Hints to Improve Automatic Load Balancing with LeWI for Hybrid Applications

Marta Garcia[a,b,*], Jesus Labarta[a,b], Julita Corbalan[a,b]

[a] *Barcelona Supercomputing Center (BSC), c/ Jordi Girona, 31 08034, Barcelona, Spain*
[b] *Department of Computer Architecture (DAC), Universitat Politecnica de Catalunya, D6-218 Campus Nord UPC, Jordi Girona. Barcelona, Spain*

Abstract

The DLB (Dynamic Load Balancing) library and LeWI (LEnd When Idle) algorithm provide a runtime solution to deal with the load imbalance of parallel applications independently of the source of imbalance. DLB relies on the usage of hybrid programming models and exploits the malleability of the second level of parallelism to redistribute computation power across processes.

When executing real applications with LeWI, although application's performance is significantly improved, we have observed in some cases efficiency values between 60% and 70%, far from our theoretical limit.

This work is a deep analysis of the sources of efficiency loss correlated with application characteristics, parallelization schemes and programming models. We have based our analysis in fine grain monitoring tools and metrics and validated our conclusions by reproducing them in synthetic experiments. As a result, this work teaches us some lessons that can be seen as hints to programmers to help LeWI make an efficient use of computational resources and obtain the maximum performance.

Keywords: application performance analysis; Load Balancing; hybrid parallel programming;

1. Introduction and Motivation

In parallel computing, the loss of efficiency is an issue that concerns both system administrators and parallel programmers. The growth in number of computing units that clusters experienced the last years has helped speeding up applications but has worsened some problems that affect the efficient use of the computational power.

One of the problems that have deteriorated with this growth is load balance. Although it is a concern that has been targeted since the beginning of parallel programming, there is not a universal solution.

*Corresponding author: Marta Garcia, BSC, c/ Jordi Girona, 31 08034 Barcelona (Spain) Telephone: +34 93 413 72 43

Email addresses: marta.garcia@bsc.es (Marta Garcia), jesus.labarta@bsc.es (Jesus Labarta), juli@ac.upc.edu (Julita Corbalan)

In the literature, we can see that load imbalance have been attacked from very different points of view (data partition, data redistribution, resource migration, etc.). These solutions only target the imbalance in one of its sources (input data imbalance, resource heterogeneity, resource sharing...). LeWI (Lend When Idle) [1] is a load balancing algorithm that provides a runtime solution for load balancing problems of hybrid applications independently of the source of imbalance.

A preliminary evaluation of LeWI demonstrated its potential when applied to some MPI+OpenMP benchmarks and its flexibility to solve different types of imbalances. In a more exhaustive evaluation, we detected that the efficiency obtained by real applications was lower than the one obtained by the synthetic benchmark used to evaluate LeWI's potential.

The synthetic benchmark showed potential to reach almost a 100% of efficiency when running with LeWI. But when running production applications, in some cases, the efficiency could be around 60% to 70% when using LeWI. Although these numbers are better than when not using LeWI, they are still far from the theoretical maximum reached by the synthetic benchmark.

In this paper, we will focus in finding the sources of this efficiency loss when using LeWI with real applications. We will identify problems related to the programming model used, some parallelization decisions of the application and the distribution of MPI processes among computation nodes.

The methodology we will follow in this study is divided in three steps. In the first step, we start doing a deep analysis of application internal behavior when using LeWI. With this analysis, we identify certain parallelization decisions that can affect the efficiency of automatic load balancing techniques: Parallelism Grain in OpenMP applications, Task duration in SMPSs applications and distribution of MPI processes among computational nodes.

The second step will be to use a synthetic benchmark to simulate different applications behavior and confirm that the aspects we pointed are limiting the efficiency obtained. Finally, as a third step, we will apply our suggestions to real applications when possible.

With this thorough study, we aim to provide useful advices to parallel application developers to facilitate the automatic load balancing of their applications and improve their performance with minimum effort. We want to release application developers from dealing with the unbalance of their applications and help them unleash all their applications potential.

In this paper, we will show that with a minimum effort and following our advice, the applications can be speedup to obtain a significantly better utilization of resources

We will see that limitations on performance are, in both programming models, related to the granularity of parallel work. In the case of SMPSs, it will be the size of the tasks and in OpenMP the size of the parallel region. We will also show the impact of MPI processes distribution among nodes in the performance of parallel applications.

