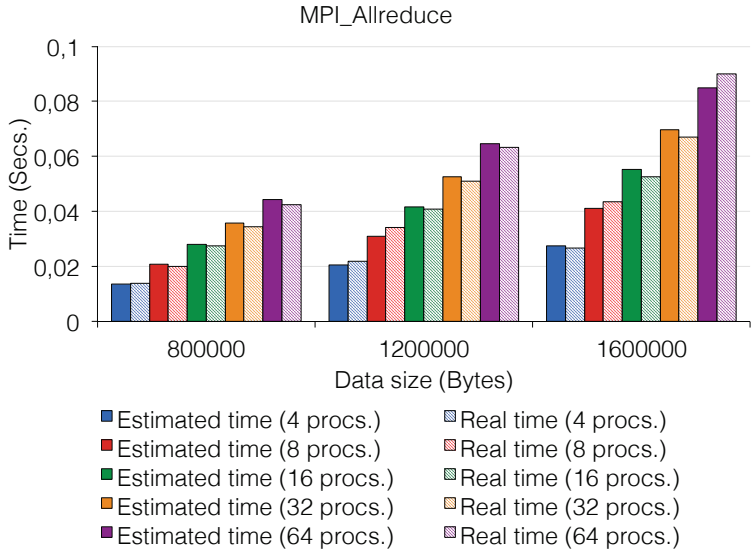


Figure 6.12: Comparison between predicted and measured times for MPI All reduce (`MPI_Allreduce`).

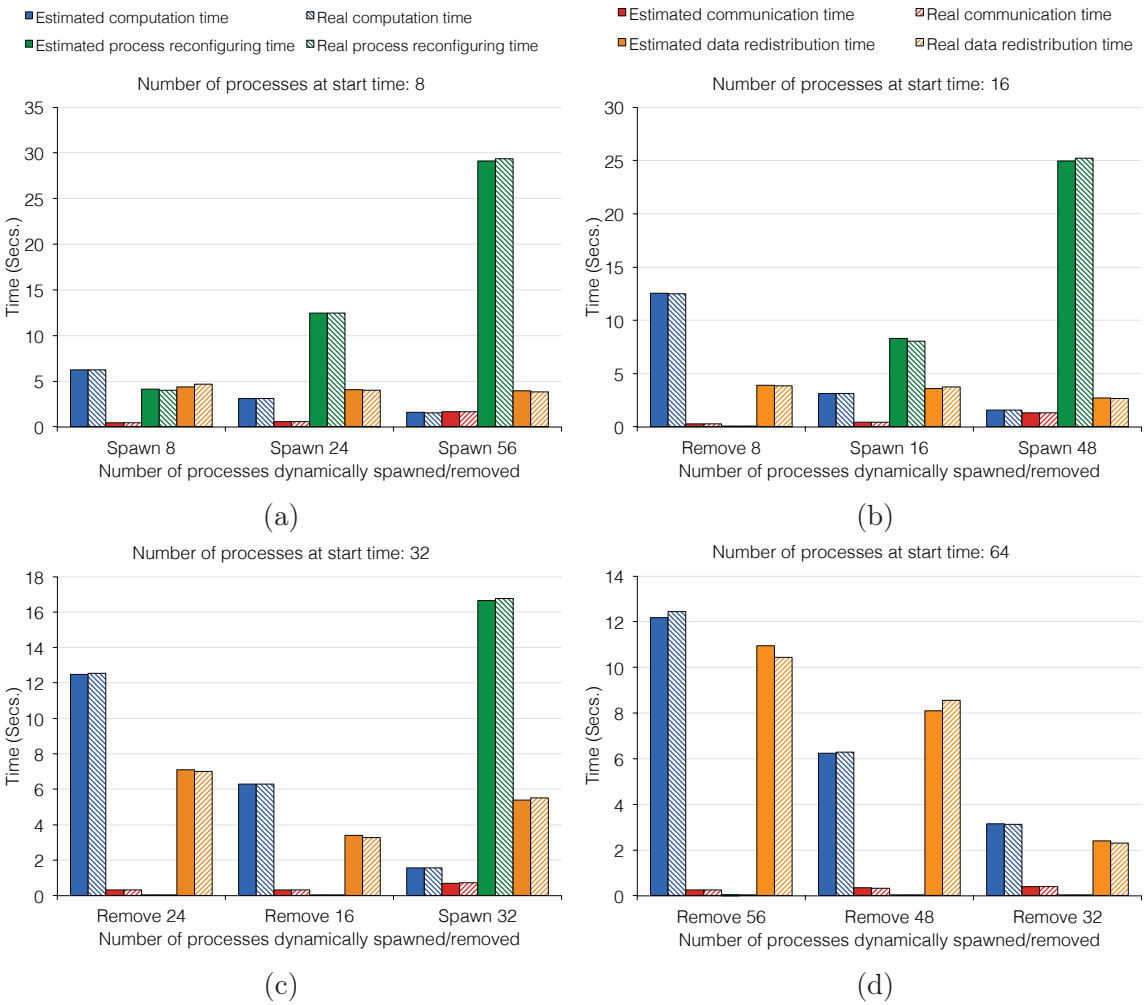


Figure 6.13: Comparison between average predicted and real times for different dynamic reconfiguring actions carried out for Jacobi benchmark running on (a) 8, (b) 16, (c) 32, and (d) 64 processes.

Table 6.8: Average relative error and relative standard deviation for CPM estimations.

Phase	**Relative Error (%)**	**Relative Standard Deviation (%)**
Computation	1.47%	0,62%
Communication	4.61%	1,31%
Process reconfiguring	2.23%	0,94%
Data redistribution	5.60%	1,63%

Figure 6.13 shows a comparison between the average 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.

Table 6.8 summarizes the average relative error (RE) and the average relative standard deviation (RSD) of the CPM to predict: (1) the computation times, (2) communication times, (3) process reconfiguring overheads, and (4) data redistribution overheads.

These results demonstrate that the computational prediction model can predict with great accuracy the performance of parallel applications prior to a reconfiguring action, either to spawn new processes or to remove currently allocated processes, ranging from 8 to 64 processes at runtime. The average values computed for RE and RSD show that the number of processes involved in the reconfiguring action does not have such a big impact on the accuracy of the prediction computed by the CPM.

The most accurate estimations correspond to those of computation and process reconfiguring times, since the benchmark considered exhibits a regular computation pattern and we feed the CPM with the overhead of process creation and termination obtained in Section 6.3.1.1. On the other hand, the estimated times for communication and data redistribution phases exhibit a slightly higher relative error due to the usual fluctuations on the network performance during runtime. These results enable the performance-aware malleability policies to reconfigure the application to the number of processes that better satisfies the goals of the policy with great accuracy.

6.4 Performance evaluation of malleable MPI parallel applications

This section summarizes the experimental evaluation results of the malleability policies: Strict Malleability Policy (SMP), High Performance Malleability Policy (HPMP), and Adaptive Malleability Policy (AMP). We focus our analysis on two main topics: the library overhead and the capabilities of the malleability policies to perform dynamic reconfiguring actions in order to achieve their performance goals. We conducted an exhaustive number of experiments in the cluster *ARCOS* for both homogeneous and heterogeneous configurations for the three considered benchmarks. The following sections describe our observations on the experiments and the results obtained.

6.4.1 Strict malleability policy

This section analyses the performance of the SMP policy (Section 5.4.1) to adjust the number of processes to the availability of system resources. The SMP policy automatically extends and shrinks the number of processes of the application depending on the current number of available processing elements in every sampling interval. The SMP does not focus on improving the performance of the application but enables the parallel program to use a variable number of processes to take advantage of idle resources at runtime without user intervention. If the number of processes is large enough, it may degrade the performance of the application due to fine-grained granularity which increases the synchronization overheads.

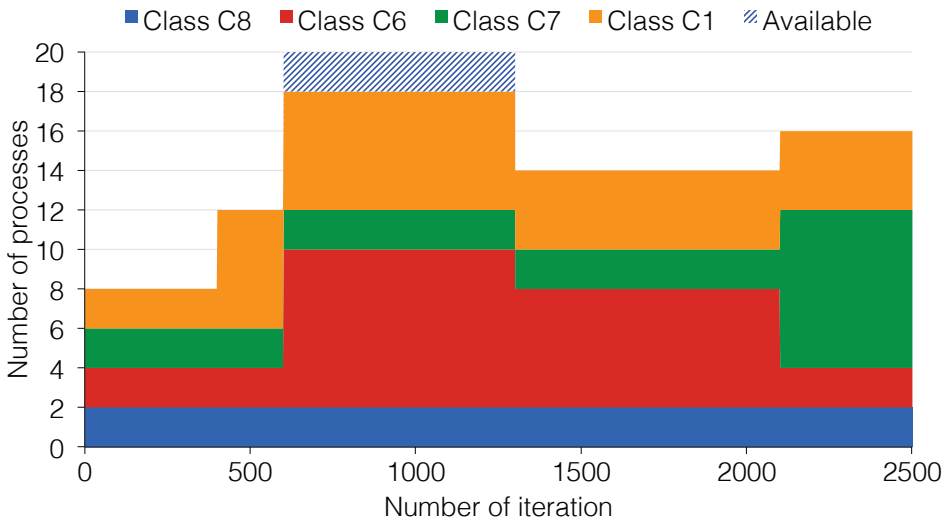


Figure 6.14: Number of processes and type of processors scheduled for Jacobi and total number processing elements available in the system.

