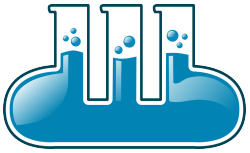
Dynamic Load Balancing for Hybrid Applications

Marta Garcia Gasulla



Advisors:
Julita Corbalan and Jesus Labarta

A thesis submitted for the degree of
Doctor of Philosophy in Computer Architecture
at the

With the support of the

Barcelona, February 2017

Abstract

It is well known that load imbalance is a major source of efficiency loss in HPC (High Performance Computing) environments. The load imbalance problem has very different sources, from static ones related to the data distribution to very dynamic ones, for example, the noise of the system.

In this thesis, we present DLB: Dynamic Load Balancing library. DLB is a framework to improve the efficient use of the computational resources of a computational node. With DLB we offer a dynamic solution to load imbalance problems. DLB is applied at runtime and does not need previous information to solve load imbalance problems, for this reason, it can deal with load imbalances coming from any source.

The DLB framework includes a novel load balancing algorithm: LeWI (Lend When Idle). The main idea of LeWI is to use the computational resources assigned to a process or thread when it is idle, to speed up another process running on the same node that it is still doing computation. We will see how this idea although being quite simple it is powerful and flexible to obtain an efficient use of resources close to the ideal one.

Contents

Chapter 1

Introduction

High performance computing (HPC) environments are always evolving. Nowadays the hardware is moving towards big clusters with multicores and manycore nodes. To achieve an efficient use of these massive new architectures, we can find multiple software layers that go from the hardware level up to the application level.

In parallel computing, the loss of efficiency is an issue that concerns both system administrators and parallel programmers. For system administrators efficiency is to obtain a high throughput[1] from the computational resources. For parallel programmers, efficiency is to obtain the maximum performance from their application.

It is well known that one of the primary sources of efficiency loss in parallel systems is load imbalance. Load imbalance implies that while the most loaded processes are computing, the less loaded ones are waiting and wasting computational resources, therefore, losing efficiency.

The growth in the number of computing units that clusters experienced in the last years has helped improving applications performance but has raised new problems or worsened old ones. One of the old problems that has deteriorated with the increase of computational resources is load imbalance.

[1] In this context, we can define throughput as the number of jobs finished per unit of time. $Throughput = \frac{FinishedJobs}{ElapsedTime}$

On the one hand it is hard to distribute the work evenly when partitioning across more resources. On the other hand, the efficiency loss related to load imbalance increases dramatically with the number of computational resources used.

If we are running with 4 processes and one of them is one second slower than the others, we will have 3 CPUs idle for 1 second, accounting for 3 CPU time seconds lost. But in the current HPC environments it is common to run into thousands and tens of thousands of CPUs. If we have the same problem when running with 1000 processes, we will be wasting 1000 seconds of computational power. The same problem when running in 10.000 CPUs would be wasting 2,8 hours of computational power.

We can define Efficiency as the percentage of time that the computational resources are doing useful work from the total CPU time consumed by the application. In this thesis we will use the following formula to compute the efficiency:

$$Efficiency = \frac{useful\_cpu\_time}{elapsed\_time * num\_CPUS} * 100$$

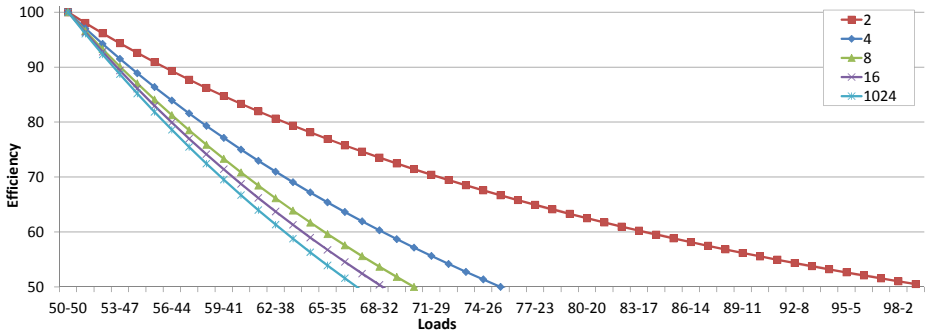


Figure 1.1: Efficiency for different loads and number of processes

In Figure 1.1 we can see how the efficiency is affected when running with different numbers of processes and different loads[2]. This makes load imbalance a crucial issue to obtain an efficient use of the HPC environments.

[2]This is a synthetic experiment assuming ideal conditions. I.e., the only impact on the efficiency is the load imbalance. In the X axis is represented different distributions of loads among processes. To simplify the first number is the load of the most loaded process and the second number is the load of the remaining processes. E.g., for 4 processes and a load

Load imbalance is a problem as old as parallel programming, and has been targeted from very different approaches but has never disappeared. In some cases, no solution has been found yet, and in others, the evolution of the system introduced new issues, for example, load imbalance caused by heterogeneity of resources.

Load imbalance appears when one or more processes of an application finish their computation faster than the others and a synchronization point is reached. In this situation, the faster processes will wait for the others. While the processes are waiting their computational resources can be idle. We will use the following formula to measure the load balance:

$$LoadBalance = \frac{\sum_{n=1}^{NumProcs} t_n}{\max_{n=1}^{NumProcs}(t_n) * NumProcs} = \frac{Average_{n=1}^{NumProcs}(t_n)}{\max_{n=1}^{NumProcs}(t_n)}$$

Where t_n is the computation time of process n. This will give a load balance value between 0 and 1; a value of 1 means a perfect load balance of the computation ($Average(t_n) == Max(t_n)$).

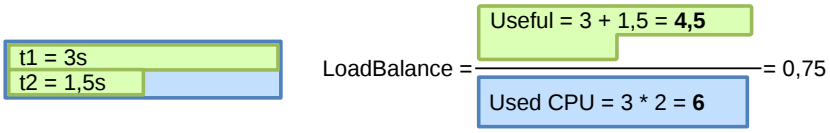


Figure 1.2: Example of load balance measured

Load imbalance can appear for very different reasons. We can divide the sources of load imbalance in three main categories: inherent to the algorithm, system functionality or system variability.

Algorithm Load imbalance can be originated in the algorithm, in some cases because of data partition, in others because of the computational load. A **computational imbalance** coming from the algorithm can appear when using sparse matrices, for example, because the number of useful values and zeros may be distributed among processes unevenly.

Mesh partitioning software (i.e. Metis [1]) is used to try to solve load imbalance coming from **data partitioning**. In most cases, this software needs

distribution of "56-44", the first process will have a load of 54, the second 44, the third 44 and the fourth 44.

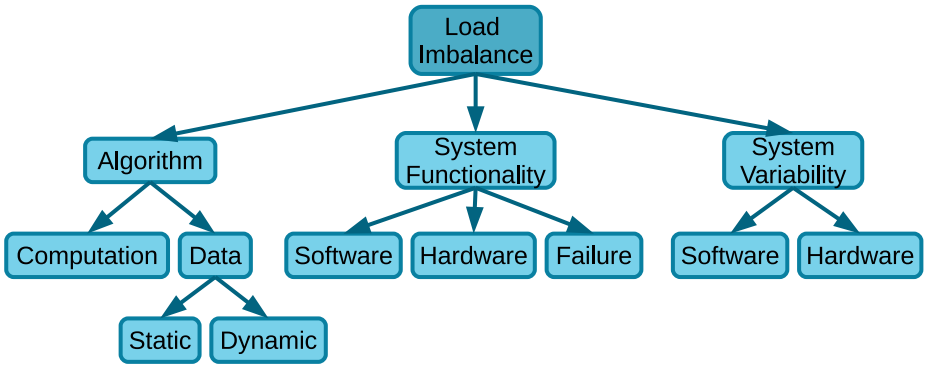


Figure 1.3: Taxonomy of sources of load imbalance

a heuristic to load balance the partition. This heuristic must be provided by the application. But, obtaining this heuristic is not always trivial (as we will see in chapter 9).

Furthermore, sometimes a data partition can be well load balanced for some parts of the application, but imbalanced for other ones. For example, when working with graphs, if the heuristic used to compute the partitions is based on the computational load of edges, this partition may not be well balanced when doing computation over the nodes.

Even if the computational work was divided evenly at the start point, load balance could change during the execution. A typical example would be the particles moving through a system and across partitions; in this case, the computational load is associated with the particles and these are moving from one partition to another. Another typical situation where we can find a **dynamic data load imbalance** is when using adaptive mesh refinement (AMR) [2]. With these techniques, the accuracy of some regions is increased by increasing the number of grid points and consequently the amount of data to compute for those regions.

System Functionality Differences in functionality can also introduce load imbalance. For example, a different IPC (Instructions per Cycle) or number of misses in any of the cache levels can end up producing a load imbalance between different processes. These kind of situations are difficult to detect, and only after a deep analysis of performance we can try to solve the imbal-

ance by changing the code to improve the data access, locality, serialization over resources or similar problems. Furthermore, we can address the problem for a particular architecture and input of the application, but the issue can reappear anytime that we change the environment or the input set.

Another kind of functionalities that might worsen the load balancing can be related to resiliency, coming from differences in the checkpointing, **failures** or restarting. Even technology designed to speed up applications, like for example, dynamic overclocking (E.g, Intel turbo boost) can introduce differences in the execution times of the processes and produce a load imbalance.

System Variability Finally, variability is a major source of load imbalance [3]. Causes for variability at the **software** level can go from the noise introduced by the operating system to thread migration among cores. But also **hardware** can present variability that affects load imbalance like contention in the network or differences in the manufacturing of the hardware. These load imbalances can not be predicted nor solved by a static approach. Only a dynamic mechanism can try to solve or minimize the impact of variability in the performance.

HPC application developers have been aware of this problem and have been trying to solve it for a long time. Their straightforward solution is to tune parallel codes by hand. In some cases, programmers include load-balancing code in their applications. In others, they need to deduce heuristics that can predict the load of the input to obtain a good data distribution. Or even redistribute the data during the execution (at the high cost of moving data around).

This is always an iterative process as depicted in Figure 1.4 of analyzing the performance of the application, modifying it to improve its performance and run again until an acceptable performance is obtained.

These practices usually result in codes optimized for specific architectures, execution conditions or inputs. They require devoting many economic and human resources to tune specific parallel codes. Furthermore, usually, these solutions do not consider or are not able to solve other factors that can introduce load imbalance such as OS noise, resource sharing, or dynamic overclocking. And even worst, the rapidly changing systems and architectures can easily deprecate these optimizations.

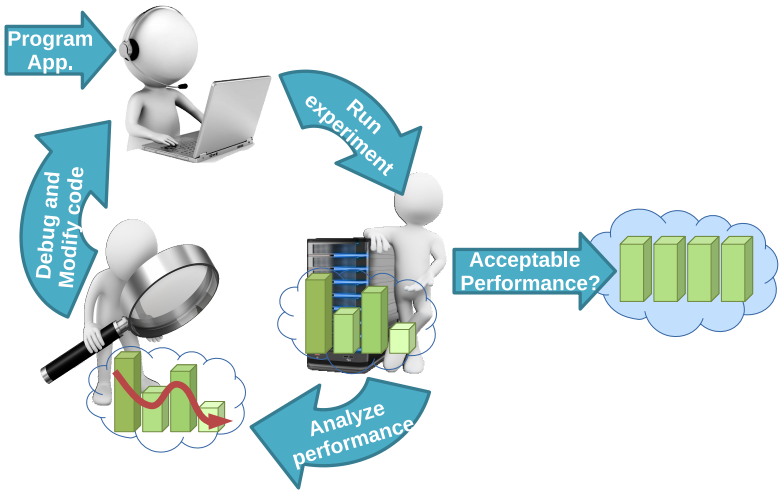


Figure 1.4: Typical HPC application life cycle

We have seen that traditionally the approach to solve the load imbalance problem has been to anticipate it. Either by estimating the load with heuristics or measuring it previously to the execution. But as we have seen the sources of load imbalance can be very dynamic and trying to fight it is a lost battle.

If we cannot anticipate it, nor fight it, then the only solution we have is to adapt to it. Our proposal is to adjust the execution dynamically to obtain a better use of the resources despite the load imbalance.

In this thesis, we will try to find a solution to the load imbalance problem that is dynamic enough to solve all the types of imbalance and granularities. To achieve this instead of relying on data load balancing, we will use the approach of computational load balancing.

Data-load balancing consists in redistributing the data to achieve a good load balance. It is the most used approach in current solutions (repartitioning the mesh, e.g. PAMPA [4] or redistributing the data, e.g. Adaptive MPI [5]). Data movements and mesh partitioning is expensive. Therefore, the grain of the load imbalance that can be solved with this approaches is coarse.

Computational-load balancing, on the other hand, consists in giving more resources to the processes that present a higher load to balance the overall execution. This approach allows a more fine-grained load balancing because the "movement" of computational resources is faster than data migration.

The load balancing based in redistributing the computational power relies on the malleability[3]. We will see that malleability is crucial at all the levels, the application, the programming model, and the system.

The solution we propose is a library that will adapt the application at runtime; we call it *Dynamic Load Balancing* (DLB) library. DLB will manage the use of resources inside a computational node to solve the load imbalance of an application. It will do this by changing the number of threads of the different processes transparently to the developer and the application. As the solution is applied at runtime, DLB can solve load imbalance coming from any source of the previously listed.

In this thesis, we will present a novel load balancing algorithm implemented in DLB called: *LeWI* (Lend When Idle). LeWI is based on the idea of lending the computational resources to another process on the same node when not using them.

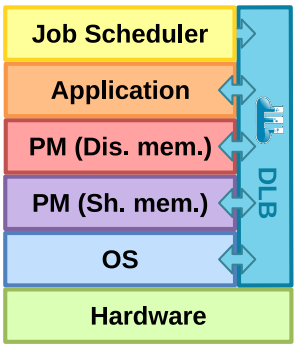


Figure 1.5: HPC software stack and DLB

As we said, the software stack in HPC environments is formed by several software layers. In Figure 1.5 we can see a typical software stack in HPC systems. All these layers share an aim: to obtain an efficient use of the computational resources, but, in general, they do not work together or coordinate.

DLB will work transversal to the other software layers and collaborate or coordinate with them to improve the efficient use of resources. As we will see, the level of integration with each layer is different. Some of them are completely transparent, and almost no integration was necessary, in other ones we have done a deep integration, and in other cases it is left as future work.

We will try to be as transparent as possible to the application, but we will also evaluate the trade-off between being completely transparent and the

[3]**Malleability**: the state of being malleable, or capable of being shaped.

performance. To this end, DLB offers an API that can be used to give hints to the load balancing mechanism to improve the performance.

DLB will need two levels of parallelism, the inner level of parallelism will be used to improve the resource utilization of the outer level of parallelism. A typical HPC application will have parallelism at the distributed memory level (across nodes) and the shared memory level (inside a node). DLB also supports a single level of parallelism and multiple applications (the application would be the second level of parallelism necessary).

At the distributed memory level we will focus on the MPI programming model and improve its load balance. Load imbalance is especially dramatic in MPI applications because data is not easily redistributed or moved between processes. Furthermore, MPI is the most used programming model in HPC applications. Nevertheless, DLB has been designed and is prepared to be easily extended to support other programming models. The interaction between DLB and MPI will be completely transparent to the application, the user, and the programming model, thanks to the PMPI interception mechanism provided by MPI.

The current version of DLB includes support for OpenMP and OmpSs as the second level of parallelism. DLB coordination with OpenMP is based on their standard public API. An integration between DLB and the OmpSs runtime has been implemented. This integration has shown the potential of a collaboration between the different layers. With the current structure of DLB and the experience of the OmpSs integration adding support for other programming models at various levels of integration should be trivial.

As a future work we plan to offer an API for the job scheduler that gives information about the performance of the application during the execution and allows to change the assigned resources to each process based on this information. This functionality will be offered by two new modules: *Stats* and *DROM*.

With the solution we propose we want to avoid the iterative process of analyzing and modifying the application to obtain a good performance, as shown in Figure 1.6. To use DLB, we do not need to change the application nor analyze its performance previously. The DLB library can be loaded dynamically, can be used linked with the application or through the integration with OmpSs.

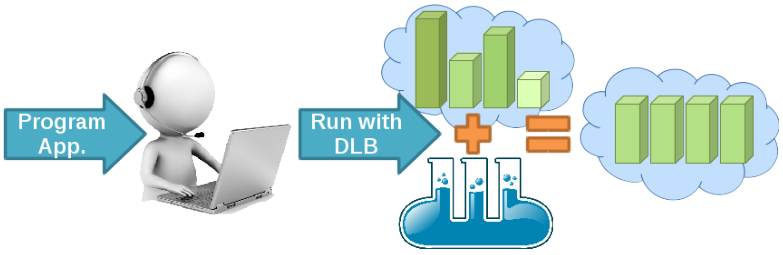


Figure 1.6: HPC application life cycle with DLB

1.1 Contributions and Publications

The main contributions of this thesis and the publications that resulted from them are the following:

- The DLB library as a framework to manage the computational resources within a node. It is ready to be extended with other load balancing algorithms and support for other programming models.
- A novel load balancing algorithm *LeWI* [6] [7].
- The integration of DLB with a parallel runtime (OmpSs). We have shown the potential of this kind of collaborations between runtimes. We have also identified the key points of coordination and defined an API for this integration.
- Study of factors that affect the load balancing mechanism based on DLB and LeWI. We have obtained conclusions that can be used by application developers or parallel runtime designers to improve the efficiency of applications when using DLB and LeWI. These factors include the degree of malleability of the programming model and the application, the distribution of MPI processes among nodes and the binding of threads to cores [8] [9].
- Extensive evaluation of DLB and LeWI in a CFD (Computational Fluid Dynamics) code: Alya. We have demonstrated the potential of DLB and LeWI at solving different kind of load balancing issues common in this type of applications. We confirm that DLB and LeWI are ready

to be used in production runs of real applications and scale up to thousands of cores [10] [11].

1.2 Organization of the document

The remaining of this document is organized as follows: In Chapter 2, we will comment the technical background in which this thesis has been developed, this includes explaining the programming models used and their key features and describing the software and the hardware used to run the experiments.

In Chapter 3, we will explain the main characteristics of the different applications and benchmarks that have been utilized for the performance evaluation. The following chapter contains a summary of the most relevant related work to our research topic.

In chapter 5 we will explain the DLB library in detail, including all the technical information, design decisions, and its main features. The two algorithms implemented within DLB will be explained and compared in Chapter 6. DWB (Dynamic Weight Balancing) [12] a load balancing algorithm from a previous work and LeWI the novel algorithm presented in this thesis.

Chapter 7 contains an extensive study of different factors that can affect the load balancing mechanism. The malleability of the programming model and the application, the distribution of MPI processes and the binding of threads to cores.

The details of the integration of DLB with a parallel runtime (OmpSs) and its performance evaluation will be explained in Chapter 8.

In Chapter 9, we will see the results of applying DLB and LeWI to a production CFD code: Alya. We will analyze in detail the performance related to load imbalance in different parts of the code and how DLB can improve it. And we will see how DLB can be used to solve different issues related to resource usage in this kind of applications.

Finally, in the last chapter, we will summarize the key findings achieved with this thesis and research lines that has been opened for future work. At the end, we can find two appendixes containing a thorough study of limits of the two algorithms included in DLB (DWB and LeWI).

Chapter 2

Technical Background

In this chapter, we explain the environment in which this thesis have been developed. First we describe the parallel programming models used and their key features. Then we depict the execution environments where we have executed the experiments and developed our project. Finally, we will briefly describe the software used for development and support.

2.1 Programming Models

2.1.1 MPI: Message Passing Interface

MPI [13] stands for *Message Passing Interface* and is a communication API that allows the development of parallel distributed applications. MPI belongs to the family of SPMD (*Single Program Multiple Data*) parallel programming models and is tailored for distributed systems. It provides an interface to the processes to easily communicate and synchronize.

MPI has become the standard in distributed parallel programming, and it is broadly used for HPC applications.

The MPI model does not allow a progressive parallelization of the application.

In MPI applications, all data is private to each process. The data that need to be shared among processes must be sent/received explicitly with MPI

calls. Since data movements are specified in the code, it is not easy to adapt the application behavior to the different input data or architectures.

Furthermore, the programmer should have an excellent knowledge of the application because the data is distributed among the different processes and each one will compute over its data.

Current implementations of the MPI runtime include optimizations to run efficiently on SMP nodes. But some studies show that the use of a second level of parallelism exploited with programming models oriented to shared memory environments is a better approach.

Rabenseifner et al. [14] discuss that a hybrid parallelization model of MPI+OpenMP has several advantages such as, improve the load balancing, reduce memory consumption or more efficient memory accesses. But they also conclude that the better approach depends on the application, input and hardware architecture.

Smith et al. [15] demonstrate that hybrid programming is not always the most efficient mechanism for SMP systems for all codes. However, in some situations, an MPI+OpenMP model can be beneficial. For example, if the MPI parallelization suffers from poor scaling with MPI processes due to load imbalance or too fine grain problem size, memory limitations due to the use of a replicated data strategy, or a restriction on the number of MPI processes combinations. Besides, if the system has a poorly optimized or limited scaling MPI implementation, then a hybrid code may increase the code performance.

The reality in HPC environments is that most applications are implemented using MPI. These codes must be optimized or present performance issues. But porting them to new programming models to solve them is not viable. On the one hand, because these codes contain hundreds of thousands of lines and changing MPI would mean a whole refactoring of the code. Not only the amount of human resources to do this is prohibitive, but, in some cases they do not have the knowledge of all the parts of the application. On the other hand, moving towards experimental models or new programming paradigms is too delicate to be adopted in production codes.

2.1.2 OpenMP: Open Multi-Processing

OpenMP (*Open Multi-Processing*) [16] is the most widely used parallel programming model for shared memory in HPC applications. It also follows the

model of SPMD, but in this case, the programmers should define the loops and regions of the code that can be executed in parallel.

The interface that OpenMP offers is formed by some preprocessor directives. If OpenMP is not activated the compiler will detect these directives as comments and they will not have any effect. Therefore, the same source code can be used as sequential code or parallel OpenMP, depending on the compilation flags.

The preprocessor directives mark the parts of the code that can be executed in parallel and can include some additional clauses concerning data sharing specification.

The OpenMP runtime is in charge of spawning the threads at the beginning of the parallel section and join them back into the master thread at the end.

With OpenMP, the parallelization of applications can follow a bottom-up approach, because it allows a progressive parallelization of the code. To parallelize with OpenMP it is not needed a broad knowledge of the application to parallelize it. And neither is required a vast knowledge of the programming model, as most options can be left with the default values.

However, if we target to achieve the best performance a reasonable knowledge of the application and the programming model is needed to tune all the options, the schedule, the number of threads, etc..

A significant advantage of OpenMP is its malleability. The number of OpenMP threads running can be changed at runtime, opposed to MPI where the number of MPI processes is fixed during the whole execution of the application. In OpenMP the number of threads can be changed only outside a parallel region, for this reason its malleability is limited to the parallel regions of the code.

2.1.3 SmpSs: SMPSuperScalar

SmpSs (*SMPSuperScalar*) [17] is a task-based programming model for shared memory systems. It was first released in 2007.

SmpSs is also based in preprocessor directives. The programmer must mark in the code the functions that can run in parallel as tasks, and identify their inputs and outputs. With this information, the runtime will detect the

dependencies between tasks and schedule them in the different threads to maximize the performance.

SmpSs presents higher malleability than OpenMP, as the number of threads can be changed at any point during the execution of the application. The only limitation is that a thread cannot leave a task unfinished.

SmpSs hybridizes nicely with MPI [18] offering a powerful approach for HPC applications.

Having access to the source code was an advantage because this provided us the opportunity to integrate some of the DLB functionalities into the runtime and have a better control of the threads.

SmpSs is the predecessor of OmpSs, once all its features were included in the OmpSs runtime the maintenance of SmpSs was discontinued. For this reason in the current version of DLB, the SmpSs support is not maintained. Nevertheless some of the early results of this thesis where obtained using SmpSs.

2.1.4 OmpSs: OpenMP SuperScalar

OmpSs (*OpenMP SuperScalar*) [19] [20] is a programming model that integrates features from SmpSs and OpenMP. It is built on top of the Mercurium compiler [21] and the Nanos++ runtime system [22]. It supports asynchronous task parallelism and OpenMP parallel loops.

OmpSs is based on tasks and dependences. Tasks are the elementary unit of work which represents a particular instance of an executable code. Dependences let the user annotate the data flow of the program. With this information the runtime determines if the parallel execution of two tasks may cause data races and avoid them.

The OmpSs compiler can also process OpenMP parallel loops constructs and transform them into tasks. This allows us to easily execute already parallelized applications with OpenMP with the OmpSs model without modifying them and at the same time obtaining the benefit of the higher malleability of OmpSs.

The Nanos++ runtime can support OmpSs but also has modules to support other programming models such as OpenMP or Chapel [23]. Nanos++ and OmpSs has been used as a forerunner for OpenMP and several features

originally implemented and tested in OmpSs are currently in the OpenMP standard, such as, task dependences or priorities.

2.1.5 Programming Models Summary

We will now summarize the characteristics that are more relevant for this thesis of the programming models explained.

To parallelize an application with MPI we need a good knowledge of the application, its data accesses and the MPI programming model. When parallelizing with OpenMP to obtain a first working version is relatively easy, just some knowledge of the programming model and the code is necessary. But when trying to obtain a good performance a more broad knowledge of the programming model and the application is necessary, the same applies to OmpSs and SmpSs.

MPI does not allow an incremental parallelization while with OpenMP, OmpSs and SmpSs it is possible to parallelize progressively.

We have seen that MPI is not malleable. On the other hand, OpenMP can be malleable (if the programmer avoids some practices, like using the number of the thread to index arrays). But OpenMP still presents some limitations in the malleability, because the number of threads can only be changed outside a parallel region. OmpSs and SmpSs does not have this limitation and are full malleable, with the only restriction of not leaving a task unfinished.

2.1.6 Hybrid Programming Models

In current HPC systems the hardware design is hierarchical, nodes that contain several sockets with multiple cores, connected via a network infrastructure. To exploit the potential of these hierarchical systems using the parallel programming models already explained there are several alternatives. Rabenseifner et al. [14] compare the different options to use a hybrid model and identify the cases where one option is superior to another.

In Figure 2.1 we can see a summary of these alternatives to use a hybrid model. As we have seen we can use only MPI launching one MPI process per core. This option will communicate always through the MPI layer. In some MPI implementations there are optimizations for shared memory systems to make communications local to the node more efficient.

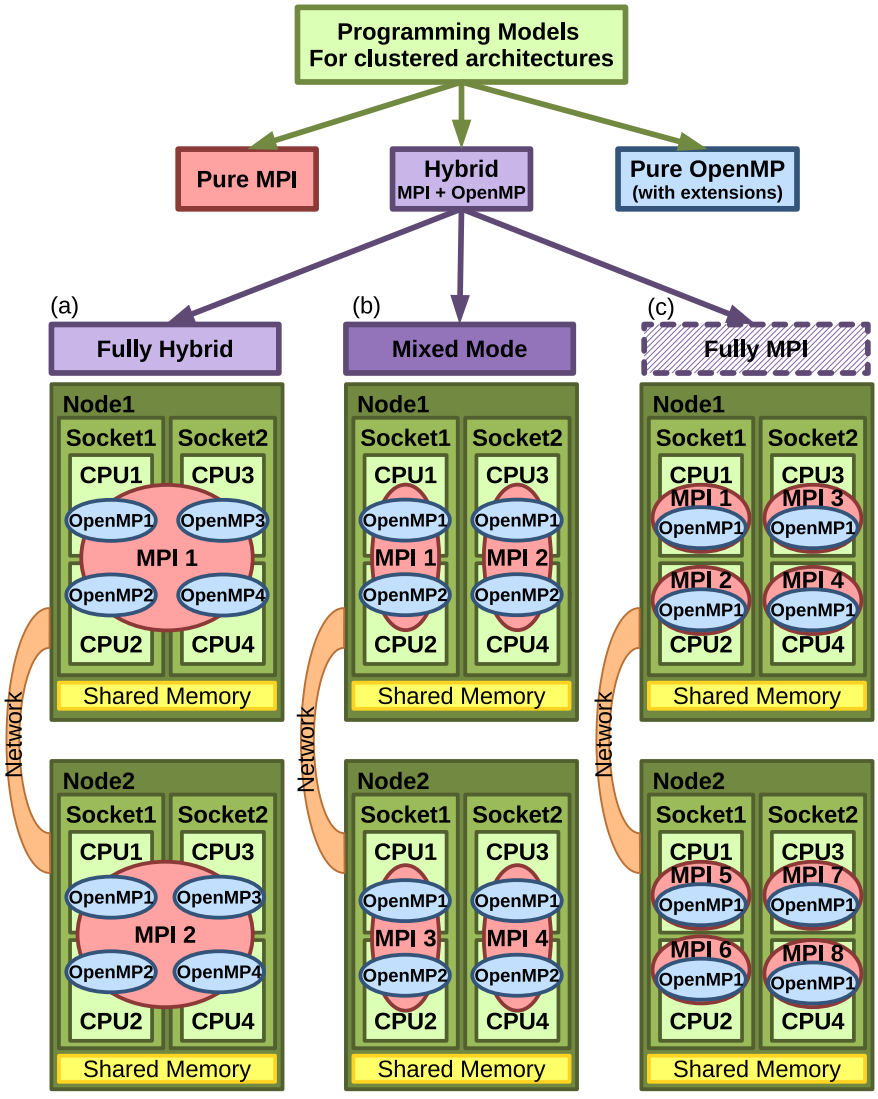


Figure 2.1: Programming models taxonomy for clustered architectures

We have added also the option of using pure OpenMP[1] but in this thesis we will not use it. There are several research work towards enabling distributed virtual shared memory. This alternative have the benefits of programming in a shared memory style while running in a distributed system. Intel offered "Cluster OpenMP" [24] but discontinued its support on 2010. OmpSs also offers a cluster flavor "OmpSs@Cluster" [25]. These works do not present a performance that can be compared with the one obtained by MPI at very large scale. A common decision in this models is to implement a relaxed memory consistency model in order to achieve a better performance.

The most "natural" approach is to use a hybrid programming model combining a distributed memory model with a shared memory one. The most widely used is to combine MPI and OpenMP. This combination allows to exploit the good performance of MPI in distributed memory systems, with the advantages of using OpenMP in the shared memory environment. When using this approach there are still different options. A fully hybrid model would launch one MPI process per node and spawn one OpenMP thread per core (Figure 2.1 (a)). Another alternative is to launch more than one MPI process per node, as the trend is to have nodes with several multi core sockets and different hierarchies of memory levels (Figure 2.1 (b)).

Finally in this thesis we will consider the possibility to launch one MPI process per core using only one OpenMP thread per MPI process (Fully MPI: Figure 2.1 (c)). In this case the OpenMP level is only used to solve load imbalance problems. The advantage of this configuration is that there is no need to have the code fully parallelized with OpenMP only the parts of the code that present load imbalance. This is important in all the pure MPI existing applications because it is not necessary to parallelize all the code that in some cases can be very extensive.

2.2 Execution Environments

Our target is a clustered architecture with shared memory nodes. During the development of this thesis we have used mainly two execution environments, until September 2012 we used Marenostrum2 to develop and run. Between

[1] In this subsection we will use OpenMP to refer to the shared memory programming model but it can always be changed by OmpSs or SmpSs

September and December 2012 Marenostrum2 was replaced by Marenostrum3. After December 2012 all the experiments and development has been performed in Marenostrum 3.

2.2.1 Marenostrum 2

Marenostrum2 [26] was based on Power PC processors; its nodes were JS21 blades with two IBM Power PC 970MP processors with two cores each and 8Gb of shared memory.

Each node had 4 cores, 8 GB of shared memory and a local SAS disk of 36 GB. Each node had a network card Myrinet type M3S-PCIXD-2-I for its connection to the high-speed interconnection and the two connections to the network Gigabit. In figure 2.2 we can see the components of a JS21 node in Marenostrum2

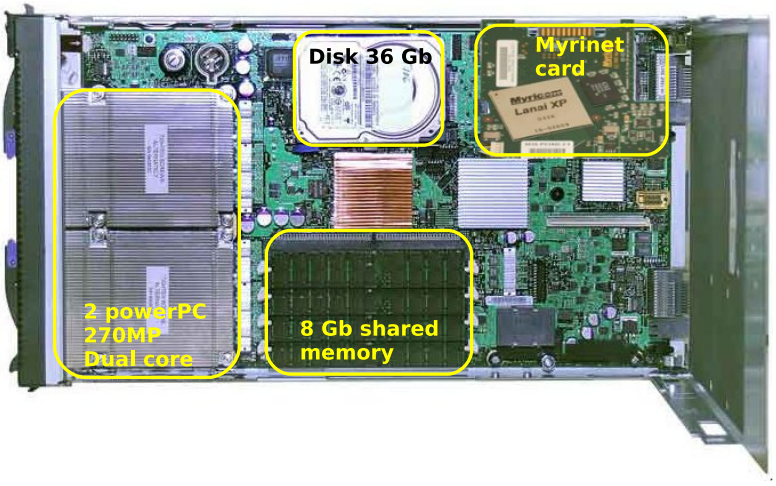


Figure 2.2: A node JS21 and its components

2.2.2 Marenostrum 3

Marenostrum3 [27] is based on Intel SandyBridge processors. Its compute nodes are IBM iDataPlex dx360 M4 X servers with two 8-core Intel Xeon processors (E5-2670) per node and 32 GB of shared memory. They also

include a hard drive of 500Gb and an MPI network card Mellanox ConnectX-3 Dual Port QDR/FDR10 Mezz Card. For management and GPFS they have two Gigabit Ethernet network cards.

In figure 2.3 we can see a schematic picture of the components of a compute node in Marenostrum3.

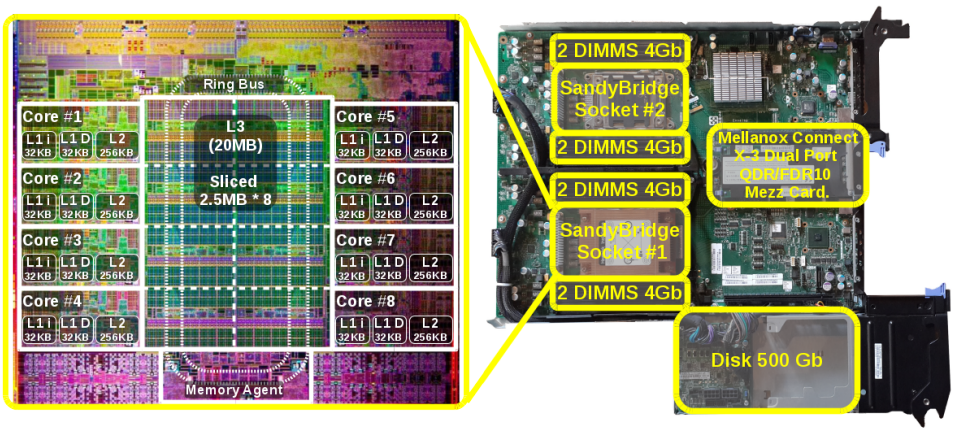


Figure 2.3: A compute node of MN3 and its components

We can see that each node has 2 sockets. Each socket has 8 cores. Caches L1 data, L1 instructions and L2 are local to the core. L3 has 20 Mb and is local to the socket.

This transition offers a new challenge for our balancing library as we changed from 4 cores per node to 16 cores per node. And opens a new study about the trade-off between the number of MPI processes and the number of threads inside a single node.

2.3 Software

2.3.1 Compilers

During the development of this thesis we have used different compilers depending on the applications requirements or the machine availability: *Intel* [28], *GNU* [29], *XL* [30] and *Mercurium* [31].

It is not within the scope of this thesis to compare the performance of different compilers. When doing performance evaluations, we have always used the same underlying compiler for all the executions. In each evaluation section, we state the exact compiler and settings used for the experiments.

2.3.2 Performance tools: Extrae and Paraver

An important part of this thesis is the in-depth evaluation of the efficiency obtained by applications and how to improve it. To do this analysis, we have used Extrae [32] and Paraver [33], two performance tools developed at BSC.

Extrae is a tracing tool that can obtain information about MPI, OpenMP, OmpSs, hardware counters (PAPI) and other performance information during the execution. We have worked together with Extrae developers to integrate DLB and Extrae, resulting in a special Extrae tracing library to use with DLB. This was necessary because both tools, DLB and Extrae, use the PMPI interface to intercept MPI calls.

Paraver is the trace visualizer we have used to analyze traces obtained from the executions. We use it also to compute some of the performance metrics we use in this thesis, such as efficiency or utilization of resources.

2.3.3 MPI libraries

The selection of the MPI library has significant restriction; we need the library to offer MPI blocking calls with a non-busy wait mode (i.e., MPI does not consume CPU while waiting on a blocking call).

When running in Marenostrum2, we used the MPICH [34] library because it was installed on the system by default and it also offered the MPI blocking calls with a CPU yield mode.

In Marenostrum3, we used OpenMPI [35], because it is the default MPI library of the system. But in some applications with special needs we used Intel MPI library [36]. Both libraries have a non-busy wait blocking mode.

To enable the non-busy wait mode usually it is enough to set an environment variable before the execution. The environment variable is different in each version of MPI. In Table 2.1 we can see the environment variable that must be set for each MPI library to enable a non-busy waiting mode.

MPI Library	Environment Variable
OpenMPI	OMPI_MCA_mpi_yield_when_idle=1
Intel MPI	I_MPI_WAIT_MODE=1
MPICH	MXMPI_RECV=blocking

Table 2.1: Environment variables necessary to set no busy waiting mode in each MPI implementation

Chapter 3

Applications and Benchmarks

In this chapter, we are going to describe the benchmarks and applications used during the evaluation of this thesis. To perform the different evaluations we have used five benchmarks with different characteristics (i.e. PILS, BT-MZ, SP-MZ, LUB, and Lulesh) and four applications (i.e., FLOWer, GROMACS, GADGET, and Alya).

For each benchmark and application used we will overview their function or the problem they solve, the programming model they use (initially or added by us) and any other relevant information.

3.1 PILS: Parallel ImbaLance Simulation

PILS (Parallel ImbaLance Simulation) is a synthetic benchmark. We developed it to help us evaluate load balancing techniques. With PILS we aim to reproduce the parallel region between MPI calls of an application.

The PILS benchmark has several configurable parameters that allow us to replicate the behavior of different types of applications and parallel regions. The different parameters are the following:

- *Programming Model:* We can compile four different versions of the benchmark: MPI, MPI+OpenMP, MPI+SmpSs, and MPI+OmpSs.

- *Load Distribution:* We can introduce the load balance of the application as the different loads for each MPI process. The more equilibrated the load distribution, the better load balance simulated. To compare executions with different load balances the sum of the loads given to the MPIs must be the same (this will give the same amount of total work to do).

- *Parallelism Grain:* Represents the percentage of computation that is run in parallel between MPI synchronizations. It is computed as the reciprocal of the number of parallel regions between MPI blocking calls ($Par.Grain = \frac{1}{Par.Regions}$). You can see an example of the relationship between Parallelism Grain and the number of parallel regions in Figure 3.1. As we will see the Parallelism Grain is directly related to the malleability of the application.

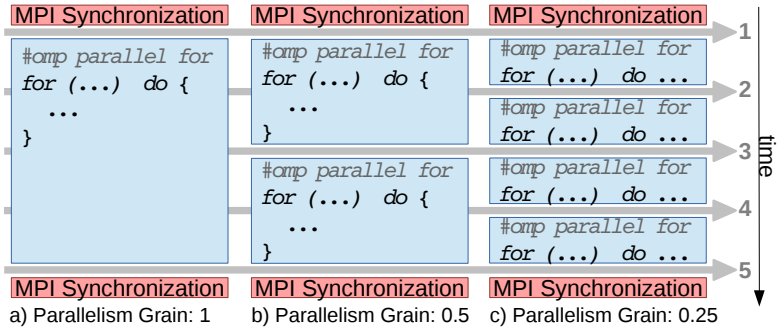


Figure 3.1: Parallelism grain explanation

- *Iterations:* The number of iterations that we will execute. Each iteration includes the computational part (parallel regions in OpenMP or SmpSs) and MPI communication.

In Algorithm 1 we can see the pseudocode of the core of PILS. The load for each MPI process is read from the input, and correspond to the number of iterations that will compute each process. Then the block size is calculated based on the *Parallel Grain* parameter. In the code, we can see the OpenMP version of the parallelization.

Algorithm 1 PILS structure

```
1: BS = myLoad * parallelGrain
2: for (i=0; i<iterations; i++) do
3:    for (j=0; j<steps; j+=BS) do
4:       #pragma omp parallel for
5:       for (k=j; k<BS; k++) do
6:          compute();
7:       end for
8:    end for
9: end for
10: MPI_call()
```

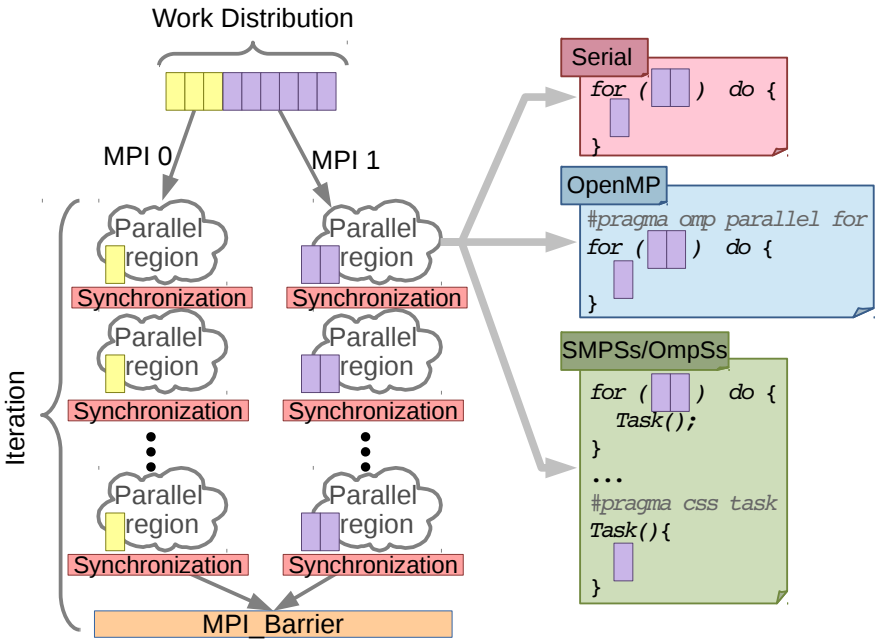


Figure 3.2: PILS benchmark

In Figure 3.2 we can see a schematic representation of PILS. The workload is given at the beginning of the execution to each MPI process (work distribution). This workload is computed in the parallel regions. The number of parallel regions depends on the parallelism grain parameter. A parallel region in the OpenMP version corresponds to a parallel loop. In the SmpSs version, a parallel region is a loop that creates several tasks and finishes with an SmpSs barrier. At the end of the iteration there is an MPI barrier to synchronize all the processes.

The core of the synthetic benchmark is a function that will do several floating point operations without data involved. In the case of the OpenMP version this function will be called each iteration, in the case of SmpSs version, this function will be taskified (each call to the function will be a parallel SmpSs task). In the right side of Figure 3.2 we can see how the code would look like for the three versions (serial, OpenMP and SmpSs).

The cost of the core function in terms of time and computation is always the same. The imbalance between MPI processes will be introduced by the number of times that the function is executed which will be given by the loads of each MPI process.

PILS do not aim to model a whole application. But a parallel application can be modeled as several instances of PILS. This is because real applications do not always present a regular behavior regarding amount and kind of imbalance or parallelism grain, among other factors that can affect the load balancing mechanism.

3.2 BT-MZ and SP-MZ from NPB-MZ (NAS Parallel Benchmarks Multizone)

The NAS Parallel Benchmarks (NPB) is a well-known benchmark suite to test parallelization tools, models or systems. The Multizone version (NPB-MZ)[37] is based on the NPB benchmarks but targets a multilevel parallelization.

In the NPB-MZ benchmark suite, we can find 3 applications: LU (Lower-Upper symmetric Gauss-Seidel), SP (Scalar Penta-diagonal), and BT (Block Tri-diagonal). All of them solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions.

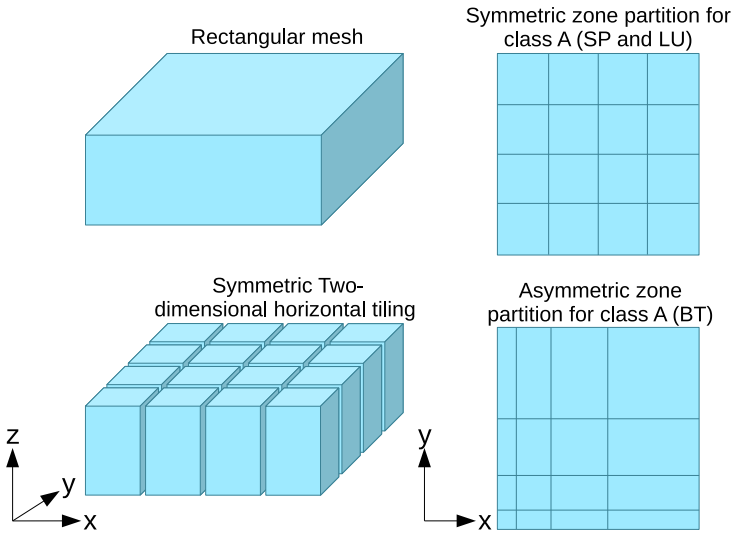


Figure 3.3: NPB-MZ benchmarks zone partition

To achieve a multilevel parallelization a logically rectangular discretization mesh is divided into a two-dimensional horizontal tiling of three-dimensional zones for each benchmark (See Figure 3.3 left-hand side).

SP-MZ and LU-MZ do the zone partition in a symmetric way (i.e. all the zones have the same size). Therefore, their executions are expected to be balanced. On the other hand, the BT-MZ zone partition is done asymmetrically, and this results in an imbalanced execution. In Figure 3.3 (right-hand side) we can see the difference between a symmetric zone partition (top, for SP and LU benchmarks) and an asymmetric zone partition (bottom, for BT benchmark).

The NAS benchmarks can run different classes that correspond to the size of the problem that it is solving. In Table 3.1 we can see the size of the mesh for each class and the number of zones in which is divided each dimension for the different multizone benchmarks.

In Figure 3.4 we can see a trace of the BT-MZ benchmark running with 4 MPI processes and 4 OpenMP threads each process. In blue color we can see the computing phases of the benchmarks and green and red are a *MPI_WaitAll* and *MPI_Barrier* respectively. When a processes reaches a

Class	Mesh Dimensions (x,y,z)	Zones (x,y,z)		
		LU-MZ	SP-MZ	BT-MZ
S	24 x 24 x 6	4 x 4 x 1	2 x 2 x 1	2 x 2 x 1
W	64 x 64 x 8	4 x 4 x 1	4 x 4 x 1	4 x 4 x 1
A	128 x 128 x 16	4 x 4 x 1	4 x 4 x 1	4 x 4 x 1
B	304 x 208 x 17	4 x 4 x 1	8 x 8 x 1	8 x 8 x 1
C	480 x 320 x 28	4 x 4 x 1	16 x 16 x 1	16 x 16 x 1
D	1632 x 1216 x 34	4 x 4 x 1	32 x 32 x 1	32 x 32 x 1

Table 3.1: NPB-MZ class problems

blocking MPI call, all its threads are idle waiting for the communication to finish (black color in the trace). In this trace we can clearly observe the imbalance introduced by the asymmetrical partition of the zones, we can see how the less loaded MPI processes (1, 2 and 3) are waiting on an MPI blocking call.

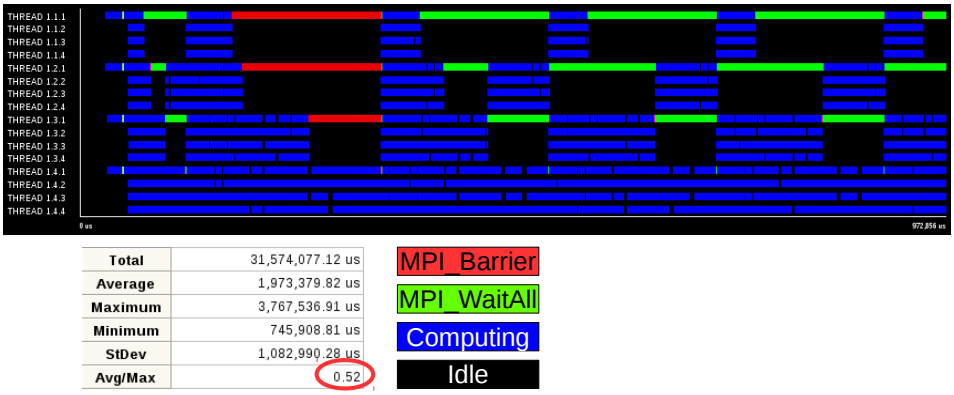


Figure 3.4: Trace of BT-MZ, 4 MPI processes and 4 OpenMP threads each

In this thesis, we will use the BT-MZ and SP-MZ benchmarks as representative of an imbalanced and a balanced application. The original version that we will use of these benchmarks is MPI+OpenMP. We have modified the BT-MZ benchmark to have an MPI+SmpSs and MPI+OmpSs version of each one.