The rest of the paper is organized as follows: Section 2 will discuss some related work. In section 3, we will introduce the two hybrid programming models used. The DLB library and LeWI algorithm will be explained in detail in section 4. The performance analysis and characteristics study are presented in section 5, and finally, in section 6 we will expose the conclusions achieved with this study.

2. Related work

It is well known that load imbalance is a source of system efficiency loss in HPC applications. 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 specially dramatic in MPI applications because data is not easily redistributed. 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 computation** and the ones that are applied **during the execution**.

2.1. Before the computation

The solutions that are applied before the computation redistribute the data to load balance it between the processes. These solutions can only be applied to certain 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 [2].

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

2.2.1. Redistribute the data

Several approaches choose the option to redistribute the data of the application so that it is better balanced. Like Charm++ [3], an object-oriented parallel programming language that employs object migration to achieve load balance. Adaptive MPI (AMPI) [4] is an implementation of MPI that uses the load balancing capabilities of Charm++. Balasubramaniam et al. [5] propose a library that dynamically balances MPI processes. The load balancing is done by redistributing the data at runtime.

The approach of redistributing the data is usually rigid in terms of the type of imbalance it can solve. Moreover, 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.

2.2.2. 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, Meraji et al. [6] present a load balancing algorithm for a discrete event simulator. These approaches can only be applied to the specific applications. We will focus in more generic approaches that can be applied to any kind of application.

The first generic approach is to combine two levels of parallelism. Smith et al. [7] 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. [8] compare MPI versus a hybrid MPI + OpenMP model in a SMP cluster. They conclude that the hybrid

model is more efficient in very load unbalanced situations. In the same kind of study on clusters of multiprocessors, Capello et al. [9] 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. [10] 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. For non-dedicated systems Sievert et al. [11] 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. [12] use a technique of process migration to load balance the applications, but they need more resources than the ones assigned at the beginning of the application to migrate the processes. We will focus in solutions that try to obtain the maximum efficiency of the given resources and environments where it is not possible to increase or modify the pool of resources.

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. [13], Duran et al. [14] and LeWI [1]. They aim to balance applications with two levels of parallelism by redistributing the computational power of the inner level. They do it at runtime without modifying the application. The first one and LeWI 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 proposed as LeWI does not need the application to present an iterative pattern, nor a regular imbalance during the execution.

All previously mentioned studies use application performance as the sole metric (i.e. speedup, elapsed time or GFlops). This kind of metrics tells us about the improvement in the application's performance, but this is not enough if we want to know whether the application is reaching its maximum and using all the computational resources available. In this study, we use metrics such as efficiency and number of CPUs used to evaluate if the application is achieving its maximum performance with the given resources.

3. Hybrid Programming Models

In distributed parallel programming, the Message Passing Interface, MPI [15], has become the standard *de facto* for communication between processes in distributed memory systems.

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

Current implementations of the MPI runtime include many optimizations to execute efficiently in SMP nodes. It has been shown that in general a second level of parallelism exploited with programming models oriented to shared memory environments is a better approach [16] [17].

In the following subsections, we will explain the two hybrid programming models we will use in our study. On one hand, MPI+OpenMP, the most widely used hybrid programming model for HPC applications. On the other hand, MPI+SMPSs, this new programming model offers an alternative that presents some advantages over OpenMP in terms of load balancing.

3.1. MPI+OpenMP

One of the most popular programming models for shared memory architectures is OpenMP [18]. OpenMP is based on the use of directives that must be inserted in the code to specify parallel regions or parallel tasks.

When used to exploit the second level of parallelism in a MPI application, parallel regions are the most used scheme of parallelization. Parallel regions follow a fork/join style where all data is shared by default.

OpenMP is a malleable programming model, meaning that the number of threads running can be changed during the application execution. With the limitation that the number of running threads can only be changed outside a parallel region.

3.2. MPI+SMPSs

SMPSuperscalar (SMPSs) [19] is a task based programming model for shared memory systems. It was first released in 2007.

A task is the basic parallel element. Each task will be executed by a thread and different tasks can run in parallel. The programmer should mark functions that can be executed as tasks (taskified) in the code with compiler directives and give all the parameters of the function (task) a directionality. The directionality of a parameter can be *input*, *output* or *inout*. Each time a taskified function is called, a task is created and added to the task graph. The task graph represents the tasks and the dependencies between them.