Figure 6.14 illustrates the number and type of resources scheduled for Jacobi when running on a subgroup of the cluster *ARCOS* which consists of two C1 nodes,

one C7 node, one C6 node, and one C8 node. Initially, the application runs on 8 processes which corresponds to the number of PEs available in the system—2 PEs per class—and we limit the maximum number of processes to 18 processes. We can observe that, as the number of available PEs increases gradually from 8 to 12, SMP spawns new processes to take advantage of these resources. At iteration 600, 8 new PEs become available. However, the SMP policy does not schedule more than 18 processes because that is the limitation that we imposed to the policy.

These results demonstrate the capabilities of FLEX-MPI to change the number of processes and redistribute the workload at runtime, while the SMP policy reacts to changes in the dynamic execution environment by adjusting the number of processes of the application to the available resources.

Then we evaluate the performance of the SMP policy using predefined process scheduling. This feature allows the user to control over the process scheduling via a configuration file that describes the number of processes, the type of involved processors, and the iteration number in which the reconfiguring action must be triggered. Figure 6.15 illustrates the performance of EpiGraph when running on a fixed number of processes (left Y axis) and simulating the spread of influenza in a population of size 2M for 60 days. We can observe that the execution time per sampling interval (in black, right Y axis) increases as the epidemic spreads on the population and the number of infected people reaches its highest point at 30 days (43200 iterations). From that point on the number of infected people decreases until the end of the simulation. This test demonstrates that irregular applications such as EpiGraph make an inefficient use of resources when using static scheduling because they can be underutilized in some phases of the execution—when the workload is very low—while more resources would be welcome in those phases of highest workload.

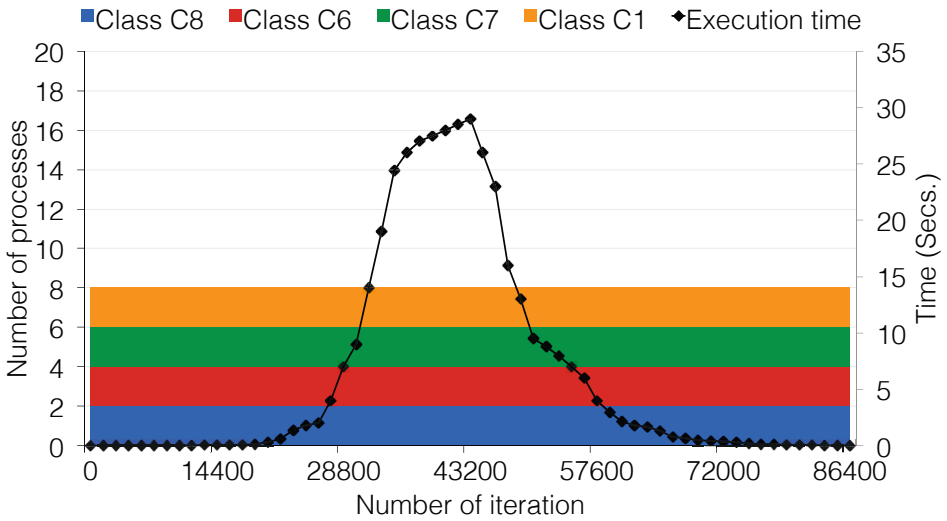


Figure 6.15: Performance evaluation of EpiGraph that illustrates the number of processes (left Y axis) and execution time (right Y axis) per sampling interval when using static scheduling without FLEX-MPI support.

Otherwise, when both the application's computation pattern and the system's performance are well known, a predefined scheduling can significantly improve the performance of the application and the efficiency usage of system resources. Figure 6.16 illustrates the performance of EpiGraph when using the SMP policy with a predefined scheduling. The number of processes scheduled (left Y axis) increases for the compute-intensive phases of the application, then the additional processes are removed when they are not required anymore. This behavior leads to an efficient usage of the resources that increases the performance of the application by reducing the execution time per sampling interval (right Y axis).

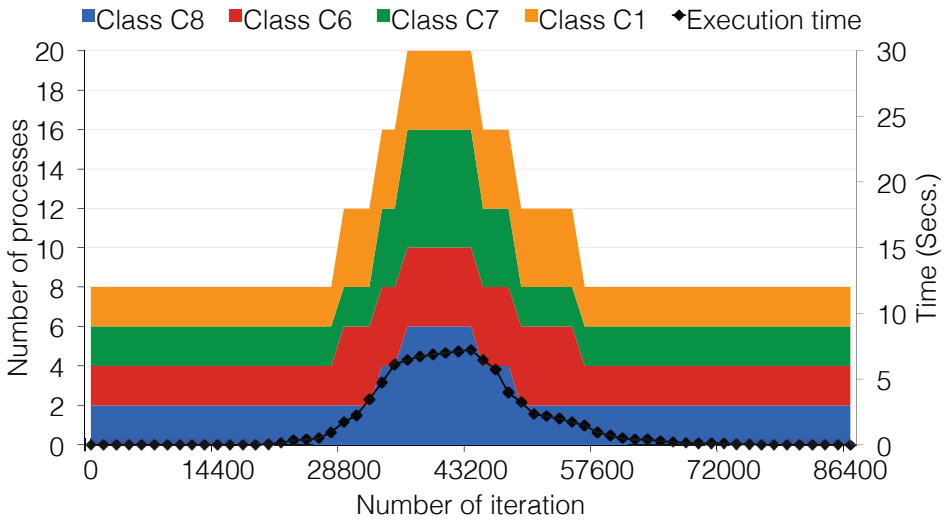
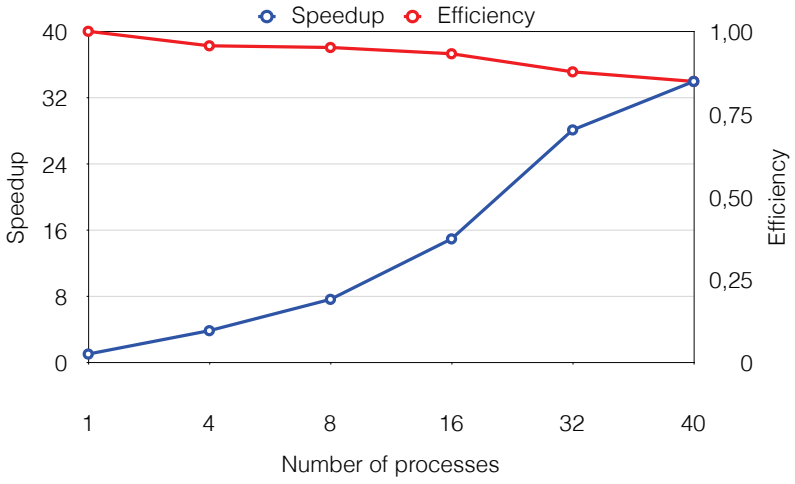


Figure 6.16: Performance evaluation of EpiGraph that illustrates the number of processes (left Y axis) and execution time (right Y axis) per sampling interval when using predefined dynamic scheduling (b).

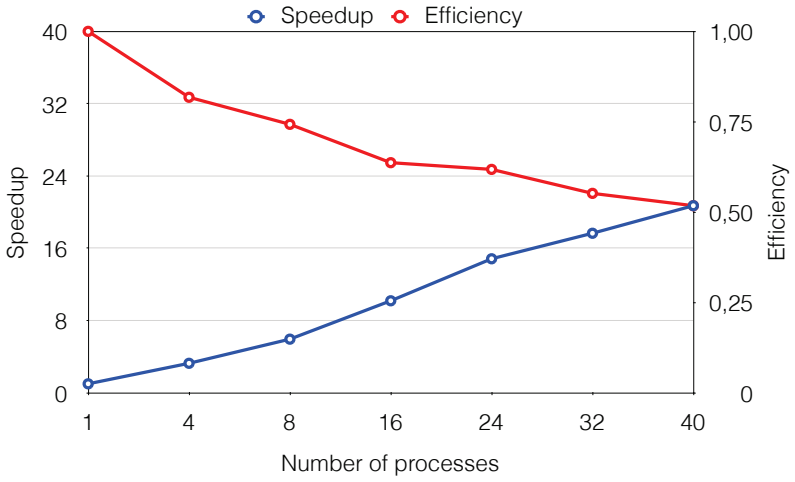
These results demonstrate that the SMP policy can significantly improve the performance of the application using predefined scheduling via simple configuration file. However, we have to note that a limitation of this policy is that it requires prior knowledge of the application's computation pattern and the underlying hardware, which is not always feasible. The SMP policy is not performance-aware so the user has to provide FLEX-MPI information about the specific systems resources to use during the program execution.

6.4.2 High performance malleability policy

This section evaluates the performance of FLEX-MPI applications executing the performance-aware HPMP policy (Section 5.4.2) to efficiently adjust the number of processes to the availability of system resources. The goal of this section is to demonstrate the capabilities of the HPMP policy to adjust the number of processes to the processor configuration that provides the highest application performance in terms of completion time.



Jacobi (a)



Conjugate Gradient (b)

Figure 6.17: Performance analysis of Jacobi (a) and Conjugated Gradient (b) that shows speedup (left Y axis) and efficiency (right Y axis).