3.3 LUB

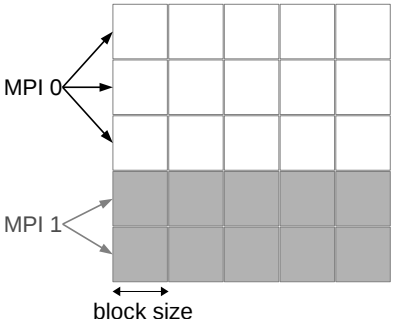


Figure 3.5: Distribution of blocks among MPI processes in LUB

LUB is a kernel performing an LU matrix factorization. The structure of data is a two-dimensional square matrix organized by blocks. The size of the matrix and the block size are input parameters of the kernel. The data is distributed by blocks of rows among the different MPI processes as can be seen in Figure 3.5.

In Figure 3.6 we can see a graphical description of the LUB algorithm used. There are four functions that are applied to the data blocks, *lu0*, *fwd*, *bdiv* and *bmod*. The *fwd* blocks can be done in parallel between them but depend on the *lu0* computation. The *bdiv* blocks can be done in parallel but depend on the *lu0* block. And the *bmod* blocks can be done in parallel but each one depends on the *fwd* block in the same column and the *bdiv* block in its row. Each iteration the *lu0* is applied to a block of the diagonal.

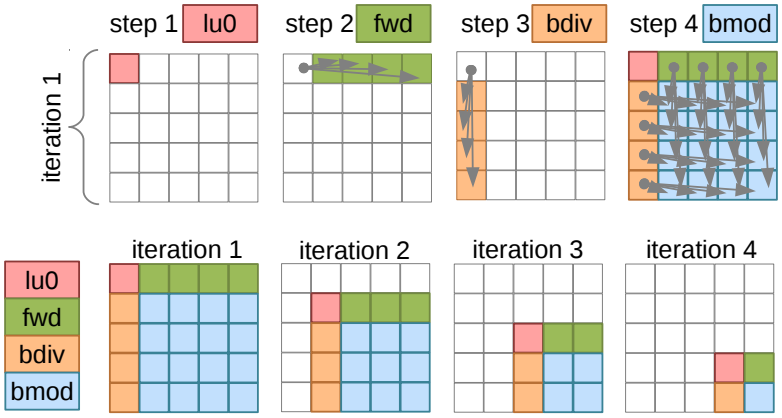


Figure 3.6: LUB behavior

To obtain a hybrid version, the straight forward solution would be to parallelize the blocks of the *fwd* and *bmod* with OpenMP.

In the following example we can see the main code of the LUB kernel (left-hand side) and a possible hybrid parallelization (right-hand side). In the example, N is the number of blocks, and A is the matrix.

```
for (i=0;i<N;i++){
    lu0(i,i);
    for(j=i+1;j<N;j++){
        fwd(i,j)
    }
    for(k=i+1;k<N;k++){
        bdiv(k,i);
        for(j=i+1;j<N;j++){
            bmod(k,j);
        }
    }
}
```

Main LUB kernel code

```
for (i=0; i<N;i++){
    lu0(i,i);
    #pragma omp parallel for
    for(j=i+1; j<N;j++){
        fwd(i,j)
    }
    MPI_Bcast(A[i]...)
    for(k=i+1; k<N;k++){
        bdiv(k,i);
        #pragma omp parallel for
        for(j=i+1; j<N;j++){
            bmod(k,j);
        }
    }
}
```

LUB kernel code parallelized with MPI+OpenMP

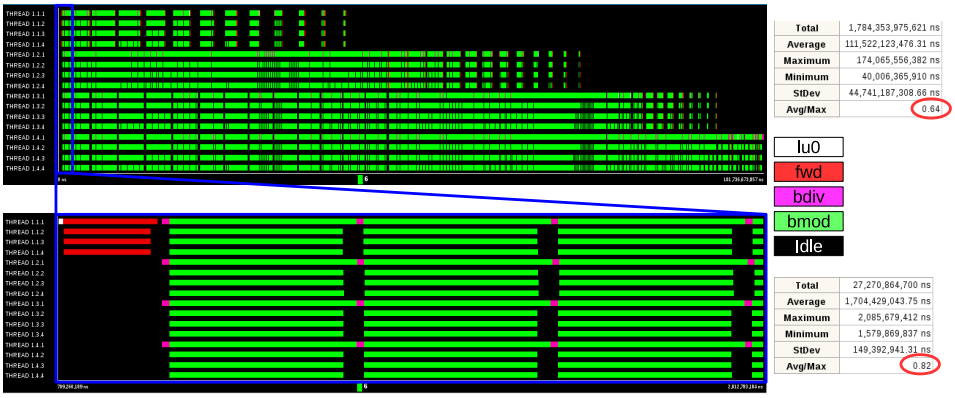


Figure 3.7: Trace of LUB, 4 MPI processes and 4 OpenMP threads each

In Figure 3.7 We can see a trace of an execution of the LUB kernel. In this execution, we were running 4 MPI processes on the same node with 4

OpenMP threads each one. In the upper trace, we can see the whole execution in the lower trace we can see a zoom of the first iteration.

We can observe that this benchmark presents a critical imbalance. On one hand, the *lu0* and *fwd* functions are only applied to a row of blocks. Therefore while the process that owns this row is computing the *lu0* and *fwd* blocks the other processes are waiting. We can observe this in the lower trace: while the MPI process 1 is computing the *lu0* and *fwd* blocks (white and red), MPI processes 2, 3 and 4 are idle (black).

We can also observe the triangular behavior of this benchmark, and how as the load decrease during the execution at the end of the trace some processes are idle all the time.

We can see how the load balance during the execution is very irregular, when computed the load balance of the whole application we obtain 0.64, while the load balance measured for the first iteration is 0.82.

On the other hand, each iteration the number of rows of blocks that have each MPI process will be different therefore the load of each MPI process will be different.

Moreover, the imbalance in this benchmark is very irregular as it changes during the execution. Each iteration a different process will be doing the *lu0* and *fwd* blocks and will have a different amount of rows of blocks.

The LUB benchmark has three versions MPI+OpenMP, MPI+SmpSs and MPI+OmpSs. The OpenMP version is parallelized at the block level, as we have shown in the code example. In the case of SmpSs and OmpSs we have defined each function (*lu0*, *fwd*, *bdiv* and *bmod*) as a task with the corresponding dependences to ensure the correct execution.

3.4 FLOWer

The FLOWer code[38] solves the compressible, three-dimensional Reynolds-averaged Navier–Stokes equations. It uses a cell-vertex, finite-volume formulation on block-structured meshes. The baseline method employs central space discretization with artificial viscosity and explicit multistage time-stepping schemes. It has been developed at the German Aerospace Center (DLR).

FLOWer is part of a project to simulate PHOENIX, which is a small-scale prototype of the Space Hopper (similar to the US Space Shuttle). FLOWer

is used to simulate the flow around the clean configuration of PHOENIX by 3D Euler and Navier-Stokes computations.

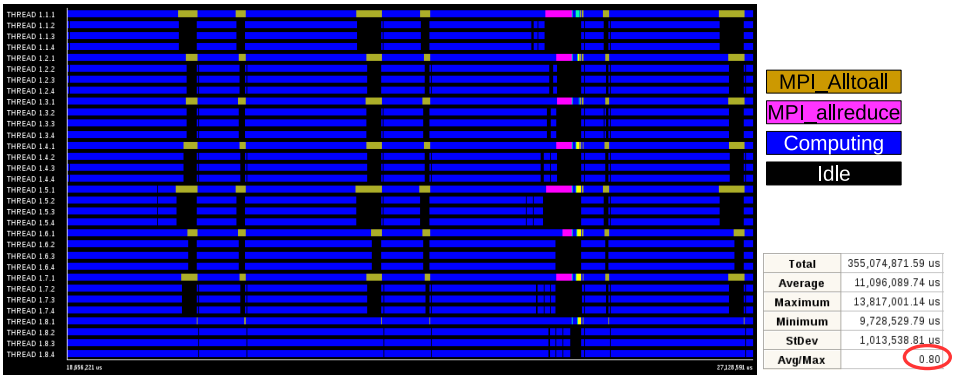


Figure 3.8: Trace of FLOWer, 8 MPI processes and 4 OpenMP threads each

This application is implemented in Fortran and provides three different parallelized versions: OpenMP, MPI, and a hybrid MPI+OpenMP. In our experiments, we have used the hybrid parallelization of FLOWer. The source code of the MPI+OpenMP version contains more than 300.000 lines of code and 363 loops parallelized with OpenMP.

We have used FLOWer without analyzing its performance nor its behavior previously. Neither it was necessary to modify the code to use it.

In Figure 3.8 we can see a trace of an execution of FLOWer with 8 MPI processes and 4 OpenMP threads each process. In blue we can see the computation, that is mainly done in parallel. We can observe some calls to collective MPI calls (i.e. MPI_Alltoall and MPI_Allreduce) in pink and yellow. We can note that the MPI process 8 is more loaded than the other MPI processes, for this reason, the other MPI processes and its threads must wait on an MPI call.

3.5 GROMACS

GROMACS [39] [40] is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules

like proteins, lipids, and nucleic acids that have a lot of complicated bonded interactions, but since GROMACS is extremely fast at calculating the non-bonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, e.g. polymers.

We use version 4.0.5 of GROMACS. We started with an MPI version of GROMACS and parallelized some parts of the code with SmpSs to have a hybrid MPI+SmpSs version. It is important to mention that this application was not parallelized completely with SmpSs. We will use the SmpSs level of parallelism only to help load balance the MPI level. For this reason, the executions must be done with only one SmpSs thread per MPI process, the same way that when running the MPI only version.

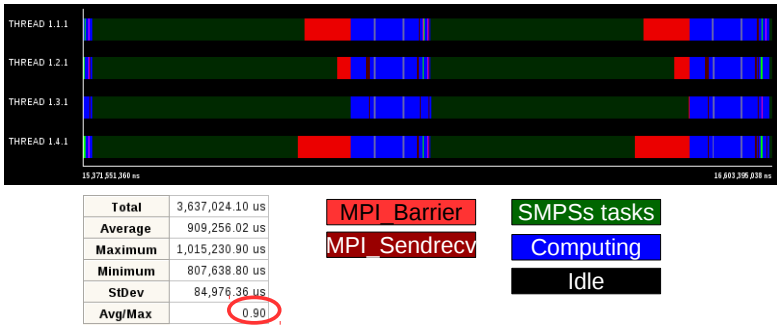


Figure 3.9: Trace of GROMACS, 4 MPI processes and 1 SmpSs thread each

In Figure 3.9 we can see a trace of GROMACS with 4 MPI processes and one SmpSs thread each one. What we visualize in the trace is a cut of the whole execution including two iterations. In green, we can see the computation that has been parallelized with SmpSs and in blue the computation that is serial code in each MPI process. We can observe that there is load imbalance between MPI processes. The load imbalance is visible in the trace because the different processes spend different amount of time in the MPI_Barrier (in red) waiting for the most loaded process (i.e. MPI process 3). In this application, the load imbalance depends on the input.

3.6 GADGET

GADGET (**GA**laxies with **D**ark matter and **G**as int**E**rac**T**), [41] [42] is a production code that performs a cosmological N-body/SPH simulation. It can be used to address different astrophysical problems such as colliding and merging galaxies or the formation of large-scale structure in the Universe.

The gravitational forces in GADGET are computed with a hierarchical tree algorithm, and the fluids are represented through smoothed particle hydrodynamics (SPH). The computation is performed by time-steps.

We used the version 2.0 of GADGET and started from a version parallelized with MPI. Although the application comes with its own load balancing code that dynamically updates the tree, there was still some imbalance problems that were not solved.

To obtain a hybrid version of the application that can be balanced with DLB we added an OpenMP and SmpSs parallelization. We only parallelized the parts of the code that could benefit from the load balancing mechanism (like previously with GROMACS), they were 3 loops out of 35.000 lines of code. The input we use runs on 800 CPUs and has a high load imbalance.

3.7 Lulesh

Lulesh (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [43] [44] [45] is a mini-app developed at the Lawrence Livermore National Lab (LLNL) and included in the CORAL benchmark suite.

Lulesh is a shock hydrodynamics code for unstructured meshes. The reference code is drawn from a production LLNL hydrodynamics code. Lulesh is large enough (more than 4,000 lines of code for the parallel MPI implementation) to be more complex than traditional benchmarks, yet compact enough to allow a large number of implementations.

The code of this mini-app is available for MPI, OpenMP, and a hybrid MPI+OpenMP version, additionally, it has been ported to different architectures or programming models (e.g. Chapel, GPUs, CHARM++). We started from the MPI version and added OmpSs to the parts of the code that could benefit from the load balancing mechanism.

When running Lulesh the number of MPI processes that can be used must be a cube of an integer. For this reason, we have executed most of the exper-

iments with 64 MPI processes (4x4x4), which fills 4 nodes of Marenostrum3 (4 nodes x 16 cores per node).

Lulesh 2.0 include a flag to set the amount of load imbalance between regions. A value of 0 in the flag means the execution presents no imbalance (i.e. perfect load balance). Increasing the value of the flag increases the load imbalance. In our experiments, we executed with values from 0 to 8.

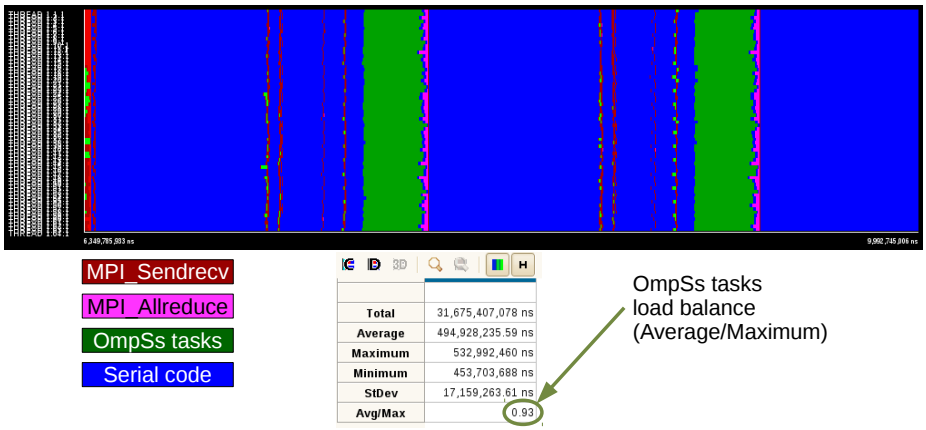


Figure 3.10: Trace of Lulesh, 64 MPI processes and 1 OmpSs thread each. Load balance 0

In Figure 3.10 we can see a cut of a Lulesh trace with two iterations. In this case, we are running 64 MPI processes with one OmpSs thread each process. In green, we can see the part of the code that was parallelized with OmpSs and in blue the remaining computing phases not parallelized. The most relevant MPI call in this code is an MPI_Allreduce that can be seen in pink. When computing the load balance of the OmpSs tasks we obtain a 0.93 (i.e. 1 is a perfect load balance and 0 the worst case).

In Figure 3.11 we can see the same execution (64 MPI processes, 1 OmpSs thread, 2 iterations) with the load balance flag set to 8. In this case, the load imbalance can be seen in the OmpSs taskified part of the code. When computing the load balance of the OmpSs tasks of this execution we obtain 0.5.

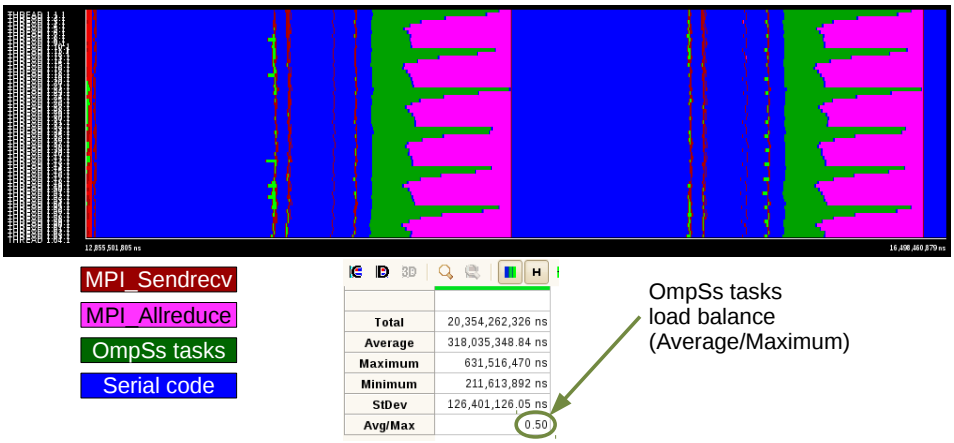


Figure 3.11: Trace of Lulesh, 64 MPI processes and 1 OmpSs thread each. Load balance 8

3.8 Alya

Alya [46] [47] is a simulation code for computational mechanics developed at BSC by the CASE department. The Alya system can solve different physics, such as incompressible flows, compressible flows, turbulence, solid mechanics or particle transport.

Alya has been designed and developed for HPC environments. It is included in the Unified European Application Benchmark Suite (UEABS) [48] which contains 12 applications currently relevant, scalable to thousands of processors and maintained into the future. Alya is parallelized with MPI and OpenMP.

In Figure 3.12 we can see a trace of an Alya execution including two time steps when solving a fluid dynamics simulation. In this trace, we can see the different computational phases inside a time step in different colors (yellow, purple, brown) the serial code appears in green and in pink, we can identify the MPI calls *MPI_Allreduce*. We can observe that the load imbalance in this execution was very high. The load imbalance depends highly on the input mesh. We will explain further the inputs used and its characteristics in Chapter 9.

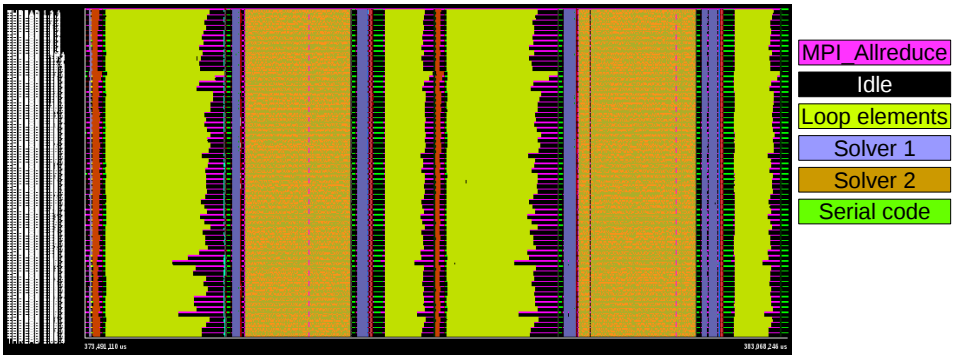


Figure 3.12: Trace of Alya, 65 MPI processes and 4 OpenMP threads each

Chapter 4

Related Work

It is well known that load imbalance is a source of system efficiency loss in HPC applications. One of the characteristics of the HPC environment is that the resources necessary to run must be specified beforehand, being not possible to ask for more resources in the middle of an execution to solve the load imbalance. The straightforward solution for this problem is to tune parallel codes by hand. Usually, programmers include load-balancing code or redistribute the data to improve the performance of their applications.

These alternatives usually result in codes optimized for specific environments or execution conditions and imply devoting many economic and human resources to tune every parallel code. Furthermore, usually these solutions do not consider or are not able to solve other factors such as heterogeneous architectures, load imbalance because of resource sharing, or irregularity of the application.

This is especially dramatic in MPI applications because data is not easily redistributed. And even more in MPI based hybrid applications because the computational power loss due to load imbalance is multiplied by the number of threads used per MPI process. For these reasons load imbalance in MPI applications is targeted in many research works.

Proposed solutions for load balancing can be divided into two main groups: The ones that are applied **before the execution** and the ones that are applied **during the execution**.

4.1 Before the execution

The solutions that are applied before the computation try to do a balanced distribution of the data and computation between the MPI processes. These solutions are usually applied to particular data distribution (graph, mesh...) and must be calculated before each execution for each different set of input data. The most representative and widely used solution in this group is Metis [1]. Shang et al. [49] discuss the impact of the different mesh partitioning solutions (Metis, ParMetis, Scotch...) in the performance of a CFD code. They conclude that the mesh partitioning strongly affects the performance, especially for large scale parallel codes.

This type of solutions has several drawbacks, on the one hand, the additional computation time needed to do the domain partitioning before the execution. On the other hand, they can only handle static imbalances coming from data distribution; they cannot handle dynamic imbalances coming from the sharing of resources, interferences from the operating system or inherent to the execution.

Another problem of this solutions is that the load can change during the execution, there are two main examples for this, on the one hand changing the geometry (moving bodies, deformations, etc.). On the other hand changes in the granularity of the mesh. To solve these problems remeshing is used during the execution (re-partition the mesh to obtain a better load balance). PAMPA [4], CIMLib [50] [51] and Yi et al. [52] are some examples that try to improve the load balance by repartitioning the mesh during the execution. These solutions are expensive in execution time because a remeshing must be done periodically. And as we will see in Chapter 9 it is not always direct to compute a well balanced distribution, either because of the characteristics of the mesh or because the heuristic need to calculate the load balance is not trivial or direct to find. Moreover, different parts of the code can present different load balance distributions, making it impossible to maximize the load balance with a single mesh partition.

4.2 During the execution

The second group tries to load balance applications during the execution, in the literature we can find two kinds of approaches: the ones that **re-**

distribute the data or the ones that **redistribute the computational power**.

4.2.1 Redistribute the data

Several approaches choose the option to redistribute the data of the application so that it is better balanced. Charm++ [53] is an object-oriented parallel programming language that employs object migration to achieve load balance. Adaptive MPI (AMPI) [5] is an implementation of MPI that uses the load balancing capabilities of Charm++. Also based in Charm++ redistribution capabilities we find the work of Deodhar et al. [54] they use a compiler load prediction to decide the migration of data for load balancing purposes.

Balasubramaniam et al. [55] also propose a library that dynamically balances MPI processes. Their load balancing is done based on a parallel loop; a dynamic scheduler will decide how many iterations will be executed by each process and the data will be migrated accordingly.

Martín et al. propose FLEX-MPI [56] an extension to MPI to support load balancing in heterogeneous dedicated or non-dedicated systems, the runtime will collect hardware counters during the execution using the MPI profiling interface and based on this information the application data is redistributed to improve the load balance.

The approach of redistributing the data is usually rigid regarding the type of imbalance it can solve. The granularity of the imbalance should be enough to allow a data migration and still obtain a performance gain. These solutions must be very careful with the frequency of the redistribution, a high frequency can introduce too much overhead, a low frequency will leave some imbalances unsolved. Moreover, to apply these solutions the data structures of the application should be able to be repartitioned and in most of the cases the applications are aware of the load balancing algorithm and are modified somehow.

4.2.2 Redistribute the computational power

The approaches that redistribute the computational power to solve the imbalance are more flexible and transparent to the applications and the programmer.

We can find a broad amount of research works for specific applications. The NAS benchmarks include [37] in the code of BT-MZ a load balance mechanism based on changing the number of threads assigned to each MPI process depending on the size of the zones it has assigned [57]. Meraji et al. [58] present a load balancing algorithm for a discrete event simulator, every C cycles they gather information of each processor, the number of events processed since the last execution of the load balancing algorithm and the number of messages sent by that processor. With this information, they will migrate logical processors to less loaded processes. Arafat et al. [59] propose a Resource Sharing Barrier to load balance random walk Monte Carlo simulations. They will redistribute the processor resources of the inner levels of parallelism when groups that complete their work early lend their resources temporarily to slower groups. The target application must be modified to use the RSB capabilities.

These approaches can only be applied to the specific applications. We will focus on more generic approaches that can be applied to any kind of application.

The first approach is to combine two levels of parallelism. Smith et al. [60] discuss the best solution between MPI, OpenMP or the hybrid (MPI + OpenMP) model. They conclude that the hybrid programming model may not be the best solution for all the codes. However, in some situations, a significant benefit can be obtained from this hybrid model. Henty et al. [61] compare MPI versus a hybrid MPI + OpenMP model in an SMP cluster. They conclude that the hybrid model is more efficient in very load imbalanced situations. In the same kind of study on clusters of multiprocessors, Cappello et al. [62] expose that the superiority of one model depends on the amount of parallelization at shared memory level, the communication patterns, and the memory access patterns.

Some relevant works that try to solve the imbalance redistributing the computational power are the following. Zhang et al. [63] presented a solution for Hyperthreaded (HT) and Simultaneous Multi Threaded (SMT) processors with a self-tuning loop scheduler that selects the number of threads that should be created to execute a parallel loop. They monitor the execution time of each iteration and try the different OpenMP schedulers and combination of threads by using the SMT or not. With this information, the best scheduler and number of threads is found.

For non-dedicated systems Sievert et al. [64] propose a system that allocates more processors than needed when the application starts. When it detects that a process is not performing as expected the system swaps the MPI process to a less loaded processor.

El Maghraoui et al. [65] use a technique of process migration to load balance iterative applications. The system will give them more resources during the execution, and they will decide to migrate processes to these new resources. They use the PMPI profiling interface to obtain performance information and take the decisions. They are limited to iterative applications and systems where it is possible to get additional resources during the execution. We will focus on solutions that try to obtain the maximum efficiency of the given resources and do not need a special kind of application structure.

One solution that propose to redistribute the computational power is *Invasive Computing* [66]. This paradigm presented by Teich et al. propose that applications are resource aware and can ask for resources when arriving a massive parallel region or deallocate resources if the program enters a retreat phase. The programmer is in charge of marking the parts of the code where more resources can be used and retreat phases. Our approach is transparent to the user, without the need of modifying the application or analyzing its performance previously.

Schreiber et al. [67] propose a load balancing mechanism in invasive computing for hybrid applications. The number of cores assigned to each MPI process can be changed based on information given by the application (developer). This information consists of a scalability graph or a workload number obtained from previous executions.

Other solutions that use the redistribution of computational power and are aimed at applications with two levels of parallelism are the works done by Spiegel et al. [68] and Duran et al. [12]. They seek to balance applications with two levels of parallelism by redistributing the computational power of the inner level, and they all do it at runtime without modifying the application. The first one balances MPI+OpenMP hybrid applications. The second one balances OpenMP applications with nested parallelism. Although their algorithms are different, the two first converge to the same solution. They both use previous iterations to detect the load of each process.

The solution we developed in this thesis belongs to the group of solutions applied at runtime, and it tries to redistribute the computational power. In

our case we do not need the application to present an iterative pattern, nor a regular imbalance during the execution. Moreover, our proposal can handle fine grained load imbalances.

When	How	Solution	Drawbacks	Who
Before Execution	**Distribute Data**	Mesh partition	Static. Not easy to determine an heuristic to represent the load	Metis[1],Shang et al.[49]
		Remeshing	Must be done periodically and is computational expensive. Not easy to determine an heuristic to represent the load	PAMPA[4],CIMLib[50] [51],Yi et al.[52]
During Execution	**Redistribute Data**	Object Migration	Specific programming language. Only suitable for some data structures. Data movements time consuming	Charm++[53],AMPI[5], Deodhar et al.[54]
		Loop Scheduling	Application must be modified. Data movements are time consuming. Only for iterative applications	Balasubramaniam et al.[55]
		Data Migration	Data movements are time consuming. Needs a regular load imbalance during the execution	FLEX-MPI[56]
	Redistribute Computational Power	Coded in Applications	Application specific	NAS[57],Meraji et al. [58], Arafat et al.[59]
		Hybrid Model	Do not really attack load imbalance	Smith et al. [60], Henty et al.[61], Cappello et al.[62]
		Self-tuning loop scheduler	Only for HT and SMT processors. Only for applications with iterative pattern and loops	Zhang et al.[63]
		Migrate MPI processes	For non-dedicated systems. Assumes more resources are available	Sievert et al.[64],El Maghraoui et al.[65]
		Invasive Computing	Needs previous scalability information of the application, provided by the application or the user	Teich et al.[66], Schreiber et al.[67]
		Use second level of parallelism	Iterative pattern needed. Attacks only coarse grain load balance	Spiegel et al.[68], Duran et al.[12]

Table 4.1: Related work summary

Chapter 5

The Approach: Dynamic Load Balancing Library

The core of this thesis is based on a runtime library: The Dynamic Load Balance library. DLB is a framework that will manage the computational resources within a node to obtain the maximum efficiency.

In this chapter, we describe the key features of the DLB framework and the main concepts of its architecture.

5.1 DLB Philosophy

The DLB library aims to improve the load balance of HPC hybrid applications (i.e., two levels of parallelism). In Figure 5.1 we can see the structure of a typical HPC hybrid application: one application that will run processes on several nodes, and each process will spawn several threads. In this thesis we will refer to these two levels of parallelism as the *inner level* of parallelism (threads) and *outer level* of parallelism (processes).

In the context of this thesis, the outer level of parallelism will be MPI. MPI is the standard de facto for HPC applications, and most scientific applications are already implemented using MPI. Moreover, MPI does not offer a solution to load balancing problems mainly because it is not malleable.

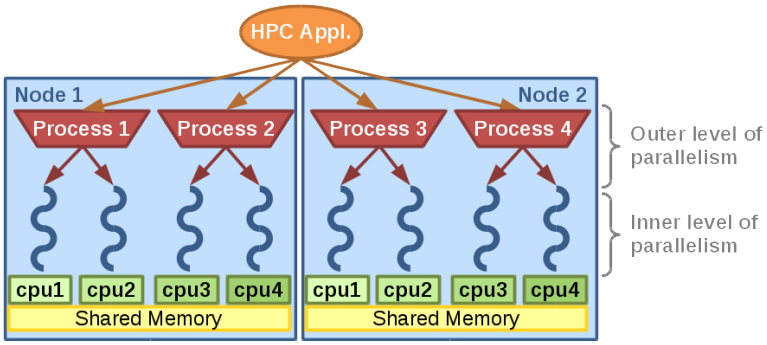


Figure 5.1: Typical structure of an HPC application

However, as we will see in the following sections, DLB is designed and prepared to be used with other programming models applied at distributed memory level (e.g. PGAS, Charm++) or parallel libraries with a reasonable integration effort.

The central idea in which the DLB library is based is to balance the outer level of parallelism by redistributing the resources of the inner level of parallelism.

The most distinctive characteristics of the DLB library are the following:

- No need for exhaustive performance analysis of the application.
- Is not necessary to modify the application.
- Can solve load imbalance problems from different sources (i.e. Data load imbalance, algorithmic load imbalance, resources load imbalance).
- Can share resources between different applications running on the same node.

5.2 Main Concepts and Terminology

DLB is a runtime that will manage the resources inside a computational node to maximize the efficiency. In the current implementation, we do not consider sharing resources between different users nor accounting of resources for a fair share.

The computational resources DLB is in charge are CPUs inside a node with shared memory.

We consider that there are running several processes in a computational node. Each process can have one or more threads. Threads are managed by a parallel runtime (i.e. OpenMP) and DLB will use an API offered by the parallel runtime to regulate the threads of each process.

Each CPU will be owned by one and only one process. The owner of a CPU is the only one to have privileges over it. The owner of a CPU does not change while the process is alive.

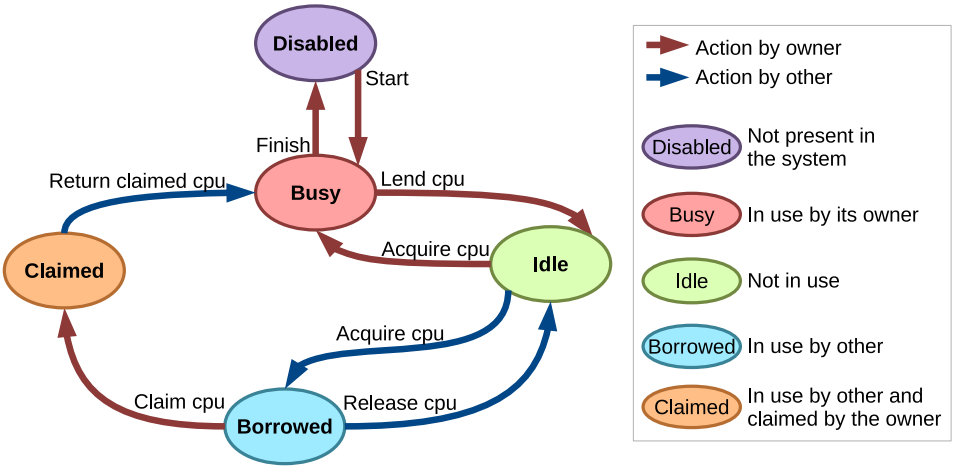


Figure 5.2: States of a CPU in DLB

A given CPU can be used by other processes that are not the owner if the owner is not using it. Only one thread can be active at the same time in a CPU. DLB does not allow oversubscription of CPUs.

The owner of a CPU can lend it to the system. When a CPU has been lent to the system, the owner does not change. When a CPU is idle because it has been lent, any process in the system can borrow it. The owner of a CPU, and only the owner can claim a previously lent CPU.

In Figure 5.2 we can see a graph summarizing the different states of a CPU in DLB and what triggers a transition between states. By default the CPUs are in *Disabled* state, meaning that they are not in the system. When a process starts its CPUs are added to the system, and while the owner is

using them they are in the *Busy* state. When the owner lends a CPU to the system, the CPU is *Idle* and ready to be used by any process in the system. If another process, not the owner, gets a CPU to use it, the CPU is *Borrowed*. When the owner of a CPU wants to use it, it has preference over the other processes in the system, if the CPU is *Idle* he can take it. If the CPU is being used by another process (*Borrowed* state), the owner will claim it, and the CPU will be in *Claimed* state until the borrowing process releases it.

5.3 DLB Framework

In this section, we are going to explain the structure of the DLB library. The DLB library has been designed and developed in a modular way. This design allows to expand the functionality of the library quickly, for example:

- Add support for different programming models at the outer level of parallelism. The current version has support for MPI and the concurrent execution of multiple applications.
- Add support for different programming models at the inner level of parallelism. The current version of the DLB library supports OpenMP and OmpSs. These are two examples of different levels of integration. With the OpenMP runtime, almost no integration was required, we used the standard OpenMP API (`omp_set_num_threads()`) to change the number of threads available. On the other hand, with OmpSs, we did a more extensive integration to exploit the benefits of having information from the runtime.
- Add new balancing algorithms.

DLB can be divided into three layers, as shown in Figure 5.3, the outer level of parallelism layer, the load balancing algorithms and the inner level of parallelism layer.

5.3.1 The outer level of parallelism layer

This layer is the entry point to the DLB library. This layer decouples the DLB library from the programming model in the outer level.

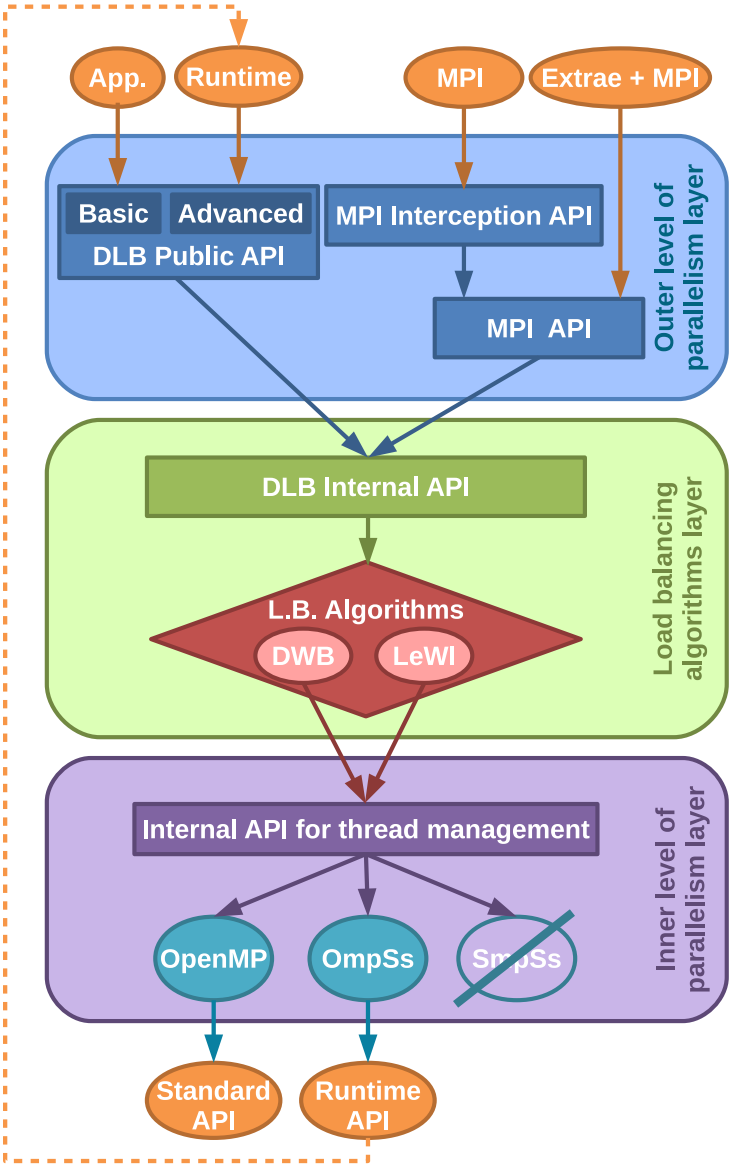


Figure 5.3: DLB framework divided in layers

As we can see in Figure 5.3 we have three different entry points to the DLB library depending on the kind of application.

MPI Application: For applications implemented using the communication library MPI. DLB will automatically intercept the MPI calls using the runtime interposition (PMPI). With this technique we do not need to recompile the application, just preloading the library it will intercept the MPI calls. We will explain this technique further in the following section 5.4.

MPI Application with Extrae tracing: For MPI applications that are being traced with the Extrae tracing library. Extrae uses the same interposition technique as DLB. Therefore, we need a special cooperation between the two libraries. A special Extrae library is available that will call the DLB API directly for MPI and also a special DLB library that does not include the interception symbols for MPI calls.

DLB Applications or Runtimes: For applications using the DLB API or runtimes with support for DLB. DLB offers a public API (we will see it in detail in section 5.5) this API can be used to give hints to DLB about the application behavior either from the application code itself or a programming model runtime. E.g., inform DLB that the process is entering a serial part of the code and will only need one thread.

This layer includes several APIs depending on the kind of application we are running. But all of them finally converge to the internal DLB API. The internal DLB API will communicate with the different load balancing algorithms.

5.3.2 The inner level of parallelism layer

This layer controls the malleability of the programming model and abstracts the DLB library from the programming model underneath. The inner level of parallelism layer will manage the threads for the different parallel runtimes supported based in the decisions taken by the load balancing algorithms.

In Figure 5.3 we can see how the layer for the inner level of parallelism can decouple DLB from the programming model below. The different load balancing algorithms will call the internal API to change the number of threads.

Internally DLB will detect the programming model used and call the corresponding API.

The current version of DLB supports OpenMP and OmpSs. SmpSs is no longer maintained because the SmpSs runtime is discontinued. But we will see some early results obtained with SmpSs in this thesis.

5.4 Runtime Interposition

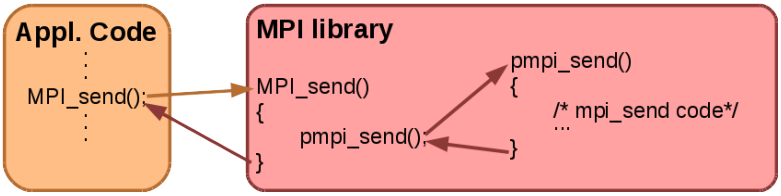
The DLB library exploits the technique of the runtime interposition of dynamically loaded libraries. This technique can be used thanks to the profiling mechanism MPI offers to instrument MPI applications. The MPI profiling mechanism consists of a different interface that is called before the real MPI call (PMPI calls). This mechanism was intended to be used by profiling tools, so it allows to detect when an MPI call starts and finishes.

In Figure 5.4 we can see with an example how the standard MPI profiling mechanism works. The Figure 5.4(a) shows the standard behavior of an MPI call, where an application calls an MPI function (*MPI_send*) and the function is executed from the MPI library that was used at link time.

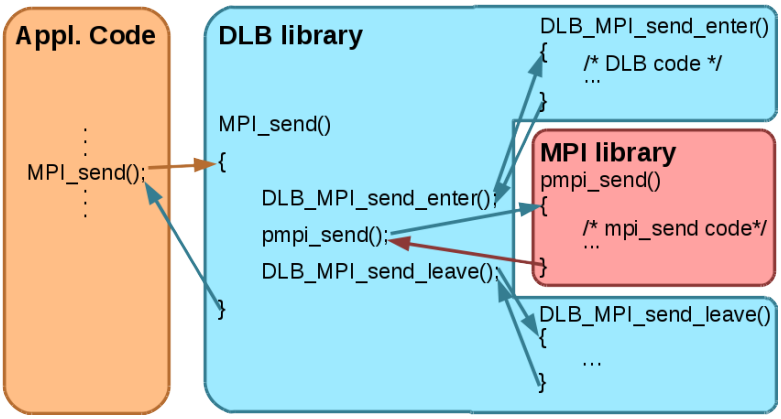
In Figure 5.4(b) we can see the sequence of calls that are executed when the runtime interposition is used. In the DLB library, there are defined functions with the same signature than the ones we want to intercept. When the application calls an MPI function the code executed is the one in the DLB library. It is then responsibility of the DLB library to call *pmpi_send()* that is the function of the MPI library that implements the functionality of the *send* function. To obtain this behavior it is necessary to load the DLB library with LD_PRELOAD when running the MPI application.

Finally, if we want to profile the application and use the DLB library we need a workaround because of the profiling mechanism and the DLB interception technique clash. In Figure 5.4(c) we can see the solution that consists of a special Extrae tracing library which will call the internal DLB API before and after each MPI call.

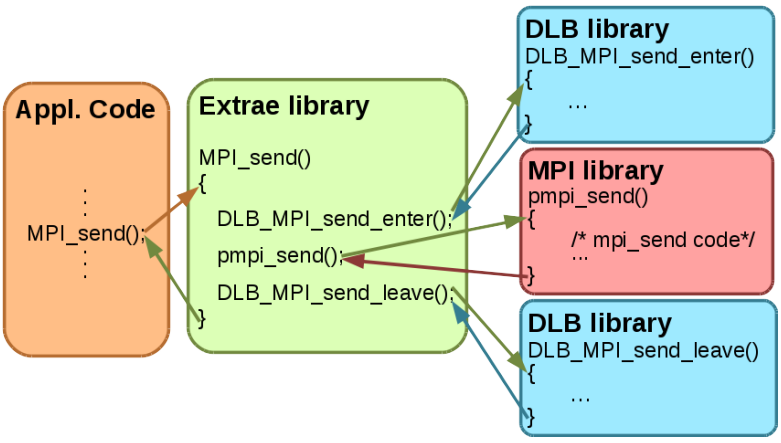
In this case, the runtime interposition is easy thanks to the profiling mechanism of MPI. But if we wanted to use the DLB library with another programming model that does not present this facility it is still possible to do it. The prerequisite for doing this is to have an application dynamically linked



(a) MPI application behavior



(b) MPI application intercepted by DLB behavior



(c) MPI application traced by Extrae and using DLB behavior

Figure 5.4: MPI profiling mechanism used by DLB and Extrae

with the programming model library. Then a binary interception should be done.

5.5 DLB public API

The DLB API can be divided into three parts:

MPI API: This is a specific API for MPI. As we can see in figure 5.3 we offer an MPI interface that will be called by Extrae if we are tracing the application or internally in the MPI intercept API. This set includes two functions for each MPI call, one that is called before the MPI call is performed, and another one that is called after the MPI call is done. For example the `MPI_Barrier` will have: `DLB_MPI_Barrier_enter` and `DLB_MPI_Barrier_leave`.

Basic set: The basic set is very simple, reduced and oriented to application developers. It is designed to be called from the application code to give hints about the application behavior or structure to DLB, so that, DLB can maximize the performance of the execution. The different functions of the API will be explained in detail in next. section(5.5.1).

Advanced set: The advanced set is oriented to programming model runtimes but can be used by applications also. It is designed to give more detailed information to DLB. E.g., the amount of parallel tasks ready, with this information DLB can take the corresponding decisions to maximize the utilization of resources. We have used the advanced API to integrate DLB with the Nanos++ runtime library, and it is ready to be used by other runtime libraries. The advanced functions will be explained in detail in section 5.5.2.

5.5.1 Basic set of DLB API

The first design of the DLB library was a dynamic library that would work completely transparent to the application. But after obtaining success with this approach, we saw that with a small effort from the application programmers' the performance could be improved significantly.

For this reason, we made public a DLB API for applications. This API is intended to give hints of the application to DLB. With these hints, DLB can make a more efficient use of resources.

- `void DLB_disable(void):`
  Will disable any DLB action. And reset the resources of the process to its default. While DLB is disabled there will not be any movement of threads for this process. Useful to limit parts of the code were DLB will not be helpful, by disabling DLB we avoid introducing any overhead.

- `void DLB_enable(void):`
  Will enable DLB. DLB is enabled by default when the execution starts. And if it was not previously disabled it will not have any effect.

- `void DLB_max_parallelism(int max_threads):`
  Will limit the maximum number of threads that can be used. Useful to mark parts of the code with a limited parallelism.

- `void DLB_single(void):`
  Will lend all the threads of the process except one. Useful to mark parts of the code that are serial. The remaining threads can be used by some other process. All the DLB functions will be disabled except lending the thread when entering or exiting an MPI call, DLB_parallel and DLB_enable.

- `void DLB_parallel(void):`
  Will claim the default threads and enable all the DLB functions. Useful when exiting a serial section of code.

We can summarize the behavior of these functions with the states graph shown in figure 5.5. We can consider three states for DLB (for each process) *Enabled*, *Disabled* and *Single*. *Enabled* would be the default state, where DLB will react to any API call.

The *Disabled* state will not allow any change in the number of threads (only a call to *DLB_enable* will have an effect). The number of threads of the process in a *Disabled* state will be the default.

The *Single* state will only react at *DLB_enable* or *DLB_parallel* API calls. The number of threads of the process in the *Single* state will be 1.

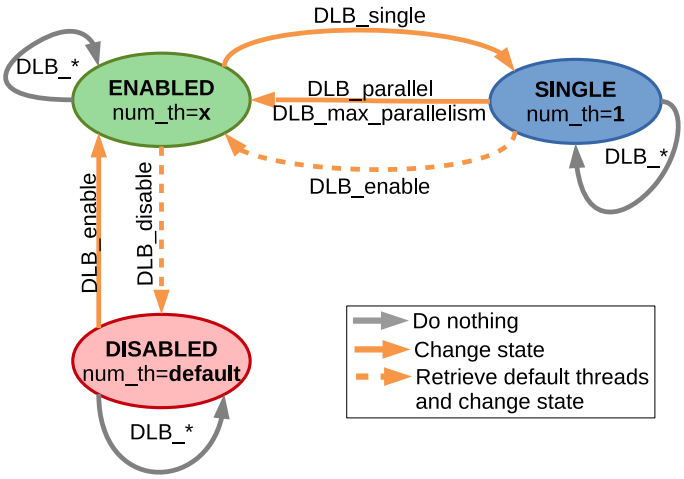


Figure 5.5: DLB states explained

5.5.2 Advanced set of DLB API

The advanced set of calls is designed to be used by runtimes, either in the outer level or the inner level of parallelism. But advanced users can also use them from applications.

- **To manage DLB:**
  - ⋆ `void DLB_Init(void):`
    Initialize the DLB library and all its internal data structures. Must be called once and only one by each process in the DLB system (if it is called more than once from the same process an error will be issued).
  - ⋆ `void DLB_Finalize(void):`
    Finalize the DLB library and clean up all its data structures. Must be called by each process before exiting the system.
- **To obtain resources from the system:**
  - ⋆ `void DLB_UpdateResources_max(int max_resources):`
    Check the state of the system to update your resources. You can

obtain more resources in case there are available CPUs. The maximum number of resources that you will get is `max_resources`. If no parameter is given the maximum resources available are given.

- ⋆ `void DLB_Retrieve(void):`
  Claim all your default resources previously lent.
- ⋆ `void DLB_ClaimCpus(int cpus):`
  Claim as many CPUs as the parameter *cpus* indicates. You can only claim your CPUs. Therefore if you are claiming more CPUs than the ones that you have lent, you will only obtain as many CPUs as you have lent.
- ⋆ `int DLB_CheckCpuAvailability(int cpu):`
  This function returns 1 if your owned CPU is available to be used, 0 otherwise. This function is only valid if the calling process is the owner of the CPU. Useful to know if a claimed CPU is ready to be used by its owner, and avoid oversubscription of threads to CPUs.

- **To lend resources to the system:**
  - ⋆ `void DLB_Lend(void):`
    Lend all your resources to the system.
  - ⋆ `int DLB_ReleaseCpu(int cpu):`
    Lend this CPU to the system. The return value is 1 if the operation was successful and 0 otherwise.

- **To return claimed resources to the system:**
  - ⋆ `void DLB_ReturnClaimedCpus(void):`
    Check if any of the resources you are using has been claimed by its owner and return it if necessary.
  - ⋆ `int DLB_ReturnClaimedCpu(int cpu):`
    Return this CPU to the system in case it was claimed by its owner. The return value would be 1 if the CPU were returned to its owner and 0 otherwise.