The parallelism between tasks is controlled by tasks' dependencies. The dependences will be computed at runtime, based on the directionality of the parameters. The runtime environment will ensure that the dependencies are fulfilled when executing the tasks.

SMPSs presents higher malleability than OpenMP because 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 [20] offering a powerful approach for HPC applications.

4. The DLB (Dynamic Load Balancing) Library

The DLB library aims to balance applications with two levels of parallelism. Currently we have modules implemented to balance hybrid MPI+OpenMP and MPI+SMPSs applications. (MPI is the outer level of parallelism and OpenMP and SMPSs the inner ones). Neither OpenMP threads nor the SMPSs threads are allowed to make MPI calls.

An important feature of the DLB library is that we use a runtime interposition technique to intercept MPI calls. With this technique we do not need to modify the application, the DLB library is loaded dynamically when running the application to load balance the execution.

The DLB library will load balance the MPI processes redistributing the computational power in the second level of parallelism. This is done by changing the number of threads each MPI process can use. This means that it will balance the load of the MPI processes that are running in the same node (shared memory).

Within DLB, we have implemented several load balancing algorithms. The algorithm that obtained the best performance in all the applications is: *LeWI* (LEnd When Idle) [1]. We will explain the details of the algorithm in the following subsection.

4.1. LeWI: Lend CPUs When Idle

The LeWI algorithm is based in the following: the imbalance between MPI processes implies that one (or more) process is blocked waiting for others. While a process is blocked waiting for a message, 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 running in the same node with a higher load finish faster.

LeWI does not need previous information to take load balancing decisions that is why it can improve all kind of applications, regular or irregular ones.

The main idea of the LeWI algorithm is to lend the threads of the second level of parallelism (CPUs) of an MPI process while it is waiting in a blocking call to another MPI process running in the same node that is still doing computation.

When the MPI process that 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.

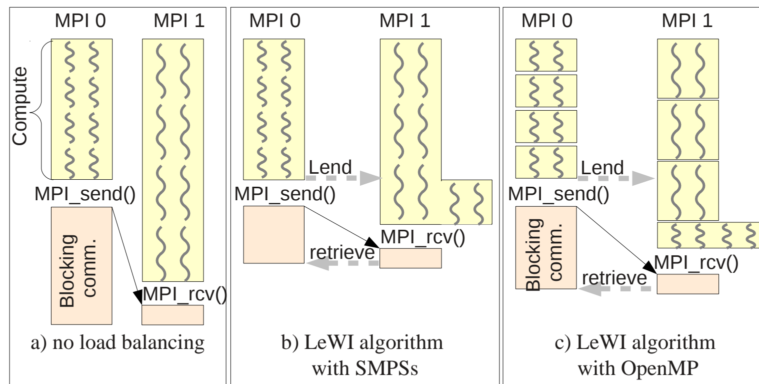

Figure 1: LeWI algorithm behavior

In Figure 1 we can see an example how the algorithm works. In the example, the application is running in a node with 4 CPUs. It starts two MPI processes in the same node and each MPI process spawns 2 threads. In Subfigure 1.a we can see the original behavior of the application, MPI process 1 is more loaded, MPI process 0 must wait in a blocking MPI call.

Subfigure 1.b shows the execution of the same application with the balancing library and the LeWI algorithm. We can see that when the MPI process 0 gets into the MPI blocking call it will lend two threads to the MPI process 1. The MPI process 1 will use the newly acquired CPUs as fast as the programming model allows it. When the MPI process 0 gets out of the blocking call it retrieves its CPUs from the MPI process 1 and the execution continues with a CPU equipartition until another blocking call is met.

We can see how the malleability of the programming model affects the behavior of the load balancing algorithm comparing Subfigure 1.c and 1.b. When using OpenMP (Subfigure 1.c) the number of threads cannot be changed until reaching a new parallel region. While in the case of SMPSs (Subfigure 1.b), the number of threads can be changed at any point during the execution. This difference is aggravated because parallel programmers tend to make parallel regions large to avoid overheads.

If there are more than two MPI processes running in the same node, the algorithm must decide to which MPI process (of the ones still running) lends the idle CPUs. An idle CPU

will be given to the first process that is available to spawn the new thread, In the case of an MPI+OpenMP application the first MPI process starting a new parallel region, and in the case of an MPI+SMPSs application the first MPI process with a thread outside a task.