We evaluate the performance of the HPMP policy on a subgroup of the cluster *ARCOS* which consists of 8 compute nodes of the C1 class—that is, 32 PEs. First we analyze the performance of our considered benchmarks when running on the evaluation platform. Figure 6.17 illustrates the speedup and parallel efficiency of Jacobi (a) and Conjugate Gradient (b). The test case for Jacobi computes the problem size B for 2,500 iterations, while the test case for CG computes the problem size B for 20,000 iterations. We can observe that for both applications the speedup increases as the number of processes grows. However, the parallel efficiency gets degraded as the number of processes increases—which means that neither Jacobi nor CG scale well for a fixed problem size. The performance analysis shows that Jacobi is more efficient and scalable than CG.

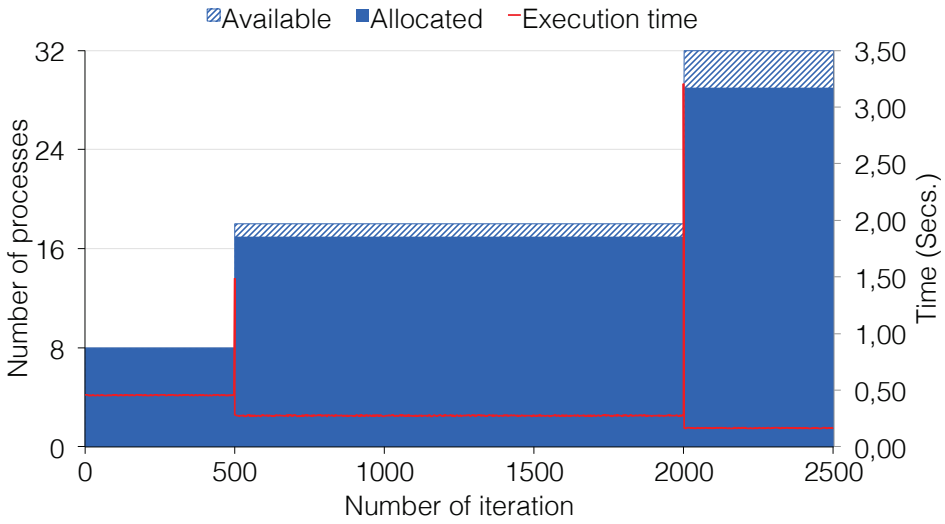


Figure 6.18: Number of processes allocated by HPMP, number of PE available (left Y axis) and execution time per iteration (right Y axis) for Jacobi.

Then we evaluate the performance of parallel applications when running with the support of the HPMP policy. Figure 6.18 illustrates performance of Jacobi when running with the support of HPMP for malleability. The figure shows the number of processing elements available and the number of processes allocated by the HPMP policy for the application (left Y axis) and the execution time per iteration (right Y axis). Initially the parallel program runs on 8 processes available in the system. At iteration 500 10 additional PEs became available. The HPMP analyzes the performance of the application and decides to allocate 9 of those PEs for the program. We have to take into account that, although one additional process may provide an improved performance, the malleability policy also computes the overhead of reconfiguration to make decisions. This overhead—which is computed in the figure as part of the execution time of the application—can have a significant impact on the performance of the application that leads to a larger completion time. At iteration 2000 14 additional PEs become available. In that case, the HPMP policy allocates 12 of them for the application. As in the prior malleable action, we observe a trade-off between the reconfiguration overhead and the application performance. In this example the overhead of reconfiguration has a significant impact on the application because, although three additional processes can enhance the performance of the program, a larger number of processes implies larger reconfiguration overheads that may degrade the overall performance of the application.

Figure 6.19 illustrates the results of the performance evaluation of CG when running with the support of the HPMP policy. We can observe that, at iteration 10,000 the malleability policy allocates a number of processes (3) that is by far smaller than the available (10). That is because the parallel efficiency of CG degrades for a large number of processes. In this case the HPMP considers that allocating more processes can have a negative impact on the performance of the program due to

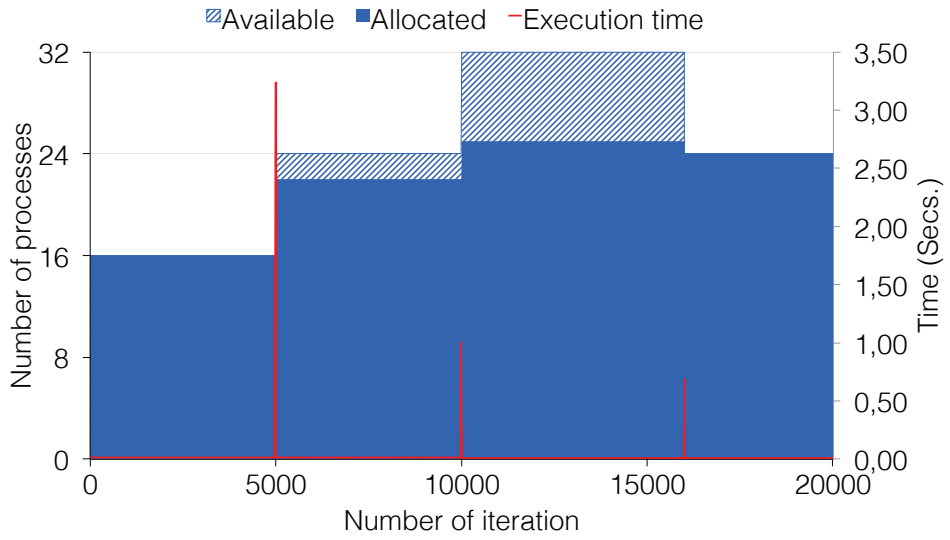


Figure 6.19: Number of processes allocated by HPMP, number of PE available (left Y axis) and execution time per iteration (right Y axis) for Conjugate Gradient.

its poor scalability and the overhead of reconfiguration. At iteration 16,000 we can observe that HPMP removes processes from the application when the RMS requests resources on behalf of another application with highest priority.

These results have demonstrated the capabilities of the HPMP policy to analyze the performance of the application and adjust the number of processes to a processor configuration that improves its performance, both for scalable and inefficient parallel applications. We have observed that the HPMP computes the overhead of processes creation and termination and data redistribution to make decisions about performance-guided reconfiguring actions. In any case, every reconfiguring action performed by the HPMP policy means a performance improvement for the parallel program in terms of completion time.

6.4.3 Adaptive malleability policy

This section evaluates the performance of FLEX-MPI applications executing the performance-aware AMP policy (Section 5.4.3) to efficiently adjust the number of processes to satisfy the user-defined performance objective under performance constraints.

This section is organized as follows: first, we present the test cases we considered to evaluate the functionalities of the AMP policy. Second, we present an elaborated performance analysis of the AMP policy for some of the test cases described in the previous subsection. Finally, we summarize the results of the performance evaluation of the AMP policy for our considered test cases.

6.4.3.1 AMP test cases

We evaluate the performance of the AMP policy for our considered benchmarks (Jacobi, CG, and EpiGraph) using a wide number of test cases by using different input matrices and initial processor configurations. Table 6.9 summarizes the test cases that we considered and the number of initial processes and types of processors for each one of them.

Table 6.9: Number of processes initially scheduled (Np) and their mapping to the available class nodes for each test case.

				Process mapping			
Test case	**Benchmark**	**Problem class**	**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

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 maximum number of processors available for each benchmark application corresponds to the number of resources of the cluster *ARCOS* as shown in Table 6.1, and we use the cost model as described in Table 6.4.

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 (which corresponds to 1 day of simulation). To provide a fair comparison we apply load balance in both static and dynamic scenarios. For Jacobi and CG benchmarks

the CPM estimates at runtime the computation times for each sampling interval; for EpiGraph we profiled a previous execution of the application to collect the number of $FLOP$ performed by the application, which results are shown in Figure 6.1.

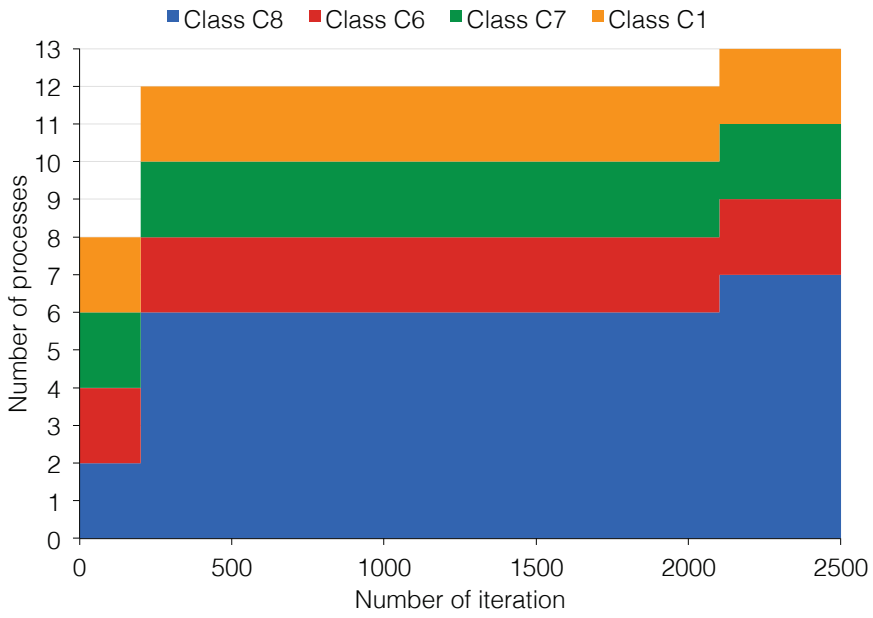
6.4.3.2 Performance analysis of AMP

This section presents an elaborated performance analysis of the AMP policy for J.B.8 test case in order to better understand the behavior of the malleability policy. We profiled the performance of the benchmark application at runtime to analyze the dynamic reconfiguring actions performed by the AMP policy.