5.6 Technical requirements

There are several technical requirements of the DLB library, although we have already commented them in the corresponding section, we are going to summarize here all of them.

Programming models: The currently supported programming models or runtimes by DLB are the following:

- MPI + OpenMP
- MPI + OmpSs
- MPI + SmpSs (not maintained)
- Nanos++ (Multiple Nanos++ Applications, can be OpenMP or OmpSs model)

Shared Memory between processes: DLB can balance processes running on the same node and sharing memory. DLB is based on shared memory and needs shared memory between all the processes sharing resources.

Preload mechanism for MPI applications: When using MPI applications we need the preload mechanism (in general available in any Linux system). This is only necessary when using the MPI interception, it is not necessary when calling DLB with the public API from the application or a runtime.

***Parallel regions* in OpenMP:** In OpenMP applications we need parallelism to open and close (*parallel region*) to be able to change the number of threads. This is because the OpenMP programming model only allows to change the number of threads outside a *parallel region*. We also need to add a call to the DLB API to update the resources used before each *parallel*. This limitation is not present when using the Nanos++ runtime.

Non-busy waiting for MPI calls: When using MPI applications we need the MPI blocking calls not to be busy waiting. However, DLB offers a mode where one CPU is reserved to wait for the MPI blocking call and is not lent to other processes.

Chapter 6

Load Balancing Algorithms

The DLB library supports several load balancing algorithms, in this chapter we are going to explain in detail the two most important algorithms that are implemented in the current version of DLB.

LeWI is the second algorithm we will see and is the main contribution and core of this thesis. The LeWI algorithm is based on the idea of lending the CPUs of a process when it is blocked waiting on an MPI call.

The other algorithm that was implemented within DLB is the Dynamic Weight Balancing algorithm (DWB). DWB dynamically detects iterative regions, with the performance information of these regions it will decide the best distribution of threads among the MPI processes running on the same node. This algorithm is based on an earlier work presented by Duran et al., and the study of its limitations lead us to develop the new algorithm Lend When Idle (LeWI). DWB was used as an inspiration and motivation to develop LeWI.

In the following sections, we will see in detail how these algorithms work, and some performance results.

6.1 DWB: Dynamic Weight Balancing

6.1.1 Algorithm

The Dynamic Weight Balancing (DWB) algorithm is based on a work presented by Duran et al. [12]. Duran et al. applied the algorithm to OpenMP applications with nested parallelism. It was used to decide the number of threads to spawn in the inner loops. We will apply the same algorithm to MPI+OpenMP applications, and we will use the algorithm to redistribute the number of threads of the MPI processes running on the same node. With the new distribution of threads, we aim to obtain a more balanced execution.

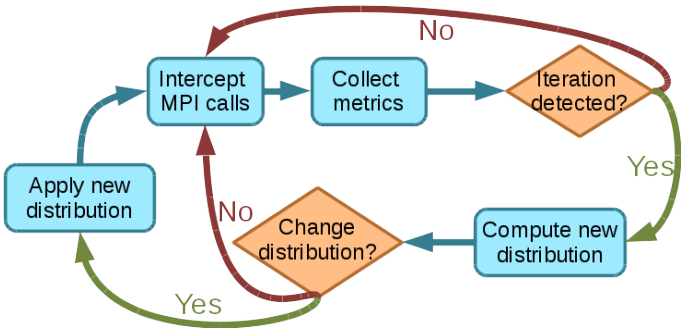


Figure 6.1: Dynamic weight Balancing (DWB) Algorithm

In Figure 6.1 we can see the DWB algorithm. The DLB library will intercept all the MPI calls and aggregate the amount of time each process spends in computation and waiting in MPI blocking calls.

DLB will detect iterative patterns in the application using the Dynamic Periodicity Detector (DPD) (Freitag et al. [69]). When an iteration is identified the DWB algorithm will compute the new optimal distribution of threads using the collected metrics. This distribution will be applied to the following parallel region.

To decide the new distribution, a computational weight is calculated for each MPI process as shown in Equation 6.1. Where $ComputeTime[i]$ is the amount of time process i has been computing (not in MPI communication). The $numCpus[i]$ is the number of CPUs that process i has been using during the last iteration detected.

$$weight[i] = \frac{ComputeTime[i] * numCpus[i] * 100}{totalComputeTime} - (weight\_1\_cpu) \quad (6.1)$$

$$totalComputeTime = \sum_{n=1}^{numMPIs} ComputeTime[n] * numCpus[n] \quad (6.2)$$

$$weight\_1\_cpu = \frac{100}{totalCpus} \quad (6.3)$$

$$NewNumCpus[i] = 1 + \frac{weight[i] * (totalCpus - numMPIs)}{\sum weight[n]} \quad (6.4)$$

The computational weight of each process is expressed as a percentage, to calculate the corresponding distribution we need to map this percentage to the number of CPUs for each process, the formula to calculate this is shown in Equation 6.4. Always committing to the condition that each process should have at least one thread. The obtained number of CPUs is rounded to the closest integer number.

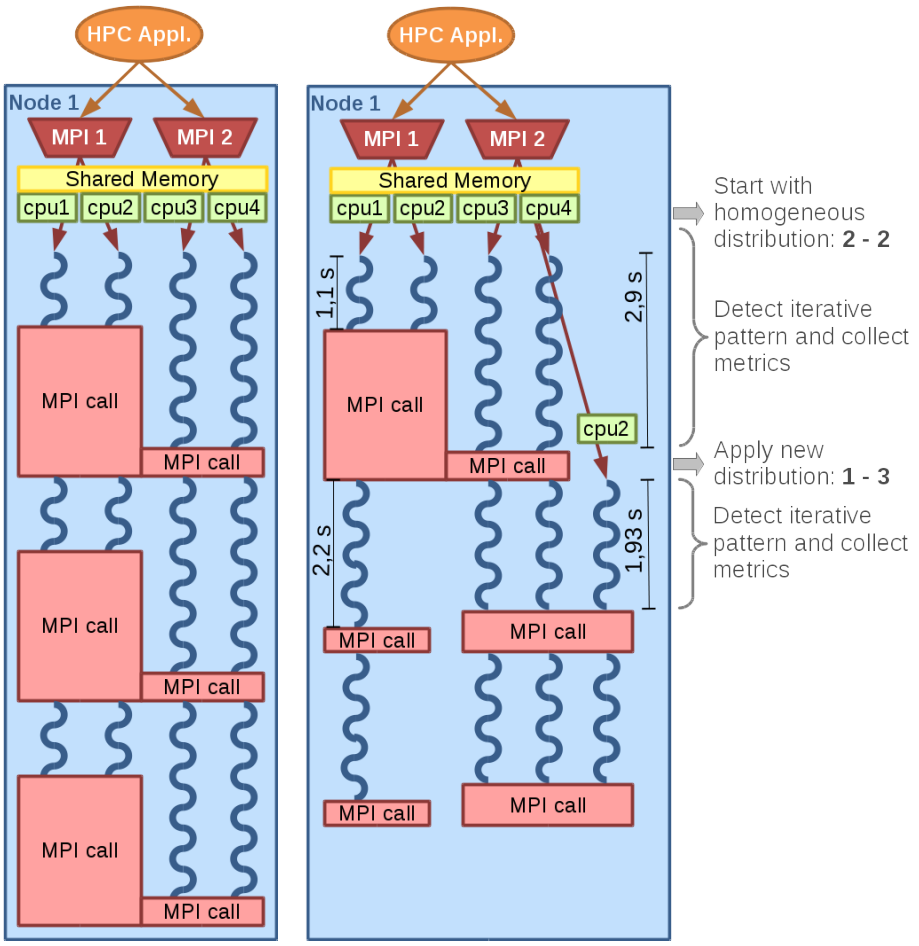
In Figure 6.2(b) we can see how the DWB algorithm can improve the performance of an imbalanced MPI application by redistributing the number of threads. In this example, an application running with 2 MPI processes is started with a homogeneous distribution of 2 threads per MPI process. During the first part of the execution, the DWB algorithm will collect metrics about the performance. When an iterative pattern is detected, and an iteration finishes the DWB algorithm will compute the optimal distribution of threads. In this case the collected metrics will be the following: $ComputeTime[1] = 1,1s$, $ComputeTime[2] = 2,9s$.

With this metrics and applying equations 6.1 through 6.4 we obtain the following:

$$totalComputeTime = (1,1 * 2) + (2,9 * 2) = 8$$

$$weight\_1\_cpu = \frac{100}{4} = 25$$

$$weight[1] = \frac{1,1s * 2 * 100}{8} - (25) = 2,5$$



(a) Imbalanced MPI application

(b) Imbalanced MPI application balanced with DWB

Figure 6.2: DWB Algorithm applied to an imbalanced application

$$weight[2] = \frac{2,9s * 2 * 100}{8} - (25) = 47,5$$

$$NewNumCpus[1] = 1 + \frac{2,5 * (4-2)}{50} = 1,1 \approx 1$$

$$NewNumCpus[2] = 1 + \frac{47,5 * (4-2)}{50} = 2,9 \approx 3$$

The new distribution is 1 thread for the MPI process 1 and 3 threads for the MPI process 2. Giving more resources to the MPI process 2 will allow a more balanced execution and better performance.

6.1.2 Study of DWB Limits

During the performance evaluation of the DWB algorithm, we detected two important limitations. On the one hand, the application must present an iterative pattern and imbalance that is constant during the different iterations. On the other hand, the performance obtained by the algorithm depends on whether the load balance of the iteration can be matched to a distribution of CPUS with the total number of CPUs in the node.

A small number of CPUs in the node offers a low granularity to solve load imbalance problems. When we are running on a node with 4 CPUs, moving a CPU from one process to another is equivalent to moving 25% of the resources. If the load imbalance is less that 25% we will not be able to improve the performance of this application.

To understand better the extent of this limitation we did an extensive study. The full results of this study can be found in Appendix A. Here we are going summarize the main results of this study.

All the results presented in this study are synthetic. They have been obtained by numerical computations, not actual executions. Supposing we have an application running with 2 MPI processes, we computed the maximum efficiency that it could get when using DWB algorithm. It is the maximum because we will suppose an ideal environment and parallelization.

We will use *Efficiency* as the metric to measure the performance of the DWB algorithm. Efficiency is the percentage of CPU time that the application is using for useful computations (not waiting due to load imbalance).

And it is computed with the following formula:

$$\text{Efficiency} = \frac{\sum_{x=1}^{numMPIs}(ComputeTime_x)}{Max_{x=1}^{numMPIs}(ComputeTime_x) * numCpus} * 100$$

We will see the efficiency obtained respect the difference in load between the two MPI processes. This difference is measured as a percentage that goes from 1% to 50%. This percentage represents the computational load of the first MPI process respect to the load of the whole application. As we will study applications with 2 MPI processes percentages going from 51% to 99% are redundant.

In Figure 6.3 we can see the efficiency that can be obtained with a different number of CPUs per node for the different load imbalances when using DWB, always with 2 MPI processes. The different load imbalances are represented in the X axis as the percentage of load of the first MPI process (MPI 1).

Figure 6.3(a) is the situation with 2 CPUs per node where DWB can not be applied (because each MPI process must have at least one CPU), in this case, the efficiency is only affected by the imbalance of the application. Note that a load of MPI 1 of 1% obtains an efficiency of 50%, this means that the application is using 50% of the resources efficiently. The other extreme is an MPI 1 load of 50%; that obtains an efficiency of 100%. This case is the baseline as this would be the efficiency obtained in all the situations when not using DWB.

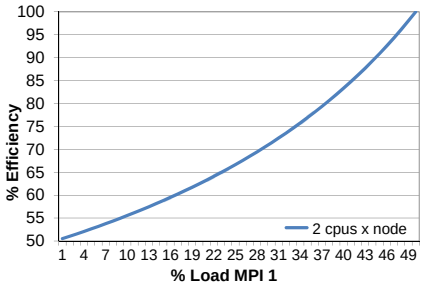
In Figure 6.3(b), the case with 4 CPUs per node, the efficiency has a new high peak, this peak corresponds to the perfect case for a distribution of 1-3 CPUs (a load of 25% for MPI 1). We can see how DWB can only improve the performance of the base case (Figure 6.3(a)) around this peak, for all the other loads DWB is not able to improve the efficiency of the execution.

We can see Figures 6.3(c) (8 CPUs per node) and 6.3(d) (16 CPUs per node) to evaluate the impact of the number of CPUs in the efficiency obtained with DWB. At least 16 CPUs per node are necessary to get an 85% or more efficiency with all the load imbalances.

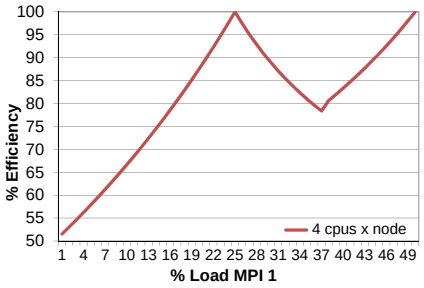
6.1.3 Conclusions for DWB

In this section we have seen how the DWB algorithm works and its implementation within the DLB library.

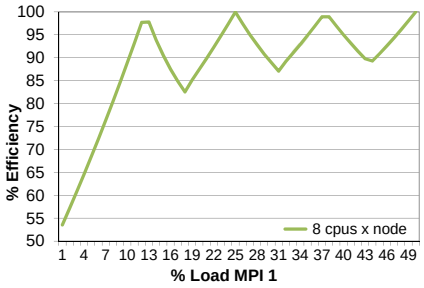
We have also seen that this algorithm presents two main limitations:



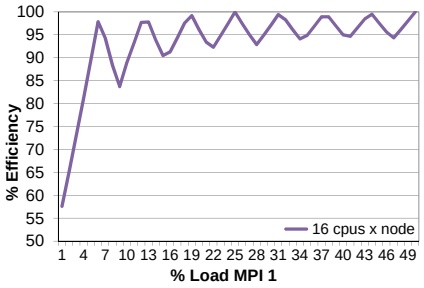
(a) Efficiency with 2 CPUs per node



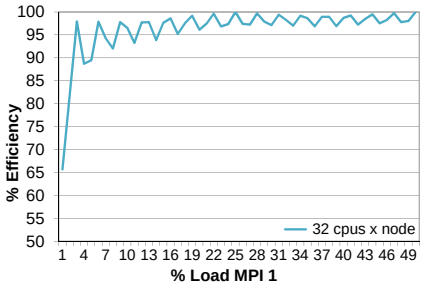
(b) Efficiency with 4 CPUs per node



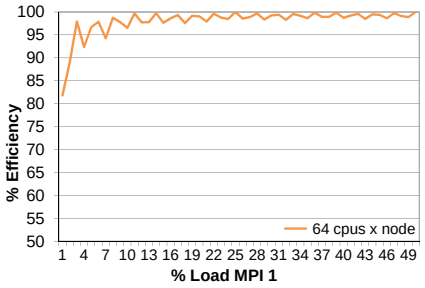
(c) Efficiency with 8 CPUs per node



(d) Efficiency with 16 CPUs per node



(e) Efficiency with 32 CPUs per node



(f) Efficiency with 64 CPUs per node

Figure 6.3: Efficiency that can be obtained from the execution of 2 MPI processes with different number of CPUs per node

- **Depends on the granularity of the resources:** In the synthetic study presented in the previous section we have seen that the performance that can be achieved with DWB algorithm is strongly dependent on the number of CPUs available in the node.
- **Needs an iterative pattern:** Needs the application to be iterative, and to present a consistent imbalance across the different iterations. Because the redistribution of CPUs is based on measurements of the previous iterations.

6.2 LeWI: Lend When Idle

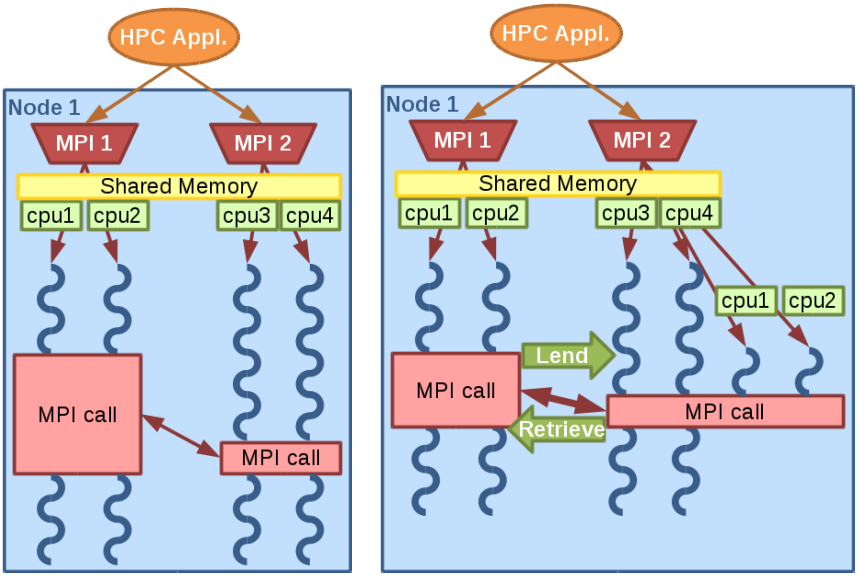
6.2.1 Algorithm

The new balancing algorithm we are presenting appears from the need to overcome the limitations discovered in the DWB algorithm.

The idea is based on the following observation: the imbalance between MPI processes implies that one (or more) process is blocked waiting for others. And while a process is blocked in an MPI call the CPUs it has assigned to run are idle.

The target of *LeWI* is to use the computational power of the idle CPUs to help the processes with a higher load finish faster.

The central idea of the *LeWI* algorithm is to lend the threads (CPUs) on the second level of parallelism of an MPI process while it is waiting on a blocking call to another MPI process running on the same node that is still doing computation. This will allow a load balancing at a more fine granularity.



(a) Imbalanced MPI application (b) MPI application balanced with LeWI

Figure 6.4: Example of LeWI algorithm

When the MPI process that has lent the CPUs gets out of the blocking call, it will recover its CPUs, and the process that was using them will be notified to stop using the lent threads.

In Figure 6.4 we can see an example about how the algorithm works. In the example, the application is running on a node with 4 CPUs. It starts two MPI processes on the same node, and each MPI process spawns 2 OpenMP threads. In Figure 6.4(a) we can see the behavior of an imbalanced application. On the other side Figure 6.4(b) shows the execution of the same application with the DLB library and the *LeWI* algorithm. We can see that when the MPI process 1 gets into the blocking call it will lend its two OpenMP threads to the MPI process 2. The second MPI process will use the newly acquired CPUs as soon as the programming model allows it. When the MPI process 1 gets out of the blocking call it retrieves its CPUs from the MPI process 2 and the execution continues with a CPU equipartition until another blocking call is met

The LeWI algorithm is entirely distributed and asynchronous. Each process will lend its resources when it is not using them to the system. And when a process can use more resources than its own will ask the system if there are idle resources. The first process that can use the resources will be the one to acquire them.

6.2.2 Study of LeWI Limits

The LeWI algorithm does not present any of the limitations detected in DWB. LeWI can solve imbalances in irregular applications because it does not depend on previous iterations. It can handle fine-grained load imbalances as it does not rely on the number of CPUs available in the node.

We are going to study the impact of the number of parallel regions between MPI calls in the performance of LeWI. This limitation is not inherent to the algorithm but derived from using the parallel programming model OpenMP. OpenMP only allows changing the number of threads outside a parallel region. LeWI relies on changing the number of threads to improve the performance and load balance, for this reason, the number of parallel regions can affect the performance of LeWI directly.

The second factor we are going to analyze is the granularity of the parallel regions. We want to see how LeWI can manage very fine-grained parallel loops when load balancing an application.

In this study, we are going to see the performance when using OpenMP. The conclusions of this study are the motivation to use other programming models more malleable, such as SmpSs or OmpSs, that we will present in the following chapter.

The data presented in this section is a summary of the conclusions obtained in the study of limits made for LeWI. The full documentation of this study can be found in Appendix B.

All the results shown in this section have been obtained using the synthetic benchmark PILS (see Section 3.1). The experiments have been executed in Marenostrum2 (4 CPUs per node) with 2 MPI processes.

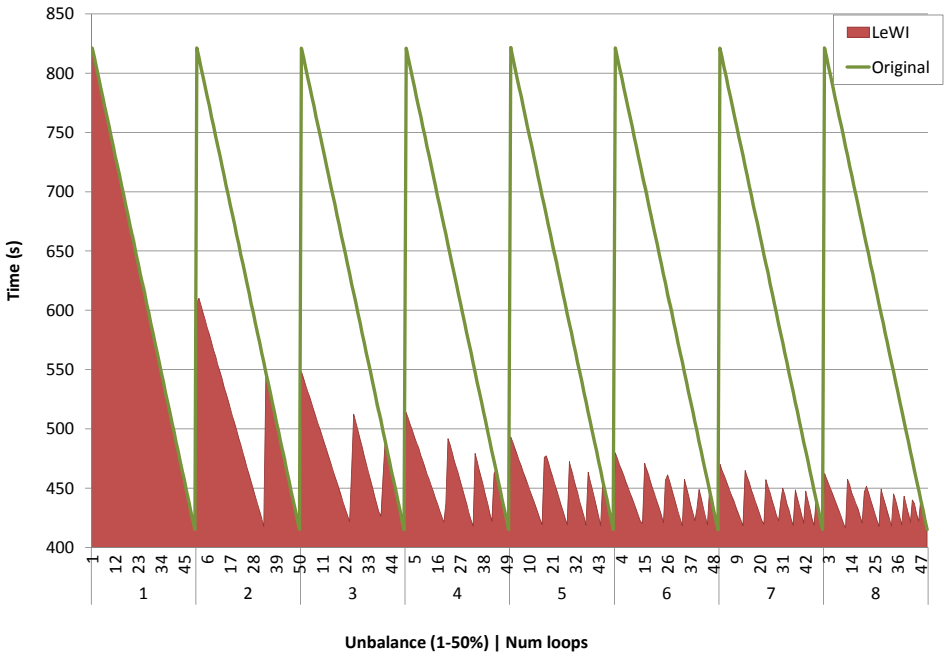


Figure 6.5: Execution time 1 to 8 loops of PILS with LeWI versus original

In Figure 6.5 we can see the impact of the number of parallel regions in the execution time. In the X axis, we can see the number of loops in which the load is divided (1 to 8). And on a smaller scale for each number of loops the % of imbalance present in the application (1% to 50%). In the Y axis is represented the execution time of the application in seconds. All the executions had the same amount of computation but distributed in a different number of parallel loops, and between the two MPI processes.

We can see how the execution time of the application without LeWI is not affected by the distribution of loops; we can see the same pattern repeated for the different number of loops. With LeWI, we can see that the case of just one parallel loop the performance is the same as the original, this is because there is no malleability in the application for LeWI to improve the load balance. For the other number of parallel loops, we can see how when using LeWI the execution time decreases as the number of loops increase. From this study, we conclude that at least 4 parallel regions must open and close between MPI blocking calls to allow an efficient load balance of the application.

	Number of parallel loops				
Class	**1**	**2**	**4**	**8**	**16**
A	332	166	83	41	21
B	663	332	166	83	41
C	1326	663	331	166	83
D	2653	1326	663	332	166
E	5305	2652	1326	663	331
F	10607	5303	2652	1326	663
G	21216	10610	5304	2651	1326

Table 6.1: Duration of each loop in ms of PILS depending on the class and the number of loops

To analyze the impact of the granularity of the computation in LeWI we have defined different classes of loads for PILS. A class represents a problem size, the load of each class is always the same independently of the number of loops. This means that the same class with more loops has a lower granularity.

In Table 6.1 we can find the duration in milliseconds of each parallel loop for each class when executed in sequential.

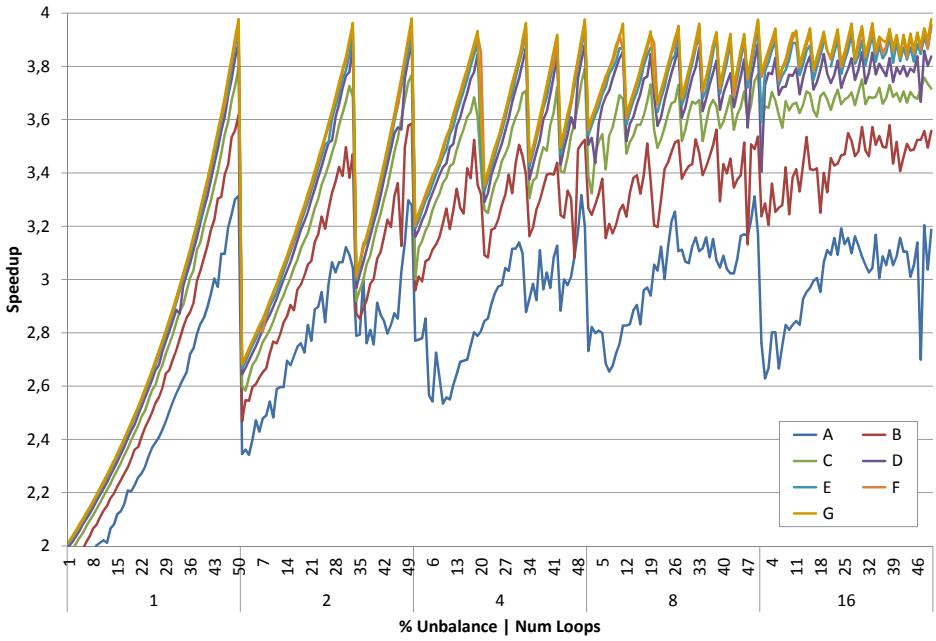


Figure 6.6: Speed up with LeWI classes A, B, C, D, E, F and G

In Figure 6.6 we can see the results obtained with the different classes run with LeWI. In the Y axis, we can see the speed up respect to the sequential execution of the same problem size. In the X axis, there is the load of the MPI process 1 (from 1% to 50%) and the number of parallel loops (from 1 to 8).

We observe that the performance is degraded with fine granularities. The threshold seems to be around 166ms of workload per parallel loop. This corresponds to class D with 16 loops or class C with 8 loops among others.

6.3 Load Balancing Policies Evaluation

In this section, we evaluate the performance of the two algorithms presented in the previous sections (DWB and LeWI) and the DLB library. For this evaluation, we will use different benchmarks and applications parallelized using MPI+OpenMP.

6.3.1 Environment

All the experiments have been executed in Marenostrum2. Marenostrum2 nodes have two processors with two cores each and 8Gb of shared memory. This means that we have nodes of 4 cores with shared memory.

We have used the MPICH library as the underlying MPI Runtime and the IBM XL C/C++ version 8.0 compiler without optimization. The operating system is a Linux 2.6.5-7.244-pseries64.

6.3.2 Methodology

We have executed the experiments with three different configurations of MPI processes and OpenMP threads per node. As we are running on nodes with 4 cores the possible combinations are the following:

- 1 MPI process per node with 4 OpenMP threads: This is the traditional configuration for hybrid applications. Our runtime can not improve the performance with this configuration because there is just one MPI per node. We show it just to compare the performance and to check that there is no overhead introduced.
- 2 MPI processes per node with 2 OpenMP threads.
- 4 MPI processes per node with 1 OpenMP thread: This configuration uses the second level of parallelism just to balance the outer one with the DLB library. In this case, the *DWB* algorithm is not able to improve the performance as the algorithm is limited to give at least one OpenMP thread per process.

In each experiment we are executing each of the three configurations with four versions:

- **ORIG:** The original application.
- **DWB:** The application with DLB and DWB algorithm.
- **LeWI:** The application with DLB and LeWI algorithm.
- **OS:** The original application with 4 threads OpenMP per MPI process. In this case, we overload the node and leave the responsibility of balancing to the operating system scheduler. With this version, we need to use a guided schedule for OpenMP (in all the other cases we are using a static schedule without chunk).

We have divided the experiments into two main types, running in a single node or running on several nodes. We executed 5 times each test and we show the average obtained.

We use the speed up to compare the performance of each experiment. The speed up is calculated as $\frac{serial\_time}{execution\_time}$ where the serial_time is the time that took the application to finish with one MPI process and 1 OpenMP thread; we use this time because in most of the cases we do not have a serial version of the application to run.

In all the charts the speed up is shown on the y axis. On the x axis are represented the different configurations (explained above) as the number of MPI processes per node and the number of OpenMP threads per MPI process at the beginning of the execution. The series labeled as *ORIG* correspond to the original application. The *DWB* and *LeWI* series are executions with DLB and the corresponding balancing algorithm. And the series marked *OS* represent the performance obtained when leaving the responsibility to balance to the Operating System Scheduler.

6.3.3 Running in a single node

As we have seen the DLB library is intended to balance the MPI processes running on the same node. Therefore our most significant contribution can be seen when running an application on a single node. In this section, we show the speed up obtained by some applications when running on a single node (4 cores).

Figure 6.7 shows the speed up obtained for the BT-MZ application. This application is very imbalanced but at the same time presents a very regular

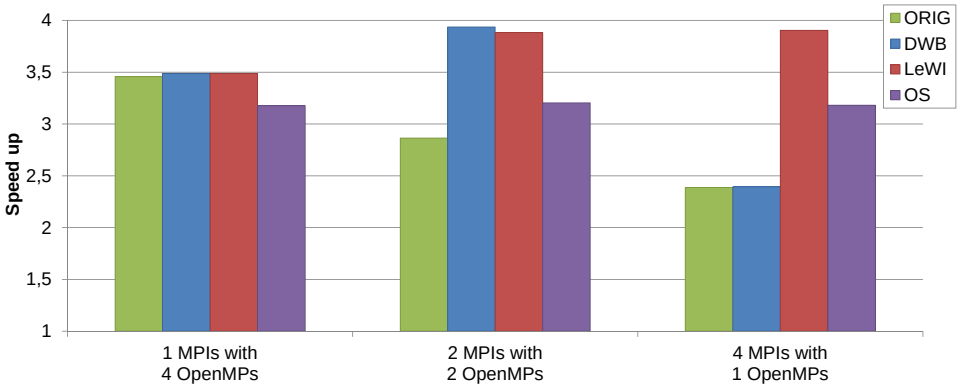


Figure 6.7: BT-MZ Class A in one node (4 cores)

imbalance that it is maintained during all it execution. We can see that the more MPI processes used the worst performance obtained with the original application. This is because the imbalance increases with the number of MPI processes.

When using DLB with the *LeWI* algorithm, the performance improves as we increase the number of MPIs per node. The best performance is obtained with 4 MPIs per node with the *LeWI* algorithm. It improves by 13% the speed up with respect to the original application with the traditional configuration of 1 MPI per node. Compared to the original application when running with 4 MPIs per node *LeWI* obtains a 64% of improvement.

In the BT-MZ application when running with two MPIs per node the performance of *DWB* is the same to the one obtained by *LeWI*. The reason is that the imbalance in this case corresponds to a perfect partition of the processors (1-3).

The SP-MZ performance is shown in Figure 6.8. As we said, SP-MZ is a well balanced application, and we can see that there is no significant overhead introduced by DLB when using neither DWB nor LeWI. On the other hand, the use of the OS schedule slows down the application when there is no load imbalance.

The speed up of the LUB application is shown in Figure 6.9. The imbalance of this application is very irregular because the most loaded MPI process changes each iteration. In this case, we can see that the *LeWI* algorithm im-

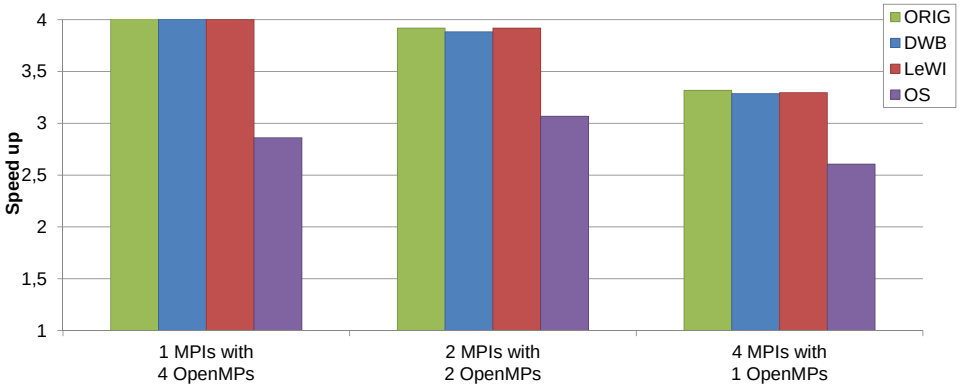


Figure 6.8: SP-MZ Class A in one node (4 cores)

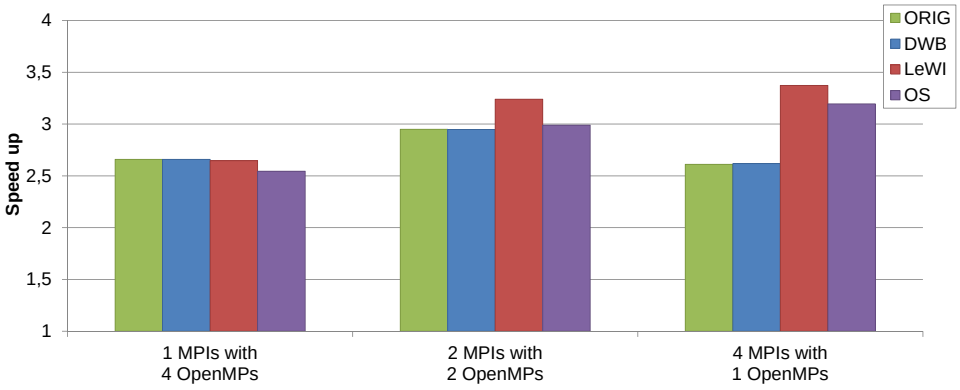


Figure 6.9: LUB in one node (4 cores)

proves the performance of the application. And again the best performance is obtained by *LeWI* running with 4 MPI processes per node. This configuration improves by 27% the speed up of the original application running with 1 MPI process per node. The *DWB* algorithm, on the other hand, is not able to improve the performance of the original application because one of the limitations of this algorithm is that it needs the same imbalance during successive iterations.

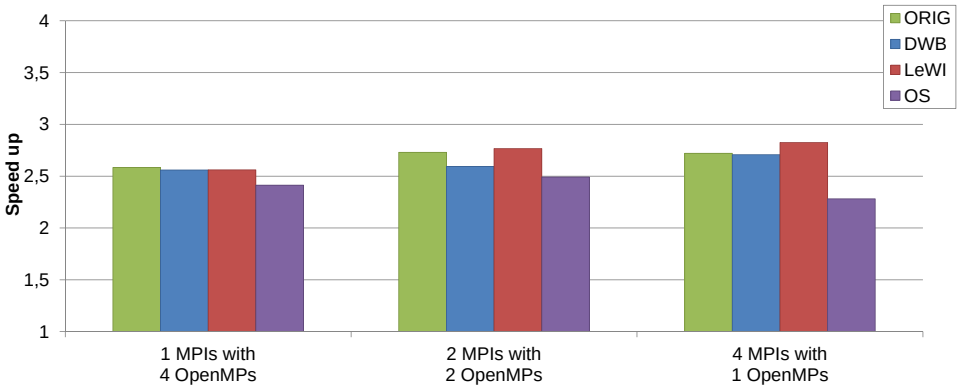


Figure 6.10: FLOWer in one node (4 cores)

We can see the performance results for the FLOWer application in Figure 6.10 when running with 1, 2 or 4 MPI processes. When running with 4 MPI processes, *LeWI* improves the performance by 9% with respect to the original application with 1 MPI process per node and a 4,5% respect the original application running with 4 MPI processes per node. Running with less than 8 MPI processes this application does not present a significant imbalance.

6.3.4 Running in several nodes

Although our library can only balance processes running on the same node, we have seen that we can improve the global performance of applications running on several nodes just balancing the processes within a node.

In Figure 6.11 and 6.12 we can see the speed up obtained with the BT-MZ and the SP-MZ applications when running on 2 nodes (8 cores). The results of *LeWI* are similar to the ones obtained when running on one node, but

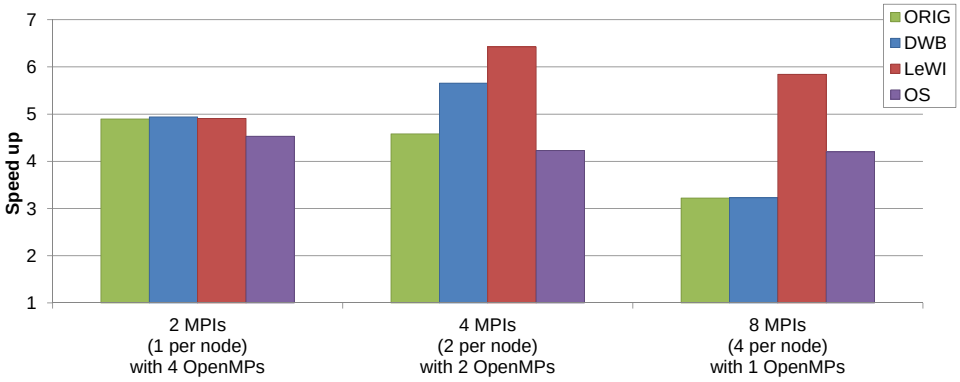


Figure 6.11: BT-MZ Class A in two nodes (8 cores)

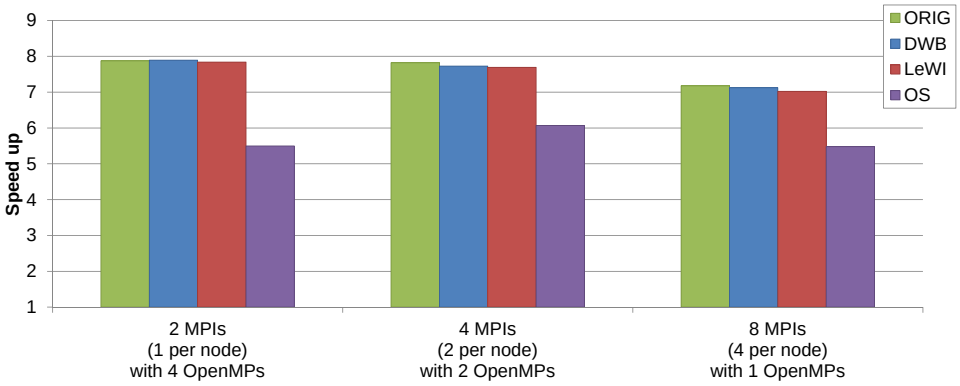


Figure 6.12: SP-MZ Class A in two nodes (8 cores)

less speed up is achieved because we can only balance the processes running on the same node. Even so, we improve the global performance of the BT-MZ original application while no overhead is introduced in the execution of the SP-MZ application. As we can not migrate processes across nodes in the current implementation, in the case of BT-MZ, to show the potential of DLB, we have mapped high loaded processes and light loaded ones in the same node.

In the case of BT-MZ, *LeWI* running with 2 MPIs per node increases the speed up by 31% respect to the original execution with 1 MPI per node. *DWB* improves the performance by 14% when running with 2 MPIs per node respect to the original application with 1 MPI per node. *DWB* performs worst because the level of imbalance does not correspond to any exact thread redistribution (this is due to the granularity limitation we explained in Section 6.1.2).

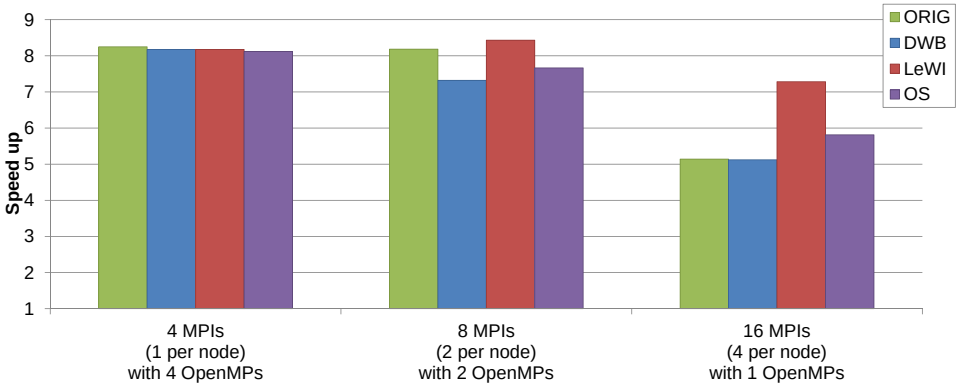


Figure 6.13: FLOWer in four nodes (16 cores)

We have executed the FLOWer application in 4 and 6 nodes, and we can see the results obtained in Figures 6.13 and 6.14. As we said, the imbalance of this application increases as the number of MPI processes increase. Therefore the performance drops as we increase the number of MPIs per node. We can see that the *LeWI* algorithm can still improve the performance of the application when running with 2 MPIs per node or 4 MPIs per node. The best execution of *LeWI* is a 2,5% better than the best execution of the original application.

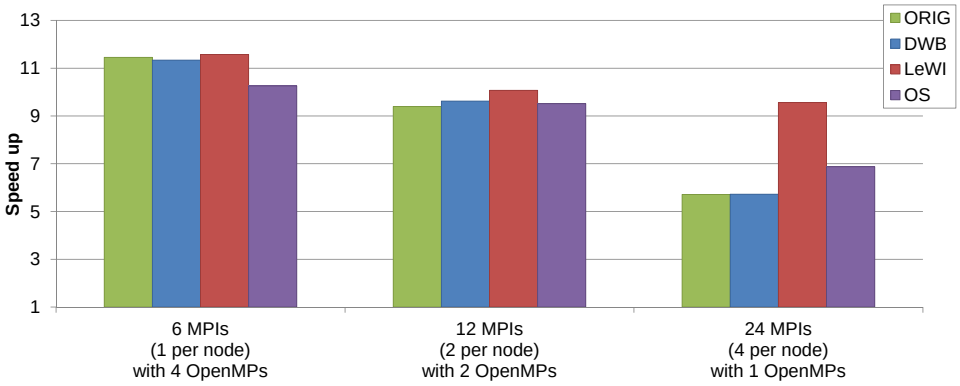


Figure 6.14: FLOWer in six nodes (24 cores)

It is interesting to see how in Figure 6.13 when running in four nodes the *LeWI* algorithm obtains a better performance when running with 2 MPIs per node than when running the original application with 1 MPI per node. This means that it is achieving a better performance than a less unbalanced execution.

6.3.5 Conclusions

In the performance evaluation section, we have shown that with DLB and the *LeWI* algorithm we can balance hybrid MPI+OpenMP applications. We can attack the load imbalance of regular and irregular applications. Moreover, we can use it with applications with an unknown pattern of imbalance (or even balanced applications), and it will improve its performance or at least will not introduce overhead.

We have seen that the *LeWI* algorithm outperforms in almost all the cases the *DWB* algorithm. Because *LeWI* does not suffer from the limitations that have the *DWB* and can balance applications with different levels of imbalance or very irregular.

And last, but not least we have been able to improve the global performance of applications running on several nodes just balancing the processes inside the nodes.

Chapter 7

Advanced Load Balancing With LeWI

In this chapter, we are going to analyze in detail different factors that can affect the load balancing of hybrid applications and especially when using DLB and LeWI. We will study the impact of the malleability of the programming model in the inner level of parallelism. The distribution of the MPI processes among the computation nodes and the binding of threads to cores.

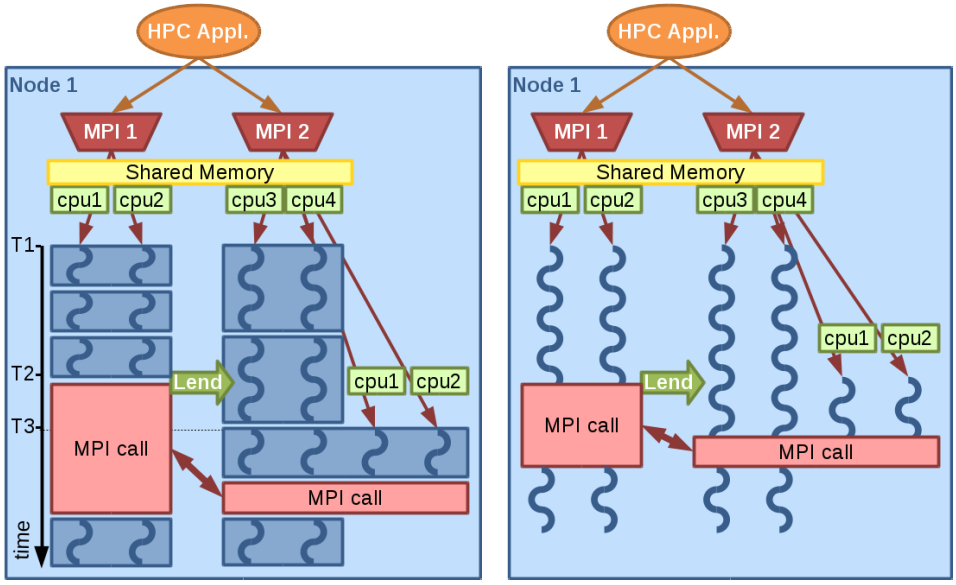
7.1 Impact of Programming Model Malleability

One of the strengths of the LeWI algorithm is its flexibility to solve different load imbalance problems. But at the same time, we have seen that malleability of the programming model that is being used affects the performance that can be obtained with LeWI.

In Section 6.2.2 we analyzed how the performance of LeWI was affected depending on the malleability of the application when using OpenMP (in Appendix B an extended study can be found).

But the limitation comes from the constraint in the malleability of the OpenMP programming model (OpenMP only allows to change the number of threads outside a parallel region). This means that when an MPI process lends its CPUs the MPI process that wants to use them is not able to do so

until reaching a new parallel region (i.e. the number of OpenMP threads can only be changed before spawning a parallel region). This is shown in Figure 7.1(a); when MPI 0 lends its CPUs to MPI 1, MPI process 1 is already executing its second parallel loop. Therefore, MPI 1 is not able to use the lent CPUs until the third loop starts.



(a) Balancing OpenMP application with LeWI

(b) Balancing SmpSs application with LeWI

Figure 7.1: Example of LeWI algorithm with OpenMP and SmpSs

This limitation makes the performance of the algorithm highly dependent on the number of parallel regions that the application presents between MPI blocking calls. I.e. if there is just one parallel loop between MPI blocking calls we can not change the number of threads, therefore, the application can not be balanced, and the performance can not be improved).

This limitation was not inherent to the algorithm but introduced by the programming model used (in this case OpenMP). To overcome this limitation we chose a shared memory programming model that allowed us to change the number of threads at any time, such as SMPSuperscalar (SmpSs). In

SMPSuperscalar there is not the concept of parallel region when there is work to do the threads are always running.

In Figure 7.1 we show the difference between using OpenMP or SmpSs when running with LeWI algorithm and how it can impact the performance. Figure 7.1(a) shows the limitation in OpenMP that can not start to use the threads until it reaches a new parallel region. In Figure 7.1(b) we can see how SmpSs can start to use the new threads as soon as they are available. This example shows us how the performance can be improved by a programming model that offers a higher malleability.

7.1.1 Performance Evaluation

In this section, we are going to evaluate the impact of the malleability of the programming model in the performance of LeWI. We are going to compare the two programming models OpenMP and SmpSs when using LeWI.

Environment and Methodolody

The experiments have been executed on Marenostrum2. Marenostrum2 has nodes of 4 cores with shared memory.

We have used the MPICH library as the underlying MPI runtime, and the operating system is a Linux 2.6.5-7.244-pseries64. The OpenMP compiler used is IBM XL version 10.1. SmpSs version used is 2.0.

In this section, we will use the speed up to compare the performance of each experiment. The speed up has been computed as the serial time divided by the parallel execution time. Each value is the average of 5 identical executions.

The serial time used to compute the speed up is the execution time of the MPI only version of the application executed with a single MPI process. We are using the MPI version with one MPI process and not the serial version of the application because we want to focus on the performance obtained by the inner programming model, in our case OpenMP or SmpSs. By using this baseline, we exclude from the computation of the speed up the overhead introduced by the MPI runtime.

We will see experiments executed with 2 and 4 MPIs processes per node. In the case of 4 MPI processes per node the second level of parallelism is only used for load balancing purposes.

In the following charts, we will be comparing four series for each application. Their meaning is as follows:

OpenMP: Execution of the MPI+OpenMP application without any load balancing.

OpenMP + LeWI: Execution of the MPI+OpenMP application with the DLB library and LeWI for load balancing.

SmpSs: Execution of the MPI+SmpSs application without any load balancing.

SmpSs + LeWI: Execution of the MPI+SmpSs application with DLB and LeWI for load balancing.

For this evaluation, we have used the synthetic benchmark PILS, LUB and BT-MZ (from NAS benchmarks).