The CPUs will be assigned one by one. That is, if the process that is blocked and lending the CPUs had two threads, the first MPI process that *sees* the available CPUs will get one, and the other one will remain available for another (or the same) MPI process to claim it.

5. Environment, methodology and applications

5.1. Environment

As our target was a clustered architecture, all our experiments have been run in Marenostrum 2 [21]. Marenostrum 2 is based on Power PC processors, its nodes are JS21 blades with two IBM Power PC 970MP 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.

5.2. Methodology

The different metrics used in this work:

- **Speedup or elapsed time:** We have used speedup or elapsed time as the main performance metrics ($speedup = \frac{parallel_execution_time}{Serial_execution_time}$).
- **Efficiency:** Percentage of the CPU time consumed that is running pure application code[1].
  $Efficiency = \frac{useful_cpu_time}{elapsed_time * cpus} * 100$
  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.

Speedup evaluates how well the application is running compared with 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 *Speedup* and *Time* have been computed as the average of 5 identical executions. The *Efficiency* and *Cpus used* have been computed from a trace of an execution. The trace have been obtained using the Extrae library [22] and analyzed with Paraver [23].

The performance section is organized in three subsections that correspond to the steps followed during the analysis. The steps are the following:

- **Extensive Performance evaluation**: An in-depth analysis of the performance obtained by each application in order to detect those cases were LeWI does not reach the expected performance.
- **Modeling parallelization characteristics that limit the automatic load balancing potential**: Reproduce the parallelization characteristics that limit the capacity of LeWI, model those characteristics with the synthetic benchmark and validate our hypothesis.
- **Improving automatic load balancing**: Apply the guidelines to some of the real applications to verify our hypothesis.

[1] Pure application code: Code inside the application, not including runtime overheads (OpenMP or SMPSs runtime code) nor MPI calls

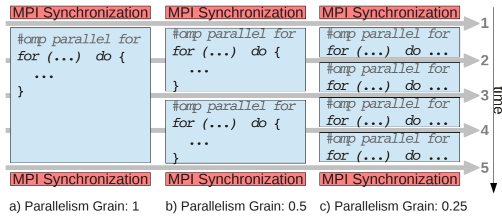

Figure 2: Parallelism grain explanation

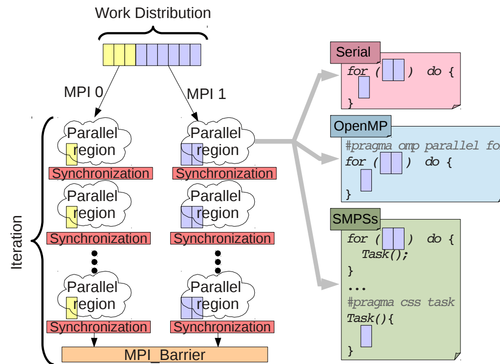

Figure 3: PILS benchmark

5.3. Applications

We will explain and analyze 3 different benchmarks (one of them developed specially to test and evaluate load balancing mechanism) and 2 real applications used in production. We choose these applications because load imbalance is an important problem for their performance.

5.3.1. PILS: Parallel ImbaLance Simulation

PILS (Parallel ImbaLance Simulation) is a synthetic benchmark to model hybrid parallel applications (MPI+OpenMP and MPI+SMPSs).

The different parameters are the following:

- *Programming Model:* MPI, MPI+OpenMP or MPI+SMPSs
- *Load Distribution:* We can introduce the load balance of the application as the different loads for each MPI process.
- *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 ($Par.Grain = \frac{1}{Par.Regions}$). You can see an example about the relationship between Parallelism Grain and the number of parallel regions in Figure 2.
- *Iterations:* Number of times the algorithm will be executed.

The core of the synthetic benchmark is a function that will do several floating point operations without data involved.

In Figure 3 we can see a schematic representation of PILS. The work load is given at the beginning of the execution to each MPI process (work distribution). This work load 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 a SMPSs barrier. At the end of the iteration, there is an MPI barrier to synchronize all the processes.