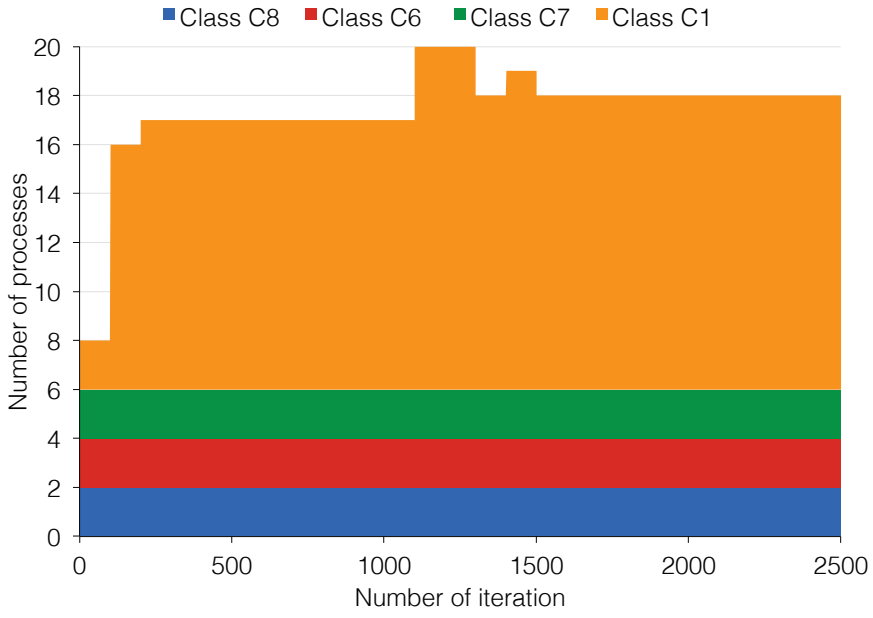
Figure 6.20 shows a comparison between the behavior of the reconfiguring policy module under efficiency (a) and cost (b) constraints when executing J.B.8 with a performance improvement objective of 35%. The statically scheduled application takes 923.453 seconds and 1,535 cost units to complete on eight processors.

When we impose the efficiency constraint—Figure 6.20 (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 6.20 (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.

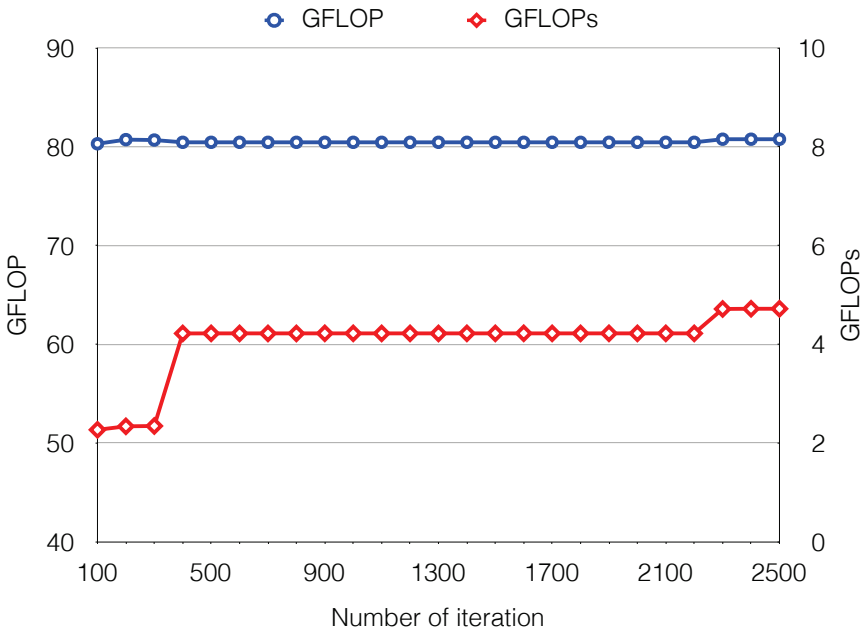


(a)

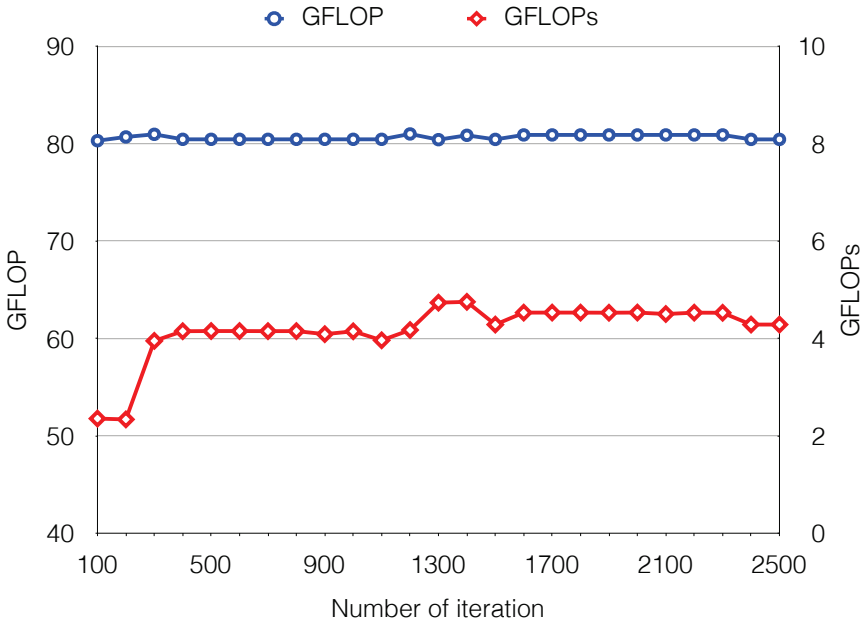


(b)

Figure 6.20: Number of processes and type of processors scheduled by the AMP policy for the execution of J.B.8 under the efficiency (a) and cost (b) constraints.



(a)



(b)

Figure 6.21: 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.

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 as the AMP policy dictates. These results show the behavior of the AMP policy algorithm to reconfigure the application at runtime in order to satisfy the performance objective and constraints.

In order to take a deeper look at the runtime performance of the benchmark, we profiled the application to collect the number of GFLOP of J.B.8 under efficiency and cost constraints. Figure 6.21 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 under the efficiency (a) and cost (b) constraints. The workload stays by and large constant in both cases, regardless of the number of simultaneously executing processes. However, we observe 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. These results demonstrate that reconfiguring actions effectively change the performance of the application but not the application workload.

6.4.3.3 Performance evaluation of AMP

This section summarizes the results of the performance evaluation of the test cases for our considered benchmarks using the AMP policy. The performance evaluation focus on analyze the completion time, number of processes, and operational cost of each test case in Table 6.9.

Figure 6.22 summarizes the performance improvement of dynamic reconfiguration compared with static scheduling for our test cases and performance objectives of 25%, 30%, and 35% performance improvement with both efficiency and cost constraints. We set these performance objectives in order to prove the capabilities of the AMP policy to accurately adjust the performance of parallel applications to different requirements. For each test case, the AMP policy allocates the computing power necessary to satisfy the performance objective, while the number of processes and type of processors depend on the user-given performance constraint. The average relative error ranges in [-1.154%, 2.436%] with a relative standard deviation in [0.981%, 1.671%]. These results reflect the accuracy of AMP to allocate the number of FLOPs necessary to satisfy the performance objective for each test case.

It is important to note that the performance results are very accurate to satisfy the different performance requirements regardless of the execution pattern of by each benchmark. The AMP policy is able to adjust the performance of either regular (Jacobi and CG) or irregular (EpiGraph) benchmarks. Besides the computation and communication performance, the ability of the CPM to estimate the overheads of reconfiguring actions necessary to modify the application performance is very noticeable in the evaluation.

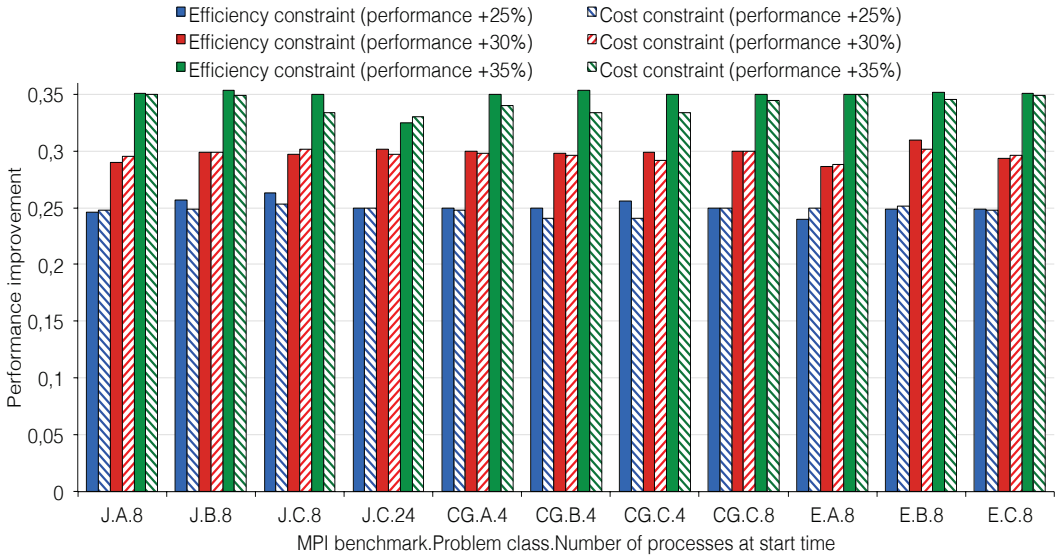


Figure 6.22: Performance evaluation of the benchmarks with the support of AMP for satisfying the performance objective.