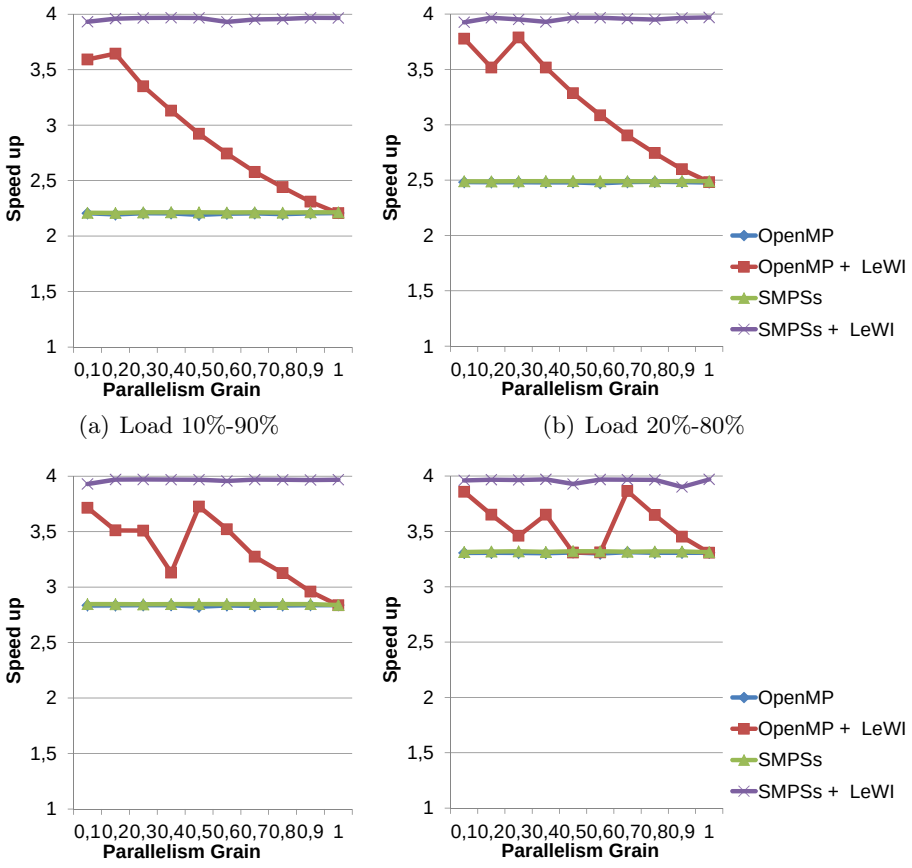
PILS

In the following experiments, we have executed PILS (synthetic benchmark explained in Section 3.1) with the parameter iterations always 1 and parallelism grain from 0.1 to 1. The parallelism grain represents the percentage of computation that can run in parallel between MPI synchronizations. It is computed as the reciprocal of the number of parallel regions between MPI blocking calls ($Par.Grain = \frac{1}{Par.Regions}$).

In Figure 7.2 we can see the speed up obtained for PILS when running with 2 MPI processes on the same node. We executed with four different workloads that go from a very imbalanced application (10-90) to an almost well balanced application (40-60). In the X axis we can see the parallelism grain and in the Y axis, the speed up obtained.

Comparing the series *SmpSs* and *OpenMP* we can see that when running the application without load balancing their performance is the same. Therefore the differences we will see when running with the LeWI load balancing algorithm do not come from the parallelization itself.

When comparing the performance of LeWI, we can see that the performance of OpenMP version depends on the parallelism grain (series *OpenMP + LeWI*). While the SmpSs version is not affected by the parallelism grain



(a) Load 10%-90%

(b) Load 20%-80%

(c) Load 30%-70%

(d) Load 40%-60%

Figure 7.2: PILS 2 MPIs per Node. Malleability impact in performance

nor the load balance of the execution (series SmpSs-LeWI) obtaining almost the maximum speed up for all the cases.

Figure 7.3 shows the performance of PILS when running with 4 MPI processes on the same node. In this case, we can see 6 different workloads with varying levels of imbalance between the MPI processes. Again the original executions without LeWI of the two models have the same performance (series *SmpSs* and *OpenMP*). But when running with LeWI, we can see how SmpSs performs much better. While SmpSs with LeWI can obtain an almost ideal speed up for all the configurations, OpenMP depends on the Parallelism Grain and hardly ever can reach the same performance as SmpSs.

LUB and BT-MZ

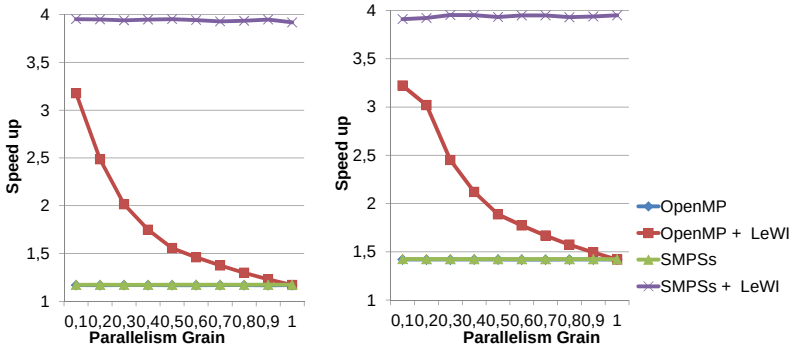
We have also used LUB and BT-MZ to evaluate the impact of the programming model malleability in the performance of LeWI. We have changed the parallelization of both applications to use SmpSs instead of OpenMP.

For the LUB application, we defined each one of the operations performed on the blocks (lu, fwd, bdiv, bmod) as tasks with the corresponding dependencies to ensure the correct execution. In the experiments, we have worked with a matrix of 5000 x 5000 elements with a block size of 300 x 300 elements.

For the BT-MZ NAS-benchmark, the code was modified to be parallelized with SmpSs but following the same structure as the OpenMP version. We will show the performance obtained when running class C.

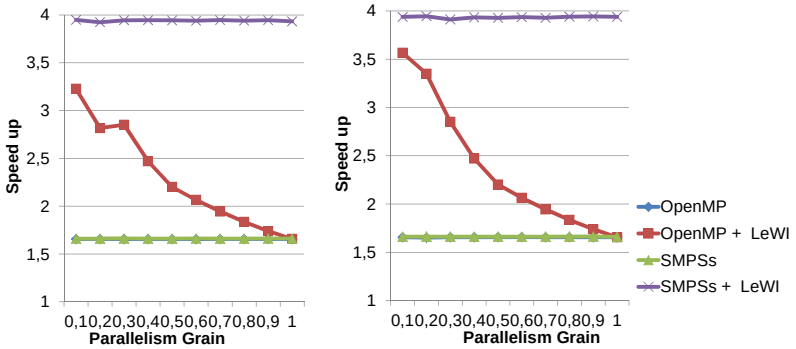
In Figure 7.4 we can see the performance obtained when running LUB in 1, 2 and 4 nodes of Marenostrum2 (4 cores per node). We have executed with 1, 2 or 4 MPI processes on the same node. The execution with 1 MPI process per node is shown just for reference as LeWI can not be applied.

We can see how the versions without load balancing (*OpenMP* and *SmpSs*) have a moderately different performance, especially when running one MPI process per node with 4 threads (20% to 24% better SmpSs than OpenMP). When running 2 MPI processes per node the improvement of SmpSs respect OpenMP is between 7% and 10%. This is because the parallelization of the algorithm with tasks and dependencies is more efficient than OpenMP because it can exploit better the parallelism. But when running with 4 MPI processes per node and just one thread per process, the second level of parallelism (OpenMP or SmpSs) is not used, and their performance is the same.



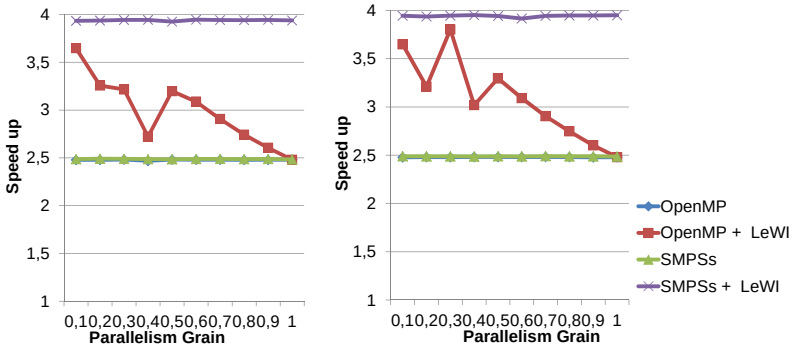
(a) Load 5%-5%-5%-10%

(b) Load 10%-10%-10%-70%



(c) Load 10%-15%-15%-60%

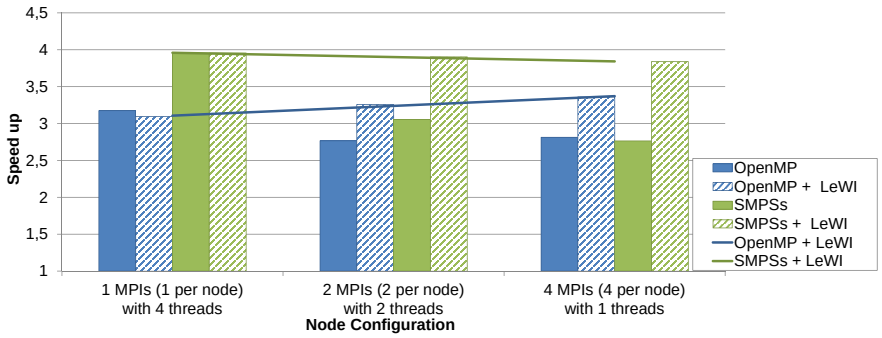
(d) Load 5%-17%-18%-60%



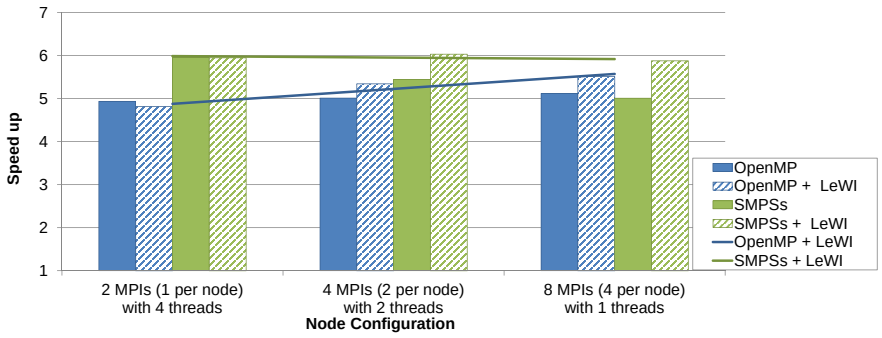
(e) Load 10%-20%-30%-40%

(f) Load 10%-25%-25%-40%

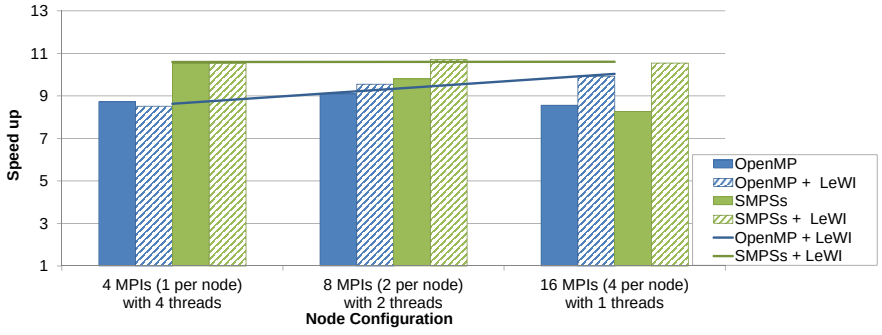
Figure 7.3: PILS 4 MPIs per Node. Malleability impact in performance



(a) LUB in 1 node (4 CPUs)



(b) LUB in 2 nodes (8 CPUs)



(c) LUB in 4 nodes (16 CPUs)

Figure 7.4: LUB in Marenostrum2. Malleability impact in LeWI performance

The execution with LeWI algorithm always improves the execution without load balancing (except when running just one MPI process per node that it does not degrade it). If we compare the performance of LeWI when using OpenMP or SmpSs, we can see that the performance is always better when using SmpSs and LeWI. In one node the SmpSs with LeWI is 19% and 14% better than OpenMP with LeWI, with 2 and 4 MPIs per node respectively. When running in 2 and 4 nodes the performance of SmpSs with LeWI is 12% better than OpenMP 2 MPIs per node and 6% better with 4 MPIs per node.

In Figure 7.5 we can see the performance obtained with BT-MZ when running in 1, 2 and 4 nodes of Marenostrum2. We have also executed with 1, 2 and 4 MPI processes per node and 4, 2 and 1 thread per process respectively. When comparing the executions without load balancing, we can see that SmpSs performs slightly better than OpenMP, around 7% better in all the cases.

Looking at the executions with LeWI we see that in all the cases it improves the performance of the original application. But the best performance is always obtained by the SmpSs version with LeWI. The improvements of LeWI with the SmpSs version respect the execution with LeWI, and the OpenMP version is around 12%-14%.

7.1.2 Extensive Performance Evaluation and Modeling Applications Characteristics that Limit LeWI Performance

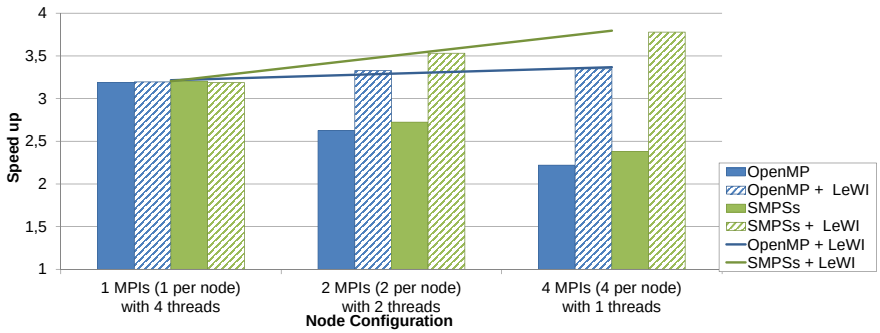
In this section we are going to use the following two metrics:

- **Efficiency:** Percentage of the CPU time consumed that is running *pure application code*. We understand by *pure application code* the code inside the application, not including runtime overheads (OpenMP or SmpSs runtime code) nor MPI calls.

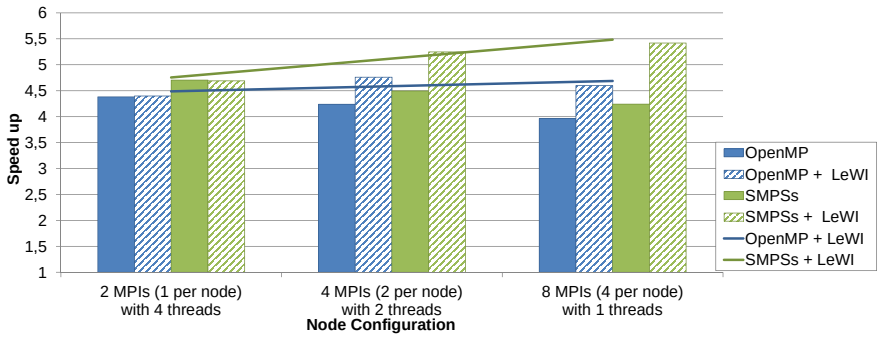
  $Efficiency = \frac{useful\_cpu\_time}{elapsed\_time * CPUs} * 100$

  useful_cpu_time = cpu_time - (MPI_time + OpenMP/SmpSs_time + DLB_time)

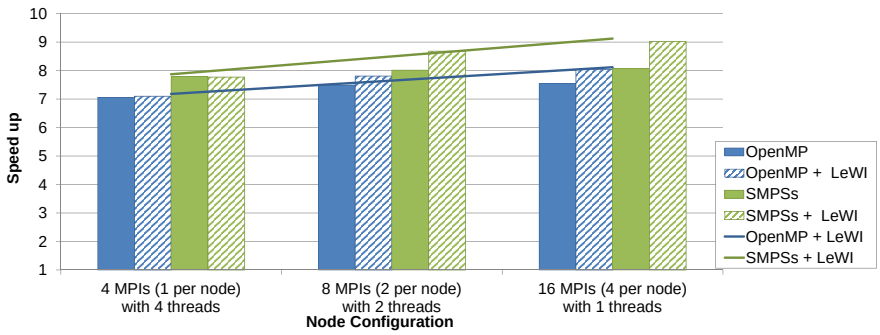
- **Cpus used:** Number of CPUs running at the same time pure application code.



(a) BT-MZ class C in 1 nodes (4 CPUs)



(b) BT-MZ class C in 2 nodes (8 CPUs)

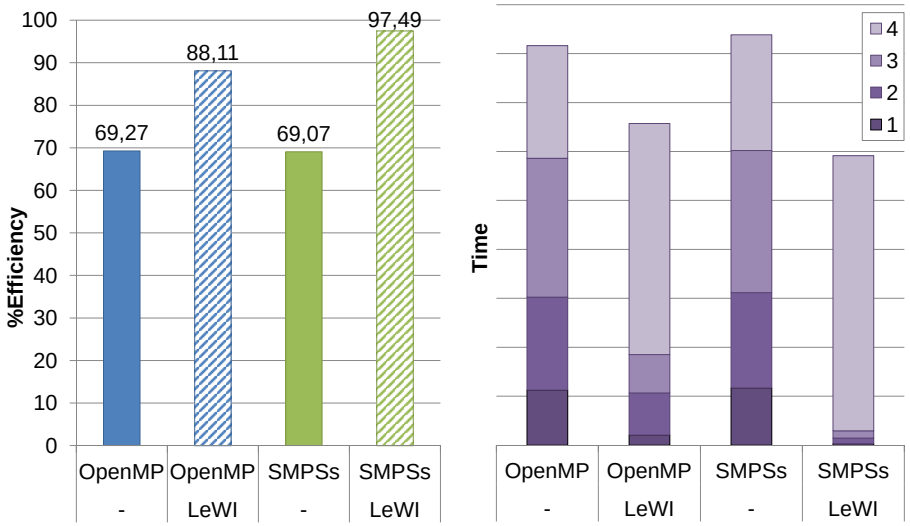


(c) BT-MZ class C in 4 nodes (16 CPUs)

Figure 7.5: BT-MZ in Marenostrum2. Malleability impact in LeWI performance

Speed up evaluates how well the application is running compared to the baseline version (usually sequential version). Efficiency (and CPUs used) evaluates how close (or far) is the utilization of resources from the desired one (100%).

The *Efficiency* and *Cpus used* have been computed from a trace of an execution. The trace has been obtained using the Extrae library [32] and analyzed with Paraver [33].



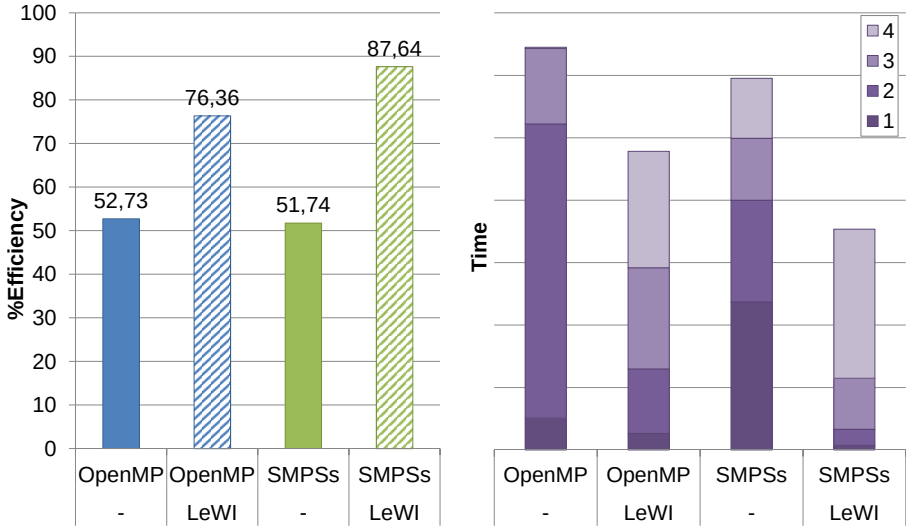
(a) Efficiency (b) Used CPUs

Figure 7.6: LUB Efficiency and CPUs used

Figure 7.6 shows two charts obtained from executions of LUB in one node with 4 MPI processes. Figure 7.6(a) shows the Efficiency of the execution (percentage of time that the CPUs were used to do useful work). We can see that the efficiency obtained by the application without load balancing is around 70% in both programming models (OpenMP and SmpSs). Also, the distribution of time that the application is using the CPUS (Figure 7.6(b)) is similar for both executions.

When using LeWI the efficiency obtained by the OpenMP version is a 10% lower than the SmpSs. This difference can be seen in the number of CPUs used. When running with SmpSs and LeWI almost 95% of the time the

application uses the 4 CPUs available while in the execution with OpenMP and LeWI, the four CPUs are used 71% of the time. The efficiency obtained by LUB with LeWI is close to the ideal.



(a) Efficiency (b) Used CPUs

Figure 7.7: BT-MZ Efficiency and CPUs used, class A in 1 node with 4 MPI processes

In Figure 7.7 we can see two charts both obtained from real executions of BT-MZ in one node, with 4 MPI processes and class A. Figure 7.7(a) shows the efficiency achieved with every combination of programming models and load balancing. In Figure 7.7(b), we see the execution time of each combination and amount of time that was running with the different number of CPUs (number of concurrent threads).

We can see that the efficiency obtained by BT-MZ is lower than the one achieved by LUB. When running the application without load balancing the efficiency obtained is around 50% (the application is wasting a half of the computational resources it is given to run). Looking at Figure 7.7(b) we can see that 73% of the time the BT-MZ application with OpenMP is using just 2 CPUs. In the case of SmpSs the distribution of cups used is different, a

40% of the time the application runs with just one CPU and 27% of the time it runs with 2 CPUs.

The execution of BT-MZ with LeWI improve the efficiency in both programming models but not at the same level. The OpenMP version with LeWI achieves a 76% of efficiency while the SmpSs version with LeWI can reach an 87% of efficiency. This difference can also be seen in the used CPUs, although LeWI makes a better use of the CPUs in both cases, only in the SmpSs version, it can obtain a 66% of the time running with the 4 CPUs.

We have seen that LeWI improves the efficient use of resources, but not always get an efficiency close to the ideal. To evaluate if the efficiency loss is bounded to the malleability of the application we will analyze two characteristics of the application evaluated:

- Parallelism Grain in OpenMP applications: In OpenMP applications, parallelism grain indicates the malleability in the number of threads that an application allows. The more parallelism grain, the fewer malleability.

- Task duration in SmpSs applications: The task duration in SmpSs applications can limit the malleability of the application, coarse grain tasks imply less malleability.

In this section, we are going to quantify these characteristics in the applications and reproduce them in PILS (the synthetic benchmark), to finally, confirm its impact in the performance.

Parallelism Grain impact in OpenMP Applications

Parallelism grain represents the percentage of computation that is run in parallel between MPI synchronizations. It is computed as the reciprocal of the number of parallel regions between MPI blocking calls ($Par.Grain = \frac{1}{Par.Regions}$). As an example, a parallelism grain of 0.25 corresponds to 4 parallel regions between MPI calls and a parallelism grain of 1 represents a single parallel region between MPI synchronizations.

We analyzed the parallelism grain in LUB and BT-MZ applications and presented it in Figure 7.8 (This data was obtained from Paraver trace of a real execution). In the X axis, we can see the parallelism grain that goes

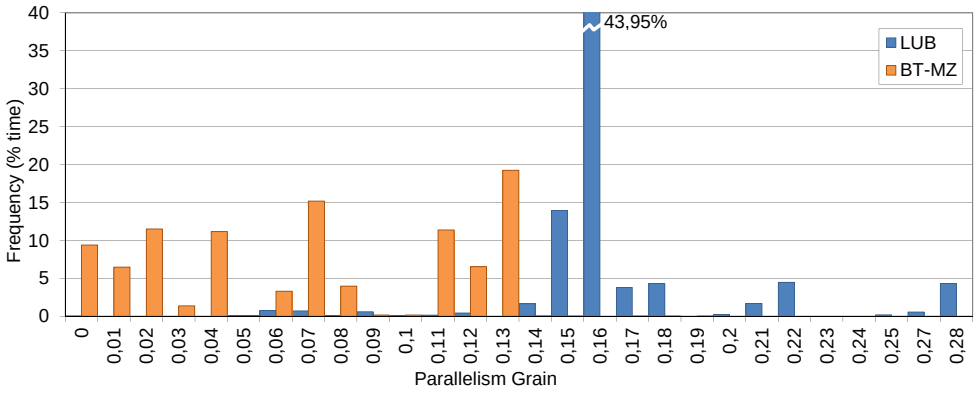


Figure 7.8: Parallelism grain of LUB and BT-MZ

from 0 to 0.28 and in the Y axis the percentage of time that this parallelism grain appears during the execution of the application (frequency).

In the case of LUB, we can see that most of the parallel time it has a parallelism grain of 0.16 and in general, it goes from 0.14 to 0.28. BT-MZ presents a lower parallelism grain that is distributed between 0.01 and 0.13.

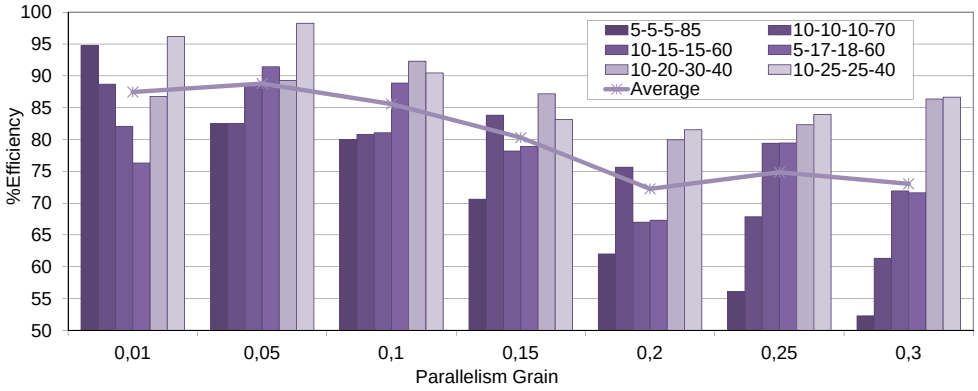


Figure 7.9: PILS efficiency with LeWI depending on parallelism grain

In Figure 7.9 we show the Efficiency obtained with PILS when using LeWI with a parallelism grain similar to the ones observed in the applications. The different series presented show different load distributions between MPI

processes and the average between them. We can see that the efficiency obtained by PILS still depends a lot on the load distribution between MPI processes although the average efficiency decreases as the parallelism grain increases.

In the case of LUB, we can simplify that the parallelism grain is around 0.15 and if we remember the efficiency obtained with LUB with LeWI it was 88%. We can see that, for some load distributions, this efficiency corresponds with the one obtained with PILS. We cannot match LUB with any load distribution because this application presents dynamic load distribution during the execution.

The parallelism grain in BT is distributed between less than 0.01 and 0.13; this is because there are different regions of code that present different parallelism grain inside the application. Therefore, it will be harder to match with the PILS results. On the other hand, BT-MZ has a quite fixed load distribution between MPI processes that is stable for the whole execution of the application. The approximate load distribution, when running with 4 MPI processes, is 9-16-28-47 respectively which would be between the 5-17-18-60 and 10-20-30-40 distributions of PILS. The efficiency obtained by BT in the previous section was 76% which would be an efficiency close to the average one achieved with 5-17-18-60 distribution.

Task duration impact in SmpSs Applications

The task duration in SmpSs is analogous to the parallelism grain in OpenMP applications. We need to find the trade off between tasks big enough not to introduce much overhead and sufficiently small to have enough flexibility to load balance the execution.

Figure 7.10 shows the duration of tasks in microseconds of LUB and BT-MZ (in 1 node with 4 MPIs for both applications). In the X axis, we show the task duration in microseconds (be aware that the scale is not contiguous to show relevant values) each point of the scale represents a range starting with the labeled number (i.e. the label 0 corresponds to task duration between 0 and 100 microseconds). In the Y axis, we can see the percentage of time represented by the tasks with this duration.

The duration of BT-MZ tasks varies from less than 100 microseconds to 1000 microsecond. LUB has a more coarse grain task duration between 222000 and 223000 microseconds.

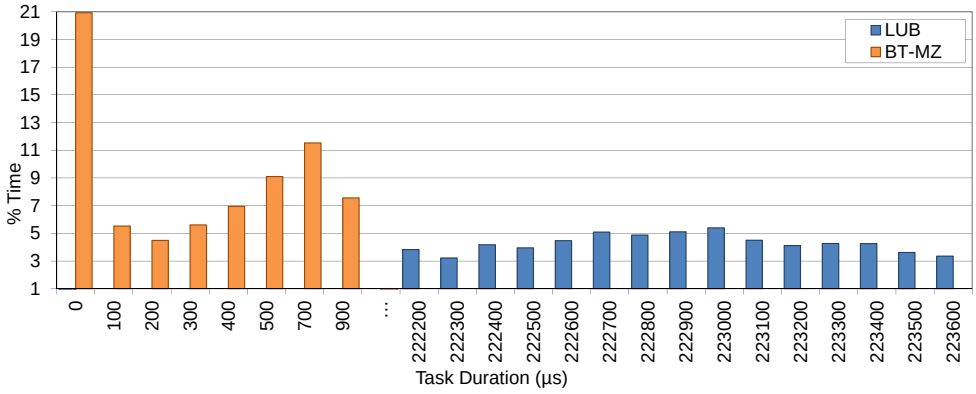


Figure 7.10: Task duration of LUB and BT-MZ

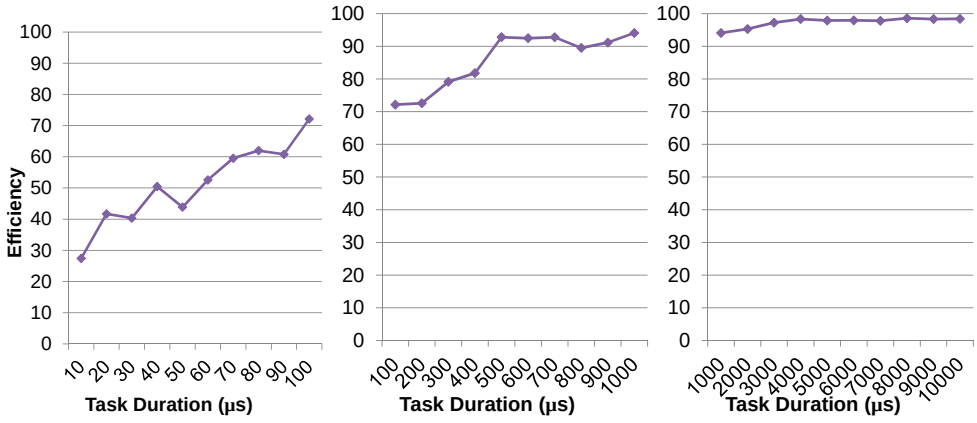


Figure 7.11: PILS efficiency depending on the task duration

We can compare the efficiency obtained with PILS when running with similar task durations in Figure 7.11. We present the results in three different charts for clarity. In the X axis, we can see the task duration in microseconds (from 0.1 to 100 microseconds in the first one, from 100 to 1000 microseconds in the second one and from 1000 to 10000 microseconds the third). The Y axis shows the efficiency obtained. We can see that the efficiency increases with the task duration and the optimum are reached with task duration of around 4000 microseconds getting an efficiency of 98%.

Now we will compare the efficiency obtained by BT-MZ with the results obtained with PILS. The duration of BT-MZ tasks was between 100 microseconds and 1000 microseconds, and the efficiency achieved was 87%. This efficiency corresponds to the average efficiency obtained by PILS with tasks between 100 and 900 microseconds. This means that most of the efficiency lost by BT-MZ is due to the granularity of its tasks.

In the case of LUB, the efficiency shown in the previous section for the SmpSs version when running with LeWI was 97%, and the duration of its tasks are around 222000 microseconds, which is the maximum obtained by PILS for tasks of 4000 microseconds or more. We can say that the SmpSs version of LUB reaches the maximum efficiency possible when running with LeWI.

7.1.3 Conclusions of the Impact of Programming Model Malleability

We have seen that LeWI relies on the malleability of the programming mode and the more malleable it is, the better performance LeWI will obtain.

From the evaluation performed we can conclude that the malleability of the programming model is crucial. But does not mean that LeWI can not speed up applications with OpenMP, but in those cases, the performance will also be affected by the malleability of the application and its load imbalance.

Regarding the malleability in the application, for MPI+OpenMP applications we have seen that the number of parallel loops can affect the performance of LeWI and that it is highly correlated with the amount of load imbalance. Our experiments showed that in general to obtain an efficiency between 80% and 85% we need a parallelism grain of 0.15 (equivalent to 6 parallel loops between MPI calls). We have shown that the original efficiency

of LUB was 69%, it was increased to 88% using LeWI. Our recommendation is to have a parallel grain below 0.1 even when that can seem to be self-defeating, we have demonstrated that when using LeWI the application will obtain a better performance.

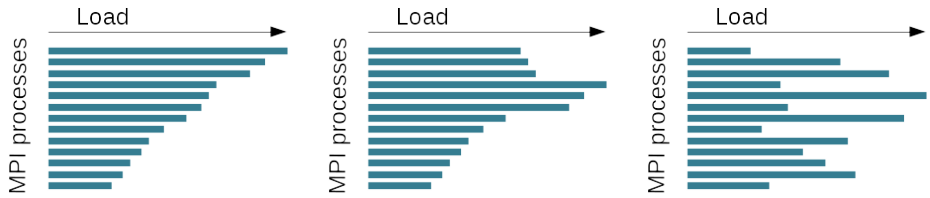
For MPI+SmpSs applications, the trade-off is in the granularity of the tasks. In this case, we have seen that we need tasks of at least 500 microseconds of duration to obtain an efficiency above the 80%. The programming model can handle tasks of less than 100 microseconds and still achieve a good efficiency, but it would be recommendable to have tasks of more than 500 microseconds of duration.

7.2 MPIs distribution among nodes

When analyzing the efficiency obtained by applications running on several nodes with DLB and LeWI, we identified the distribution of MPI processes among nodes as a factor that can affect the performance. As DLB can only load balance processes running in the same shared memory node, the performance it can achieve depends on the load imbalance between nodes. The better load balance between nodes the better performance LeWI will obtain.

The distribution of MPI processes between nodes is not a parallelization decision but an execution decision. This makes it easy to change and as we will see it has a substantial impact on the efficient use of the resources.

We can classify the load imbalance of applications into three categories:



(a) Triangular (b) Clustered (c) Random

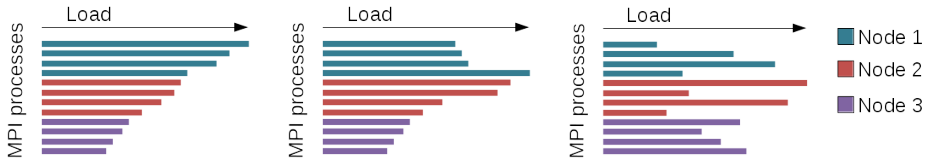
Figure 7.12: Classification of load distribution

Triangular: The load of the MPI processes goes from more loaded processes to less loaded ones (or vice-versa), Figure 7.12(a).

Clustered: The most loaded processes are consecutive, Figure 7.12(b). There can be one or more clusters of highly loaded MPI processes.

Random: There is no pattern in the load distribution among MPI processes, Figure 7.12(c)

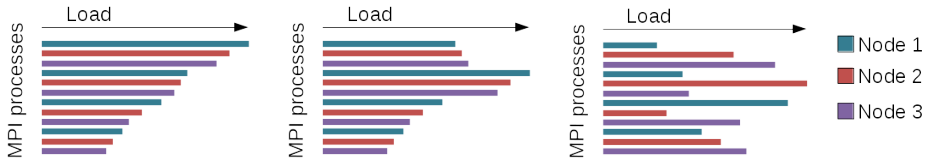
By default applications will be executed with consecutive assignment of MPIs, moreover, the communication pattern of some applications is designed to work with this kind of distribution. An application running on 3 nodes with 12 MPI processes will run MPI processes 0, 1, 2 and 3 in node 1; MPI processes 4, 5, 6, and 7 in node 2 and MPI processes 8, 9, 10 and 11 in node 3.



(a) Triangular (b) Clustered (c) Random

Figure 7.13: Distribution of MPI processes with a consecutive placement

But this kind of distribution will assign the more loaded processes together in the same node if the load distribution of the application is triangular or clustered, as we can see in Figures 7.13(a) and 7.13(b).



(a) Triangular (b) Clustered (c) Random

Figure 7.14: Distribution of MPI processes with a round robin placement

In almost all HPC systems, we can choose a different distribution or even decide one by hand. An alternative distribution available in almost all the environments is *Round Robin*, A Round Robin distribution assigns the MPI processes in a cyclic way among the nodes. An application running on 3 nodes with 12 MPI processes, will run MPI processes 0, 3, 6 and 9 in node 1 and MPI processes 1, 4, 7 and 10 in node 2 an MPI process 2, 5, 8 and 11 in node 3. As we can see in Figure 7.14 applications with a triangular or clustered load distribution will usually obtain a more balanced distribution of loads among nodes with a Round Robin placement.

7.2.1 Performance Evaluation

In this section, we are going to evaluate the impact of the malleability of the programming model in the performance of LeWI. We are going to compare the two programming models OpenMP and SmpSs when using LeWI.

Environment and Methodolody

The experiments have been executed on Marenostrum2 and Marenostrum3. Marenostrum2 has nodes of 4 cores with shared memory and Marenotrum3 16 cores per node.

The experiments executed in Marenostrum2 use the MPICH library as the underlying MPI runtime and the OpenMP compiler used is IBM XL version 10.1. The SmpSs version used is 2.0.

In Marenotrum3 we used the Intel MPI library version 4.1.3 and the underlying compiler is Intel 13.0.1.

In this section, we will use the execution time to compare the performance of each experiment. The execution time is the wall time that each application takes to finish. We will use execution time instead of speed up because for Gadget and Gromacs is not possible to obtain a serial execution (the data does not fit in the memory of a single node). For clarity in the section, we will use the same metric for all the applications. Each value is the average from 5 identical executions.

In the following charts, we will be comparing four series for each application. Their meaning is as follows:

Consecutive: Execution of the application with a consecutive placement of MPI processes among nodes.

Consecutive + LeWI: Execution of the application with DLB and LeWI for load balancing and a consecutive placement of MPI processes among nodes.

Round Robin: Execution of the application with a round robin placement of MPI processes among nodes.

Round Robin + LeWI: Execution of the application with DLB and LeWI for load balancing and a round robin placement of MPI processes among nodes.

For this evaluation, we have used BT-MZ, Lulesh, Gromacs, and Gadget. For BT-MZ and Gromacs we are using the MPI+SmpSs version, Gadget use MPI+OpenMP and Lulesh is parallelized with MPI+OmpSs.

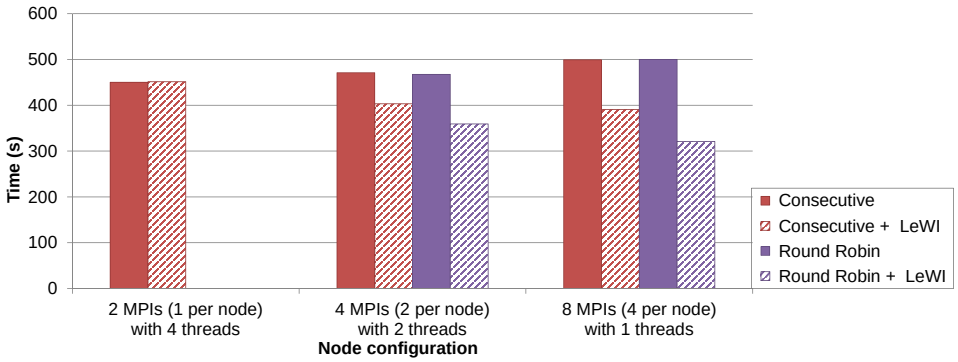
Application	HPC Server	Program. model	Nodes	CPUs	MPIs per node
BT-MZ	MN2	MPI+SmpSs	2 and 4	8, 16	1, 2 and 4
Lulesh	MN3	MPI+OmpSs	4	64	4
Gromacs	MN2	MPI+SmpSs	1, 2, 4, 8, 16, 32 and 64	4, 8, 16, 32, 64 and 128	4
Gadget	MN2	MPI+OpenMP	200	800	4

Table 7.1: Applications used for evaluation

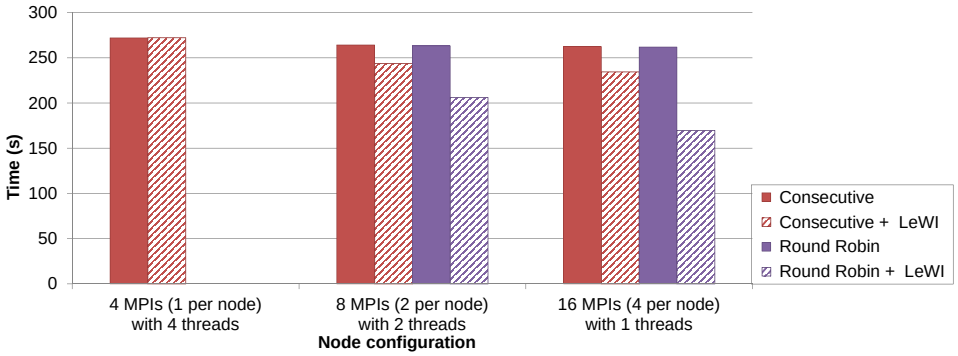
Each application runs in a different number of nodes depending on its needs. BT-MZ is the only application that runs with 1, 2 and 4 MPI processes per node, the case with 1 MPI process per node is shown just for reference because LeWI can not be applied. Lulesh, Gromacs, and Gadget are always executed with 4 MPIs per node, because in all of them the parallelization of the second level is not complete, and is used just for load balancing purposed. In Table 7.1 we can see a summary of the applications used in this evaluation and their characteristics.

Results and Discussion

In Figure 7.15 we can see how the distribution of MPI processes affects the performance of the BT-MZ application when running in 2 and 4 nodes. In the Y axis, we can see the execution time and in the X axis the different configurations inside a node (1, 2 or 4 MPIs in the same node). We can observe that the speed up of the application is much better when running with LeWI and the Round Robin distribution. It is important to notice that the executions without load balancing have the same performance with Consecutive and Round Robin distributions in all the cases. This means that the communication pattern of this application is not affected by the



(a) BT-MZ class C in 2 nodes (8 CPUs)



(b) BT-MZ class C in 4 nodes (16 CPUs)

Figure 7.15: BT-MZ in Marenostrum2. MPIs distribution impact in LeWI performance

MPI distribution. It is just the automatic load balancing mechanism that is benefited with the Round Robin distribution.

In the case of running on 2 nodes (Figure 7.15(a)) it is interesting to notice that the execution time of the application without load balancing increases with the number of MPIs. But when using LeWI, this trend is reversed, and the best configuration is running with 4 MPI processes per node and a Round Robin distribution.

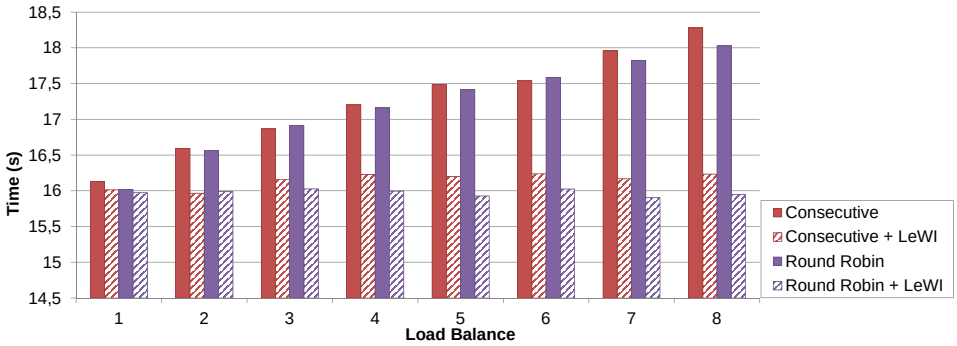


Figure 7.16: Lulesh in Marenostrum3. MPIs distribution impact in LeWI performance

In Figure 7.16 the execution time of Lulesh with 64 MPI processes in 4 nodes. In the X axis, we can see the value of the flag that controls the load imbalance of the application, a higher value of the flag implies a higher load imbalance of the application. The performance of the application is slightly affected by the distribution of MPI processes among nodes. The difference in the performance is less than the 1% in almost all the executions without load balancing. But when using LeWI the difference in the performance between a Consecutive or Round Robin placement of MPI processes is around the 1,5%. The speed up obtained with LeWI respect the application without load balancing goes from 3% to 11% depending on the load imbalance.

The impact of the MPI distribution in the performance of Gromacs can be seen in Figure 7.17. In this chart in the X axis, we find the number of nodes used for the execution from 1 to 64 nodes (4 to 256 cores). We can see that, in this application, the Round Robin distribution is not always beneficial for the application. This means that the communication pattern in Gromacs is

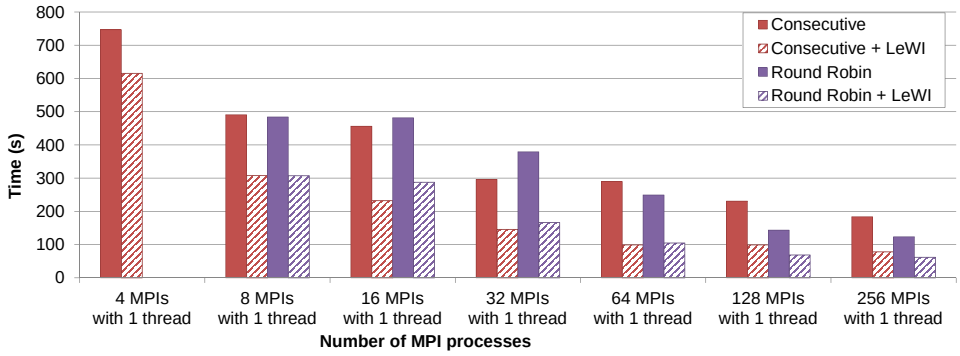


Figure 7.17: Gromacs in Marenostrum2. MPIs distribution impact in LeWI performance

affected by the distribution of MPI processes among nodes. The execution with LeWI always improves the performance of the application. But using a Round Robin distribution is not the best option in all the cases. The speed up obtained when running with LeWI is between 37% and 57% in the case of a consecutive placement of MPI processes and between 36% and 49% for a Round Robin distribution.

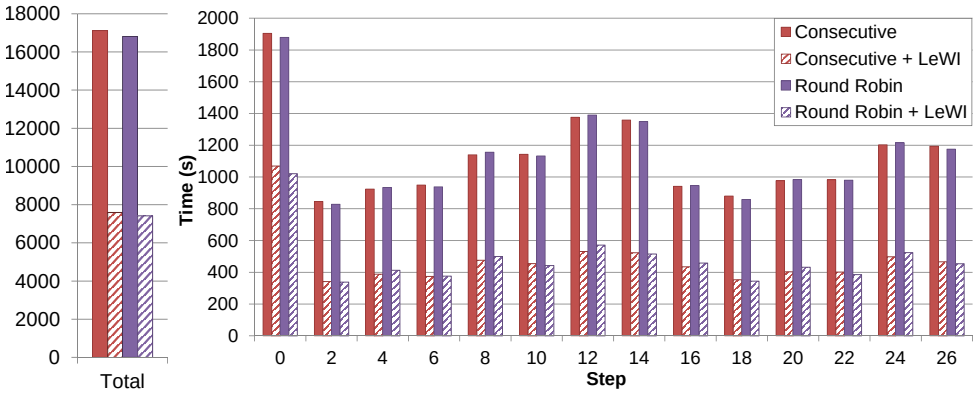


Figure 7.18: Gadget in Marenostrum2. MPIs distribution impact in LeWI performance

In Figure 7.18 we can see the execution time of Gadget with different MPI distributions with and without load balancing. Gadget, in general, is not affected by changing the distribution of MPIs, in the details of the time steps, we can see that some of them are affected negatively and other positively by a Round Robin distribution, in all the cases the difference is negligible. The improvement in performance of using LeWI with Gadget is 55,6% with a Consecutive placement of MPI processes and 55,8% with a Round Robin distribution.

7.2.2 Conclusions of the Impact of MPIs Distribution Among Nodes

In the distribution of MPI processes among nodes, we can not conclude that a universal solution is suitable for all the applications. But we have seen that is an important topic. The placement of MPI processes could affect the performance of the application if the communication pattern was designed for a specific distribution.

In the case of using LeWI, we have seen that the MPIs distribution between the nodes can have a significant impact on the performance of the application. In the case of the BT-MZ (4 MPIs per node running in 4 nodes) the improvement when using LeWI with a Consecutive placement of MPI processes was 10%, compared to an improvement of 35% when running LeWI with a Round Robin distribution.

The effect of the distribution, however, will depend on the load distribution among processes. For this reason, it is difficult to generalize if changing the MPI distribution will improve the performance of LeWI. However, usually the data distribution among MPIs in scientific applications is consecutive (highly loaded MPI processes are consecutive) for this reason, we can say that in general a Round Robin distribution of MPI processes obtains better results than the Consecutive distribution.

7.3 To bind or not to bind

As we have seen in previous chapters, LeWI is based in lending the CPUs from a process that is not using it to another one that still has work to do. In Figure 7.19 we can see the internal behavior of the policy. We have some DLB data in a shared memory accessible by all the process of the application. In this shared data we keep the number of idle CPUs at this moment. At the beginning of the execution, the number of idle CPUs will be 0. When a process lends its resources to the system, the number of idle CPUs will be increased accordingly. When a process wants to ask for more resources will check the number of idle CPUs in the system and decrease it if necessary.