5.3.2. Benchmarks: BT-MZ (NAS Benchmarks) and LUB

The BT application is one of the benchmarks in the NAS Multizone suite [24]. The original version is a hybrid parallelization of MPI+OpenMP that we will see in the charts as *OMP1*. We have modified this benchmark to be parallelized with SMPSs (labeled *SMPSs*).

LUB is a kernel performing a LU matrix factorization [24]. The structure is a two dimensional matrix organized by blocks. The data is distributed by blocks of rows among the different MPI processes. In figure 4 we can see a schematic representation of the LUB kernel that consists of several iterations and each iteration of four steps.

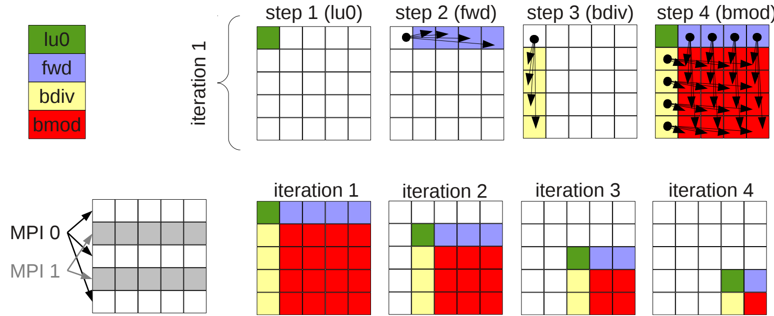

Figure 4: LUB behavior

This application is available in MPI+SMPSs and MPI+OpenMP. The MPI+SMPSs parallelization consider each block as a task, while the MPI+OpenMP version parallelizes the loops executing the fwd, bdiv and bmod blocks.

5.3.3. Production Codes: Gromacs and Gadget

GROMACS [25] is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles.

We started with an MPI version of GROMACS and parallelized some parts of its code to have a hybrid MPI+SMPSs version. It is important to mention that this application was not parallelized completely with SMPSs. For this reason, the executions are done like the MPI only version, meaning that we run 4 MPI processes per node (Marenostrum nodes have 4 cores). SMPSs parallelism is only exploited when we need to load balance.

The imbalance of this application depends heavily on the number of MPI process used, for this reason we will be doing experiments with different number of nodes. We will use from 1 to 64 nodes (4 to 256 CPUs).

Gadget [26] 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 original version of Gadget was MPI only. 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.

In this application we only parallelized the parts of the code that could benefit from the load balancing mechanism (like Gromacs), they were 3 loops out of 35.000 lines of code. The input we used runs in 800 CPUs and has a high load imbalance.

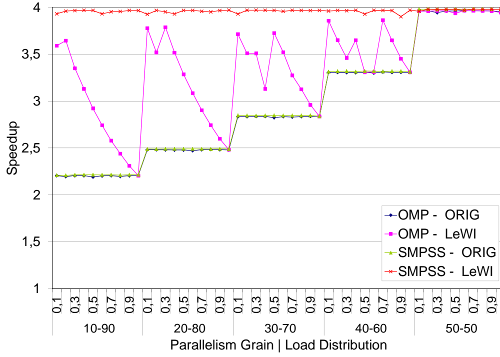

Figure 5: PILS 2 MPI processes

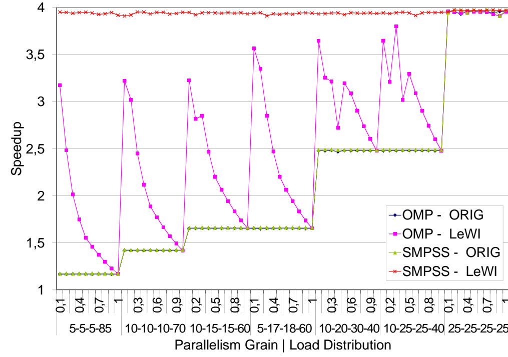

Figure 6: PILS 4 MPI processes

In table 1 we can see for each application used its original version, the available versions for execution, and the amount of nodes in which the application will be executed.

Application	**Original version**	**MPI+ OpenMP**	**MPI+ SMPSs**	**Executed in nodes (cpus)**
PILS	MPI+OpenMP MPI+SMPSs	X	X	1 (4)
BT-MZ	MPI+OpenMP	X	X	1,2,4 (4,8,16)
LUB	MPI+OpenMP MPI+SMPSs	X	X	1,2,4 (4,8,16)
Gromacs	MPI		X	1..64 (4..256)
Gadget	MPI	X		200 (800)

Table 1: Summary of applications used for the evaluation

6. Performance analysis: Sources of efficiency loss with LeWI

6.1. PILS: Parallel ImbaLance Simulation

Figures 5 and 6 show the speedup obtained with different configurations of PILS with 2 and 4 MPIs respectively.