Once we have concluded the capabilities of AMP to satisfy the performance objective, we analyze its capabilities to schedule the number of processes and type of processors necessary to satisfy the performance constraint. Figure 6.23 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 the type of processes allocated by the AMP policy to satisfy the cost constraint can be seen in Figure 6.24, 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.

We emphasize the ability of the AMP policy to take into account the wide variety of number and characteristics of the class nodes available in the cluster *ARCOS* when reconfiguring the application to the most appropriate processor configuration that satisfies the performance requirements. For instance in the J.C.24 test case, the AMP policy reconfigures the application from the initial one which uses 6 processes of each node class to a processor configuration that allocates 14 processes of class C1, 14 nodes of class C6, 6 nodes of class C7 and 12 nodes of class C8 under the efficiency constraint, and 34 processes of class C1, 6 nodes of class C6, 6 nodes of class C7 and 8 nodes of class C8 under the cost constraint.

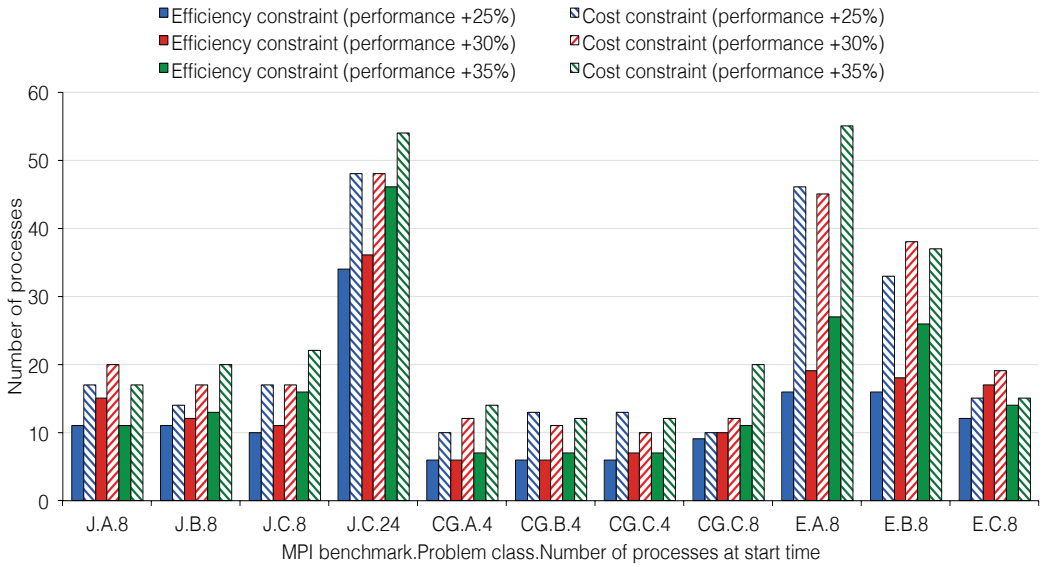


Figure 6.23: Number of processes scheduled by AMP for satisfying the performance objective for the benchmarks.

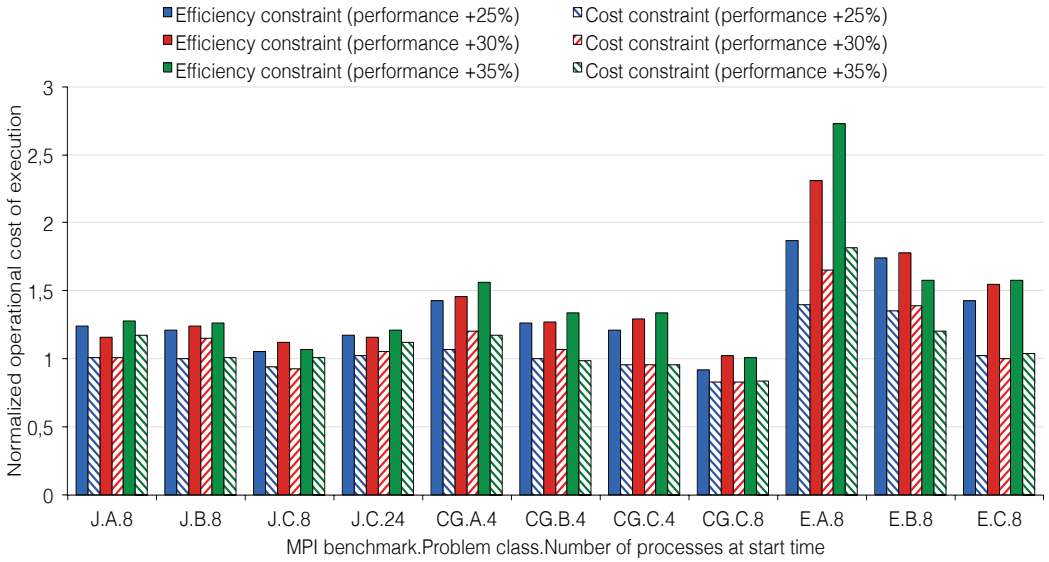


Figure 6.24: Cost analysis of the benchmarks with AMP support for satisfying the performance objective.

These results reflect that using FLEX-MPI to run MPI applications under the efficiency constraint permits to enhance the overall throughput of the system by allowing other applications to use available resources without affecting the performance of our application. On the other hand, MPI applications can benefit from the capabilities of the AMP policy to minimize the operational cost of the application with no performance penalty. This makes FLEX-MPI very interesting for scenarios in which we have a large number of available resources to run our application or we need to reduce the cost incurred by the application. Minimizing the operational cost has become a critical factor in recent years due to the large amount of energetic and economic cost associated to the execution of scientific applications in HPC systems. We emphasize that in some of the test cases (i.e. J.C.8, CG.C.4, and CG.C.8) the AMP policy satisfies the performance objective with an operational cost that is lower than the cost of running the application with static scheduling. It is important to note that, in the best cases, the performance of the benchmarks using static scheduling in 25% lower than the application running with the AMP policy.

We can conclude that the AMP policy offers several benefits for the execution of scientific parallel application that run on HPC systems. This policy allows to make an efficient usage of the resources both from the point of view of the overall throughput and the operational cost.

6.5 Overhead analysis

This section evaluates the overhead of the FLEX-MPI library when comparing to the execution of an MPI application running with the MPICH library, and its impact on the application performance. One of the goals of the library implementation is to minimize its overhead on the execution time. Though reconfiguring actions have a significant impact on the overhead, the FLEX-MPI library takes this into account when performing a process reconfiguration.

In order to evaluate the FLEX-MPI overhead we compare the execution times for Jacobi, initially running on 8 processes (each process pair is mapped to a node class C1, C7, C6, and C8), executed to compute matrix B (Table 6.2) for 2,500 iterations—J.B.8 test case—for the following cases:

1. The benchmark executes legacy MPI code (compiled with MPICH v3.0.4) with static process allocation.
2. The benchmark executes FLEX-MPI code with static scheduling.
3. The benchmark executes FLEX-MPI code with dynamic reconfiguration using the Adaptive malleability policy (AMP) under the efficiency constraint.
4. The benchmark executes FLEX-MPI code with dynamic reconfiguration using the AMP policy under the cost constraint.

The completion time for static scheduling is the sum of computation and communication times. Dynamic reconfiguration incurs overheads of process creation and

termination, in addition to load balance and data redistribution overheads associated with the reconfiguring actions. Figure 6.25 reflects the time allocated to the different phases—computation, communication, process reconfiguring, data operations, and other—for each of these four cases. The completion time for the application using static scheduling (8 processes) and legacy MPI code is 922.489 seconds and 1,512 cost units—considering the values of the cost model described in Table 6.4. The statically scheduled application using FLEX-MPI takes 923.453 seconds to complete and 1,535 cost units. These results show that the overhead when using the FLEX-MPI library with the adaptability functionalities turned off is negligible for the application performance.