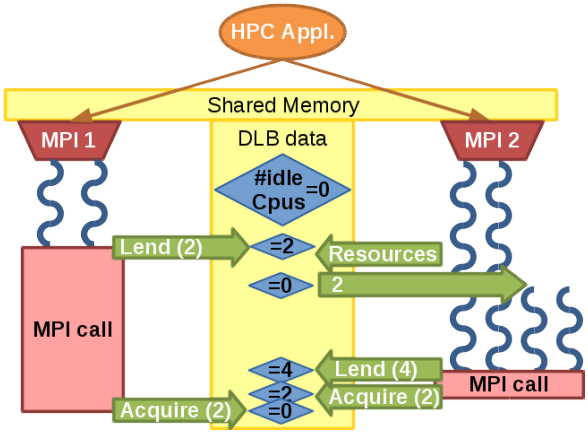


Figure 7.19: LeWI internal behavior

When analyzing some performance results in detail, we detected that when LeWI changed the number of threads, the threads needed some time to adjust correctly to the CPUs that were idle. This is because we are relying on the Operating System (OS) scheduler to assign each thread to an empty CPU. In Figure 7.20 we can see a trace of an application with 2 MPI processes and 6 threads each and LeWI. The color of each thread represents the CPU where it is running. We can see how the new started threads of MPI process 2, when borrowed from MPI process 1, do not use the correct CPU. Instead, some of the threads start sharing the same CPU (same color at the same time, marked with circles in the trace).

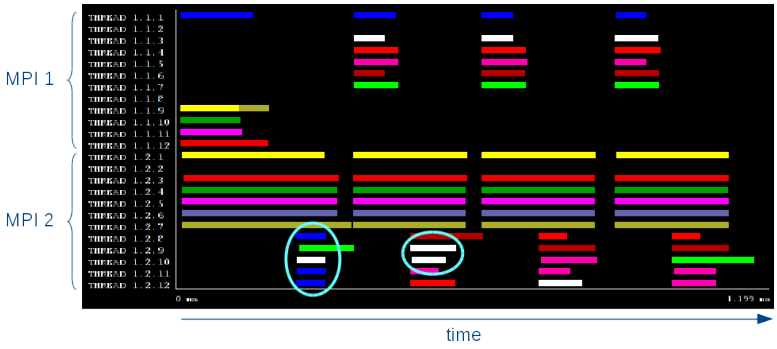


Figure 7.20: Trace: example of LeWI and thread assignment to CPUs

To avoid this, we implemented a new version of LeWI based on masks of CPUs; we will refer to this new policy as *LeWI_mask*. In Figure 7.21 we can see the behavior of the new policy. In this case, in the shared memory, we will keep a mask of idle CPUs. When a process lends its resources, it will give to the system a mask of CPUs to be lent. The mask of idle CPUs will be updated accordingly. When a process asks for resources, it will be given a mask of idle CPUs where it can run new threads.

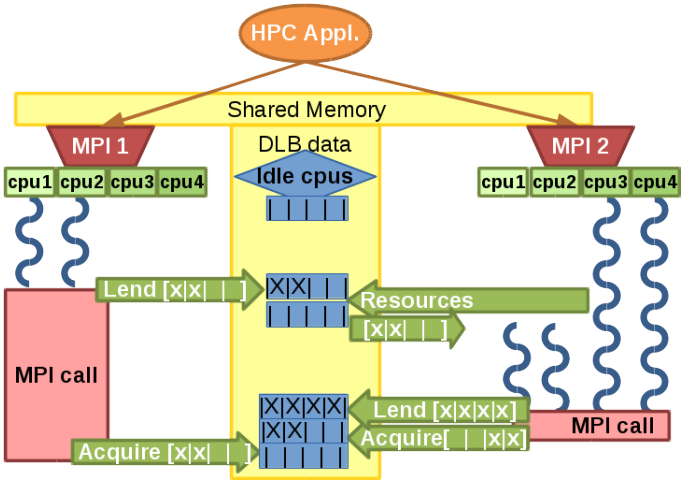


Figure 7.21: LeWI mask internal behavior

To develop this feature we need support from the runtime in the inner level of parallelism, in our case OpenMP. The standard OpenMP does not offer an API to assign threads to specific CPUs. For this reason we implemented a new API in Nanos++ [22] to change the number of active threads with a mask of CPUs. The two calls added to the API were the following:

- `void nanos_omp_get_process_mask ( cpu_set_t *cpu_set );`
- `int nanos_omp_set_process_mask ( const cpu_set_t *cpu_set );`

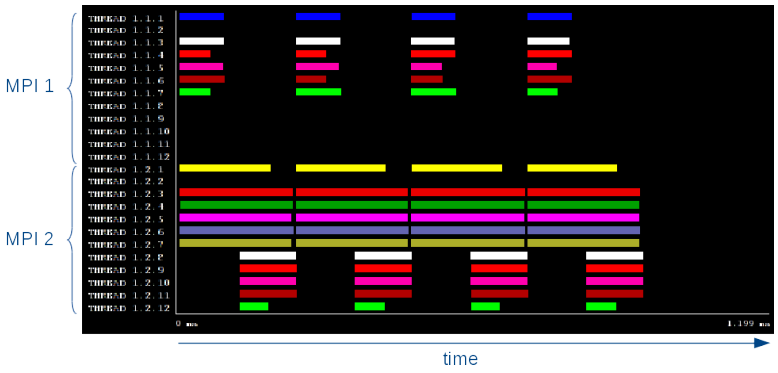


Figure 7.22: Trace: example of LeWI_mask and thread assignment to CPUs

In Figure 7.22 we can see a trace of an application with 2 MPI processes, with 6 threads each, running with LeWI_mask. The colors in the trace represent the CPU where the thread is bound and running. We can see how now a CPU is not being used by two threads at the same time.

In the following section, we are going to evaluate the performance of LeWI_mask. As we detected that the impact of binding the threads is highly related to the architecture of the node, we will show the performance obtained in different kind of nodes.

7.3.1 Performance Evaluation

In this section, we are going to evaluate the impact of binding or not the threads to CPUs in the performance of LeWI. And how the size of the node and its memory structure affects the performance.

Environment and Methodology

The experiments have been executed on Marenostrum3. We have used the Intel MPI library version 4.1.3, and the underlying compiler used by Mercurium is Intel 13.0.1. The experiments shown have been obtained using the Nanos++ runtime.

In this section, the metric we will use is the speed up respect the serial execution of the application (we consider the serial execution as the execution with one MPI process and 1 thread). The values shown for each scenario are the average of 10 runs.

In this evaluation, we have used the BT-MZ benchmark from the NAS-MZ benchmark suite and Lulesh. We have executed the BT-MZ with all the possible combinations of MPI processes and OpenMP threads in one node. The version of the Lulesh application we are using is not fully parallelized with OpenMP, the second level of parallelism is only used for load balancing, for this reason, all the experiments have been done with one thread per MPI process and filling the nodes with MPI processes. Also, Lulesh can only run with a certain number of MPI ranks; we have always executed with 64 MPI processes in total.

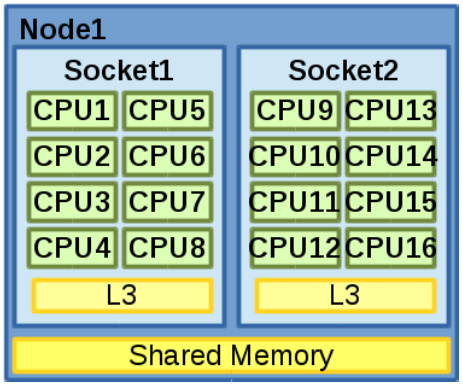


Figure 7.23: Marenostrum3 node structure

Each node of Marenostrum3 has two sockets of 8 CPUs each, caches L1 and L2 are local to the CPU, and L3 is shared for each socket. A schematic representation of a Marenostrum3 node can be seen in Figure 7.23. To evaluate the impact of the node size and memory structure we have emulated different nodes sizes using Marenostrum3 nodes and limiting the CPUs that the application can use. To see the impact of the memory hierarchy we have selected CPUs from the same socket, calling it *compact* or CPUs from both sockets, calling it *scatter*.

We have executed in the following 5 different scenarios:

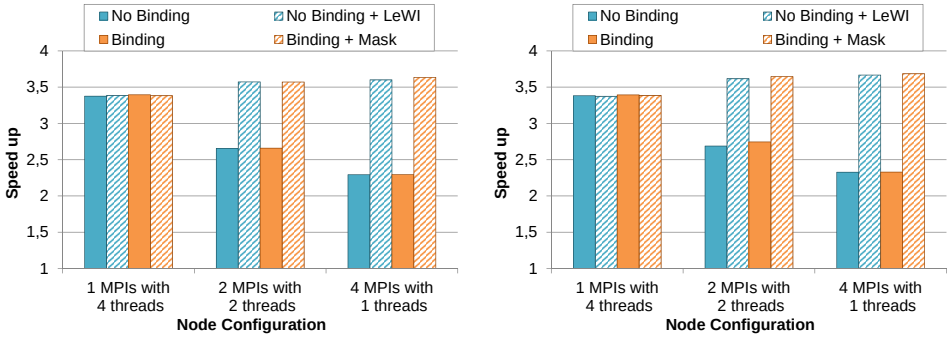
- **4 CPUs Scatter memory:** Using only CPUs 1, 2, 9 and 10 of each node.
- **4 CPUs Compact memory:** Using only CPUs 1, 2, 3 and 4 of each node.
- **8 CPUs Scatter memory:** Using only CPUs 1, 2, 3, 4, 9, 10, 11, 12 of each node.
- **8 CPUs Compact memory:** Using only CPUs 1, 2, 3, 4, 5, 6, 7 and 8 of each node (socket 1).
- **16 CPUs Scatter memory:** Using the whole Marenostrum3 node.

In the following evaluation, we will be comparing four series for each application. Their meaning is as follows:

- **No Binding:** Execution of the original application without CPU binding.
- **No Binding + LeWI:** Execution of the application without CPU binding and with DLB and LeWI for load balancing.
- **Binding:** Execution of the original application with binding of threads to CPUs.
- **Binding + Mask:** Execution of the application with binding of threads to CPUs and with DLB and LeWI_mask for load balancing.

Results and Discussion

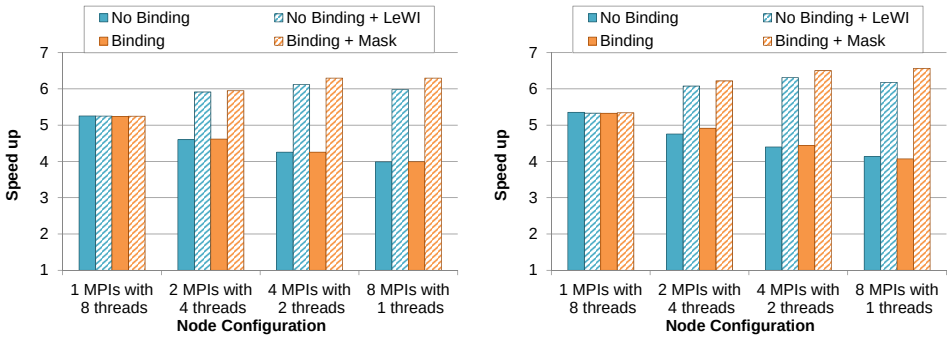
In Figure 7.24 we can see the speed up obtained by the BT-MZ application when running in a node of 4 CPUs. In the Y axis, we can see the different configurations of MPI processes and threads per process inside the node. The performance of the original application decreases when increasing the number of MPI processes because the imbalance increases with the number of MPI processes. The right hand side chart represent a node with compact



(a) Compact (b) Scatter

Figure 7.24: BT-MZ Speed up, class C in 1 node with 4 CPUs

memory (Figure 7.24(a), the left hand side chart is a node with scatter memory (Figure 7.24(b)). In this case, we can observe that there is no significant difference in the performance obtained with the two types of memory hierarchy. In both cases there is no difference between binding or not binding the threads in the original application or when using DLB with LeWI or LeWI_mask.



(a) Compact (b) Scatter

Figure 7.25: BT-MZ Speed up, class C in 1 node with 8 CPUs

Figure 7.25 shows the speed up obtained by BT-MZ in one node with 8 CPUs. In this case, we can see that the original execution is not affected

by the binding of threads. But when using DLB we can observe that the speed up obtained is better with LeWI_mask than with LeWI. Moreover, the difference in performance increases with the number of MPI processes per node.

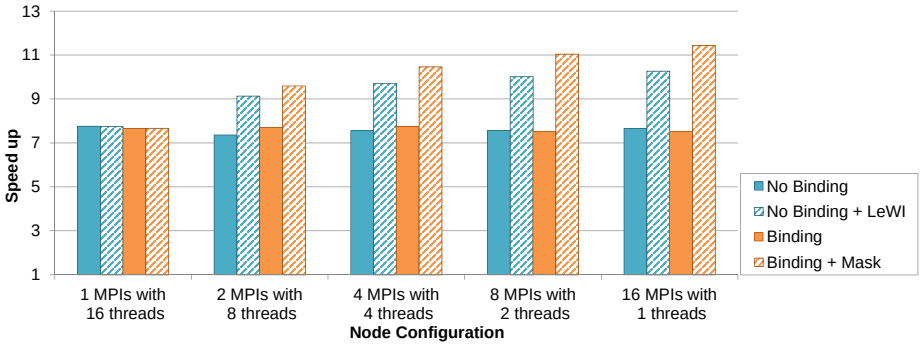
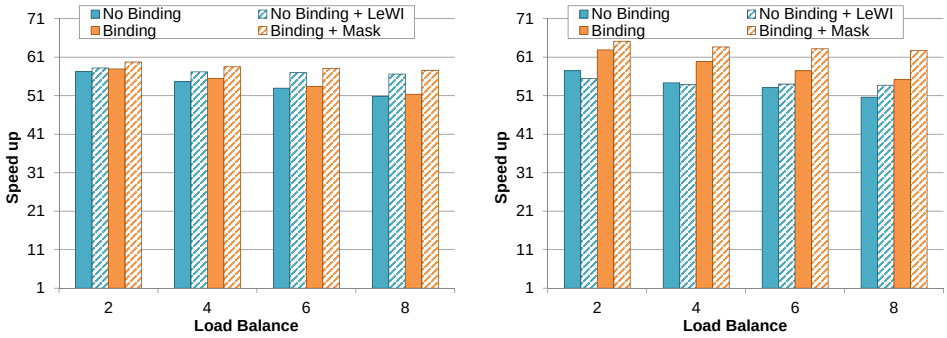


Figure 7.26: BT-MZ Speed up, class C in 1 node with 16 CPUs

In Figure 7.26 we can see the speed up obtained by BT-MZ in a node with 16 CPUs. We can observe that the performance of the original application is not affected by the binding of threads, this is because at the beginning of the execution the OS schedules each thread in one CPU and there are no migrations. When running with DLB, we can see that using LeWI_mask and binding the threads achieves a better speed up than when using LeWI without binding the threads. In this case, the threads are going to sleep and waking up, and the OS scheduler needs to place them in the idle CPUs.

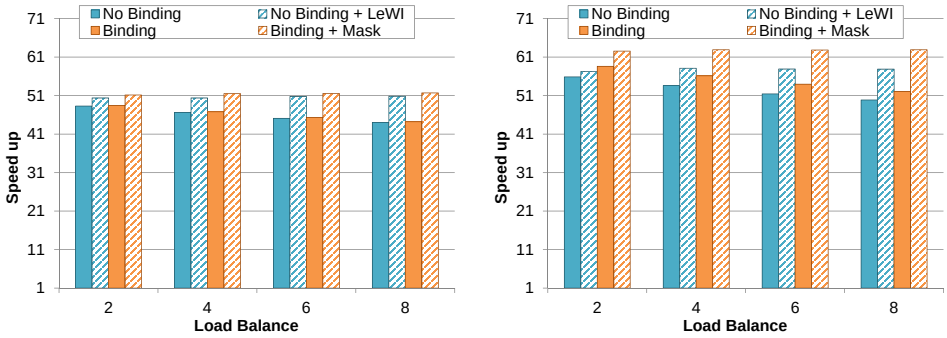
In Figure 7.27 we can see the speed up obtained by Lulesh when running on 16 nodes with 4 CPUs. In the X axis, we can see different values for the parameter that controls the amount of load imbalance of the application. Higher values represent a greater imbalance. In the case of a compact memory hierarchy we can see that there is almost no difference between binding or not binding the threads to cores, neither in the original application nor when using LeWI or LeWI_mask. On the other hand, with scatter memory we can see a significant difference in performance when binding threads to CPUs or not. The original application benefits from binding threads to CPUs in all the situations. When using DLB with LeWI and LeWI_mask we can see that



(a) Compact (b) Scatter

Figure 7.27: Lulesh Speed up, in nodes with 4 CPUs, 64 MPI processes in 16 nodes

the performance of binding the threads to CPUs (LeWI_mask) is better than using LeWI, and the performance gain is higher than the use of binding in the original application.



(a) Compact (b) Scatter

Figure 7.28: Lulesh Speed up, in nodes with 8 CPUs, 64 MPI processes in 8 nodes

Figure 7.28 shows the speed up obtained by Lulesh in 8 nodes of 8 CPUs each. The observations in the case of 8 CPUs per node are very similar to the ones of 4 CPUs per node. When using a compact memory hierarchy

the performance is not affected by the binding of the threads to CPUs, not in the original application, nor when using DLB with LeWI. With a scatter memory hierarchy the speed up obtained is higher with binding of threads in both cases. But the benefit in the speed up of binding the threads is greater when using DLB with LeWI_mask.

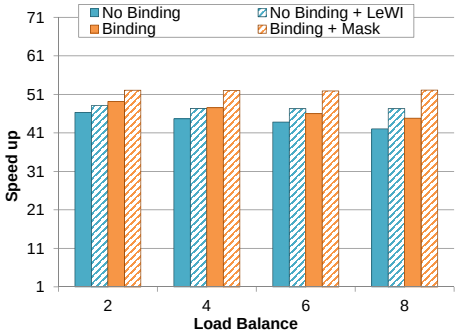


Figure 7.29: Lulesh Speed up, in nodes with 16 CPUs, 64 MPI processes in 4 nodes

In Figure 7.29 we can see the speed up obtained in 4 nodes with 16 CPUs by Lulesh. We can observe that binding the threads to the CPUs improves the performance of the application and that the improvement is higher when using DLB with LeWI_mask.

7.3.2 Conclusions about to bind or not to bind

In this section, we have evaluated the impact of the binding of threads to the CPUs when using DLB and LeWI. We have compared the performance of the original application when binding or not and when using LeWI or LeWI_mask with DLB. For the evaluation, we have used different sizes of nodes and memory hierarchy.

We have seen that the binding of threads to CPUs affects the performance of applications, but when using DLB with LeWI the impact is higher. This is due to the fact that in an execution without DLB the threads are constant and once they have been assigned to a CPU the Operating System scheduler does not have reasons to migrate them. When using DLB the threads are going to sleep and waking up, and when they wake up, they go to the CPU

where they were running. The operating system scheduler needs some time to detect the conflict and migrate one of the threads to an idle CPU.

In this evaluation, we have confirmed that the more CPUs in the node, the more necessary to bind threads to CPUs. Not only when using DLB but also in executions without DLB the binding is an important performance factor. Now that the trend is to have bigger nodes this practice is becoming more common and necessary.

An important factor in the impact of the binding in the performance is the hierarchy of memory. Having levels of memory that are not shared among all the CPUs makes binding more important. In this case migrating a thread from a CPU to another not only penalizes the time it has been sharing the CPU but also the fact that its data must migrate also.

Finally, we have seen that the number of MPI processes that are running in the node also have an impact on the performance. The more MPI processes we are placing in a node, the more performance loss if not using binding of CPUs when using DLB.

We can conclude that the use of binding of CPUs is necessary both when using DLB and when not. But especially with DLB, it is important to have a LeWI_mask policy with binding of CPUs to obtain the maximum performance.

Chapter 8

Integration with a Parallel Runtime

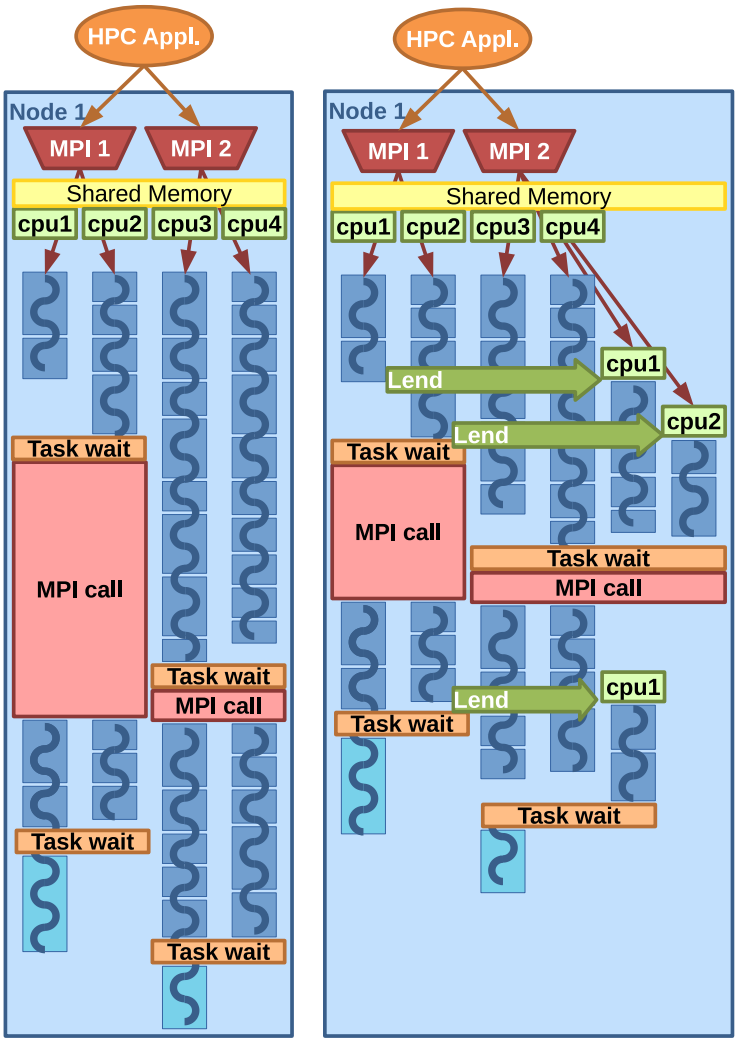
In this Section, we will explain the integration of DLB with a parallel runtime, Nanos++ (see Section 2.1.4), and evaluate its impact on the performance of LeWI and DLB.

The integration of the DLB library with the Nanos++ runtime will allow us to have more control over the threads of each process, more access points to the library and more information about the internal state of the process.

The main advantage of this integration is that the threads will be autonomous to decide when to release its CPU to the system. In situations of load imbalance within the process or limited parallelism the resources can be used by other processes.

In Figure 8.1 we can see different examples of situations where we can benefit from this integration, if there is an imbalance between threads of the same process, the resources can be used by another process. After the MPI call we can see a parallel region and in light blue a serial code, in this case, we can use the resources during the serial phase for another process.

Another advantage of the integration is the increase in the responsiveness of DLB. For example, when a CPU is claimed by its owner, as soon as the thread that is borrowing the CPU reaches a safe point it will release the CPU to its owner. Without the integration, the process had to wait for the master



(a) Imbalanced MPI+OmpSs application

(b) Balancing application with LeWI and autonomous threads

Figure 8.1: Example of LeWI algorithm with autonomous threads

thread to finish its task, then it could check if any resource of the process was claimed, and notify the corresponding thread to sleep, finally when the thread finished its task would release the claimed resource.

We also have direct information from the runtime that can be used by DLB. As the runtime knows the amount of parallel work left to do it can ask for more or fewer resources depending on this information. For example, if there are 4 idle CPUs in the system and a process has 2 parallel tasks pending it will ask for 2 CPUs instead of borrowing all of them.

In the following section we are going to explain the technical details of the integration, and in the next one, we will present the performance evaluation.

8.1 Technical Details

The technical details of the integration are important because we want to demonstrate that DLB is ready to be used by a parallel runtime. It has also been important to identify the key points of the parallel runtime that must be located for the integration.

DLB is integrated with Nanos++ since version 0.7.

The integration maintains the independence between the two runtimes: DLB and Nanos++. DLB offers an advanced API targeted to parallel runtimes. And the Nanos++ runtime calls the DLB API when necessary.

Nanos++ can be installed with DLB support by adding a flag in the configuration step to the DLB installation directory. At execution time, DLB can be enabled with a flag given to the Nanos++ runtime.

We have identified the following key points in the runtime where calls to the DLB library can be added:

1. When a thread detects that it does not have work left to do.

2. When it is safe for a thread to go to sleep and release its CPU. Usually after finishing a task (or unit of work).

3. When the process can use more resources than the assigned ones. At this point, the runtime can tell DLB how many resources it can efficiently use.

4. When it is safe to return resources claimed by other processes.

Algorithm 2 Integration of DLB in Nanos++

```
1:  while (!queue.IsEmpty()) do
2:    if (DLB_ReturnClaimedCpu()) then
3:      /* Check flag and sleep if necessary */
4:      CheckGoToSleepFlag();
5:    end if
6:    if (queue.size > currentResources) then
7:      if (currentResources < defaultResources) then
8:        /* Claim muy CPUS */
9:        DLB_ClaimCpus(queue.size-currentResources);
10:     end if
11:   end if
12:   if (queue.size > currentResources) then
13:     /* Ask for more resources if available */
14:     DLB_UpdateResources(queue.size-currentResources)
15:   end if
16:   task=queue.getTask();
17:   execute(task);
18: end while
19: /* No tasks to execute I release my CPU */
20: DLB_ReleaseCpu(myCPU);
21: /* Check flag and sleep if necessary */
22: CheckGoToSleepFlag();
```

For points 3 and 4, it is important to locate places in the code that are not in the critical path and that are executed frequently enough to give an acceptable response time to changes in the assignment of resources. But at the same time, they are not called too often to degrade the performance of the application.

In algorithm 2 we can see a summarized pseudo-code of the integration between the Nanos++ runtime and DLB. Before starting a new task, a thread will check if its CPU has been claimed and it has to go to sleep (lines 2 to 4).

If the number of tasks in the queue is greater than the current resources and the current resources is less than the owned resources by this process it will claim some CPUs (line 9). If the number of tasks in the queue is greater than the number of resources it will ask for idle resources to DLB (line 14).

Finally, when there are no tasks left in the queue, the thread will lend its CPU to the system (line 20), and go to sleep if necessary.

These were the main changes needed in the Nanos++ runtime and what we learned from it. Now we are going to see what the integration implied from the point of view of DLB.

The integration of DLB with Nanos++ supposed a significant change in the philosophy of the library. Up to now only one thread of each MPI process would enter the DLB library. Therefore a synchronization mechanism was only necessary between processes. By allowing every thread to call the DLB library, we needed to add mutual exclusion between them.

The integration has been done with the LeWI algorithm, and although the main idea is the same, the changes in the code made necessary to implement a new policy, we will refer to it as *auto_LeWI* (from autonomous threads).

An API for runtimes was added in DLB; it can be found in detail in Section 5.5.2. We can summarize it in the following points:

1. Release a CPU to the system.

2. Acquire more resources, up to a given maximum.

3. Check if a CPU is claimed, and return it if necessary.

In Figure 8.2 we can see an example of the integration, in this case, we depict the situation where one thread detects that there is not work to do and lends its CPU to the DLB system. At this point, it will call the DLB

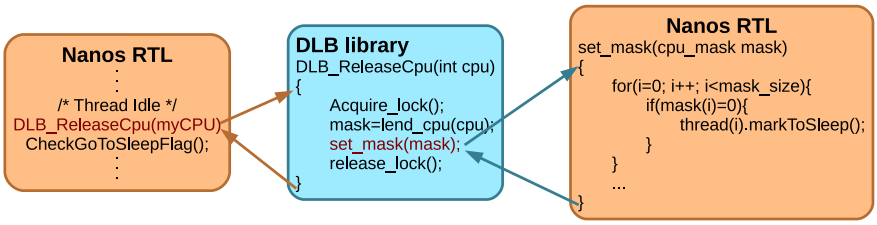


Figure 8.2: Diagram of Nanos++ and DLB integration

API `DLB_ReleaseCpu` with the number of CPU where it is running. When DLB gets this call, first, it will acquire the lock, then it will add the CPU to the pool of idle CPUs, and finally it will set the new mask (the previous mask without the lent CPU). To set the new mask, DLB will call the API offered by the Nanos++ runtime `set_mask`. The `set_mask` call will change the flags of the threads that must go to sleep and will wake up the threads that must run according to the information on the new mask. Finally, after returning from the DLB call, the thread will check its flag to see if he must go to sleep, and at this point it will sleep and stop consuming CPU.

8.2 Performance Evaluation

8.2.1 Environment and Methodolody

The experiments for this chapter have been executed on Marenostrum3. Marenotrum3 has nodes of 16 cores with 32Gb of shared memory. We used the Intel MPI library version 4.1.3 and the underlying compiler is Intel 13.0.1.

The performance metric that we will use is speed up. We have computed it respect the serial execution with one MPI process and one thread.

In the following charts, we will see three different series of data; their meaning is as follows:

no DLB: Execution of the original application without load balancing.

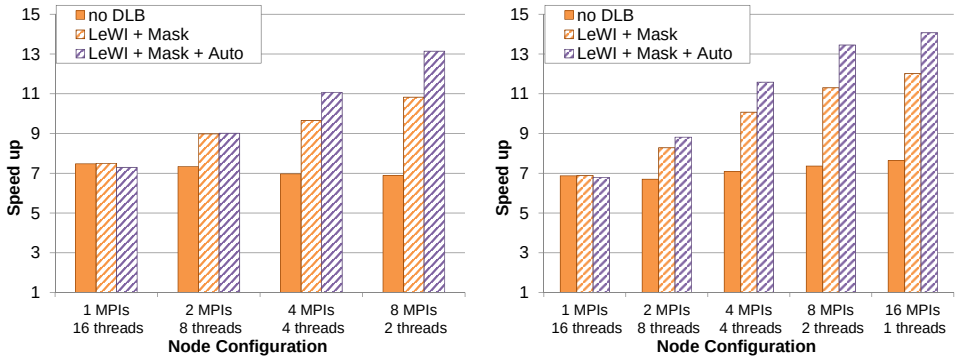
LeWI + Mask: Execution of the application with the load balancing algorithm LeWI and using the option of binding threads to cores.

LeWI + Mask + Auto: Execution of the application with the load balancing algorithm LeWI, using the option of binding threads to cores and using the *Autonomous threads*.

In this evaluation we have used the BT-MZ benchmark from the NAS-MZ benchmark suite and Lulesh, both of them are hybrid using MPI+OmpSs. We have executed the BT-MZ with all the possible combinations of MPI processes and OmpSs threads in one node and classes C and B.

The version of the Lulesh application we are using is not fully parallelized with OmpSs, the second level of parallelism is only used for load balancing, for this reason, all the experiments have been done with one thread per MPI process and filling the nodes with MPI processes. Also, Lulesh can only run with a certain number of MPI ranks; we have always executed with 64 MPI processes in total.

8.2.2 Results and Discussion



(a) Class B (b) Class C

Figure 8.3: BT-MZ Speed up in one node (16 cpus)

In Figure 8.3 we can see the speed up obtained by BT-MZ when running in one node of Marenostrum3 (16 cores) class B and C. In the X axis we can find the configuration of MPI processes and threads in each node. The mesh size of class B does not allow a partition in more than 8 MPI processes, for this reason in the x axis class B only reaches 8 MPI processes per node.

The speed up obtained by class B can be seen in Figure 8.3(a), when running without DLB the speed up is around 7 independently of the configuration of MPI processes and threads inside the node. When using LeWI with binding of CPUs (*LeWI + Mask*), we can see that the performance improves as the number of MPI increases, because it gives more malleability to DLB, and the maximum speed up obtained is 11. But when using LeWI with binding of CPUs and Autonomous threads (*LeWI + Mask + Auto*), we can see that the maximum speed up obtained is around 13.

In Figure 8.3(b) we can see the same experiments with class C. We can observe that the performance of class C is very similar to the performance of class B when not using DLB. When using LeWI and binding of threads, the performance increases with the number of MPI processes per node and obtaining a better performance than class B. In the case of LeWI with binding and Autonomous threads, the speed up can be increased up to 14.

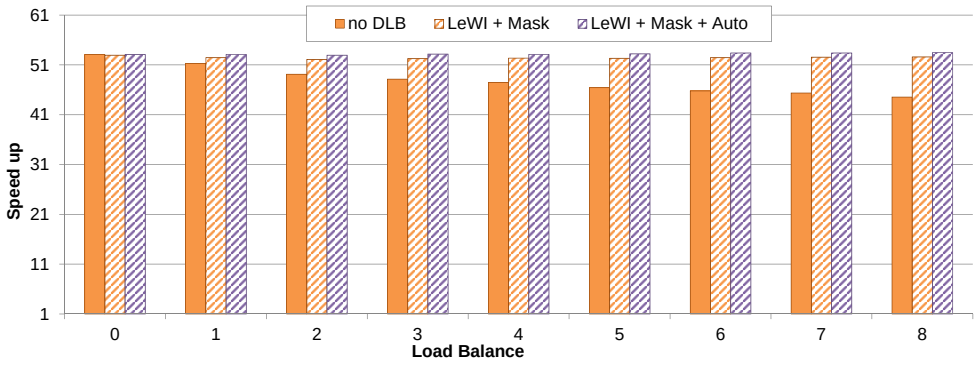


Figure 8.4: Lulesh Speed up in 4 nodes (64 cpus)

In Figure 8.4 we can see the speed up obtained with Lulesh when running in four nodes of Marenostrum3 (64 cores). In the X axis is represented the load balance flag, this flag can go from 0, well balanced execution, to 8, very imbalanced execution. Lulesh is only parallelized with OmpSs partially, for this reason, the execution is done with one thread per MPI process, and the second level of parallelism (OmpSs) is only used for load balancing.

When running Lulesh without DLB, we can see how the performance decreases as the load imbalance increases. When using LeWI with binding of threads, the speed up is constant independently of the amount of load imbal-

ance. In the case of LeWI with binding of threads and Autonomous threads, the speed up is also constant regardless of the amount of load imbalance, and higher that the obtained without Autonomous threads. We can observe that the performance achieved with this version is comparable to the performance obtained with the well balanced application.

It is important to notice that when running with the load balance flag 0 the performance of the Autonomous threads is the same as the execution of the original application and LeWI without Autonomous threads, this means that the Autonomous threads do not introduce an additional overhead to the execution.

8.3 Conclusions

In this chapter we have seen the integration of DLB with a parallel runtime, for this, we have used the Nanos++ runtime. We have used Nanos++ because it supports OpenMP (which is widely employed in HPC applications) and we had the source code and support from the developers.

This integration aimed to prove that DLB is ready to be used by a parallel runtime and that the effort of integration is worth the benefits that will be obtained.

We have presented a simple API that can be utilized by parallel runtimes. With this experience, we have identified the key points that must be located in the runtime to add the calls to the DLB API.

In the performance evaluation, we have seen that the overhead of the threads accessing the DLB library is negligible.

We observed that the performance of applications can be improved using Autonomous threads. The amount of improvement depends on the characteristics of the application. Not only on the load imbalance but the granularity of parallel work or the load balance at the second level of parallelism.

Chapter 9

Alya: A Case Study

Alya [46][47][70] is a computational mechanics code developed at BSC. It can solve different physics problems that include: incompressible/compressible flow, solid mechanics, chemistry, particle transport, heat transfer or turbulence modeling, among others.

Alya was originally designed and developed for HPC environments, and it is used in production runs that use thousands of cores during several hours. Currently, it is part of the UEABS [48] (Unified European Applications Benchmark Suite), a selection of 12 codes. The selection of these codes was based on its scalability, portability, and relevance for the scientific community.

On of the distinctive features of Alya is that it can simulate multiphysics problems. In this case, the solution can be achieved in a single code or coupling different instances of Alya, where each instance will solve a different physic.

The development in Alya pushes into two directions. On the one hand, research in its numerical methods to perform more precise, faster and more detailed simulations. On the other hand, its code is constantly optimized to scale up to thousands of cores in a wide range of supercomputers.

Alya is written in Fortran and organized in modules. Modules can be compiled and executed independently, and each one solves a physical prob-

lem. Its main parallelization is done using MPI, but in some of its modules we can use a hybrid parallelization using MPI+OpenMP.

Alya relies on Metis [1] to partition the input mesh. The original mesh is partitioned in N subdomains, where N is the number of MPI processes. Each MPI process will work over its subdomain.

9.1 The Load Balance Problem

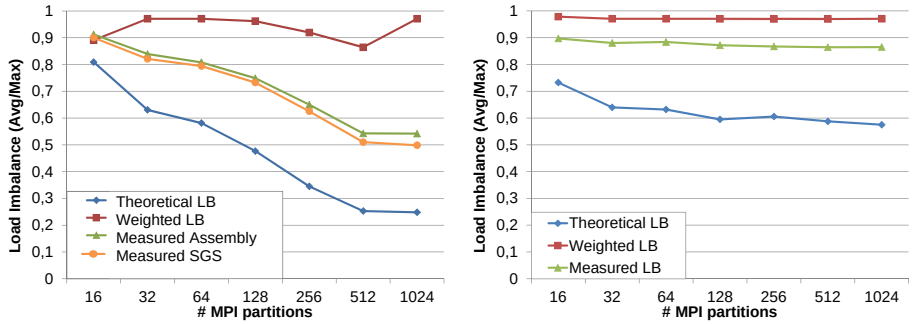
While doing the performance analysis of some executions of Alya we detected several load imbalance problems that were not solved. Moreover, they cannot be solved with the current tools and mechanisms. These kind of load balancing problems are not particular to Alya, they are common in this type of applications. We can categorize the sources of load imbalance like follows:

- **Mesh partition issues:** As we said Alya relies on Metis to partition the mesh among the different MPI processes, this partition will determine the load balance of the executions because each MPI process will compute the data in its subdomain. In order to partition the mesh one or more directives can be given to Metis. In general Metis will try to keep the boundaries to its minimum and to load balance the number of elements per partition.

  - ⋆ **Complex mesh structure:** Some very irregular meshes or with complex structures are hard to partition for Metis.
  - ⋆ **Heuristic used to Load Balance the partition:** Alya computes a heuristic that tries to guess the load of each element and Metis will partition the mesh based on this heuristic. The heuristic given to Metis to load balance the partition is an approximation based on the number of integration points of each element.

    In Figure 9.1 we can see two charts that correspond to two of the simulations that we will analyze in this chapter. In the X axis we can see the number of partitions and in the Y axis the load balance computed as: $LB = \frac{Average(time_{MPIs})}{Max(time_{MPIs})}$.

    The label *Theoretical LB* is the load balance based on the number of elements assigned to each partition. The *Weighted LB* is the load balance obtained when using the heuristic (i.e. giving to each



(a) Load balance for the *Respiratory System*

(b) Load Balance for the *Iter* Simulation

Figure 9.1: Load Balance for different inputs of Alya

element a weight based on its integration points). And the *Measured LB* show the real load balance observed in the execution. We can observe that the load balance obtained by the two inputs is very different. In the *Respiratory System* (fluid mechanics problem) the load balance that Metis achieves when partitioning (Weighted LB) is above 0.9 in almost all the cases, but the real load balance decreases from 0.9 to 0.3 (partitioning in 16 or 1024 partitions). On the other hand, the *Iter* simulation (solid mechanics problem) has an almost constant load balance around 0.88 for all the number of partitions.

- ⋆ **Different phases of the code:** Some computations are performed over the elements, but in other phases, the computation is done over the nodes. Having a balanced execution for one of the phases does not guaranty a load balance in another phase.

- **Load change during time:** Some simulations have a variable load during the execution time. For example, in the simulation of particles, the particles start in one MPI subdomain and during the simulation cross boundaries and belong to another subdomain. Another example would be a solid that break; the load will change during the execution as the crack is propagated.

In these situations, a balanced partition at the beginning of the simulation will get imbalanced during the execution. Only a repartition of the mesh could solve this problem, and this kind of solutions are very expensive in execution time.

- **Coupling of codes issues:** When running coupled codes, an important decision is how many resources are given to each code. It is not easy to know beforehand the optimal distribution of resources, and a wrong decision will produce an imbalance between the two codes.

9.2 Improving the Load Balancing

In this section, we are going to present the performance results obtained in Alya with two different test cases. The two test cases used are different in the type of problem they solve, but also in the performance and load balance that they achieve. On the one hand, the *Respiratory System* that solve an incompressible flow and presents a high load imbalance. On the other hand, the *Iter* simulation solve a solid mechanics problem and the partition achieves a good load balance.

We will explain in detail the input cases and the modifications that were made to the code to apply DLB. We will focus on the parallelization of the second level and evaluate the impact of the parallelization performance in the performance of LeWI.

Finally, we will also present some scalability tests executed in up to 16.000 cores.

9.2.1 Input simulations

Respiratory System

For this evaluation, we are going to use the respiratory system [71]. This simulation is used to study the airflow in the human airways during a rapid inhalation. It includes from the face to the seventh branch generation of the bronchopulmonary tree and a hemisphere of the subject's face exterior. In Figure 9.2 we can see depicted the described mesh for this simulation.

The mesh used for this simulation was obtained from a contrast-enhanced computed tomography scan clinically acquired. The unstructured mesh contains 17M elements of four different types: tetrahedra, hexahedra, prisms, and pyramids.

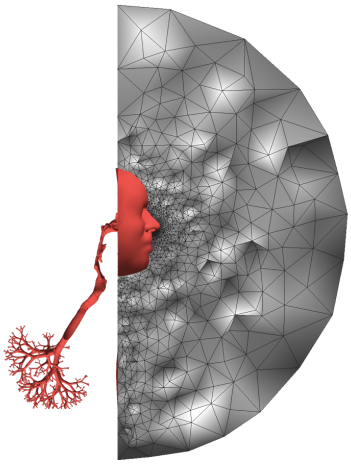


Figure 9.2: Respiratory system

The whole simulation of the respiratory system lasted 50.000 time steps. In Figure 9.3 we can see the trace of one time step running in 256 cores (256 MPI ranks).

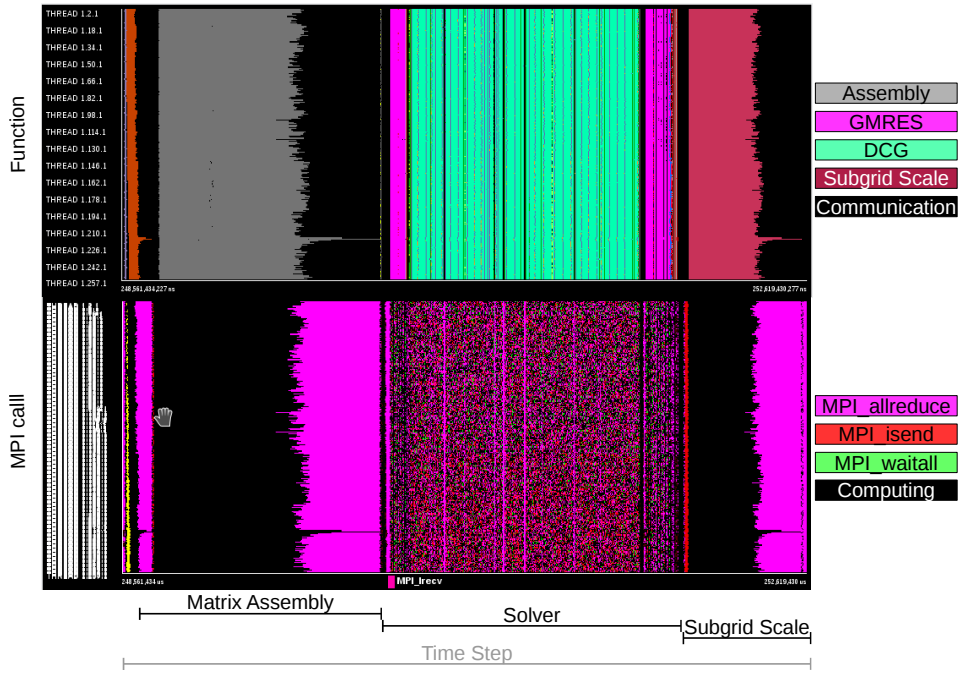


Figure 9.3: Trace of one time step of the *Respiratory System*

Each time step can be divided into 3 phases very differentiated and with a particular behavior. First, the matrix assembly can be seen in gray in the trace, it is a loop over the elements and does not present MPI communication. For this input, the matrix assembly shows a high load imbalance.

Afterward, we can see the algebraic solver, in green and pink, in this case, they solve twice the momentum equation, with the GMRES method (pink), and once the continuity equation with the Deflated Conjugate Gradient method (green). The operations in the iterative solvers are loops over the nodes and contain a high amount of MPI communications, both global and point to point.

Finally, the subgrid scale is computed (red), this phase loops over the elements and does not need MPI communication. The subgrid scale computation has a high load imbalance also.

Iter: Fusion Reactor

ITER [72] (International Thermonuclear Experimental Reactor) is an experimental reactor which will reproduce the physical reaction - fusion - that occurs in the sun and stars. It is a joint research project including 35 different countries that will last for 35 years. In Europe, it is administrated under the Fusion For Energy project.

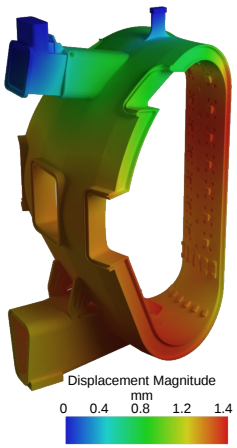


Figure 9.4: Iter

The final aim of it ITER project is to build the largest tokamak nuclear fusion reactor. The vacuum vessel is the central part of the ITER machine. It houses the fusion reactions and acts as a first safety containment barrier. The vacuum vessel is a torus shaped chamber.

The mesh we will be simulating represents a slice of this torus. An image of it can be seen in Figure 9.4. This mesh contains 32 Millions of elements.

The input we will use simulates the deformation of the slice of the vacuum vessel when different forces are applied to it.

Each step of the simulation will have two phases, first the assembly of the matrix and second the solver. In Figure 9.5 we can see the trace of one time step running in 257 MPI processes.

The matrix assembly is performed over the elements, and it does not need communication at the MPI level. On the other hand, the solver has a high amount of MPI communication and its computation is performed over the nodes.

In this trace, we can observe that the imbalance at the matrix assembly is very different than the one in the respiratory system. In general, we can see a better load balance among the different MPI processes.

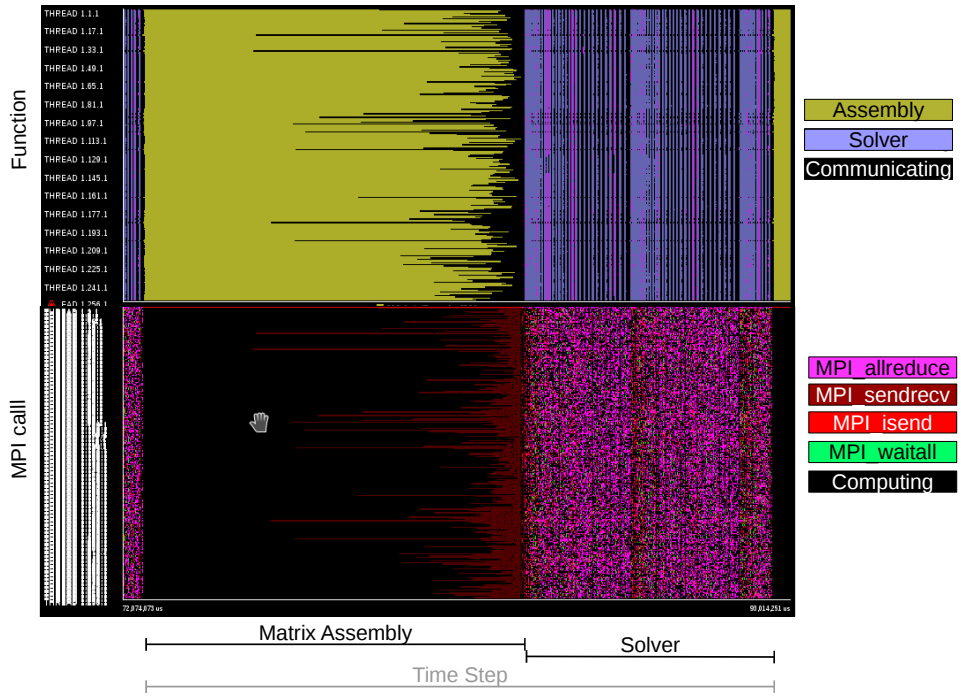


Figure 9.5: Trace of one time step of the *Iter* simulation

9.2.2 Applying DLB to the *Respiratory System* and *Iter*

We have seen that the application has different phases with very different behavior, we have identified that the load balancing problem is in the matrix assembly and the subgrid scale. The performance of the solver also needs to be targeted, but load imbalance is not its biggest problem.

For this reason, we will focus on the matrix assembly and the subgrid scale phases and disable DLB when computing the algebraic solver. Both the matrix assembly and the subgrid scale consist of a loop over the elements of the MPI subdomain.