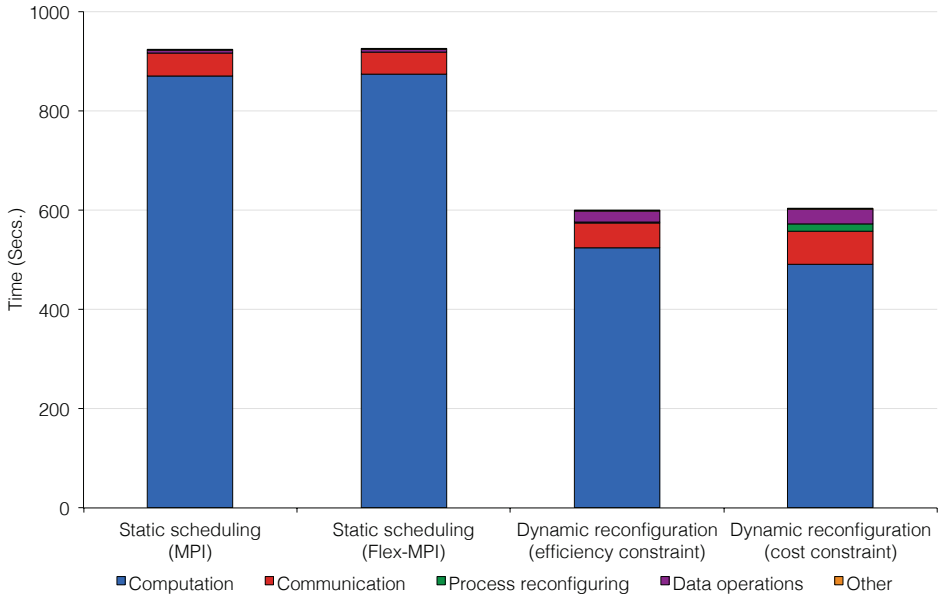


Figure 6.25: Performance overhead for legacy MPI with static scheduling and FLEX-MPI with static static scheduling and dynamic reconfiguration.

On the other hand, 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). Therefore, the goal of the AMP policy is to improve the performance of the application using dynamic reconfiguration such the application completes its execution in 600 seconds. Scenario 3 uses the efficiency constraint to minimize the number of processes allocated to satisfy the user-defined performance objective, while scenario 4 uses the cost constraint to minimize the operational cost of the execution to satisfy the performance objective.

FLEX-MPI effectively reconfigures the application in scenarios 3 and 4 to complete its execution respectively in 597.456 seconds and 601.387 seconds, with an operational cost of 1,928 cost units for scenario 3 and 1,543 cost units for scenario 4. The application requires 13 processes to satisfy the performance objective under the efficiency constraint. On the other hand, the application uses 20 processes to satisfy the objective under the cost constraint.

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. On the other hand, 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. However, this overhead is countered by the performance improvement exhibited by the application when using dynamic reconfiguration, and the application completes its execution in the user-defined time interval. These results reflect the trade off between performance improvement and the overhead of the FLEX-MPI library.

6.6 Summary

This chapter summarizes the results obtained in the execution of the experiments that we have conducted in order to validate the functionalities of FLEX-MPI and analyze the performance of parallel applications running on top of the runtime system. These experiments have been grouped into three categories: the first group of experiments analyzes the performance of the dynamic load balancing algorithm, the second one validates the computational prediction model, and the third group analyzes the performance of the malleability policies introduced by FLEX-MPI.

- Section 6.2 has demonstrated the significant improvement in terms of performance and scalability of FLEX-MPI applications using the dynamic load balancing algorithm, even for irregular applications that run on heterogeneous systems. Additionally, these results reflect the capabilities of the algorithm to adapt the application workload to the dynamic execution environment such as load balancing actions are carried out depending on the magnitude of the interferences of external applications that compete for the underlying resources.

- Section 6.3 has evaluated the capabilities of the computational prediction model integrated into FLEX-MPI to predict the performance of parallel applications, therefore allowing the reconfiguring policies to provision a number and type of processors that satisfies the performance criteria of each policy. This section shows a detailed analysis of the model validation considering each phase of the program execution: computation, communication and overhead of dynamic operations (creation and termination of processes and data redistribution).

- Section 6.4 has evaluated the performance of the malleability policies integrated into FLEX-MPI: SMP, HPMP and AMP policies. The SMP policy allows the application to change the number of processes to cope with the availability of

resources in the system. The HPMP policy enables the application to reconfigure the number of processes to the processor configuration that provides the maximum performance to the application. The AMP policy allows the user to define a performance goal and constraint, which can be either efficiency or cost, so that the application finalizes its execution in a user-defined time interval while maximizes the efficiency or minimizes the operational cost of the execution. These results have demonstrated that FLEX-MPI can meet the goals of the policy for several benchmarks that exhibit dynamic characteristics such as irregular computation and communication patterns and sparse data structures.

- Finally, Section 6.5 has analyzed the overhead of FLEX-MPI when comparing to the execution of legacy MPI applications. These results have demonstrated that the overhead of FLEX-MPI without dynamic reconfiguration is negligible when comparing to the execution of the native MPI code. On the other hand, the small overhead measured when executing FLEX-MPI with dynamic reconfiguration reflects the trade off between the performance gains and reconfiguring actions.

Chapter 7

Conclusions

This thesis proposes a set of optimization techniques for enhancing the performance of MPI applications. These techniques are embedded in FLEX-MPI, a runtime system that confers adaptability to MPI applications. FLEX-MPI includes both low-level and high-level functionalities to support dynamic load balancing and performance-aware dynamic reconfiguration in MPI programs. We have shown that MPI applications exhibit a significant improvement in performance and scalability, and make an efficient usage of the resources of the platform when using these adaptability techniques. FLEX-MPI introduces a dynamic execution model that allows MPI programs to execute efficiently in dynamic execution environments.

This chapter presents a detailed review of the main contributions of this work following the objectives of the thesis. This chapter also provides the list of journal and international conference publications related to this thesis, and concludes with the directions for future work arising from this work.

7.1 Main contributions

Following the results achieved in this work, we can conclude that the objectives of this thesis have been successfully accomplished. This section discusses the main contributions of this thesis, that can be classified into four main categories.

- **Dynamic load balancing algorithm**. Our novel dynamic load balancing algorithm for iterative MPI applications uses performance metrics collected at runtime to make decisions with respect to the workload distribution. The algorithm considers both homogeneous and heterogeneous platforms, dense and sparse data structures, regular and irregular parallel applications, and dedicated and non-dedicated systems. The algorithm optimizes the application to adapt to changes in the execution of the application at runtime (e.g. variable performance of the processing elements or variable computation pattern of the application). One of the main advantages of our approach is that the dynamic load balancing technique does not require prior knowledge of the underlying hardware.

The experimental results (Section 6.2) have demonstrated that our novel dynamic load balancing algorithm can significantly improve the performance and scalability of parallel applications running on homogeneous and heterogeneous systems, which can be either dedicated or non-dedicated.

- **Computational prediction model**. The novel computational prediction model uses performance metrics collected at runtime to predict the performance of parallel applications. Our model effectively computes the execution time of a parallel application in each phase: computation and computation, and additionally predicts the overheads of data redistribution and process reconfiguring actions. The network performance model uses the MPICH's cost functions that are based on the Hockney model to predict the performance of parallel communications in MPI applications.

  The experimental results (Section 6.4) demonstrate that the computational prediction model can estimate with high accuracy the performance of parallel applications for each of the considered execution phases under different execution scenarios.

- **Performance-aware dynamic reconfiguration policies**. This thesis introduces three novel performance-aware dynamic reconfiguring policies that allow to automatically change the number of processes of the application at runtime to satisfy performance criteria. This functionality is high-level because reconfiguring actions are performed according to the malleability policy without user intervention. FLEX-MPI features three high-level malleability policies: *Strict malleability policy* (SMP), *High performance malleability policy* (HPMP), and *Adaptive malleability policy* (AMP).

  The SMP policy allows a FLEX-MPI program to automatically change the number of processes at runtime depending on the availability of resources in the system. HPMP and AMP are performance-aware because, in addition to the availability of resources, they take into account the performance of the application to guide the process reconfiguring actions. HPMP automatically reconfigures the application to run on the processor configuration which provides the highest performance to the application. The AMP policy uses a user-given performance objective and constraints to guide reconfiguring actions such the application completes within a specified time interval.

  The experimental results (Section 6.3) have demonstrated that using these policies the parallel application achieves a significant improvement on its overall performance and throughput, and improves the efficiency of the resources of the system.

- **FLEX-MPI**. This thesis presents a novel system architecture, FLEX-MPI, that implements the adaptability techniques described in this work for dynamic load balancing and performance-aware dynamic reconfiguration. Furthermore, FLEX-MPI features a set of low-level components that offer basic mechanisms to collect runtime performance metrics of the MPI program, change the number of processes of the application, expand and shrink the MPI communicator,

schedule dynamic MPI processes, and move data between processes as a result of workload redistributions. The high-level components include the adaptability techniques and the computational prediction model.

FLEX-MPI communicates at runtime with the PMPI interface, the PAPI library, the resource management system and the MPI application, and is implemented as a library on top of MPICH implementation. This makes any MPI program compatible with FLEX-MPI.

- **EpiGraph**. EpiGraph is an epidemic simulation tool that has been designed and implemented as a scalable, fully distributed application based on MPI. EpiGraph was designed as a deterministic, scalable tool which simulates the propagation of influenza in scenarios that cover extended geographic areas. It consists of three main components: the social model, the epidemic model, and the transportation model.

  Though the context of EpiGraph is out of scope of this work, it definitely had a significant impact on FLEX-MPI motivation and approach. The irregular data structures, computation, and communication patterns of EpiGraph became challenging to the efficient execution of simulations on parallel architectures.

  The experimental results (Section 6.3) have demonstrated the capabilities of FLEX-MPI to efficiently execute irregular applications such as EpiGraph that run on dynamic execution environments.

7.2 Publications related to this thesis

The main contributions of this thesis have been published in several papers in international conferences and journals.

- Journals
  - Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu and Jesús Carretero. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, 2015. Impact Factor: 1.890. [MSMC15a]
  - Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu and Jesús Carretero. Towards efficient large scale epidemiological simulations in EpiGraph. *Parallel Computing*, 2015. Impact Factor: 1.890. [MSMC15b]
  - Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh and Jesús Carretero. Leveraging social networks for understanding the evolution of epidemics. *BMC Systems Biology*, 2011. Impact Factor: 3.148. [MMSC11b]

- International conferences
  - Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh and Jesús Carretero. FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems. *Euro-Par*, 2013. Core A. [MMSC13]
  - Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh and Jesús Carretero. Parallel algorithm for simulating the spatial transmission of influenza in EpiGraph. *EuroMPI*, 2013. Core C. [MSMC13]
  - Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh and Jesús Carretero. EpiGraph: A scalable simulation tool for epidemiological studies. *BIOCOMP*, 2011. Core C. [MMSC11a]
- National conferences
  - Manuel Rodríguez-Gonzalo, Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, Javier García Blas and Jesús Carretero. Flex-MPI: una biblioteca para proporcionar maleabilidad dinámica en aplicaciones MPI. *XXVI Jornadas de Paralelismo*, 2015. [RGMS+15]
- Posters
  - Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu and Jesús Carretero. Runtime Support for Adaptive Resource Provisioning in MPI Applications. *EuroMPI*, 2012. [MSMC12]

7.3 Future work

This thesis has opened several interesting research directions for future work:

- Extending the execution model of FLEX-MPI to consider parallel applications developed following the principles of the task-based parallelism. We aim to design a new malleability policy that can take advantage of this parallelization approach to guide reconfiguring actions based on the number of tasks and the availability of resources, then adjusting the granularity of the application.
- Extending the AMP policy to model the power consumption of the computer as a new performance metric and add it as a performance objective to FLEX-MPI. In recent years, power consumption has become a topic which attracts much interest due to its importance on the economic costs associated to HPC. We aim to develop new techniques that can reduce the energy footprint of large-scale MPI applications.
- Improving the functionalities of the network performance component of the CPM to predict the performance of different categories of network topologies. Additionally, extending the model to support applications with asynchronous communications, which may overlap communication and computation.

- Extending the optimization techniques for adaptability of FLEX-MPI applications to Grid and 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, consider the topology and physical location to make decisions about task scheduling, and evaluate their impact on the performance of HPC applications.
- Making dynamic the number of iterations of the sampling interval instead of being a user-defined parameter.
- Extending the integration of FLEX-MPI with the resource management system, including a execution priority system for MPI applications and a global batch scheduler for FLEX-MPI jobs, so the RMS can consider multiple malleable applications together to optimize overall system performance.

Bibliography

[A+00] InfiniBand Trade Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the Clouds: A Berkeley View of Cloud Computing. *University of California at Berkeley*, 2009.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AG89] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[AISS95] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model-one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[B+99] Rajkumar Buyya et al. High Performance Cluster Computing: Architectures and Systems (Volume 1). *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[BDGR95] Shirley Browne, Jack Dongarra, Eric Grosse, and Tom Rowan. The Netlib Mathematical Software Repository. *D-Lib Magazine*, 1995.

[BDMT99] Eric Blayo, Laurent Debreu, Gregory Mounie, and Denis Trystram. Dynamic load balancing for ocean circulation model with adaptive meshing. In *EuroPar'99 Parallel Processing*, pages 303–312. Springer, 1999.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Bel89] C Gordon Bell. The future of high performance computers in science and engineering. *Communications of the ACM*, 32(9):1091–1101, 1989.

[BGA14] Sumit Bagga, Deepak Garg, and Abhishek Arora. Moldable load scheduling using demand adjustable policies. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 143–150. IEEE, 2014.

[BGB06] M. Beltran, A. Guzman, and J.L. Bosque. Dealing with heterogeneity in load balancing algorithms. In *ISPDC'06*, pages 123–132. IEEE, 2006.

[Bha97] P. Bhargava. MPI-LITE user manual. Technical report, Parallel Computing Lab, University of California, 1997.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216. ACM, 1995.

[BKS+95] Nanette J Boden, Alan E Kulawik, Charles L Seitz, Danny Cohen, Robert E Felderman, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995.

[BMG06a] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10–pp. IEEE, 2006.

[BMG06b] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 86–95. Springer, 2006.

[BMS94] RAA Bruce, JG Mills, and AG Smith. CHIMP/MPI user guide. Technical report, Edinburgh Parallel Computing Centre, 1994.

[CCN97] Michele Cermele, Michele Colajanni, and G Necci. Dynamic load balancing of distributed SPMD computations with explicit message-passing. In *Heterogeneous Computing Workshop, 1997.(HCW'97) Proceedings., Sixth*, pages 2–16. IEEE, 1997.

[CDCG+05] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 776–783. IEEE, 2005.

[Cer12] Márcia Cristina Cera. *Providing adaptability to MPI applications on current parallel architectures*. PhD thesis, Universidade Federal do Rio Grande do Sul, August 2012.

[CGR+09] Márcia Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, Philippe Navaux, et al. Supporting MPI malleable applications upon the OAR resource manager. In *Colibri: Colloque d'Informatique: Brésil/INRIA, Coopérations, Avancées et Défis*, 2009.

[CGR+10] Márcia Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe Navaux. Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI. *Distributed Computing and Networking*, pages 242–257, 2010.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[Com07] Hewlett-Packard Development Company. HP-MPI User's Guide, 2007.

[Cor11] Intel Corporation. Intel MPI Library - Reference Manual, 2011.

[Cor15] NVIDIA Corporation. Nvidia cuda programming guide, 2015.

[CPM+06] Márcia Cera, Guilherme Pezzi, Elton Mathias, Nicolas Maillard, and Philippe Navaux. Improving the dynamic creation of processes in MPI-2. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 247–255. Springer, 2006.

[CPP+07] Márcia Cera, Guilherme Pezzi, Maurício Pilla, Nicolas Maillard, and Philippe Navaux. Scheduling dynamically spawned processes in MPI-2. In *Job Scheduling Strategies for Parallel Processing*, pages 33–46. Springer, 2007.

[CSG99] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999.

[Dan98] George B Dantzig. *Linear programming and extensions*. Princeton University Press, 1998.

[DBSDBM13] Marc Duranton, David Black-Schaffer, Koen De Bosschere, and Jonas Maebe. The HIPEAC vision for advanced computing in horizon 2020. 2013.

[DGNP88] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[DH11] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, December 2011.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[Don98] Jack Dongarra. Overview of PVM and MPI. `http://www.netlib.org/utk/people/JackDongarra/pdf/pvm-mpi.pdf`, 1998. Accessed: 2014-02-01.

[DS98] Kevin Dowd and Charles Severance. *High Performance Computing*. O'Reilly, Second edition, 1998.

[DYDS+10] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, et al. The HiPEAC vision. *Report, European Network of Excellence on High Performance and Embedded Architecture and Compilation*, 12, 2010.

[EH08] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA*, 2008.

[EMDSV06] Kaoutar El Maghraoui, Travis J Desell, Boleslaw K Szymanski, and Carlos A Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications*, 20(4):467–480, 2006.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[Fos02] Ian Foster. What is the Grid? A three point checklist. *GRIDtoday*, 6, July 2002.

[FR96] Dror Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 1996.

[FTM+14] John T Feo, Antonino Tumeo, Timothy G Mattson, Oreste Villa, and Simone Secchi. Special Issue of Journal of Parallel and Distributed Computing: Architectures and Algorithms for Irregular Applications. *Journal of Parallel and Distributed Computing*, 1(74):2027–2028, 2014.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[GABC08] Ismael Galindo, Francisco Almeida, and José Manuel Badía-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 64–74. Springer, 2008.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, 1994.

[GC01] Duncan Grove and Paul Coddington. Precise MPI performance measurement using MPIBench. In *Proceedings of HPC Asia*, pages 24–28. Citeseer, 2001.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GK09] Richard L Graham and Rainer Keller. Dynamic communicators in MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 116–123. Springer, 2009.

[GKM14] Elisabeth Günther, Felix G König, and Nicole Megow. Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width. *Journal of Combinatorial Optimization*, 27(1):164–181, 2014.

[GKP96] GA Geist, J.A. Kohl, and P.M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[GL98] William Gropp and Ewing Lusk. PVM and MPI are completely different. In *Fourth European PVM-MPI Users Group Meeting*. Citeseer, 1998.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[GLP03] GNU GLPK. Linear Programming Kit. `http://www.gnu.org/software/glpk/glpk.html`, 2003. Accessed: 2013-08-01.

[GM12] Kate Gregory and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. O'Reilly Media, Inc., 2012.

[Gro02] William Gropp. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–7. Springer, 2002.

[Gui15] MATLAB User's Guide. The Mathworks. *Inc., Natick, MA*, 8.6, 2015.

[Hal09] Bruce Hallberg. *Networking, A Beginner's Guide*. McGraw-Hill, Inc., 2009.

[Har94] J.C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In *Scalable Parallel Libraries Conference, 1994., Proceedings of the 1994*, pages 68–77. IEEE, 1994.

[Hem96] R. Hempel. The status of the MPI message-passing standard and its relation to PVM. *Parallel Virtual Machine–EuroPVM'96*, pages 14–21, 1996.

[HLK04] C. Huang, O. Lawlor, and L. Kale. Adaptive MPI. *Languages and Compilers for Parallel Computing*, pages 306–322, 2004.