The original version of this part of the code was already parallelized using OpenMP. The subgrid scale does not present any problem, and all the elements of a subdomain can be processed in parallel. On the other hand, the matrix assembly needs to update the matrix of global nodes; this means that

two elements that share a node will update the same position of the matrix. For this reason we need an `OMP ATOMIC` in Algorithm 3 in lines 7 and 9.

Algorithm 3 Matrix Assembly without Coloring

```
1: DLB_enable()
2: !$OMP PARALLEL DO SCHEDULE (DYNAMIC,Chunk_Size) &
3: !$OMP PRIVATE (...) &
4: !$OMP SHARED  (...)
5: for Elements e in my subdomain do
6:    Compute element matrix and RHS: A^e, b^e
7:    !$OMP ATOMIC
8:    Assemble matrix A^e into A
9:    !$OMP ATOMIC
10:   Assemble RHS b^e into b
11: end for
12: DLB_disable()
```

The performance of this parallelization was far from optimal as we will see in the performance evaluation, being two times slower than the pure MPI version. The common solution to avoid the use of `OMP ATOMIC` is to apply coloring to the loop. A coloring algorithm will assign a color to each element, and two elements that share a node will have different colors. Meaning that elements of the same color can be safely computed in parallel because they do not update the same node.

The Coloring version has a better performance than the No Coloring, but it does not reach the performance of the pure MPI version. The reason is that this implementation does not exploit the data locality of the elements. To improve this, we implemented a new parallelization using two novelty features of the OmpSs programming model: the multidependencies and commutative clause.

Multidependences are dependencies between tasks that are computed at runtime and allows a different number of dependencies for different instances of the same type of task. To use Multidependencies, we will partition the subdomain assigned to an MPI process into tasks and obtain a connectivity graph of these tasks. With the connectivity graph, we will be able to identify

Algorithm 4 Matrix Assembly with Coloring

```
1: DLB_enable()
2: Partition my subdomain in colors
3: for Color c in my colors do
4:   !$OMP PARALLEL DO SCHEDULE (DYNAMIC,Chunk_Size) &
5:   !$OMP PRIVATE (...) &
6:   !$OMP SHARED  (...)
7:   for Elements e in c do
8:     Compute element matrix and RHS: A^e, b^e
9:     Assemble matrix A^e into A
10:    Assemble RHS b^e into b
11:  end for
12: end for
13: DLB_disable()
```

Algorithm 5 Matrix Assembly with commutative Multidependences

```
1: DLB_enable()
2: Partition my subdomain in tasks of size Chunksize
3: Store connectivity graph in neigh
4: for Tasks t in my tasks do
5:   num_neigh = SIZE(t%neigh)
6:   !$OMP TASK
7:   !$OMP COMMUTATIVE([tasks(t%neigh(J)), J=1, num_neigh])
8:   !$OMP PRIORITY(num_neigh)
9:   !$OMP PRIVATE (t, ...)
10:  !$OMP SHARED  (...)
11:  for Elements e in t do
12:    Compute element matrix and RHS: A^e, b^e
13:    Assemble matrix A^e into A
14:    Assemble RHS b^e into b
15:  end for
16: end for
17: DLB_disable()
```

the tasks that can not be executed at the same time because they share nodes.

The `Commutative` clause indicates that two tasks can not be executed in parallel, but the execution order between them is not relevant. In Algorithm 5 we can see the pseudocode of the matrix assembly when using commutative Multidependences.

This version of the parallelization has good data locality because the elements inside a task are consecutive and avoid the use of the `ATOMIC` clause.

In Figure 9.6 we can see a summary of the three versions of parallelization that we implemented and will evaluate.

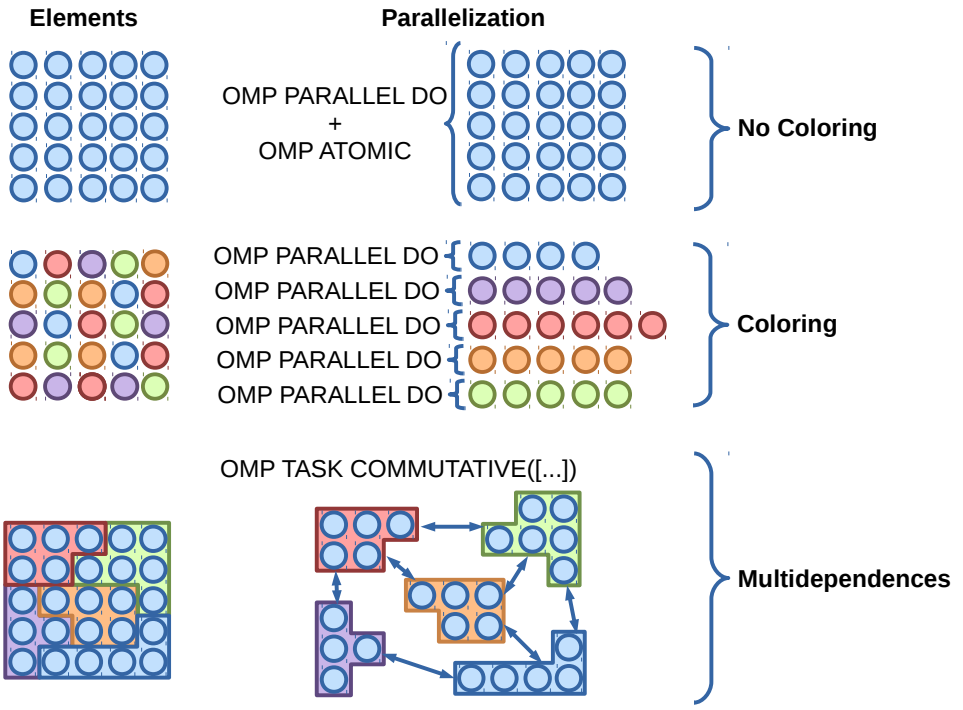


Figure 9.6: Parallelization alternatives of matrix assembly

9.2.3 Environment and Methodology

All the experiments have been executed in Marenostrum3, the nodes of Marenostrum3 have 16 cores with 32Gb of shared memory.

We have used the Intel MPI library version 4.1.3 and as the underlying Fortran compiler Intel 13.0.1. For OpenMP we have used Nanos 0.12a with the source to source compiler Mercurium, this will allow us to use the version of LeWI with autonomous threads and binding of threads to cores (see Section 6.2 and Chapter 8).

The MPI processes are distributed in a Round Robin scheme among the nodes.

For the *Respiratory System* we will show the performance obtained in the Assembly and Subgrid Scale. The values shown in the evaluation correspond to the average of 10 time steps. In the case of the *Iter* test case we will show the performance of the Assembly and the results are the average of 5 time steps that include 17 executions of the matrix assembly.

For each experiment we will use the following configurations:

- **16, 32 and 64 Nodes:** We have executed the experiments in 16, 32 and 64 nodes of Marenostrum3 that correspond to 256, 512 and 1024 cores.
- **Inner parallelization:** We will see the results of using Coloring, No Coloring (use of Atomics) and Commutative Multidependences as the shared memory parallelization.
- **DLB:** We will run without DLB and with DLB using LeWI with binding of threads to cores and autonomous threads. The solid lines correspond to executions without load balancing and the dashed lines to the executions with LeWI.
- **Node configuration:** For each experiment we will see 5 different configurations of MPI processes and threads inside the node:
  - ⋆ **1x16:** 1 MPI process with 16 threads, this is the pure hybrid approach, where MPI is used across nodes and OpenMP/OmpSs inside the shared memory node. In this configuration, DLB cannot load balance, but we show it for completeness.

  - ⋆ **2x8:** 2 MPI processes with 8 threads each. This is another typical configuration when running in nodes with two sockets, each MPI process is mapped to one socket, and 8 threads are spawn in each socket.
  - ⋆ **4x4:** 4 MPI processes with 4 threads each.
  - ⋆ **8x2:** 8 MPI processes with 2 threads each.
  - ⋆ **16x1:** 16 MPI processes with 1 thread each. In this case, the shared memory level is only used for load balancing. This configuration is useful when the application is not fully parallelized with OpenMP/OmpSs.
- **Pure MPI:** As a reference we show the performance of the pure MPI version of the application.

We have divided the evaluation into four parts:

- **Chunk size:** We will study the impact of the chunk size in the performance of the different parallelizations and in particular when using DLB and LeWI. From this evaluation, we will decide the chunk size to use for the following experiments.
- **Execution Time:** We will evaluate the performance of DLB and LeWI in the different simulations. We will compare the 3 possibilities in the shared memory parallelization: No Coloring, Coloring, and Multidependences. And we will see how the performance of the parallelization impacts in the performance of DLB and LeWI.
- **IPC:** We will see the IPC obtained in the matrix assembly and subgrid scale of each input for the different parallelizations. This will support the assumptions made about the performance in the previous section.
- **Scalability:** We will show some scalability test when running the *Respiratory System* in up to 16K cores in Marenostrum3.

9.2.4 Performance Evaluation

Chunk Size Study

In this section, we are going to see the impact of the chunk size on the performance of the parallelization and LeWI. All the experiments have been

executed with 16 MPI processes per node and 1 thread per process. With this configuration, we aim to evaluate the impact of the task duration, the amount of tasks in the queue and the impact in the malleability when using LeWI. Using only one thread will avoid seeing the overhead of several threads accessing to shared structures or invalidating the memory.

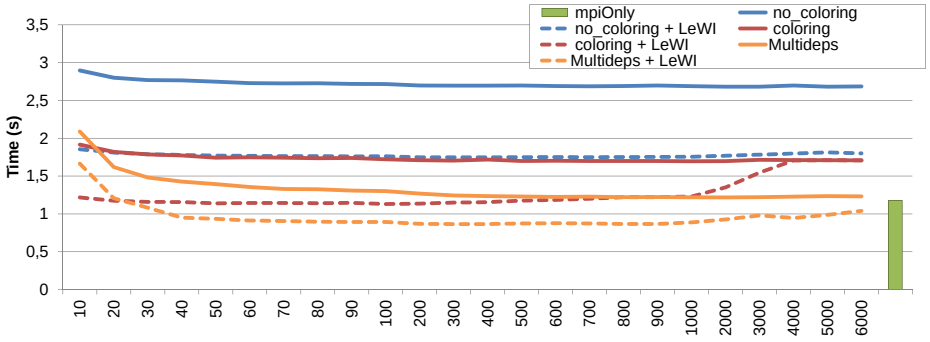
In Figure 9.7 we can see the execution time of the matrix assembly in the *Respiratory* simulation. In the X axis are represented the different chunk sizes used. Each Figure (9.7(a), 9.7(b) and 9.7(c)) correspond to executions on 16, 32 and 64 nodes of Marenostrum3.

In the case of the execution without LeWI, No Coloring and Coloring are not affected by the chunk size, until reaching very small values, around 10 or 20. Using bigger values does not represent a problem. When using Multidependencies the degradation in performance happens with larger chunk sizes around 100.

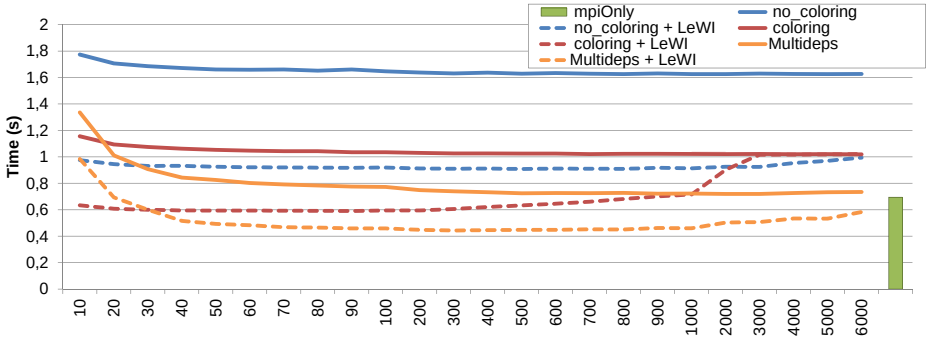
The impact of the chunk size when using LeWI comes from the fact that bigger tasks imply less malleability because threads can not leave a task until it is finished. The Coloring version with LeWI is the more affected by big chunks, after chunks of size 200 the performance slowly degrades until it reaches the same performance as the Coloring version with LeWI. This is because the parallelism in the Coloring version, open and closes for each color and bigger chunks mean fewer chunks to be distributed among the threads, therefore, less parallelism. If there is not enough parallelism when LeWI tries to spawn more threads to load balance, there is not enough work for all of them.

We can see that LeWI improves the performance of the original execution with the same configuration in all the cases. When choosing bigger chunks, LeWI is not able to improve the performance, but it does not introduce overhead. In these cases, it reaches the same execution time as the original application.

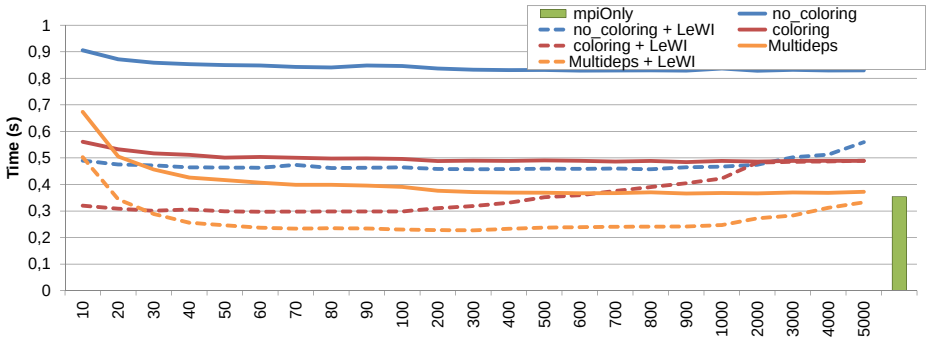
In Figure 9.8 we can see the execution time of the subgrid scale in the *Respiratory* simulation when varying the chunk size. The conclusions for this experiments are very similar to the previous ones. The limit in the chunk size is around 20, from this size the performance of the Coloring and No Coloring versions degrade. For very small chunks sizes the overhead of creating the tasks and scheduling them is too high. In this case, the limit chunk size is bigger because the durations of the tasks are smaller if we compare the



(a) 16 Nodes (256 cores)

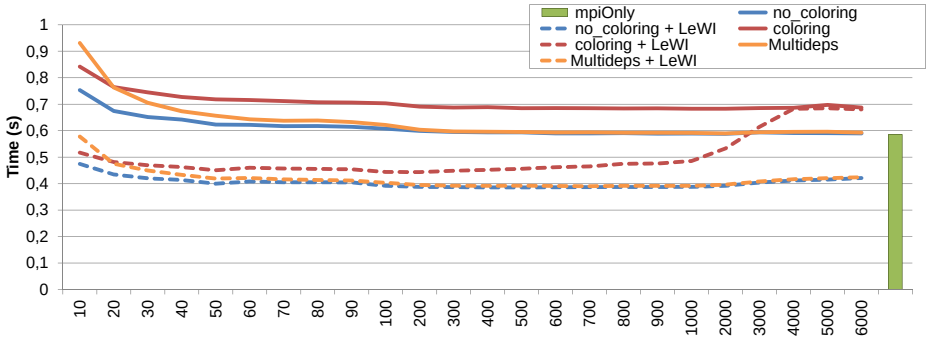


(b) 32 Nodes (512 cores)

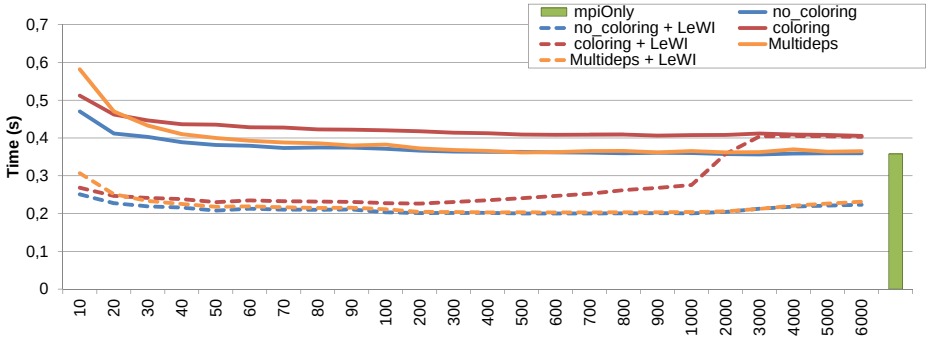


(c) 64 Nodes (1024 cores)

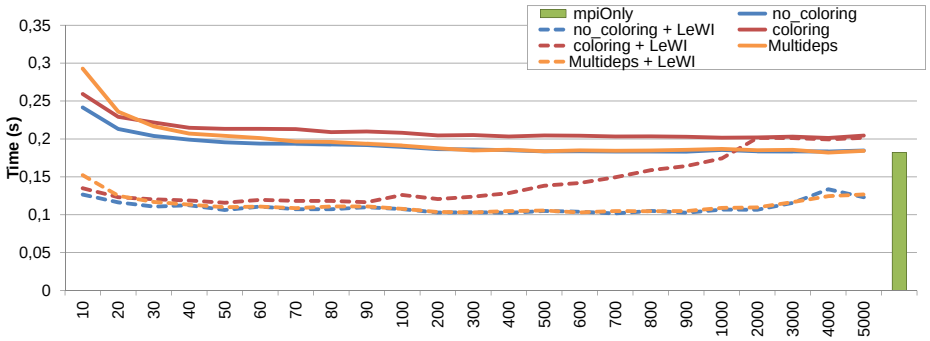
Figure 9.7: Performance of Matrix Assembly in the respiratory simulation depending on the chunk size



(a) 16 Nodes (256 cores)



(b) 32 Nodes (512 cores)



(c) 64 Nodes (1024 cores)

Figure 9.8: Performance of Subgrid Scale in the respiratory simulation depending on the chunk size

total execution time and considering that the number of elements that are processed in the parallel loop is the same.

The higher limit for the chunk size, like before, affect when running with LeWI only. For the Coloring version is around 1000. Although, for all the cases the optimum is around 200. This will be the chunk size used in the experiments executed in the following sections for this simulation.

We can see the execution time of the matrix assembly in the Iter simulation in Figure 9.9. In this charts, we have adjusted the scale of the Y axis to see some differences between the different series. Each chart corresponds to a number of nodes (16,32 and 64), on the X axis, we can see the different chunk sizes. In this case, the Coloring and No Coloring versions are not affected by the chunk size. This is because the duration of these tasks in this simulation is much higher. On the other hand, the Multidependences version is still limited by small chunk sizes, because its limitation does not only come from the duration of the task but also from the amount of tasks in the queue and with a commutative relationship.

When using DLB and LeWI the use of big chunk sizes implies a loss of performance. The three versions are affected proportionally in a similar way.

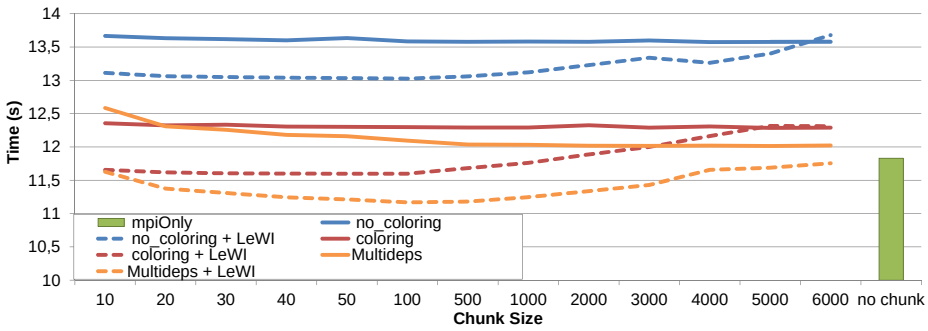
For this simulation the optimal chunk size in all the series is 100. This the chunk size that we will use for this simulation in the following experiments.

Execution Time

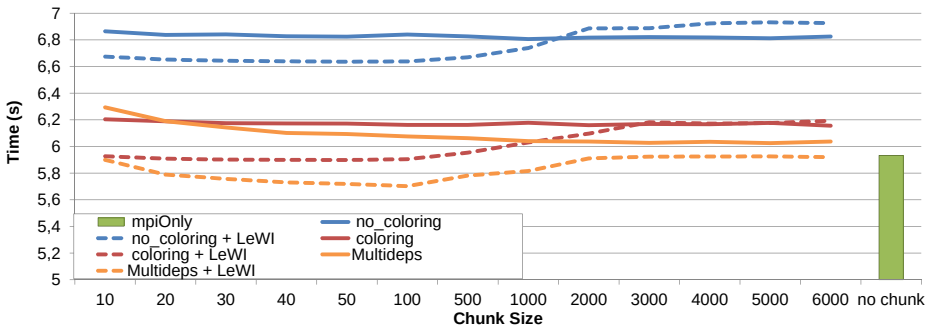
In this section we will compare the execution time of the matrix assembly and subgrid scale computations for the *respiratory system* and the *Iter* simulation with and without LeWI for each parallelization.

In Figure 9.10 we can see the execution time of the assembly when running the *Respiratory* simulation. Figures 9.10(a),9.10(b) and 9.10(c) correspond to executions in 16, 32 and 64 nodes respectively. In the X axis we can see the different configuration of MPI processes and threads used in each case. In the Y axis is represented the average execution time of the assembly for ten time steps.

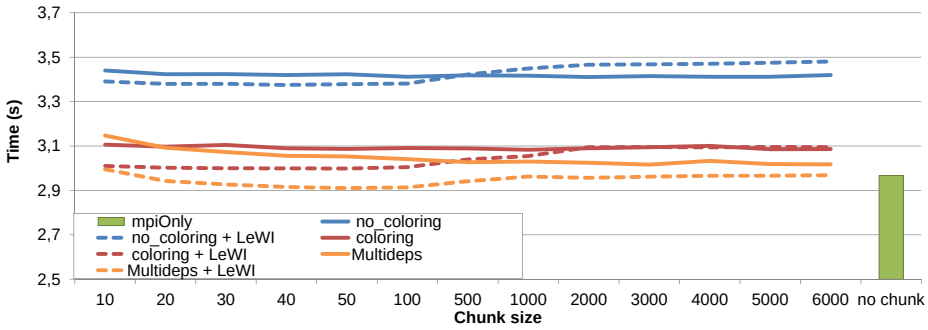
When comparing the three different implementations of the parallelization without LeWI, we can see the difference in performance that they obtain. Being the No Coloring version the worst one versus the commutative Multidependences that achieves a better performance than the pure MPI version for some configurations.



(a) 16 Nodes (256 cores)

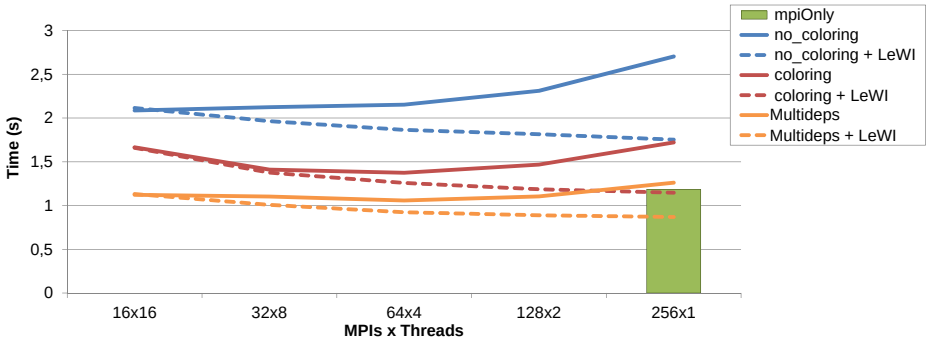


(b) 32 Nodes (512 cores)

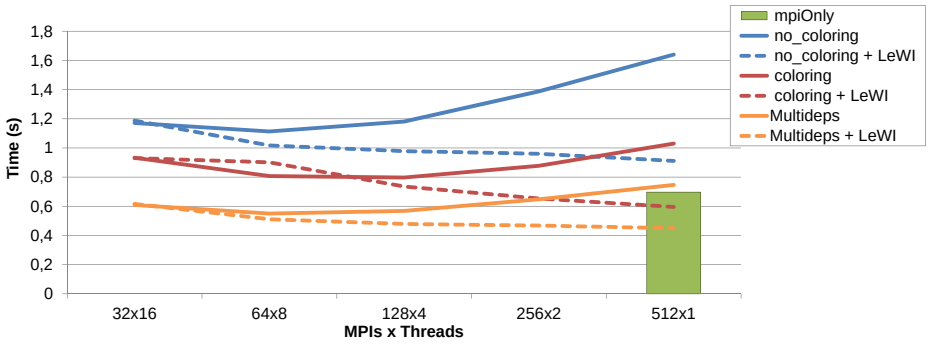


(c) 64 Nodes (1024 cores)

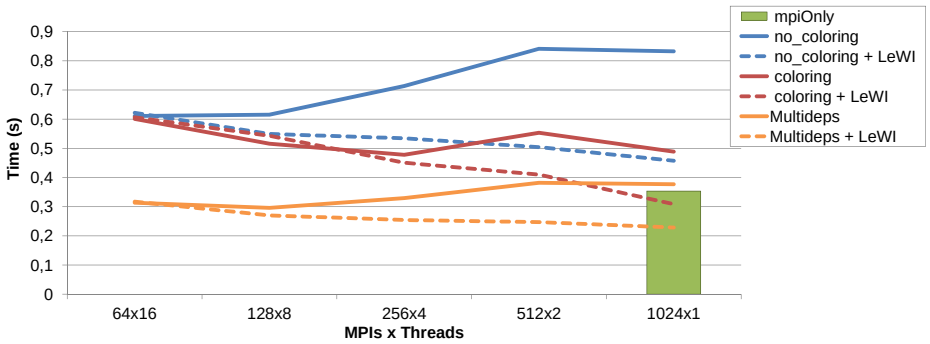
Figure 9.9: Performance of Matrix Assembly in the Iter simulation depending on the chunk size



(a) 16 Nodes (256 cores)



(b) 32 Nodes (512 cores)



(c) 64 Nodes (1024 cores)

Figure 9.10: Performance of Matrix Assembly in the respiratory simulation

All the versions present a worse imbalance when increasing the number of MPI processes per node (and decreasing the number of threads) this is because the load imbalance increases with the number of MPI processes (see Figure 9.1(a)). Except in the case of just one MPI process per node and 16 threads, where the threads may access memory belonging to the other socket of the node and these data accesses are slower. When using 8, 4 or 2 MPI processes per node, each MPI process is pinned to one of the sockets of the node. Therefore, all the data accesses will be to the local memory of the socket.

When using LeWI, we can see how the performance is improved respect the same execution without LeWI. Although the performance of the parallelization affects LeWI, in some cases, the load balance can overcome the overheads of the parallelization and obtain a better performance than the pure MPI version.

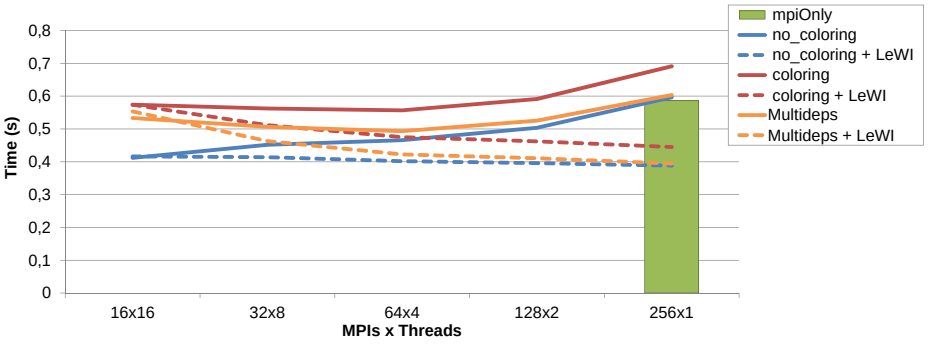
It is interesting to see how the performance of LeWI improves with the number of MPI processes on the node, being the best configuration to fill the nodes with MPI processes and only one thread for OpenMP/OmpSs. This can be explained because having more MPI processes gives LeWI more flexibility to load balance. i.e. if we use 2 MPI processes per node configuration, the load balance can only be applied to the two MPI processes running on the same node.

In all the cases the best situation is to use the commutative Multidependences with LeWI, which can represent a 37% faster execution than the pure MPI version.

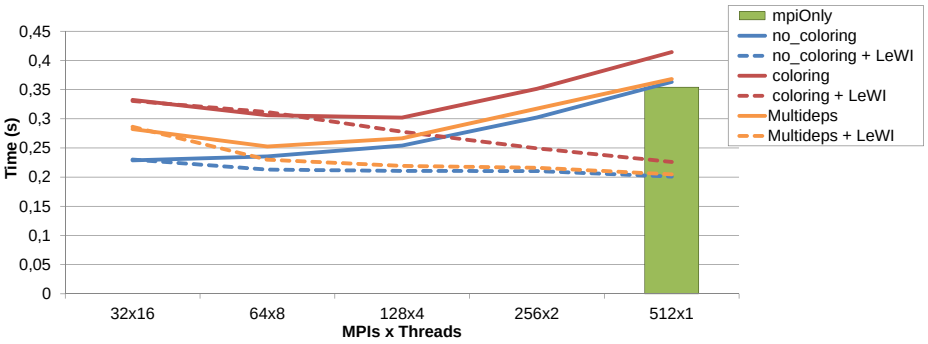
In Figure 9.11 we can see the execution time of the Subgrid Scale for the same experiments. The main difference that we can observe is that the performance of the No Coloring version is almost the same as the Multidependences, because in the subgrid scale the `ATOMIC` clause is not necessary.

The other conclusions are the same as for the matrix assembly. When using LeWI, the performance is improved in all the cases. The best configuration with LeWI is to use 16 MPI processes per node and one thread per process. In this case, when running in 64 nodes, the version of Multidependences with LeWI is 44% faster than the pure MPI version.

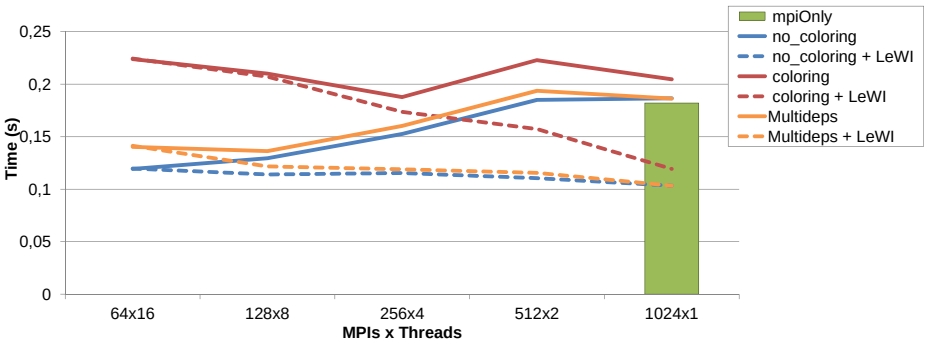
In Figure 9.12 we can see the average execution time of the matrix assembly phase in the *Iter* simulation. The different charts correspond to executions in 16, 32 and 64 nodes. In the X axis, we can see the different



(a) 16 Nodes (256 cores)

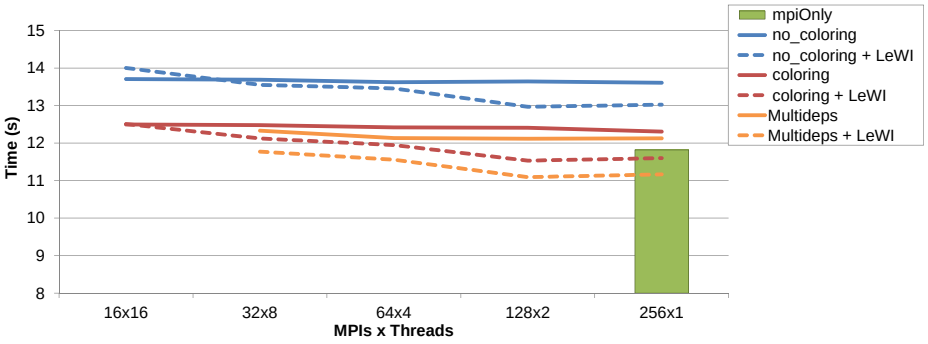


(b) 32 Nodes (512 cores)

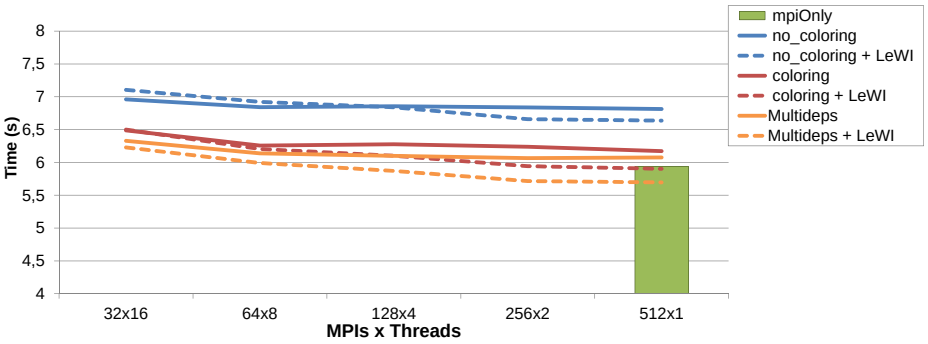


(c) 64 Nodes (1024 cores)

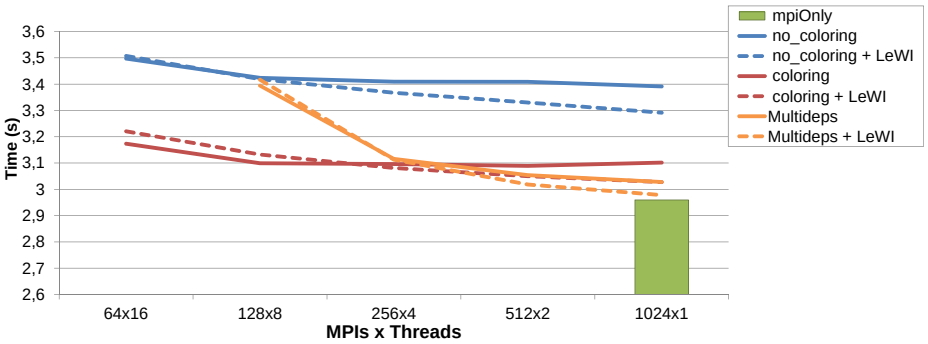
Figure 9.11: Performance of Subgrid Scale in the respiratory simulation



(a) 16 Nodes (256 cores)



(b) 32 Nodes (512 cores)



(c) 64 Nodes (1024 cores)

Figure 9.12: Performance of Matrix Assembly in the Iter Simulation

configurations of MPI processes and OpenMP/OmpSs threads. As we have seen in Figure 9.1(b) the imbalance in this input is not very high. Therefore, the performance improvement that we can expect with LeWI will not be as high as the ones obtained in the *Respiratory Simulation*. Note that in this case; we have increased the scale of the X axis in order to discern better the different values.

When comparing the performance of the different parallelizations, we can see that the No Coloring version is much slower that Coloring and Multidependences because of the `ATOMIC` overhead. In this case, the difference between the Coloring and Multidependences is not very significant because in this simulation the amount of computation per element is higher than when solving the fluid (*Respiratory system*). This means that the data locality has less impact on the overall performance.

The simulations executed with LeWI are always faster than their analogous ones. The better performance is obtained when using 16 MPI processes per node and one thread per MPI process. When using LeWI with Multidependences and 16 MPI processes per node, the performance is better that the pure MPI version (around 5% improvement).

IPC Study

In this subsection, we are going to see a summary of the IPC obtained by the different parallelizations (No Coloring, Coloring, and Multidependences) to support some of the performance explanations we have used in the previous experiments.

The data shown in the following charts have been obtained with Paraver from an Extrae trace of a real execution. We have measured the IPC obtained during the Matrix Assembly and the Subgrid Scale respectively.

In Figure 9.13 We can see the IPC in the *Respiratory* simulation during the assembly phase. In the X axis, we can see the percentage of time that presented the corresponding IPC respect the total CPU time spent in the matrix assembly. We can see that the pure MPI version had an IPC between 2.1 and 2.3. When using the No Coloring version of the parallelization the IPC went down to 1.1. This matches the previous performance results were the No Coloring version was two times slower than the pure MPI.

The IPC of the Coloring version is between 1.5 and 1.7, better than the No Coloring but still far from the IPC obtained by the pure MPI version.

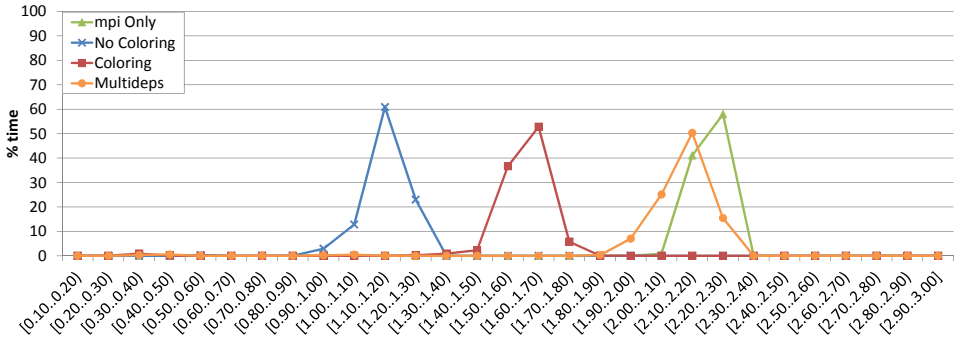


Figure 9.13: IPC at Matrix Assembly for the respiratory simulation

When using the Multidependences parallelization, the IPC is between 2 and 2.2 almost the same as the one achieved by the pure MPI version.

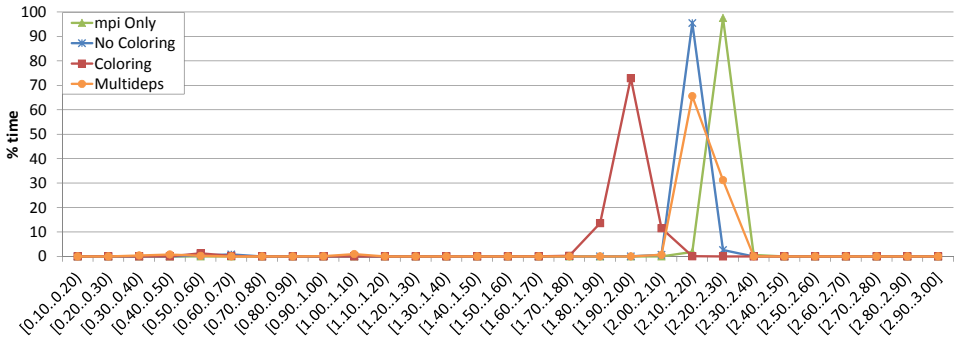


Figure 9.14: IPC at Subgrid Scale for the respiratory simulation

In Figure 9.14 we can see the IPC in the subgrid scale phase. The IPC for the pure MPI and the Multidependences versions are the same as in the matrix assembly. The No Coloring version in this phase presents a much higher IPC because it does not need the `ATOMIC` clause. Obtaining IPC equivalent to the one achieved with Multidependences. On the other hand, the Coloring parallelization has a worse IPC than the others because of the loss in data locality. But we can see that the performance loss is not as important as in the matrix assembly phase, this is because in the Subgrid

Scale the pressure over the memory is not as high as in the matrix assembly phase.

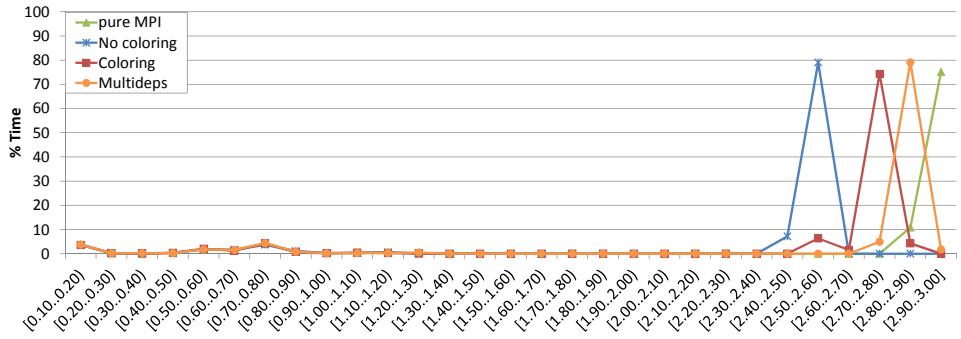


Figure 9.15: IPC at Matrix Assembly for the Iter simulation

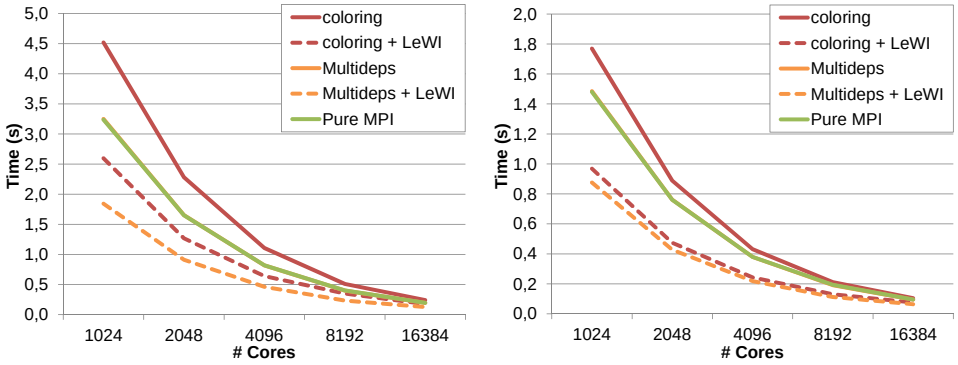
Figure 9.15 shows the IPC obtained in the matrix assembly during the *Iter* simulation. As we said before the solution of this problem has a higher computational load per element, and this can be seen in the higher IPC in all the cases. The pure MPI version has an IPC around 3 almost during the whole phase. The No Coloring version goes down to an IPC of 2.5 but still far from dividing the IPC by two that we observed in the *Respiratory* simulation. Again, this confirms that there is more computation going on, and the impact of the `ATOMIC` clause is not as high as in the other case.

The Coloring parallelization presents an IPC of 2.7 because of the worst data locality, and the Multidependences version obtains an IPC of 2.9 achieving almost the same performance as the pure MPI version.

Scalability

During this evaluation, we had the opportunity to run some strong scalability tests in Marenostrum3 with up to 16384 cores (1024 nodes). In these experiments, we want to demonstrate that DLB and LeWI can scale up to using thousands of cores. But also that even working at the node level the use of DLB can help improve the performance significantly in this kind of executions.

In these executions we have simulated the *Respiratory System* using the best configurations observed in the previous experiments, 16 MPI processes per node with 1 thread per process and chunks of 200 elements.



(a) Matrix Assembly

(b) Subgrid Scale

Figure 9.16: Execution time of Respiratory Simulation up to 16k cores

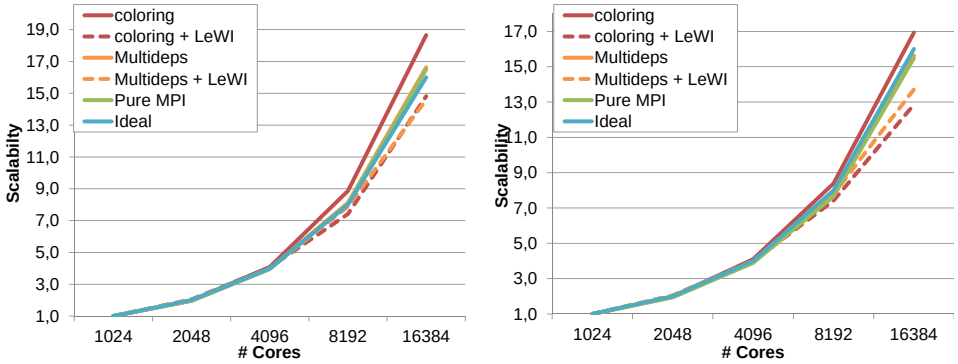
In Figure 9.16 we can see the execution time of the matrix assembly and the subgrid scale. In the X axis, we can see the number of cores used to run and in the Y axis the elapsed time in seconds. We can see how using LeWI with Multidependences or Coloring can reduce the execution time significantly. The Multidependences version shows the same performance as the pure MPI version.

In Figure 9.17 we can see the scalability as is usually presented by application developers, using as base case the smallest number of resources used:

$$Scalability_x = \frac{elapsedTime_{1024}}{elapsedTime_x}$$

We want to show how misleading this metric can be, in this chart the best scalability is obtained by the Coloring version without LeWI. But this version is the one that gets the worst execution time. On the other hand, the executions with LeWI (both with Coloring and Multidependences) has a worse scalability curve, but a better execution time.

In Figure 9.18 we can see the speed up respect the pure MPI version in 1024 cores. In this case, all the versions are computed versus the same baseline scenario.

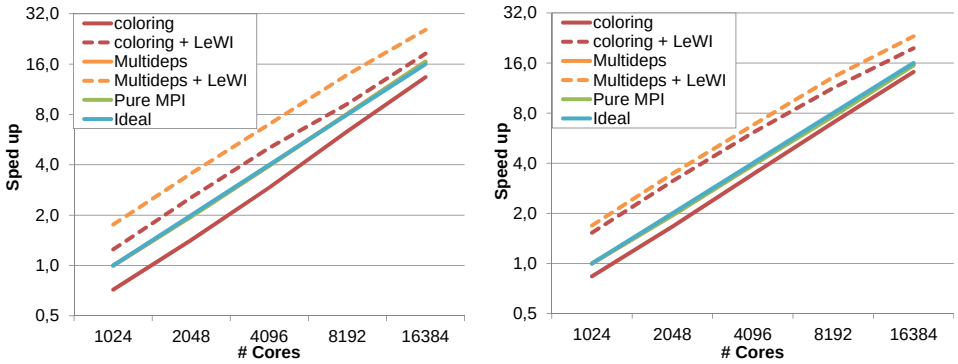


(a) Matrix Assembly (b) Subgrid Scale

Figure 9.17: Scalability of Respiratory Simulation up to 16k cores

$$Speedup_x = \frac{elapsedTimePureMPI_{1024}}{elapsedTime_x}$$

In this chart, we can see how the performance of the LeWI versions is significantly better even when running in a large number of cores. Being able to obtain a speed up of 23 when using 16 times more resources.



(a) Matrix Assembly (b) Subgrid Scale

Figure 9.18: Speed up of Respiratory Simulation up to 16k cores

9.3 Improving the Performance of Coupled Codes

In this section, we are going to evaluate the benefits of using our DLB library when running coupled codes. Coupling of codes is a common practice in HPC environments. In atmospheric modeling, for example, a code will simulate the ice, another one the earth and another one the oceans.

When a user needs to start a coupled simulation the first question that arises is: *How many processes/resources I should give to each code?* This is relevant to obtain a well balanced execution. But it is not easy to determine because it is a trial and error process. Moreover, the loads can change during the execution, and different configurations may be the optimum in different phases of the application.

In this context we will demonstrate how using DLB and LeWI can release the users of this decision. And can help to obtain an efficient simulation independently of the distribution of resources among the different codes.

9.3.1 Input simulation: *Sniff*

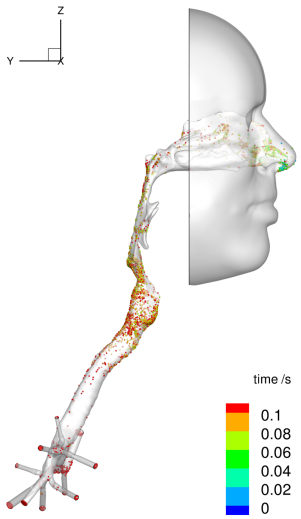


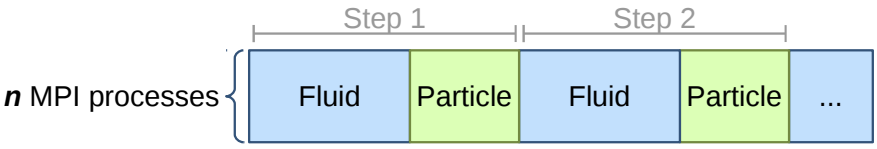
Figure 9.19: *Sniff* simulation

In this evaluation, we will use a simulation of the respiratory system when inhaling a medicinal spray [73]. The objective of this simulation is to track the particles after they are injected and inhaled in the nasal cavity through the airways.

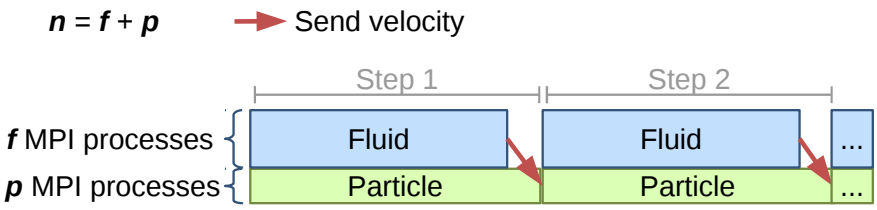
For this study, it is necessary to simulate the airflow when doing a rapid and short inhalation, called *Sniff*. The flow in the airways is simulated using the computational fluid dynamics. Lagrangian particle tracking is used to simulate the trajectory and deposition of the particles injected with the spray.

In this experiment the coupling is one-way, meaning that solving the fluid equations is necessary to know the velocity applied to the particles, but the particles have no effect on the fluid.

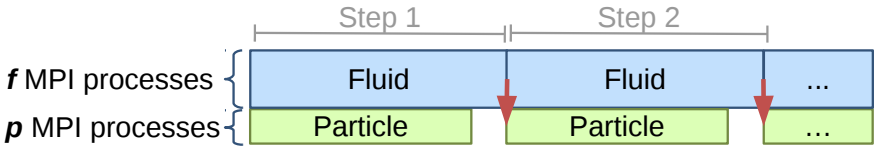
This simulation can be executed in a synchronous way using a single instance of Alya that uses all the resources available. When running the synchronous version, first, the fluid equations are solved. Once the velocity has computed, the transport of particles can be calculated, as can be seen in Figure 9.20(a).



(a) Synchronous execution



(b) Asynchronous execution particle dominated



(c) Asynchronous execution fluid dominated

Figure 9.20: Execution scenarios for the multi-physics problem: *Sniff*

When running a coupled version of the simulation two instances of Alya are used, one of them will solve the fluid and the other one the particle transport. The two codes will be able to run concurrently and when the fluid finishes will send the velocity to the particles. The available computational resources will be shared among the two instances of Alya.

Using an asynchronous version implies a different partition of the mesh for the two codes. In Figure 9.21 we can see the partition of the mesh obtained when running in 256 cores, the nodes represent the gravity center for each MPI partition and the lines the connectivity between the different partitions. When using the synchronous version, there is only one partition

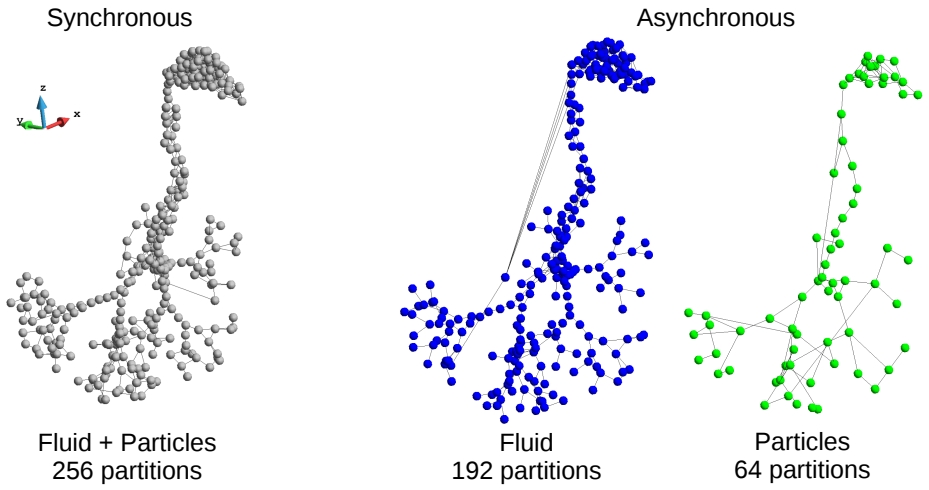


Figure 9.21: Partition of the mesh for 256 cores in the synchronous (256) and asynchronous (192+64) versions

for the two codes in 256 subdomains (left-hand side image). When using the asynchronous version the fluid code will partition the mesh in 192 subdomains and the particles code will have a different partition of 64 subdomains.

To evaluate all the possible situations we have considered two scenarios:

Injection of 0,5M Particles: In this scenario, the computational load of the fluid is higher than the one for the particles, we call it a fluid dominated execution. This means that the code solving the particle transport will wait for the fluid to finish and send the velocity, as can be seen in Figure 9.20(c)

Injection of 10M Particles: In this scenario, the computational load of the fluid is lower than the one for the particles, we call it a particle dominated execution. This means that the code solving the fluid will wait for the particle instance of Alya, as can be seen in Figure 9.20(b)

The mesh used contains 17M elements, and the whole simulation lasted 10.000 time-steps.

The load imbalance of the fluid code in this mesh has been studied in detail in the previous section. In the particles code, we will see that the

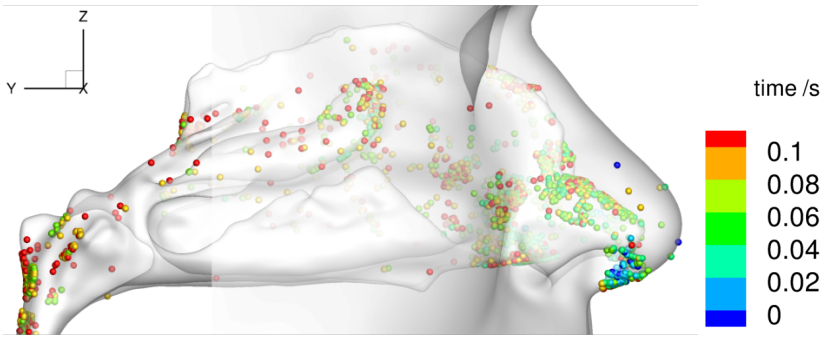


Figure 9.22: Particles injected in the nasal cavity

load imbalance is quite extreme because at the beginning of the simulation all the particles are injected into the nasal cavity. Therefore all the particles are concentrated in few MPI processes. In Figure 9.22 we can see an image extracted from the simulation where the color of each particle represent the instant in time in this position. In this image, we have a zoom of the nose, because all the particles are in this part of the system at the beginning of the simulation, although the mesh and simulation include the whole respiratory system.

Moreover, during the simulation the particles are moving through the respiratory system and crossing MPI subdomains, this means that the load balance will change during the simulation. Even if an ideal distribution of MPI processes between fluids and particles was found it would constantly change during the execution.

9.3.2 Applying DLB to the *Sniff* Simulation

The use of DLB in this simulation will solve 3 load balancing problems and independent between them: The load imbalance in the fluid, the load imbalance in the particles and the distribution of MPI processes between fluids and particles.

To apply DLB to the *Sniff* simulation we started from the production version of Alya with some minor modifications. In the fluid code, we have used the parallelization explained in the previous section with coloring (At

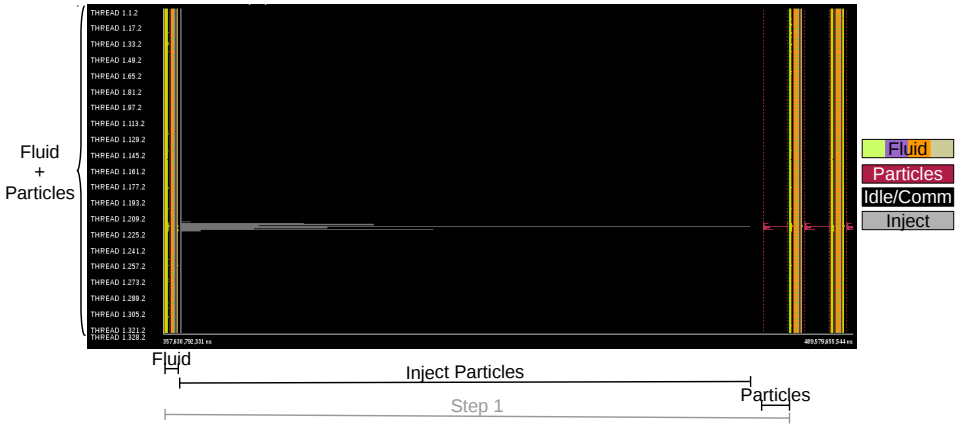
the moment of this evaluation the parallelization with commutative multidependencies was not available yet).

Algorithm 6 Algorithm for the parallel transport of particles.

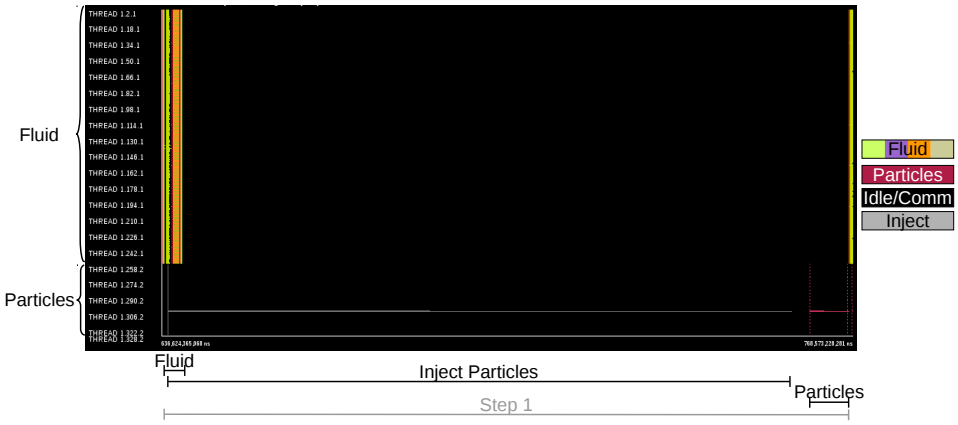
```
1: DLB_enable()
2: N_over = 0
3: while N_over ≠ N_p do
4:   !$OMP PARALLEL DO SCHEDULE (DYNAMIC,1000)
5:   !$OMP ...
6:   for Particles p in my subdomain do
7:     Compute particle p
8:     if particle p is in halo element then
9:       Save particle to be sent to neighbor
10:    else
11:      N_over = N_over + 1
12:    end if
13:  end for
14:  !$OMP END PARALLEL DO
15:  MPI_AllReduce(N_over)
16:  MPI_Send particles in halo elements to corresponding neighbors
17:  MPI_Recv particles from neighbors
18: end while
19: DLB_disable()
```

In the particles code, we added the OpenMP parallelization to two loops, the injection of particles into the system and the loop to compute the transport of particles. We also added calls to the DLB API to enable DLB in the parts of the code parallelized with OpenMP and disable it the remaining time. In Algorithm 6 we can see the pseudo-code for the loop over the particles and the calls to the DLB API added.

In Figure 9.23 we can see the first step of the simulation that includes the injection of particles. The injection of particles is done only at the beginning of the simulation, and compared with the full simulation that lasts 50.000 time steps we can consider it negligible. But, on the other hand, when injecting 10M of particles it took 2 minutes on 656 cores, for this reason, we have parallelized the injection of particles and enabled DLB.

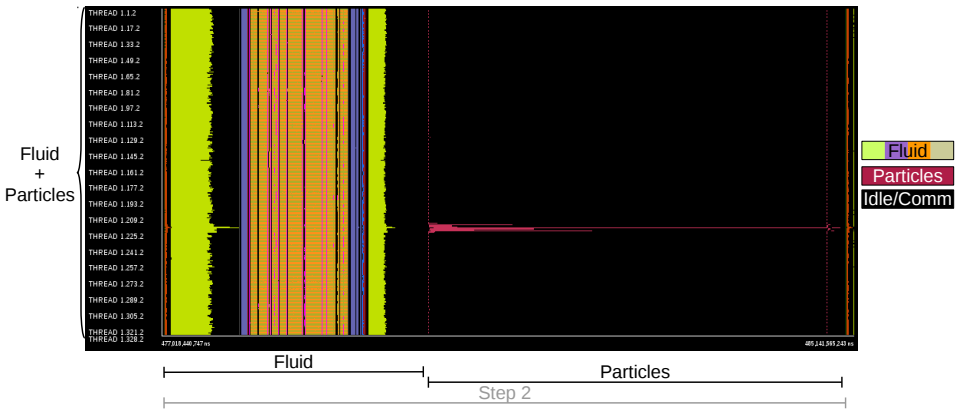


(a) Synchronous execution

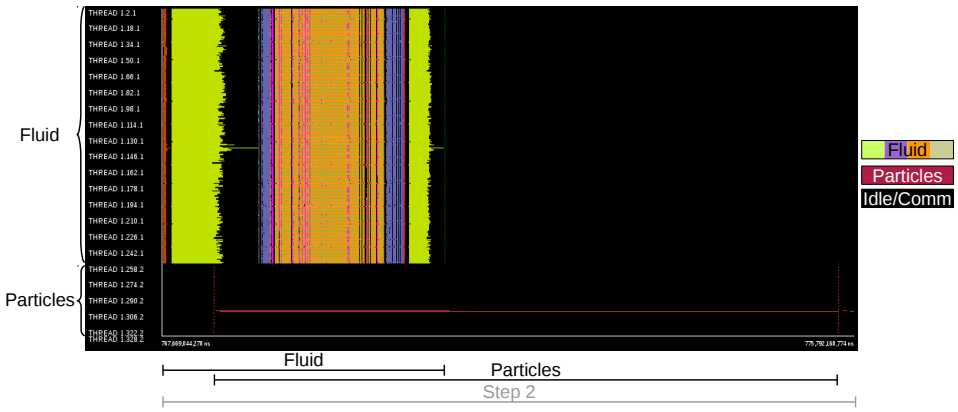


(b) Asynchronous execution

Figure 9.23: Trace of the first step of the *Sniff* with 10M of particles



(a) Synchronous execution



(b) Asynchronous execution

Figure 9.24: Trace of the second step of the *Sniff* with 10M of particles

In Figure 9.24 we can see a trace of the second step of the *Sniff* simulation, this simulation is running in 328 MPI process with 2 OpenMP threads each one. In Subfigure 9.24(a) we can observe one time step of the synchronous version. In this case first is solved the fluid in the 656 cores, and afterward the particles in the same resources with the same domain partition. In Subfigure 9.24(b) we can see the asynchronous execution of a time step using 256 MPI processes for the fluid and 72 MPI processes for particles. In this version, the fluids and particles are running in parallel.

In Figure 9.25 we can see two traces, in these traces, we can observe the detail of one of the nodes (the more loaded one, therefore, the bottleneck). The traces include one time step of the asynchronous execution. In Subfigure 9.25(a) we can observe how the computation of fluids and particles are overlapped we can also see some amount of load imbalance at the fluid code. In this node we find two processes computing the particles (with 2 OpenMP threads each one), but only one of them have particles to compute, the other one is idle all the time (black).

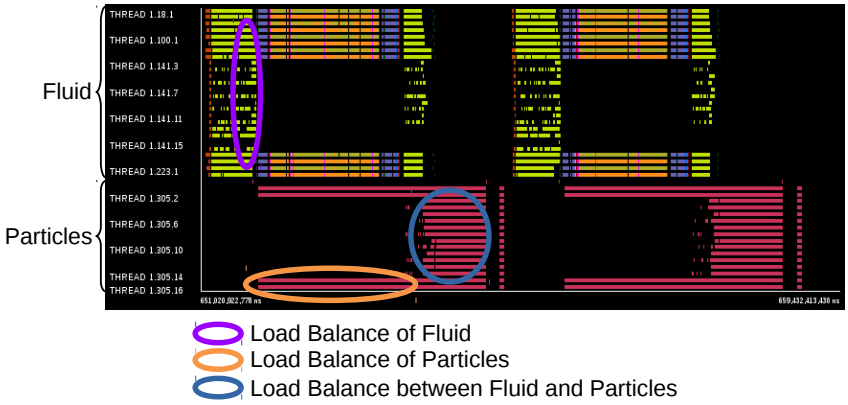
In Subfigure 9.25(b) we can see the trace of the same node when using DLB. In this trace, we can identify the three situations where DLB acts to improve the performance. In a purple circle, we can see how the load imbalance at the fluids code is solved. Inside the orange circle, we can see how the process of particles that does not have particles to track, lends its cores to the process with particles in the same node (enabling an execution with 4 OpenMP threads). And finally in the blue circle, we can see how the process computing the particles can use the cores from the fluids processes to run with up to 16 OpenMP threads.

9.3.3 Environment and Methodology

All the experiments of this section have been executed in Marenostrum3. The nodes of Marenostrum3 are based in SandyBridge with 2 sockets of 8 cores each one and 32 Gb of shared memory.

The Fortran compiler used is Intel 13.0.1. For MPI we have used the Intel MPI library version 4.1.3.049. The OpenMP runtime used is Nanos 0.12a, this allows us to exploit the integration of DLB with the Nanos runtime and use the *autonomous threads* version of the *LeWI* policy (See Section 6.2 and Chapter 8).

(a) Asynchronous execution



(b) Asynchronous execution with DLB

Figure 9.25: Trace of the bottleneck node when running 256 processes for the fluid and 72 for particles

For all the simulations, the MPI processes are started with one OpenMP thread. This means that OpenMP is exclusively used for load balance with LeWI.

The MPI processes have been distributed in a Round Robin scheme among the computational nodes; this allows to have processes solving the fluid and processes solving the particles mixed in all the nodes.

For this evaluation several numerical and computational configurations have been considered:

- **Synchronous *vs* asynchronous:** We will compare the performance of the synchronous version with the asynchronous one. When running the synchronous code, we will fill the nodes with MPI processes (i.e. 16 MPI ranks per node). For the asynchronous executions, we have used four different proportions of MPI processes between the two codes (fluid and particle):

  - **8 + 8:** Starting 8 MPI processes for the fluid and 8 MPI processes for particles on each node.
  - **12 + 4:** Starting 12 MPI processes for the fluid and 4 MPI processes for particles on each node.
  - **14 + 2:** Starting 14 MPI processes for the fluid and 2 MPI processes for particles on each node.
  - **15 + 1:** Starting 15 MPI processes for the fluid and 1 MPI proces for particles on each node.

  Table 9.3.3 shows a summary of the total number of MPI processes used in each code for all the possible configurations.

- **Original *vs* DLB:** All the experiments have been executed with and without DLB. The executions without DLB are labeled *Original*. The results obtained with DLB using LeWI, binding of threads to cores and autonomous threads are labeled as *LeWI + Mask + Auto*.

- **16, 32, 64 computing nodes:** We will present the performance results obtained on 16, 32, 64 computing nodes of Marenostrum3 that correspond to 256, 512 and 1024 cores respectively.

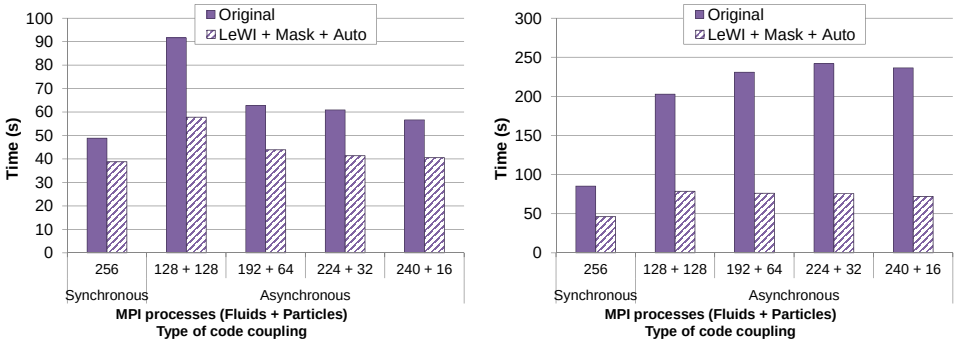
Code	Per node	16 nodes	32 nodes	64 nodes
Sync	16	256	512	1024
Asynch	fluid + part	fluid + part	fluid + part	fluid + part
	8 + 8	128 + 128	256 + 256	512 + 512
	12 + 4	192 + 64	384 + 128	768 + 256
	14 + 2	224 + 32	448 + 64	896 + 128
	15 + 1	240 + 16	480 + 32	960 + 64

Table 9.1: Distribution of MPI processes for fluid+particle, per node and in total.

- **Amount of particles:** We have considered two numbers of particles, 0.5 and 10 million. This will allow us to compare the performance in a fluid dominating and particle dominating situation.
  - ⋆ 0.5M particles → Fluid dominates
  - ⋆ 10M particles → Particle dominates
- **Execution of 10 time steps:** We have executed 10 time steps starting with the particle injection in the vestibule of the nasal cavity. In the performance results, we will present the average execution time of 10 time steps excluding the first one that includes the injection of particles. And the total execution time of the first 10 time steps, in this case, the injection of particles is included.

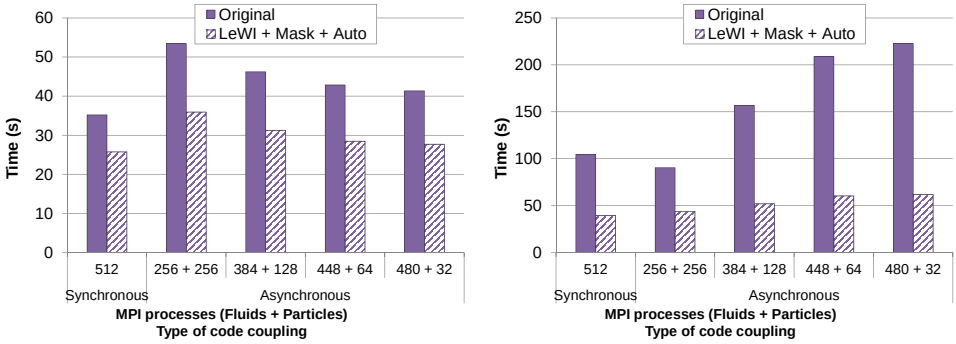
9.3.4 Performance Evaluation

In Figure 9.26, 9.27 and 9.28 we can see the average execution times for the first 10 time steps excluding the injection of particles in 16, 32 and 64 nodes respectively. The charts on the left-hand side correspond to the execution that is *fluid dominated* (i.e. injection of 0.5M particles), the right-hand charts present the results of the *particle dominated* scenario where 10M particles are injected. In the Y axis, we can see the average time step in seconds and in the X axis the different configurations of MPI processes among the two codes and the synchronous or asynchronous execution. The solid bars represent the execution of the original code and the stripped bars the same execution when using DLB with LeWI.



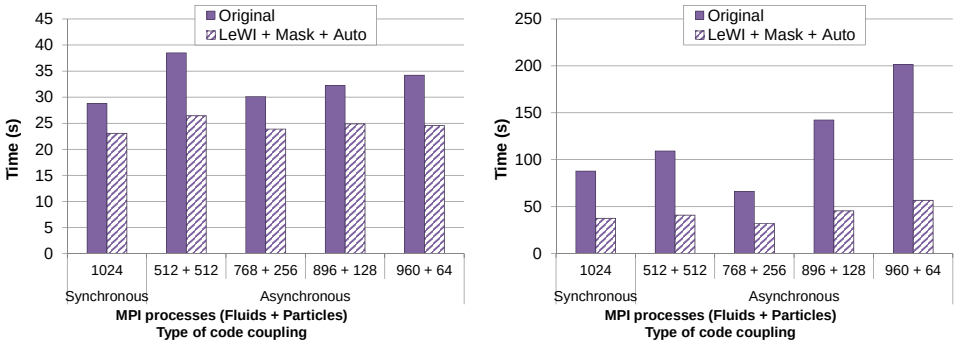
(a) 0,5 Millions of particles (b) 10 Millions of particles

Figure 9.26: Average time step duration of *Sniff* in 16 nodes



(a) 0,5 Millions of particles (b) 10 Millions of particles

Figure 9.27: Average time step duration of *Sniff* in 32 nodes



(a) 0,5 Millions of particles (b) 10 Millions of particles

Figure 9.28: Average time step duration of *Sniff* in 64 nodes

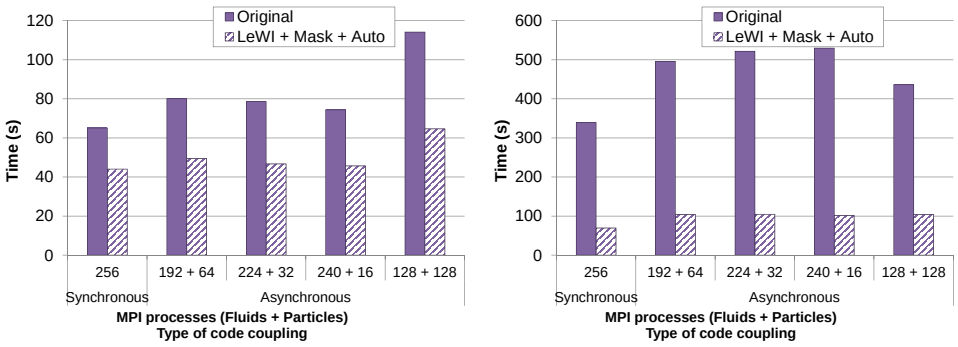
We can observe that the performance of the original execution, when the computation is dominated by the fluid (0,5M particles, left-hand side charts), differs from the one obtained by the particle dominated execution (10M particles, right-hand side of the figure). This is because the fluid and the particle codes have different scalability behaviors, while the fluid code scales quite well [74], the particle code has a poor scalability due to the high load imbalance. Therefore, the global scalability highly depends on which code dominates the computation.

We can see how the executions with LeWI improve the performance in all the individual situations.

When analyzing the performance of the asynchronous version, we can see that it depends on the distribution of MPI processes among the fluids and the particles codes. Almost in all the cases we can find a configuration of MPI processes that perform better that the synchronous version. But at the same time when choosing any of the other configurations, we can see how the performance drops, for some of the configurations the asynchronous code with a bad distribution can be two times slower than the synchronous one.

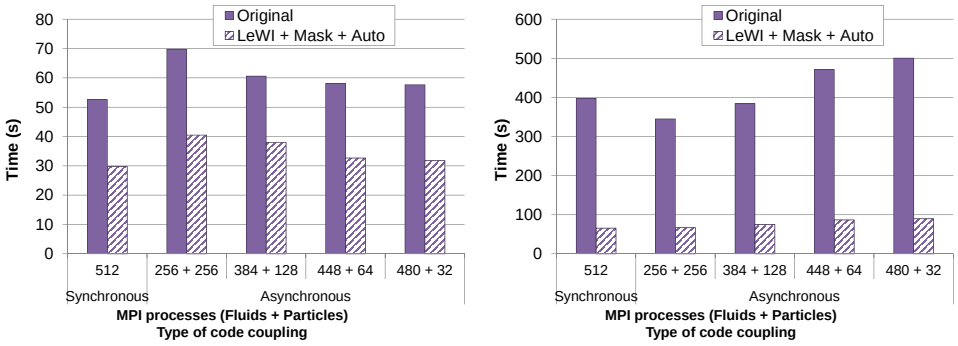
We can observe that using DLB and LeWI can hide the performance problem when using an inadequate distribution of MPI processes among the codes (fluids and particles). The execution time when using LeWI is almost constant independently of which version we are running (synchronous or asynchronous) and how many MPI processes for fluids and particles we

are running. This means that LeWI can manage the load balance between the two codes independently of the initial distribution of MPI processes.



(a) 0,5 Millions of particles (b) 10 Millions of particles

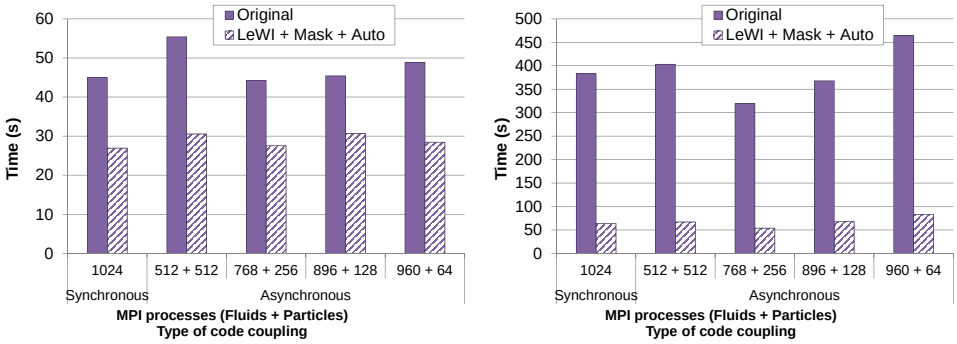
Figure 9.29: Execution time of 10 time steps of *Sniff* in 16 nodes



(a) 0,5 Millions of particles (b) 10 Millions of particles

Figure 9.30: Execution time of 10 time steps of *Sniff* in 32 nodes

In Figures 9.29, 9.30 and 9.31 we can see the execution time of 10 steps including the injection of particles. Although, in this experiment, the injection of particles is only performed once and the whole simulation can run for over 10.000 time steps. Nevertheless, in other simulations can happen more than once.



(a) 0,5 Millions of particles

(b) 10 Millions of particles

Figure 9.31: Execution time of 10 time steps of *Sniff* in 64 nodes

We can see that in the *particle dominated* executions (right-hand side charts) the impact of the particle injection is very high, and using LeWI can improve the performance extremely. Moreover, LeWI can maintain a regular performance across the different distribution of MPI processes among the two codes, or even when using the synchronous version.

In the *fluid dominated* simulation (left-hand side charts) LeWI also hides the imbalance between the codes, obtaining a similar performance with the different configurations of processes.

9.4 Conclusions for Alya

In this chapter, we have demonstrated that DLB and LeWI are ready to be used in production codes with minimum effort.

We have evaluated the use of LeWI to load balance an HPCM (High performance Computational Mechanics) code when solving two different problems: A computational fluid dynamics and a solid mechanics. One of them with a high load imbalance and the other with a reasonable good load balance. The use of LeWI can improve the performance when a high load imbalance is present and speed up the execution up to a 44% faster (in the same number of resources). At the same time, it does not penalize the execution of a well balanced simulation.

We have evaluated the impact of the shared memory parallelization in the performance of DLB and LeWI. In some cases, we have seen that LeWI can overcome the overhead of the parallelization and obtain a better performance than the pure MPI parallelization.

In general, the best configuration to use LeWI is to use one thread per MPI process, using the second level of parallelism only to load balance. Even though, this implies a higher number of partitions, which usually present a higher load imbalance.

We have also analyzed the impact of the chunk size on the performance of LeWI. We have observed that the chunk size can limit the benefits of using LeWI, big chunk sizes imply less malleability to change the number of threads and too big chunk sizes can even disable the effect of using DLB. On the other hand, extremely small chunks can penalize the OpenMP/OmpSs parallelization. Nevertheless, the range of size chunks in which all the versions obtain an optimum performance is very broad.

In this chapter, we have tested the performance of LeWI when scaling up to thousands of cores. We have demonstrated not only, that LeWI can scale up to 16k cores. But also, that even working only within the node it can improve the performance of this kind of executions.

Finally, we have evaluated the use of LeWI when running coupled codes. In this kind of executions, the decision of how many resources to dedicate to each code can have a significant impact in the performance. We have seen that the use of LeWI can alleviate this decision. When using LeWI we can achieve a similar performance in all the configurations, even when selecting the worst ones.

We have seen that when simulating 10M of particles DLB with LeWI can make the execution between a 45 and 72% faster (speed up between 1.8 and 3.5 with the same number of resources). When running with 0.5M of particles, LeWI can run between 20 and 32% faster (speed up between 1.2 and 1.5 of the original code).

Chapter 10

Conclusions and Future Work

10.1 Conclusions

In this thesis we approach the load imbalance problem. Our main objective is to minimize its impact in the performance of applications and in the efficient use of computational resources.

We have seen that load imbalance is a dynamic problem. It can have very different sources and can change with the system, the software or the execution environment. For this reason, it can not be solved a priori and a dynamic solution applied at runtime is necessary.

The solution we propose is used at runtime, through a library that can be loaded dynamically and without modifying the source code of the application. It can react to load imbalance coming from any source, independently of the system, the code or the input of the execution. This have been developed within DLB: Dynamic Load Balancing library.

The DLB library includes a novel load balancing algorithm: LeWI (Lend When Idle). The main idea of LeWI is to use computational resources assigned to processes or threads that are idle, to other processes that are doing useful computation. For example, in the case of an MPI+OpenMP application, when a processes reaches an MPI blocking call it will lend its computational resources to another process running in the same node to speed up its execution.

We have evaluated the performance of DLB and LeWI in different scenarios. From these evaluations we have seen that DLB and LeWI can solve load imbalances from different sources. We have als learned the following general conclusions.

The software stack in HPC systems have several layers that traditionally do not work together or cooperate, we have observed the importance of the cooperation between these layers to obtain an efficient use of the computational resources. Our library interacts with several layers of the HPC software stack and tries to coordinate them.

We have seen that *Malleability* (the capacity to change or adapt during the execution) is crucial to improving the efficiency of HPC systems at runtime. Malleability is necessary for programming models not only to allow a change in the amount of resources used but also to do it with enough frequency. Malleability must be present also in applications; it is important to avoid programming practices that are dependent on the number of resources (processes or threads), i.e. allocate memory structures depending on the number of threads running.

The original objective of DLB was to be an entirely transparent solution to the application, where it was not necessary a previous analysis of performance nor to modify the source code. We achieved this objective in the first version and evaluation of the library using the interposition technique of MPI and the standard API from OpenMP the change the number of threads.

We also observed that the application developers have a knowledge of their application that can be useful to give some hints to the DLB library. With this information, DLB will be able to make a more efficient use of the resources. Therefore, we trade-off some transparency for better performance.

In the evaluations that we have performed in general the best configuration of MPI processes per threads when using DLB and LeWI is to fill the nodes with MPI processes and use only one thread per MPI process. With this configuration, the second level of parallelism is used only to load balance the MPI level. For application developers, this can be very interesting, because having a full parallelization with OpenMP (or OmpSs) is not always trivial or possible. Using this approach not only allows a progressive parallelization but also to parallelize only the parts of the code where the load imbalance can be improved.

10.2 Contributions

We have presented DLB as a framework to improve the use of resources at the node level. We have demonstrated that the DLB framework is stable using it in production runs with Alya, an HPC computational mechanics code. We have also proved that it scales up to thousands of cores in Marenostrum3.

The DLB framework coordinates and interacts with several layers of the HPC software stack. The current version supports OpenMP, OmpSs and MPI programming models and is ready to be extended to other programming models or runtime libraries.

DLB can be used transparently to the application, without modifying it, but it also offers and API that can be used to give hints from the application.

Within the DLB framework we have implemented two load balancing algorithms, the first one based on related works and our novel algorithm: LeWI, Lend when Idle.

LeWI reacts to the load imbalance at runtime and does not need previous information, either from other executions or previous iterations. For this reason, it can respond to load imbalances coming from any source.

We have seen that LeWI can improve the performance of applications independently of the decisions taken by the user when choosing the distribution of MPI processes or configuration of MPI processes and threads per node.

One of the strengths of LeWI is that it can load balance very fine granularities. But, to handle fine grain imbalances it needs a fine grain malleability both from the programming model and the application.

We have integrated DLB and LeWI with a parallel runtime library: Nanos++. This integration has allowed us to define, develop and test an API that now is ready to be used by other runtime libraries. We have also seen the potential of collaborating with the different layers involved in the performance of HPC applications.

Finally, we have evaluated the use of DLB and LeWI with a production code, Alya, when solving relevant scientific problems in an HPC environment. With Alya, we have seen that LeWI can be improve significantly the performance of real applications and has no problem scaling up to 16.000 cores (In Marenostrum3).

The evaluation with Alya has presented a new scenario where DLB can help HPC developers and users, the coupled executions. Coupled executions are common in HPC environments where different applications collaborate to address a problem, in this case, the developers must decide the amount of resources given to each application. Developers can make a blind decision or try different configurations to find an optimum one. The use of DLB and LeWI can alleviate this decision, as the resources can be shared when necessary. Moreover, we have seen that the performance of coupled executions can be maintained constant independently of the distribution decided by the user when using DLB and LeWI.

10.3 Open research topics

We have several research topics related to the DLB framework that we plan to develop in the future and some of them that are already in development.

New modules: DROM and Stats The DLB framework will be divided into three independent modules but compatible between them (Figure 10.1). The micro-Load Balancing module (μLB) is the main core of this thesis and is in charge of solving load imbalances at very fine granularities.

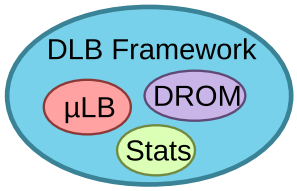


Figure 10.1: DLB framework and its modules

The *DROM* (Dynamic Resource Ownership Management) module, offers the functionality of changing the assignment of resources by changing the ownership of cores. This module has been already implemented and currently we are testing and evaluating it. The *Stats* module collects different metrics about the performance and provides and API to check them.

These modules are designed to be used by other entities that will be in charge of taking the corresponding decisions. For example, the job scheduler can ask to the *Stats* module if a node is underutilized and decide to reduce the number of resources assigned to one of the processes to start a new application in the same node. It can reduce the resources of a running process using the *DROM* module.

The first draft of the proposed API for DROM is the following:

- `void DLB_Drom_Init(void):`
  Allows a process to enter the system to be able to use the DROM API. Note that this function is for the resource manager, the processes managed by the system will register into the system through the DLB_Init call explained in Section 5.5.2.

- `void DLB_Drom_Finalize(void):`
  Finalize the use of the DROM module by the resource manager.

- `int DLB_Drom_GetProcessMask(int pid, dlb_cpu_set_t mask):`
  Obtain the current mask of owned resources for a given process.

- `int DLB_Drom_SetProcessMask(int pid, const_dlb_cpu_set_t mask):`
  Change the mask of owned resources of a given process. This call is synchronous, meaning that it will not return until the target process effectively applies the new mask.

Integration with Resource Manager: Slurm Through the API offered by the new modules DROM and Stats, we plan to integrate DLB with a resource manager. In this case, we will be working with Slurm [75] [76] because it is open source and can be easily extended through plugins.

This integration will allow Slurm to obtain information about the performance of each node through the Stats API. With this information, Slurm can take decisions during the execution based on the load of the node or the performance of a particular process.

The DROM API offers an interface to Slurm to change the resources used by any running process. This functionality allows several scenarios:

- A user wants to send a second application in the resources previously allocated to another application. For example, when running a simulation that can use hundreds of cores during several hours or even days, the user may want to analyze some of the results to see the progress of the simulation. Instead of requesting for more resources he can launch the analysis application in the same resources as the simulation was running. The resources used by the simulation will be reduced while the analysis is running. When the analysis finishes, the simulation can resume using all the resources.

- Slurm detects that a node is being underutilized, and decides to reduce the number of resources assigned to the processes running on that node. In these freed resources Slurm can spawn a new process or application if needed.

This integration is already in development and testing and the next step will be to evaluate its potential.

Port to many-core system: KNL We have seen that DLB can scale up to a thousand of nodes of 16 cores per node. The next step is to port DLB to a many-core system, like for example KNL from the Intel Xeon Phi family. This will allow DLB to have more flexibility to load balance because of the higher number of cores. It will allow testing more configurations of MPI processes per node and threads per process.

At the same time, we will stress DLB with a high number of threads accessing the system. We will evaluate if DLB can support this pressure or if there is a performance problem. With more cores available it might be interesting to evaluate more elaborated policies to decide to which process to lend the core.

Extend Load Balancing Policies We want to implement other load balancing policies. The main idea of lending the cores when not using them would be the same, but the decision of which process get the lent core can be based on different policies.

We can take into account if the user is the same, or prioritize lending cores to processes belonging to the same application. To implement this kind of policies, we must keep information inside DLB about the user and the application to which each process belong. And maybe we can apply some accounting to ensure that the resources are shared in a fair way among the different applications or users.

DLB Node Barrier In order to use the resources of a process while waiting for an MPI call, we need to set the MPI runtime wait mode as non-polling. We have observed that setting this mode can delay the end of MPI calls. In some applications or phases with a high number of communications, these can degrade the performance significantly.

We have implemented a barrier in the shared memory of the node; this barrier does not consume CPU because the threads are sleeping in a semaphore. We can use this barrier directly using the API `DLB_Barrier()`.

We have a preliminary implementation of the DLB_Barrier that can only be used through the DLB API. We plan to evaluate its performance compared with the different blocking modes offered by MPI libraries. If the evaluation shows that DLB_Barrier has potential we will add its use transparently to the user. We will do this using the PMPI interposition, we can use the DLB_Barrier before each call to a collective MPI call. This implementation would allow dividing the collective MPI calls into two phases a first non-CPU consuming done inside the shared memory node, and a second step is done in a polling mode with the processes outside the node.

10.4 Outreach

The development of this thesis have produced several publications ([6] [7] [8] [9] [10] [11]).

The current stable version of DLB is 1.2. The code of DLB is open source with a LGPLv3 license. The DLB Framework is available as a software package to be used that can be downloaded from: `https://pm.bsc.es/dlb-downloads`.

The user guide of DLB can be found here: `https://pm.bsc.es/dlb-docs/user-guide/`

We have a public web page to disseminate DLB: `https://pm.bsc.es/dlb`. In it we can find general information about DLB and related links.

Appendix A

DWB Algorithm Limits

During the evaluation of this algorithm we detected an important limitation related to the granularity of the resources. I.e. in the experiments that we will show in the following sections we run in nodes with 4 cpus this means that the granularity that we can use to load balance is a 25%. Each time we move a cpu from one process to another we are moving the 25% of the resources.

In this section we are going to try to answer two questions:

- How many cpus per node do we need to balance an application?
- With X cpus, which is the % of Efficiency that we can expect of an application?

This study will tell us which is the maximum performance we can expect depending on the number of cpus that the node has available.

In this section, all the numbers are synthetic (i.e obtained with numerical computations not real executions). All the numbers in this section are obtained supposing that there are two MPI processes per node and the 100% of the code of the application is parallelized. In other words, the % of Efficiency shown in this section is the maximum that can be obtained with that number of processors.

To obtain the % of Efficiency and time we have first calculated the distribution of cpus in the same way as the ***redistribution of cpus model***

does, the number of cpus per process is directly proportional to the load of each process, and each process should get at least one cpu. The equation A.1 is the formula we have used to calculate the number of cpus that should be assigned to the least loaded process (i.e. its load is between 1% and 50%).

$$\text{Assigned procs} = Max\left(1, Round\left(\frac{\%load * total\_cpus}{100}\right)\right) \tag{A.1}$$

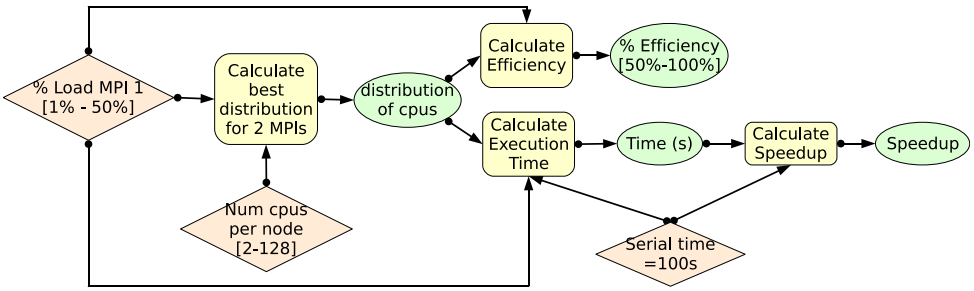


Figure A.1: Flow of data to calculate synthetically Efficiency, Time and Speed up

In Figure A.1 is shown a data flow that depicts the procedure to calculate the % of Efficiency, time and speed up synthetically. The red rhombus represent the input data and its possible values, the green circles represent the output data obtained, and the yellow squares the computation procedures needed.

One last clarification before examining the results obtained is that the imbalance of the application is represented as the percentage of load of the first MPI process. So in the charts will usually appear the imbalance from 1 to 50, an imbalance of 1 means that the first MPI process have the 1% of the load and the second one the 99% remaining, so represents a very imbalanced application (probably not a realistic situation). On the other hand an imbalance of 50 means that the first MPI process does the 50% of the computation and the second one the other 50% and represents a perfectly balanced application. The imbalance from 51 to 100 will not be shown because the information is redundant.

We have separated the section in subsections depending on the metric we are measuring at each time: % of Efficiency, time or speed up.

% of Efficiency

In this subsection we will measure the performance that is obtained by an application with a metric that we will call *% of Efficiency*. This metric represents the efficiency in the use of the resources as the percentage of time that the cpus are doing computation (not idle, blocked in a MPI call) respect the total execution time.

The formula used to compute the % of Efficiency is shown in Equation A.2.

$$\%\text{Efficiency} = \frac{\sum_{num\_cpus}^{x=1}(cpu\_time_x)}{Max_{num\_cpus}^{x=1}(cpu\_time_x) * num\_cpus} * 100 \qquad \text{(A.2)}$$

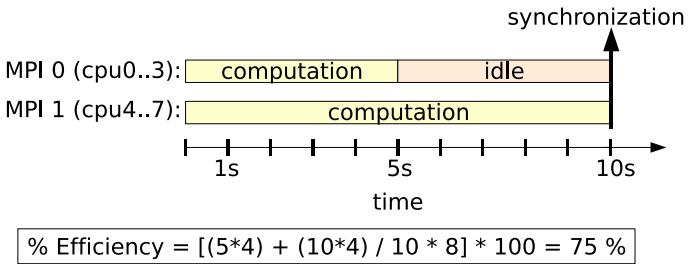


Figure A.2: Example of how to compute the % Efficiency

In Figure A.2 is shown an example on how is computed the % of Efficiency. In this example we suppose we have 8 cpus in the node (4 threads per MPI process) and presents an imbalance of 33%-67%. The first process is doing computation during 5 seconds, then waits to synchronize with the second one. The second process finishes the computation in 10 seconds. Therefor of a total of 80 (10 * 8) seconds of cpu time that they are using they are computing 60 (20 + 40) seconds. This is equivalent to be using the 75% of the time assigned.

When we try to balance an application what we target is to achieve a 100% of Efficiency, or what is the same that there are not processors idle waiting for the others.

As example or guideline in Figure A.3 is shown the % of Efficiency that can be obtained when running an application with equipartition and with different percentages of imbalance.

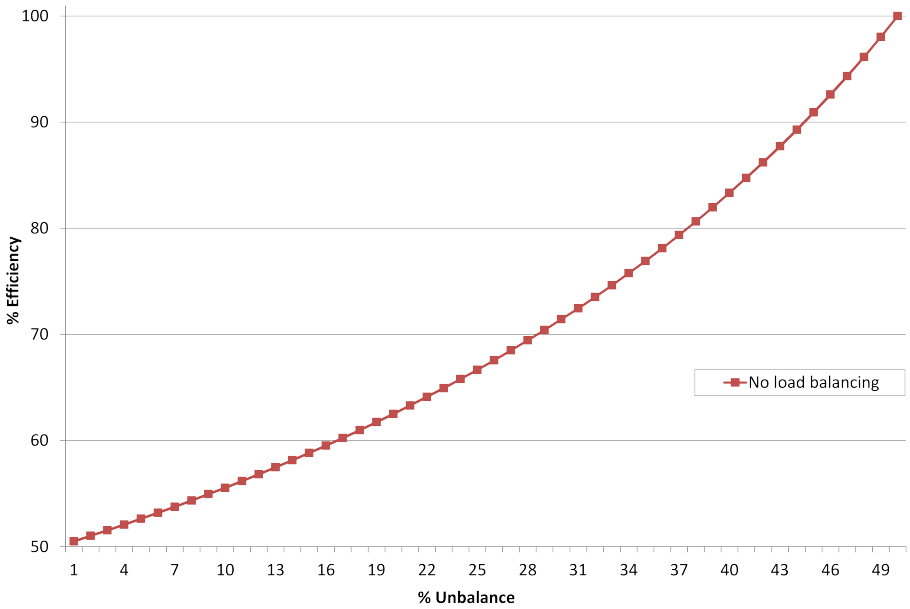


Figure A.3: Efficiency obtained without DLB

The average of % of Efficiency that can be obtained with different number of processors per node can be seen in Figure A.4. The number of cpus of the node appear in the X axis. Each blue bar represent the average % of Efficiency that can be obtained between the different values of imbalance. An average of 88,5% of Efficiency can be obtained with 8 cpus per node.

The green bars are the average excluding the values from 1 to 10 % of imbalance because we consider this cases as very extrem and wanted to see the difference it makes in the evaluation. With 8 processors per node we can achieve a 92,94% of Efficiency in average if we consider only applications with an imbalance lower than 90%-10%.

In Figure A.4 is represented also the standard deviation from the average. Is interesting to see that the more number of processors per node the less variance in the Efficiency obtained between the different percentage of imbalance. The standard deviation for the numbers excluding the imbalance from 1 to 10% is significantly smaller than the one containing all the values.

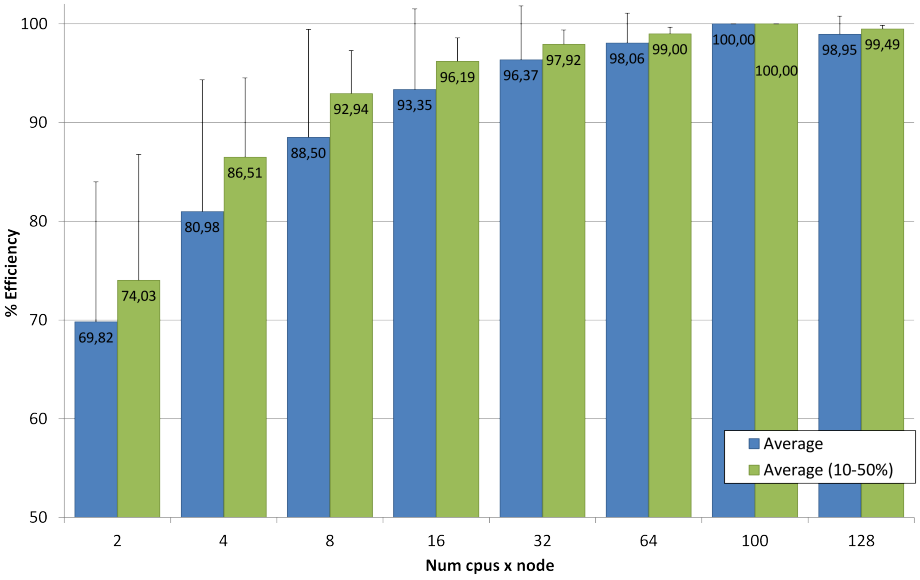


Figure A.4: Average of % of Efficiency

In Figure A.5 we can see the % of Efficiency that can be obtained for each number of processors per node and depending on the imbalance that the application presents. It is a detailed view of the previous one were we can see the effect that the imbalance have on the Efficiency. Each line represents the behavior of a number of processors per node. From this chart we can observe that for some percentages of imbalance is hard to get good Efficiency, specially the very imbalanced applications (from 1%-99% to 7%-93%). For instance , with an imbalance of 8%-92% we need 32 cpus to obtain a % of Efficiency of 92%. We can also observe from this chart that the Effi-

ciency respect the imbalance does not get stabilized but fluctuates with the imbalance.