[Hoc94] Roger W Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[Hun04] Jan Hungershofer. On the combined scheduling of malleable and rigid jobs. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 206–213. IEEE, 2004.

[HZK+10] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. Case study for running HPC applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 395–401. ACM, 2010.

[IOY+11] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick HJ Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[JRM+10] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[KBM02] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[KBV00] Thilo Kielmann, Henri E Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *Parallel and Distributed Processing*, pages 1176–1183. Springer, 2000.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, 1994.

[KJP08] Matthew J Koop, Terry Jones, and Dhabaleswar K Panda. MVAPICH-aptus: Scalable high-performance multi-transport MPI over infiniband. In *Parallel and Distributed Processing (IPDPS), 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[KK93] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[KKD02] Laxmikant V Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 230–230. IEEE, 2002.

[KM27] W.O. Kermack and A.G. McKendrick. Contributions to the mathematical theory of epidemics. In *Proceedings of the Royal society of London. Series A*, volume 115, pages 700–721, 1927.

[KP11] Cristian Klein and Christian Pérez. An RMS for non-predictably evolving applications. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 326–334. IEEE, 2011.

[Kwi06] Jan Kwiatkowski. Evaluation of parallel programs by measurement of its granularity. In *Parallel Processing and Applied Mathematics*, pages 145–153. Springer, 2006.

[Lov93] D.B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[LR06] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

[LS06] Claudia Leopold and Michael 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*, pages 285–292. Springer, 2006.

[LSB06] Claudia Leopold, Michael Süß, and Jens Breitbart. Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. In *Proceedings of the European Conference on Modelling and Simulation*, pages 665–670, 2006.

[LTW02] Renaud Lepère, Denis Trystram, and Gerhard J Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science*, 13(04):613–627, 2002.

[MAG+11] JA Martínez, Francisco Almeida, Ester M Garzón, Alejandro Acosta, and V Blanco. Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. *The Journal of Supercomputing*, 58(3):385–393, 2011.

[MB76] Robert M Metcalfe and David R Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[MBDH99] P.J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[MCS+13] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, et al. AutoTune: a plugin-driven approach to the automatic tuning of parallel applications. In *Applied Parallel and Scientific Computing*, pages 328–342. Springer, 2013.

[MDSV08] KE Maghraoui, T.J. Desell, B.K. Szymanski, and C.A. Varela. Dynamic malleability in iterative MPI applications. In *7th Int. Symposium on Cluster Computing and the Grid*, pages 591–598, 2008.

[Mes94] Message Passing Interface Forum. Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/`, 1994. Accessed: 2014-02-01.

[MGPG11] José A Martínez, Ester M Garzón, Antonio Plaza, and Inmaculada García. Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. *The Journal of Supercomputing*, 58(2):151–159, 2011.

[MMSC11a] G. Martín, M.C. Marinescu, D.E. Singh, and J. Carretero. EpiGraph: A scalable simulation tool for epidemiological studies. *The 2011 International Conference on Bioinformatics and Computational Biology*, pages 529–536, 2011.

[MMSC11b] G. Martín, M.C. Marinescu, D.E. Singh, and J. Carretero. Leveraging social networks for understanding the evolution of epidemics. *BMC Syst Biol*, 5(S3), 2011.

[MMSC13] G. Martín, M.C. Marinescu, D.E. Singh, and J. Carretero. FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems. In *Euro-Par 2013 Parallel Processing*, pages 138–149. Springer Berlin Heidelberg, 2013.

[MPI] MPI Forum. Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/`. Accessed: 2014-02-01.

[MSDS12] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. TOP500 Supercomputer sites. `http://www.top500.org`, 2012. Accessed: 2015-01-31.

[MSMC12] G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. Runtime Support for Adaptive Resource Provisioning in MPI Applications. *Recent Advances in the Message Passing Interface*, pages 294–295, 2012.

[MSMC13] G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. Parallel algorithm for simulating the spatial transmission of influenza in EpiGraph. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 205–210. ACM, 2013.

[MSMC15a] G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, 46(C):60 – 77, 2015.

[MSMC15b] G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. Towards efficient large scale epidemiological simulations in EpiGraph. *Parallel Computing*, 42(0):88 – 102, 2015. Parallelism in Bioinformatics.

[MSV06] K.E. Maghraoui, B. Szymanski, and C. Varela. An architecture for reconfigurable iterative mpi applications in dynamic environments. *Parallel Processing and Applied Mathematics*, pages 258–271, 2006.

[MZ94] Cathy McCann and John Zahorjan. *Processor allocation policies for message-passing parallel computers*, volume 22. ACM, 1994.

[NBF96] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads programming.* O'Reilly Media, Incorporated, 1996.

[Net09] Network-Based Computing Laboratory (NBCL) - Ohio State University. OSU Micro Benchmarks. `http://nowlab.cse.ohio-state.edu/`, 2009. Accessed: 2014-11-31.

[Pan] DK Panda. MVAPICH 1.0 User and Tuning Guide. *Network-Based Computing Laboratory, Ohio State University.*

[Phe08] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[PIR+14] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix Wolf. A Batch System with Fair Scheduling for Evolving Applications. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 351–360. IEEE, 2014.

[PNR+15] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V Kale. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 429–438. IEEE, 2015.

[PWDC05] A Petitet, RC Whaley, J Dongarra, and A Cleary. HPL–a portable implementation of the high–performance Linpack benchmark for distributed–memory computers. `http://www.netlib.org/benchmark/hpl`, 2005. Accessed: 2014-11-31.

[RBA11a] A. Raveendran, T. Bicer, and G. Agrawal. An autonomic framework for time and cost driven execution of mpi programs on cloud environments. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 218–219. IEEE Computer Society, 2011.

[RBA11b] A. Raveendran, T. Bicer, and G. Agrawal. A framework for elastic execution of existing mpi programs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 940–947. IEEE, 2011.

[RBL+09] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I.M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.

[RGMS+15] M. Rodríguez-Gonzalo, G. Martín, D.E. Singh, M.C. Marinescu, J. García Blas, and J. Carretero. Flex-MPI: una biblioteca para proporcionar maleabilidad dinámica en aplicaciones MPI. *XXVI Jornadas de Paralelismo*, 2015.

[RHJ09] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[RSPM98] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent advances in parallel virtual machine and message passing interface*, pages 52–59. Springer, 1998.

[RWS11] Nicholas Radcliffe, Layne Watson, and Masha Sosonkina. A comparison of alternatives for communicating with spawned processes. In *Proceedings of the 49th Annual Southeast Regional Conference*, pages 132–137. ACM, 2011.

[Sap94] W. Saphir. Devil's advocate: Reasons not to use PVM. In *PVM User Group Meeting*, volume 20, 1994.

[SD03] S.V. Sathish and J.J. Dongarra. SRS – a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.

[SD05] S.V. Sathish and J.J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[SGS10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[SNdS+07] A. Sena, A. Nascimento, J.A. da Silva, D.Q.C. Vianna, C. Boeres, and V. Rebello. On the Advantages of an Alternative MPI Execution Model for Grids. In *Cluster Computing and the Grid (CCGRID), 2007. Seventh IEEE International Symposium on*, pages 575–582, May 2007.

[SR07] Rajesh Sudarsan and Calvin J Ribbens. ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Parallel Processing (ICPP), 2007. International Conference on*, pages 44–44. IEEE, 2007.

[SR09] Rajesh Sudarsan and Calvin J Ribbens. Scheduling resizable parallel applications. In *Parallel & Distributed Processing (IPDPS), 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[SR10] Rajesh Sudarsan and Calvin J Ribbens. Design and performance of a scheduling framework for resizable parallel applications. *Parallel Computing*, 36(1):48–64, 2010.

[ST96] Sartaj Sahni and Venkat Thanvantri. Performance metrics: Keeping the focus on runtime. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(1):43–56, 1996.

[Sta06] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[Ste04] Steven Kehlet. tcpping network performance tool. `http://www.kehlet.cx/download/tcpping-0.3.tar.gz`, 2004. Accessed: 2014-11-31.

[TF15] Antonio Tumeo and John Feo. Intrusion detection: A brief history and overview (supplement to computer magazine). *Computer*, 48(8):14–16, 2015.

[TG03] Rajeev Thakur and William D Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 257–267. Springer, 2003.

[TQD+05] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast.nlanr.net/Projects*, 2005.

[TRG05] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[TVR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, December 1985.

[UCL04] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on MPI jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 215–224. IEEE Computer Society, 2004.

[Utr10] Gladys Utrera. *Virtual Malleability Applied to MPI Jobs to Improve Their Execution in a Multiprogrammed Environment.* PhD thesis, Universitat Politècnica de Catalunya, 2010.

[VBS+06] Cécile Viboud, Ottar N Bjørnstad, David L Smith, Lone Simonsen, Mark A Miller, and Bryan T Grenfell. Synchrony, waves, and spatial hierarchies in the spread of influenza. *Science*, 312(5772):447–451, 2006.

[VD03] Sathish S Vadhiyar and Jack J Dongarra. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.

[Vog08] WA Vogels. Head in the Clouds–The Power of Infrastructure as a Service. In *First workshop on Cloud Computing and in Applications*, October 2008.

[VRMCL08] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[WLNL03] D.B. Weatherly, D.K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 5–5. IEEE, 2003.

[WLNL06] D.B. Weatherly, D.K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on medium-scale, non-dedicated clusters. *Journal of Parallel and Distributed Computing*, 66(6):822–838, 2006.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.