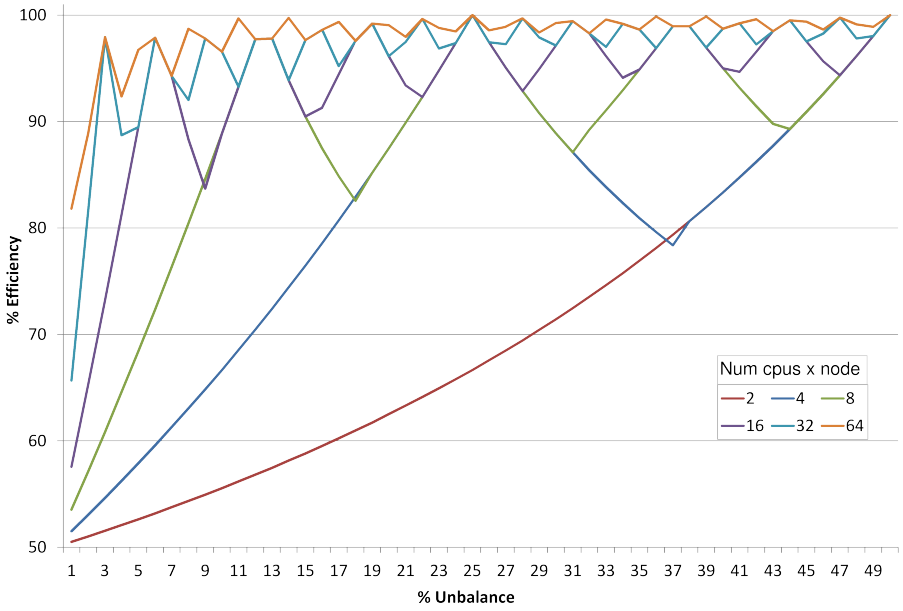


Figure A.5: % of Efficiency that can be obtained with different number of cpuS using the redistribution of cpus model

In general we can see that with 8 cpus per node the % of Efficiency is over 85% most of the time, except in extremely imbalanced situations (from 1%-99% to 10%-90%). If we go one step further with 16 cpus per node the % of Efficiency is over 90% and around 95% for almost all the imbalances.

The main observation we obtain from this study is how the Efficiency obtained depends both on the number of cpus per node available and the imbalance between the MPI processes. If the imbalance that the application presents is favorable to be divided in the given number of cores the Efficiency will be high, even if the number of cores available is small.

Being conservative we can say that with 8 processors per node the % of Efficiency that can be obtained is acceptable. But if we are really worried by balancing issues 16 cpus per node would be a more secure choice.

Execution time

The metric that we have shown until now is the Efficiency that is a measure that tries to maximize the usage of the resources. In the following pages we are going to show an evaluation based on execution time that is a measure more user oriented.

In the evaluation that follows we have emulated the time that will take an application to finish depending on the number of processors available. The suppositions we have made is that the application runs in one node with two MPI processes, that when running in serial the application takes 100 seconds to finish and that with an application well balanced (% Unbalance is 50%) the speed up is ideal (i.e with 2 processors the speed up will be 2).

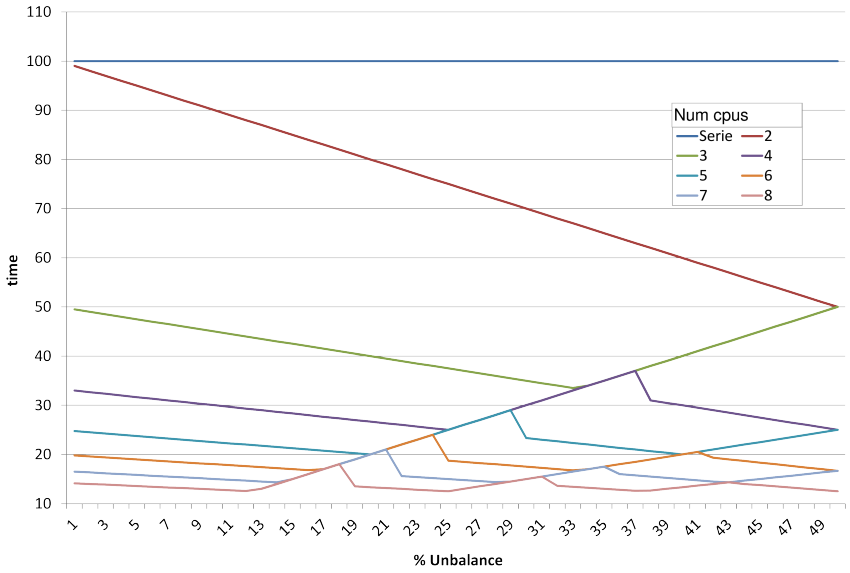


Figure A.6: Execution time for 2 to 8 cpus per node with redistribution model

In Figure A.6 we can see the time in seconds that we estimate an application will take to run depending on the number of processors available, the imbalance that presents and the distribution of cpus calculated by the algorithm of redistribution.

The graph shows that if we are running an application with 4 processors and the imbalance it presents is between 33% and 37% we can release or decrease the frequency of one cpu without impact in the execution time. In all the cases where there are two lines overlapped it means that the execution time is the same, therefor we can use the lowest number of processors to achieve the same performance of the application.

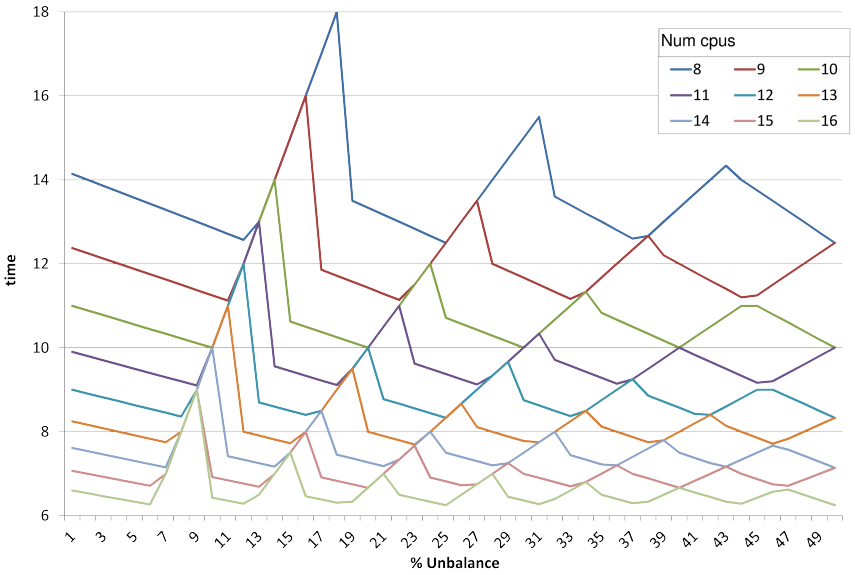


Figure A.7: Execution time for 8 to 16 cpus per node with redistribution model

We have divided into two charts the execution time just because a matter of visibility. In Figure A.7 we can find the execution time when running an application from 8 to 16 processors. In this chart seems that there is more overlap of lines. This means that the more number of processors we are using the more number of processors we will be able to release without

penalizing the application. For example when running with 16 processors if the application have an imbalance of 9% we could release 4 processors (run with 12) and finish in the same amount of time. But on the other hand what we also notice is that the margin of imbalance in which we are moving gets smaller.

To show the numbers we are talking in Table A.1 is shown the execution time for the different imbalances (vertical) and number of procs per node (horitzontal). The numbers that are colored with the same color in the same line are equal. Meaning that an aplicacion with that imbalance will present the same execution time with different nuimber of cpus. For example, if we are executing an application that presents an imbalance of 12%-88% with 12 cpus per node it can finish in 12 seconds, but with 9 cpus per node the time obtained is also 12 seconds. This numbers suggest that maybe with the cpus we have we can not improve the performance of the application

Speed up

The speed up presented in this subsection have been calculated with the execution time calculated in the previous subsection and a serial time of the application supposed of 100 seconds.

In Figures A.8 and A.9 we can see the speed up that can be obtained when running with different number of processors and depending on the imbalance present in the application. In these charts one can observe when the maximums are achieved (i.e the maximum for 8 processors is a speed up of 8).

Another interesting observation to understand better the behavior of the algorithm is the number of lower peaks that presents each line. We can see that the more number of processors used the more number of lower peaks. If we concentrate in Figure A.8 in the line representing the execution with 8 processors we can see that the lower peaks represent the "change" in the distribution of cpus. If we calculate the distribution for the different % of imbalance we will be able to check this. For an imbalance of 1%-99% the number of cpus assigned to the first MPI process is $= \frac{8*1}{100} = 0,08$ as we can not assign less than one cpu per MPI process the distribution is: 1-7. For an imbalance of 18%-82% the number of cpus assigned to the first MPI process is $= \frac{8*18}{100} = 1,44$, so the distribution is still 1-7 because 1,44 is rounded to 1. If we calculate the distribution for an imbalance of 19%-81% we obtain that

Appendix A. DWB Algorithm Limits

%Unbalance	Num cpus x node														
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	99,00	49,50	33,00	24,75	19,80	16,50	14,14	12,38	11,00	9,90	9,00	8,25	7,62	7,07	6,60
2	98,00	49,00	32,67	24,50	19,60	16,33	14,00	12,25	10,89	9,80	8,91	8,17	7,54	7,00	6,53
3	97,00	48,50	32,33	24,25	19,40	16,17	13,86	12,13	10,78	9,70	8,82	8,08	7,46	6,93	6,47
4	96,00	48,00	32,00	24,00	19,20	16,00	13,71	12,00	10,67	9,60	8,73	8,00	7,38	6,86	6,40
5	95,00	47,50	31,67	23,75	19,00	15,83	13,57	11,88	10,56	9,50	8,64	7,92	7,31	6,79	6,33
6	94,00	47,00	31,33	23,50	18,80	15,67	13,43	11,75	10,44	9,40	8,55	7,83	7,23	6,71	6,27
7	93,00	46,50	31,00	23,25	18,60	15,50	13,29	11,63	10,33	9,30	8,45	7,75	7,15	7,00	7,00
8	92,00	46,00	30,67	23,00	18,40	15,33	13,14	11,50	10,22	9,20	8,36	8,00	8,00	8,00	8,00
9	91,00	45,50	30,33	22,75	18,20	15,17	13,00	11,38	10,11	9,10	9,00	9,00	9,00	9,00	9,00
10	90,00	45,00	30,00	22,50	18,00	15,00	12,86	11,25	10,00	10,00	10,00	10,00	10,00	6,92	6,43
11	89,00	44,50	29,67	22,25	17,80	14,83	12,71	11,13	11,00	11,00	11,00	11,00	7,42	6,85	6,36
12	88,00	44,00	29,33	22,00	17,60	14,67	12,57	12,00	12,00	12,00	12,00	8,00	7,33	6,77	6,29
13	87,00	43,50	29,00	21,75	17,40	14,50	13,00	13,00	13,00	13,00	8,70	7,91	7,25	6,69	6,50
14	86,00	43,00	28,67	21,50	17,20	14,33	14,00	14,00	14,00	9,56	8,60	7,82	7,17	7,00	7,00
15	85,00	42,50	28,33	21,25	17,00	15,00	15,00	15,00	10,63	9,44	8,50	7,73	7,50	7,50	7,50
16	84,00	42,00	28,00	21,00	16,80	16,00	16,00	16,00	10,50	9,33	8,40	8,00	8,00	8,00	6,46
17	83,00	41,50	27,67	20,75	17,00	17,00	17,00	11,86	10,38	9,22	8,50	8,50	8,50	6,92	6,38
18	82,00	41,00	27,33	20,50	18,00	18,00	18,00	11,71	10,25	9,11	9,00	9,00	7,45	6,83	6,31
19	81,00	40,50	27,00	20,25	19,00	19,00	13,50	11,57	10,13	9,50	9,50	9,50	7,36	6,75	6,33
20	80,00	40,00	26,67	20,00	20,00	20,00	13,33	11,43	10,00	10,00	10,00	8,00	7,27	6,67	6,67
21	79,00	39,50	26,33	21,00	21,00	21,00	13,17	11,29	10,50	10,50	8,78	7,90	7,18	7,00	7,00
22	78,00	39,00	26,00	22,00	22,00	15,60	13,00	11,14	11,00	11,00	8,67	7,80	7,33	7,33	6,50
23	77,00	38,50	25,67	23,00	23,00	15,40	12,83	11,50	11,50	9,63	8,56	7,70	7,67	7,67	6,42
24	76,00	38,00	25,33	24,00	24,00	15,20	12,67	12,00	12,00	9,50	8,44	8,00	8,00	6,91	6,33
25	75,00	37,50	25,00	25,00	18,75	15,00	12,50	12,50	10,71	9,38	8,33	8,33	7,50	6,82	6,25
26	74,00	37,00	26,00	26,00	18,50	14,80	13,00	13,00	10,57	9,25	8,67	8,67	7,40	6,73	6,50
27	73,00	36,50	27,00	27,00	18,25	14,60	13,50	13,50	10,43	9,13	9,00	8,11	7,30	6,75	6,75
28	72,00	36,00	28,00	28,00	18,00	14,40	14,00	12,00	10,29	9,33	9,33	8,00	7,20	7,00	7,00
29	71,00	35,50	29,00	29,00	17,75	14,50	14,50	11,83	10,14	9,67	9,67	7,89	7,25	7,25	6,45
30	70,00	35,00	30,00	23,33	17,50	15,00	15,00	11,67	10,00	10,00	8,75	7,78	7,50	7,00	6,36
31	69,00	34,50	31,00	23,00	17,25	15,50	15,50	11,50	10,33	10,33	8,63	7,75	7,75	6,90	6,27
32	68,00	34,00	32,00	22,67	17,00	16,00	13,60	11,33	10,67	9,71	8,50	8,00	8,00	6,80	6,40
33	67,00	33,50	33,00	22,33	16,75	16,50	13,40	11,17	11,00	9,57	8,38	8,25	7,44	6,70	6,60
34	66,00	34,00	34,00	22,00	17,00	17,00	13,20	11,33	11,33	9,43	8,50	8,50	7,33	6,80	6,80
35	65,00	35,00	35,00	21,67	17,50	17,50	13,00	11,67	10,83	9,29	8,75	8,13	7,22	7,00	6,50
36	64,00	36,00	36,00	21,33	18,00	16,00	12,80	12,00	10,67	9,14	9,00	8,00	7,20	7,20	6,40
37	63,00	37,00	37,00	21,00	18,50	15,75	12,60	12,33	10,50	9,25	9,25	7,88	7,40	7,00	6,30
38	62,00	38,00	31,00	20,67	19,00	15,50	12,67	12,67	10,33	9,50	8,86	7,75	7,60	6,89	6,33
39	61,00	39,00	30,50	20,33	19,50	15,25	13,00	12,20	10,17	9,75	8,71	7,80	7,80	6,78	6,50
40	60,00	40,00	30,00	20,00	20,00	15,00	13,33	12,00	10,00	10,00	8,57	8,00	7,50	6,67	6,67
41	59,00	41,00	29,50	20,50	20,50	14,75	13,67	11,80	10,25	9,83	8,43	8,20	7,38	6,83	6,56
42	58,00	42,00	29,00	21,00	19,33	14,50	14,00	11,60	10,50	9,67	8,40	8,40	7,25	7,00	6,44
43	57,00	43,00	28,50	21,50	19,00	14,33	14,33	11,40	10,75	9,50	8,60	8,14	7,17	7,17	6,33
44	56,00	44,00	28,00	22,00	18,67	14,67	14,00	11,20	11,00	9,33	8,80	8,00	7,33	7,00	6,29
45	55,00	45,00	27,50	22,50	18,33	15,00	13,75	11,25	11,00	9,17	9,00	7,86	7,50	6,88	6,43
46	54,00	46,00	27,00	23,00	18,00	15,33	13,50	11,50	10,80	9,20	9,00	7,71	7,67	6,75	6,57
47	53,00	47,00	26,50	23,50	17,67	15,67	13,25	11,75	10,60	9,40	8,83	7,83	7,57	6,71	6,63
48	52,00	48,00	26,00	24,00	17,33	16,00	13,00	12,00	10,40	9,60	8,67	8,00	7,43	6,86	6,50
49	51,00	49,00	25,50	24,50	17,00	16,33	12,75	12,25	10,20	9,80	8,50	8,17	7,29	7,00	6,38
50	50,00	50,00	25,00	25,00	16,67	16,67	12,50	12,50	10,00	10,00	8,33	8,33	7,14	7,14	6,25

Table A.1: Execution times with redistribution of cpus model

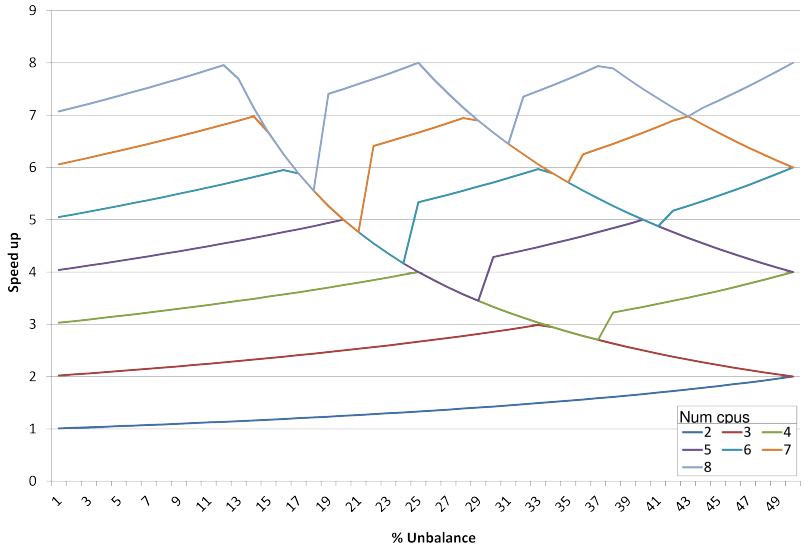


Figure A.8: Speed up for 2 to 8 cpus per node with redistribution model

the number of processors assigned to the first MPI process is $= \frac{8*19}{100} = 1,52$ in this case 1,52 is rounded to 2, then the distribution of cpus is 2-6.

If we repeat this process for the other peaks we will find out that the lower peaks correspond to the changes of distribution or what is the same, that the values in the number of cpus before rounding them is close to X,5. On the other hand, the higher peaks correspond to the values close to a whole number. As an example we can calculate the distribution for an imbalance of 12%-88% $= \frac{8*12}{100} = 0,96$ and for 13%-87% $= \frac{8*13}{100} = 1,04$, both are close to 1.

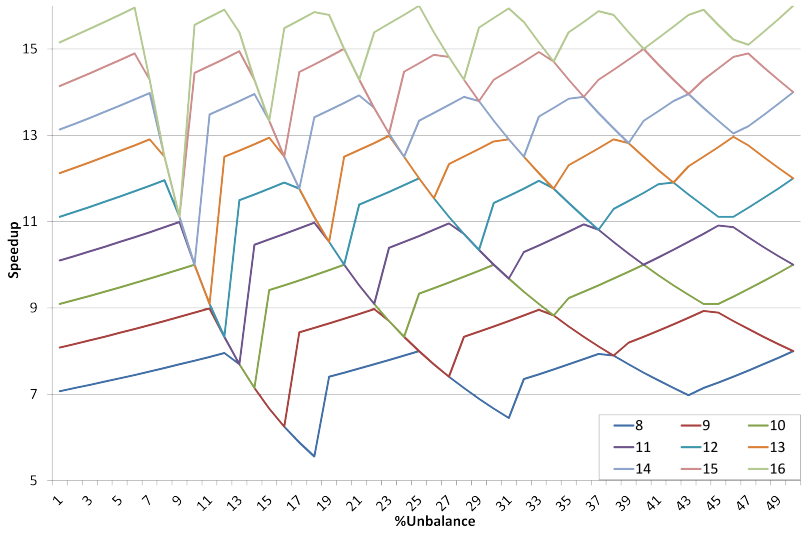


Figure A.9: Speed up for 8 to 16 cpus per node with redistribution model

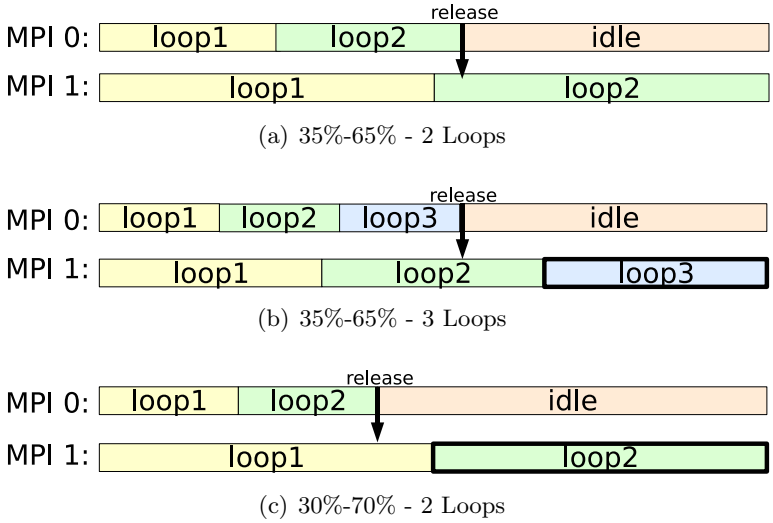
Appendix B

LeWI Algorithm Limits

In this section we are going to study the limitations of the ***LeWI*** algorithm. The limitation that we have observed for this algorithm is the number of parallel regions between MPI calls. This is due to the fact that with the current programming model we are using (OpenMP) we can only increase or decrease the number of threads per process before starting a parallel loop.

To show this effect in Figure B.1 are represented different situations, the first one (Figure B.1(a)) presents an imbalance of 35%-65% and the application have two loops, when the first process finishes its computation and lends the cpus the second process have just started the second loop (and last one) therefor the second process will not be able to use the lent cpus and there will not be any improvement in the execution time respect the original.

Let's see now an example with the same imbalance (35%-65%) but in this case it has 3 loops per iteration (Figure B.1(b)). Now when the first process finishes the computation phase and lends the cpus the second process is in the middle of the second loop. In this case when the second process starts the third loop it will have more processors available and will be able to use them for the last loop (the loops that are able to use the lent cpus are marked with a thicker line). - To show that the potential of the algorithm not only depends on the number of loops in Figure B.1(c) we show a case with 2 loops as the first example but a slightly different imbalance (30%-70%). In this scenario the first process gets into the communication phase and lends the



(a) 35%-65% - 2 Loops

(b) 35%-65% - 3 Loops

(c) 30%-70% - 2 Loops

Figure B.1: Example of impact of the number of loops and the imbalance in the performance of LeWI

cpus at the end of the first loop of the first process. When the second process starts the second loop it has available the recently lent cpus and adds them to the computation.

In the first part of this section we are going to evaluate the behavior of the model when running with different number of loops per iteration. On the second part we will try to determine if there is a limitation in the size of the loops and which is it.

All the data presented in this section has been obtained running PILS, the synthetic application introduced in Section 3.1. The application have been executed in Marenostrum2 always with 4 cpus per node and 2 MPIs processes. When evaluating the impact of the number of loops per iteration the global computational load of the application is constant but divided in different number of loops.

Number of loops per iteration

In this subsection we try to determine the minimum number of loops necessary per iteration to obtain good performance results with the LeWI algorithm.

The experiments contain from 1 to 8 loops. We must keep in mind that the load is always the same, but divided in different number of loops. In the case of one loop there is nothing the DLB library with LeWI can do, but we show it to check that no overhead is introduced and to keep it as reference. In the experiments we show all the loops have the same amount of load.

As we have done previously in the other sections we have executed all the experiments with all the imbalances from 1%-99% to 50%-50%

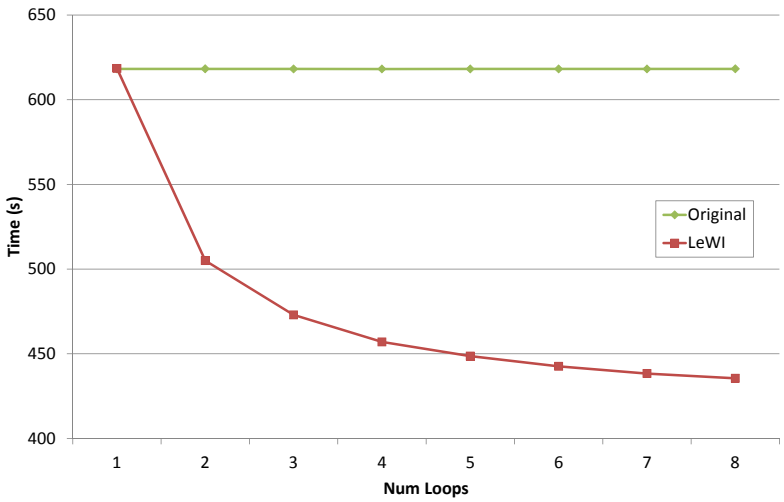


Figure B.2: Average time versus Number of Loops

In Figure B.2 is shown the average execution time obtained with all the imbalances when running the original application with DLB using the LeWI algorithm and without DLB (label *Original*).

These charts show that with one loop nothing can be done but with just 2 loops some improvement is possible. The minimum desirable seems to be around 4 loops per iteration because is where the line seems to stabilize. With 4 loops when running with LeWI we can obtain an speed up of 3,6 compared

with the 2,7 that is obtained by the application without DLB. Now we are going to see more in detail what happens with the different imbalances.

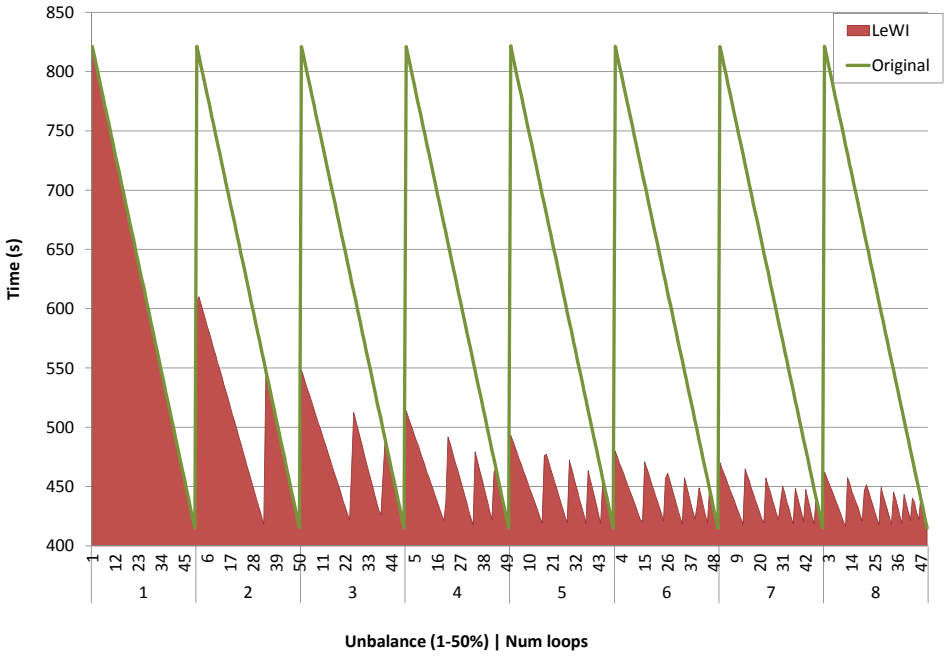


Figure B.3: Execution time 1 to 8 loops with LeWI versus original

In Figure B.3 we can see the execution time spent to run the synthetic application with LeWI versus the application without DLB. In the X axis we can see the number of loops in which the load is divided and in a smaller scale the % of imbalance present in the application. The imbalance goes from 1 to 50% (left to right). In the Y axis is represented the execution time of the application in seconds.

The red area represents the time spent with LeWI and the green line the time spent with the application without LeWI. The difference between the two areas is the time we are gaining when using LeWI versus the execution of the original application without dynamic load balancing.

As we said before, the global load of the application is the same independently of the number of loops; so we can see that when running the

application without DLB the execution time behaves the same for the different number of loops. But the execution time when running the application with LeWI decreases as the number of loops increase.

For the case of just one parallel loop there is nothing that LeWI can do and it has the same area as the original version of the application.

In this chart we can observe that with LeWI there are the same number of higher peaks as loops, that effect is due to the fact that depending on the % of imbalance that appears in the application the threads that are lent arrive in a different moment of the loop, and depending on the moment that the threads are available (beginning, middle, end...) they can be used a different amount of time.

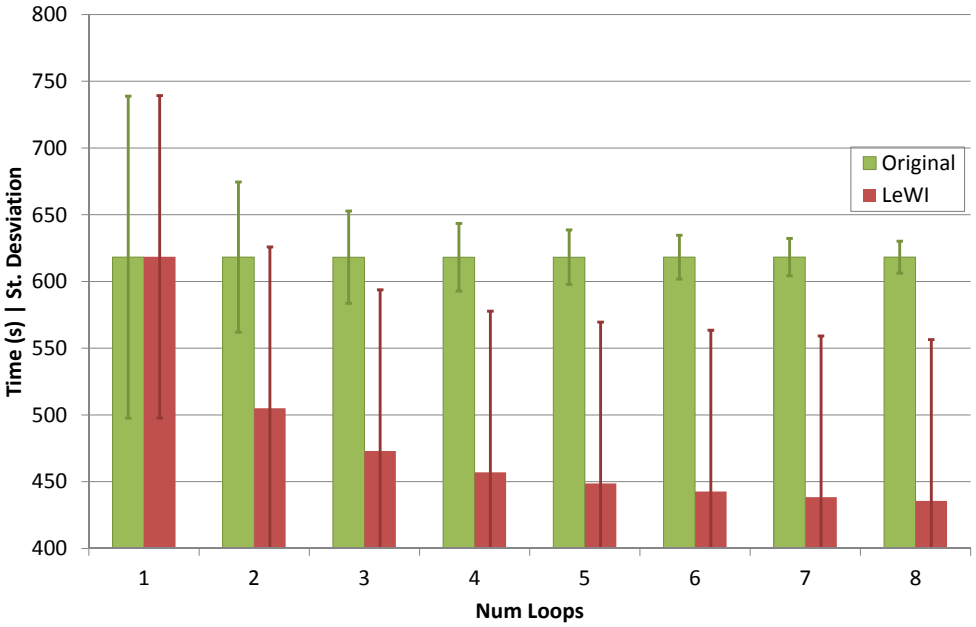


Figure B.4: Standard Deviation in the average time

In Figure B.4 the execution time is represented in the Y axis, the number of loops appear in the X axis. In this chart is also represented the standard deviation of the average time for the original execution and the execution with LeWI.

The yellow bars represent the execution time of the original application and their standard deviation, both are constant for all the number of loops because as we said it is independent from how is distributed the load as long as it is always the same amount.

On the other hand the red bars represent the average when running the application with the DLB library and LeWI. In this case we can see how the execution time decreases as the number of loops increases and also the standard deviation decreases.

The conclusion is that an application should contain at least 4 loops per iteration to obtain good performance results. By the way, with 2 and 3 loops some performance can be obtained and, as there is no overhead introduced, LeWI can be used anyway with an application with this characteristics to try to obtain some benefits.

Size of loops per iteration

In the previous subsection we analyzed the number of loops per iteration necessary to get good results with LeWI. In the following subsection we are going to study the impact of the size of the loops on the performance of LeWI.

Although the experiments we have done until now show that the DLB library does not introduce overhead the duration of the loops can have impact in the performance. In case the loops are too small can happen that the lent cpus don't have time to be used in the next loops.

	Load	**Num Loops / Load per loop**				
Class	**Total**	**1**	**2**	**4**	**8**	**16**
A	1600	1600	800	400	200	100
B	3200	3200	1600	800	400	200
C	6400	6400	3200	1600	800	400
D	12800	12800	6400	3200	1600	800
E	25600	25600	12800	6400	3200	1600
F	51200	51200	25600	12800	6400	3200
G	102400	102400	51200	25600	12800	6400

Table B.1: Load for each class

In table B.1 is shown the load that corresponds to each class. The number that appears in the table represent the work that should be done, and it is divided between the two MPI processes depending on the corresponding imbalance.

	Duration Loop in serial (ms)				
Class	**1**	**2**	**4**	**8**	**16**
A	332	166	83	41	21
B	663	332	166	83	41
C	1326	663	331	166	83
D	2653	1326	663	332	166
E	5305	2652	1326	663	331
F	10607	5303	2652	1326	663
G	21216	10610	5304	2651	1326

Table B.2: Duration of each loop in serial execution depending on the class and the number of loops

We have tried with different classes of load and for each load with different number of loops. Class A is the smallest and G is the biggest. The smallest class has been dimensioned by the higher number of loops that is 16 and if the load is 100 in the case of an imbalance of 1%-99% the MPI process 1 will get 1 loop iteration and the MPI process 2 will get 99.

In table B.2 we can find the duration in milliseconds of the loops corresponding to each class and number of loops. The time shown in the table has been calculated executing the original application without parallelism. This means that in the parallel version of the application the duration of a loop of class A with 16 loops if the imbalance is %50-%50 is $\simeq \frac{21ms}{4} = 5,25ms$, but with a different degree of imbalance the duration will be different.

We have run the synthetic application with classes A, B , C, D, E, F and G and number of loops 1, 2, 4, 8 and 16. The execution time obtained can be seen in Figure B.5 and B.6 for classes G and A, the smallest and the biggest.

In Figure B.5 can be seen an almost perfect pattern as the loops are very big and the behavior is very regular, we will take this pattern as the ideal to be obtained and we are going to compare it with the smallest class to see the deviation that it presents.

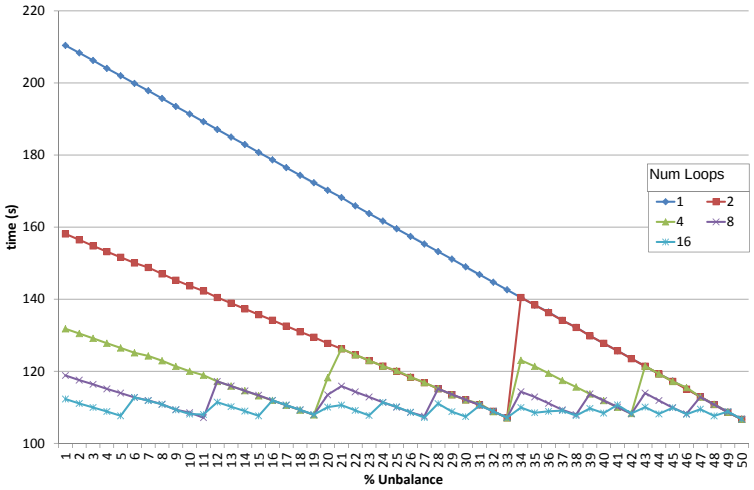


Figure B.5: Execution time with LeWI and load class G

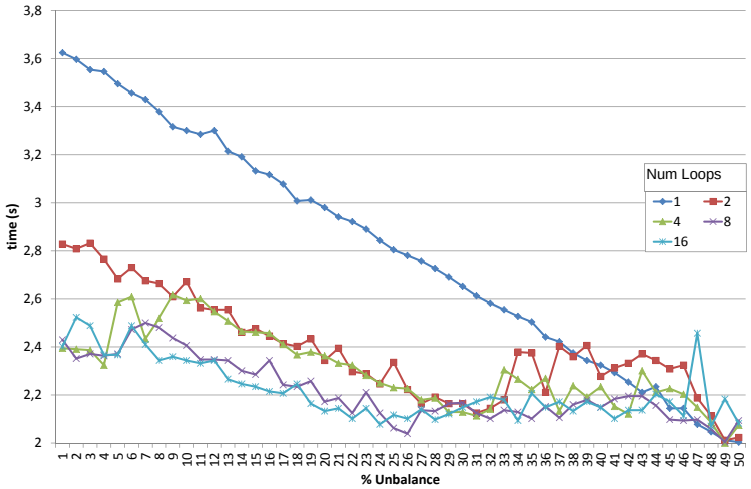


Figure B.6: Execution time with LeWI and load class A

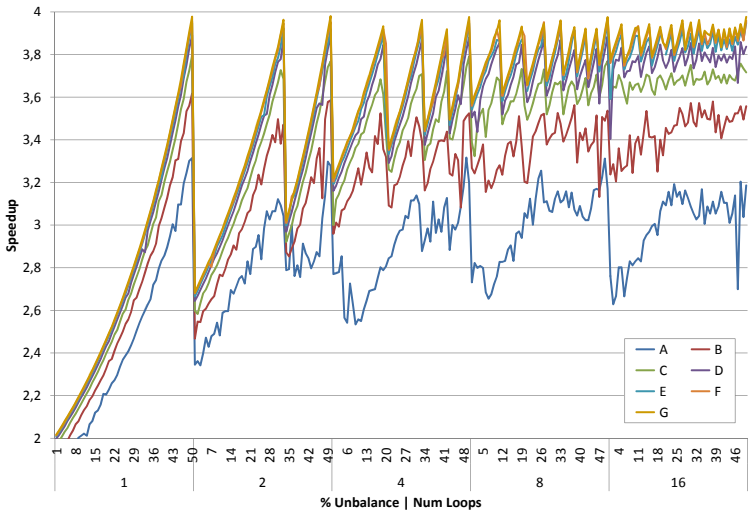


Figure B.7: Speed up with LeWI classes A, B, C, D, E, F and G

The execution time for class A, as can be see in Figure B.6, has much variation. With one loop the behavior is more or less the same as for class G. For two loops it is not difficult to follow the pattern at least for the more imbalanced experiments. The worst performance is found for the more balanced experiments from 35%-65% to 50%-50%. This can be due to the fact that the time spent in each loop is smaller as the application is more balanced, and the time "remaining" to exchange processors is also shorter.

What it is important to notice from this data is that the performance for class A is not the best one (as can be obtained with bigger classes) but that it only performs worst than the original in few cases, that is when the application is not imbalanced (45%-55%).

The comparison between the performance of all the classes can be seen in Figure B.7. In this chart is shown the speed up for each class. In the X axis is shown the number of loops and the % of imbalance.

The performance drops for class A and B, and specially when there are more than 4 loops (remember that the size of the loops is not constant per class, but that the more loops the smaller they are). On the other hand classes C, D, E, F and G present almost no difference in their performance

It is interesting to notice from this evaluation that for classes bigger than B and more than 4 loops the speed up obtained is over 3,5 that is quite close to the ideal of 4.

We have seen that for loops smaller than class B and 4 loops the performance drops a bit. This is equivalent to approximately a loop of 83 milliseconds in serial and around 20,75 milliseconds in parallel with 2 MPIs and 2 OpenMP threads per MPI.

List of Figures

LIST OF FIGURES

List of Tables

Bibliography

[1] G. Karypis and V. Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[2] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.

[3] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[4] C Lachat, C Dobrzynski, and F Pellegrini. Parallel mesh adaptation using parallel graph partitioning. In Eugenio Oñate, Xavier Oliver, and Antonio Huerta, editors, *5th European Conference on Computational Mechanics (ECCM V)*, volume 3 of *Minisymposia in the frame of ECCM V*, pages 2612–2623, Barcelone, Spain, July 2014. IACM & ECCOMAS, CIMNE - International Center for Numerical Methods in Engineering. ISBN 978-84-942844-7-2.

[5] Milind A. Bhandarkar, Laxmikant V. Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for mpi programs. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, 2001.

[6] Marta Garcia, Julita. Corbalan, and Jesus Labarta. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In *Proceedings of the International Conference on Parallel Processing (ICPP09)*, 2009.

[7] Dirk Brömmel, Paul Gibbon, Marta Garcia, Víctor López, Vladimir Marjanović, and Jesús Labarta. Experience with the MPI/STARSS programming model on a large production code. In *Advances in Parallel Computing*, volume 25: Parallel Computing: Accelerating Computational Science and Engineering (CSE), pages 357 – 366. 2014.

[8] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with LeWI for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781 – 2794, 2014.

[9] Marta Garcia, Julita Corbalan, Rosa Maria Badia, and Jesus Labarta. A Dynamic Load Balancing approach with SMPSuperscalar and MPI. In *Facing the Multicore-Challenge II*, 2011.

[10] Guillaume Houzeaux, Marta Garcia, Juan Carlos Cajas, Antoni Artigues, Edgar Olivares, Jesús Labarta, and Mariano Vázquez. Dynamic load balance applied to particle transport in fluids. *International Journal of Computational Fluid Dynamics*, 30(6):408–418, 2016.

[11] Marta Garcia-Gasulla, Guillaume Houzeaux, Antoni Artigues, Jesus Labarta, and Mariano Vazquez. Load balancing of an MPI parallel unstructured CFD code using DLB and OpenMP. *International Journal of High Performance Computing Applications*, 2017 (Submited).

[12] Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. Automatic thread distribution for nested parallelism in openmp. In *Proceedings of the 19th annual international conference on Supercomputing (ICS)*, pages 121–130, 2005.

[13] MPI Forum. Message passing interface version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf` - Last accessed Nov. 2012, 2009.

[14] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing-Volume 00*, pages 427–436. IEEE Computer Society, 2009.

[15] L. Smith and M. Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[16] OpenMP Architecture Review Board. Openmp application program interface version 2.5. `http://www.openmp.org/mp-documents/spec25.pdf` - Last accessed Nov. 2012, May 2005.

[17] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2008.

[18] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010.

[19] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193, 2011-03-01 2011.

[20] Ompss programming model. `https://pm.bsc.es/ompss` - Last accessed Jul. 2016.

[21] Mercurium compiler. `https://pm.bsc.es/mcxx` - Last accessed Jul. 2016.

[22] Nanos++ runtime. `https://pm.bsc.es/nanox` - Last accessed Jul. 2016.

[23] Chapel. `http://chapel.cray.com/` - Last accessed January. 2016.

[24] Jay P. Hoeflinger. Extending OpenMP to Clusters. 2006.

[25] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with OmpSs. *Euro-Par 2011 Parallel Processing*, pages 555–566, 2011.

[26] Marenostrum2. `http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture` - Last accessed January. 2016.

[27] Marenostrum3. `http://www.bsc.es/marenostrum-support-services/mn3` - Last accessed January. 2016.

[28] Intel compilers. https://software.intel.com/en-us/intel-compilers - Last accessed January. 2016.

[29] Gnu compilers. https://gcc.gnu.org/ - Last accessed January. 2016.

[30] Ibm xl compilers. http://www-03.ibm.com/software/products/en/category/SW780 - Last accessed January. 2016.

[31] Mercurium compiler. http://pm.bsc.es/mcxx - Last accessed January. 2016.

[32] Extrae. http://www.bsc.es/computer-sciences/extrae - Last accessed March 2015.

[33] Paraver. http://www.bsc.es/computer-sciences/performance-tools/paraver - Last accessed March 2015.

[34] Mpich: Mpi library. https://www.mpich.org/ - Last accessed January. 2016.

[35] Openmpi: Mpi library. http://www.open-mpi.org/ - Last accessed January. 2016.

[36] Intel mpi library. https://software.intel.com/en-us/intel-mpi-library - Last accessed January. 2016.

[37] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. *J. Parallel Distrib. Comput.*, 66(5):674–685, 2006.

[38] M. Hesse, B. U. Reinartz, and J. Ballmann. Inviscid flow computation for the shuttle-like configuration phoenix. *Notes on Numerical Fluid Mechanics and Multidisciplinary Desing*, 2004.

[39] Erik Lindahl, Berk Hess, and David van der Spoel. Gromacs 3.0: a package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*, 7:306–317, 2001. 10.1007/s008940100045.

[40] Gromacs. http://www.gromacs.org/ - Last accessed September. 2016.

[41] Volker Springel, Naoki Yoshida, and Simon D.M. White. Gadget: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79 – 117, 2001.

[42] Gadget. `http://wwwmpa.mpa-garching.mpg.de/gadget/` - Last accessed September. 2016.

[43] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[44] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[45] Lulesh. `https://codesign.llnl.gov/lulesh.php` - Last accessed September. 2016.

[46] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, E.D. Burness, J.M. Cela, and M. Valero. Alya: Multiphysics engineering simulation towards exascale. *J. Comput. Sci.*, 14:15–27, 2016.

[47] Alya. `https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational` - Last accessed Nov. 2016.

[48] Unified european applications benchmark suite. `http://www.prace-ri.eu/ueabs/` - Last accessed Nov. 2016.

[49] Zhi Shang. Impact of mesh partitioning methods in {CFD} for large scale parallel computing. *Computers & Fluids*, 103:1 – 5, 2014.

[50] Youssef Mesri, Hugues Digonnet, and Thierry Coupez. Advanced parallel computing in material forming with cimlib. *European Journal of Computational Mechanics*, 18(7-8):669–694, 2009.

[51] Youssef Mesri, Walid Zerguine, Hugues Digonnet, Luisa Silva, and Thierry Coupez. *Dynamic Parallel Adaption for Three Dimensional Unstructured Meshes: Application to Interface Tracking*, pages 195–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[52] Jin Li, Xiangren Geng, Dingwu Jiang, and Jianqiang Chen. Dynamic load balance scheme for the dsmc algorithm. *AIP Conference Proceedings*, 1628(1):288–295, 2014.

[53] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, pages 175–213. 1996.

[54] V. Deodhar, H. Parikh, A. Gavrilovska, and S. Pande. Compiler assisted load balancing on large clusters. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 280–291, Oct 2015.

[55] Mahadevan Balasubramaniam, Kevin Barker, Ioana Banicescu, Nikos Chrisochoides, Jaderick P. Pabico, and Ricolindo L. Carino. A novel dynamic load balancing library for cluster computing. In *Proceedings of the Third International Symposium on Parallel and Distributed Computing(ISPDC)*, 2004.

[56] Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh, and Jesús Carretero. *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, chapter FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems, pages 138–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[57] Haoqiang Jin and Rob F Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 6. IEEE, 2004.

[58] S. Meraji and C. Tropper. Optimizing techniques for parallel digital logic simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 23(6):1135–1146, June 2012.

[59] H. Arafat, P. Sadayappan, J. Dinan, S. Krishnamoorthy, and T. L. Windus. Load balancing of dynamical nucleation theory monte carlo simulations through resource sharing barriers. In *Parallel Distributed*

Processing Symposium (IPDPS), 2012 IEEE 26th International, pages 285–295, May 2012.

[60] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3), 2001.

[61] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[62] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, 2000.

[63] Yun Zhang, Mihai Burcea, Victor Cheng, Ron Ho, and Michael Voss. An adaptive openmp loop scheduler for hyperthreaded smps. In *ISCA PDCS*, pages 256–263, 2004.

[64] Otto Sievert and Henri Casanova. A simple mpi process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341–352, 2004.

[65] Kaoutar El Maghraoui, Boleslaw Szymanski, and Carlos Varela. An architecture for reconfigurable iterative mpi applications in dynamic environments. In *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM)*, 2005.

[66] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, chapter Invasive Computing: An Overview, pages 241–268. Springer New York, New York, NY, 2011.

[67] M. Schreiber, C. Riesinger, T. Neckel, and H. J. Bungartz. Invasive compute balancing for applications with hybrid parallelization. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 136–143, Oct 2013.

[68] Alexander Spiegel, Dieter an Mey, and Christian H. Bischof. Hybrid parallelization of cfd applications with dynamic thread balancing. In *PARA*, pages 433–441, 2004.

[69] Felix Freitag, Julita Corbalan, and Jesus Labarta. A dynamic periodicity detector: Application to speedup computation. In *15th International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[70] Alya software. `https://www.bsc.es/research-and-development/software-and-apps/software-list/alya` - Last accessed Nov. 2016.

[71] H. Calmet, A. Gambaruto, A. Bates, M. Vázquez, G. Houzeaux, and D. Doorly. Large-scale CFD simulations of the transitional and turbulent regime for the large human airways during rapid inhalation. 69:166–180, 2016.

[72] Iter. `http://www.iter.org//` - Last accessed Nov. 2016.

[73] A. Gambaruto, E. Olivares, H. Calmet, G. Houzeaux, A. Bates, and D. Doorly. Transport and deposition in the upper human airways during a sniff. In *Computational Engineering and Science for Safety and Environmental Problems COMPSAFE2014*, Sendai (Japan), April 13-16 2014.

[74] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, E.D. Burness, J.M. Cela, and M. Valero. Alya: Multiphysics engineering simulation towards exascale. *J. Comput. Sci.*, 14:15–27, 2016.

[75] Andy B. Yoo, Morris A. Jette, and Mark Grondona. *SLURM: Simple Linux Utility for Resource Management*, pages 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[76] Slurm. `https://slurm.schedmd.com/` - Last accessed Jan. 2017.