In the Y axis, we present speedup respect the serial execution, and in the X axis we present the load distribution (coarse grain scale) and the parallelism grain (fine grain scale).

ORIG versions achieve exactly the same performance with OpenMP and with SMPSs, and because of this, only the SMPSs performance values are visible since they completely cover OpenMP values.

The LeWI versions show us two interesting facts. On one hand, the LeWI execution of both OpenMP and SMPSs versions have a higher speedup than the ORIG execution of the same version; therefore, LeWI improves the performance of hybrid microbenchmarks independently of the level of imbalance that it presents.

On the other hand, we can observe a big difference in performance results of SMPSs and OpenMP versions with LeWI. SMPSs+LeWI version is close to the optimal in both versions (2

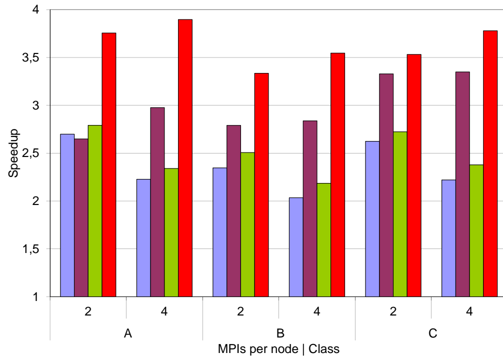

Figure 7: BT-MZ in 1 node (class A, B, C)

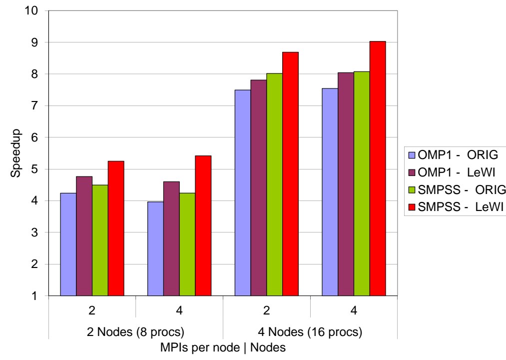

Figure 8: BT-MZ in 2 and 4 nodes (class C)

and 4 MPIs), obtaining almost a speedup of 4. While OpenMP+LeWI version is significantly influenced by parallelism grain, needing the application to have enough malleability to obtain a good performance when load balancing.

6.2. BT-MZ

Figure 7 show the BT-MZ speedup respect the serial execution when running in one node. We can observe how in all cases the baseline with SMPSs is slightly better than with OpenMP, but the maximum improvement is less than 7% so we can say they have *similar* performance. However, when running with LeWI, the differences with the baseline and between programming models are quite significant. The SMPSs-LeWI version achieves between 97% and 80% of the maximum speedup with the different configurations. OMP-LeWI version can obtain up to an 83% of the maximum speedup (note that the maximum speedup is 4 because we are running in a single node with 4 cores).

An interesting observation from this chart is to compare for the same class the original execution without load balancing. We can see that the speedup with 2 MPI processes is always better than with 4 MPI processes, independently of the programming model (OpenMP or SMPSs). This is because this application presents a high level of imbalance between MPI processes, thus using more MPI processes causes a more inefficient execution. When using load balancing the best performance is obtained when running with 4 MPI processes instead of 2. This is because LeWI has more flexibility to load balance when running with more MPI processes.

The relevance of these observations is that, in some cases, the best performance that we can achieve from an application when using load balancing is not obtained using the best configuration of the application, but the one that helps the load balancing mechanism.

In Figure 8 we can see the speedup of BT-MZ when running in 2 and 4 nodes respectively. Since each node has four CPUs, the maximum number of MPIs per node is four processes.

As before, the performance when running the original SMPSs version is slightly better than the original OpenMP version. Also, the LeWI version always obtains a better performance than the original execution.

In this case, the maximum performance is limited to 67% of the ideal speedup for the SMPSs version with LeWI and 2 MPI processes. As we will see in next sections, this is because we are running in several nodes and inter-node load distribution cannot be handled by LeWI.

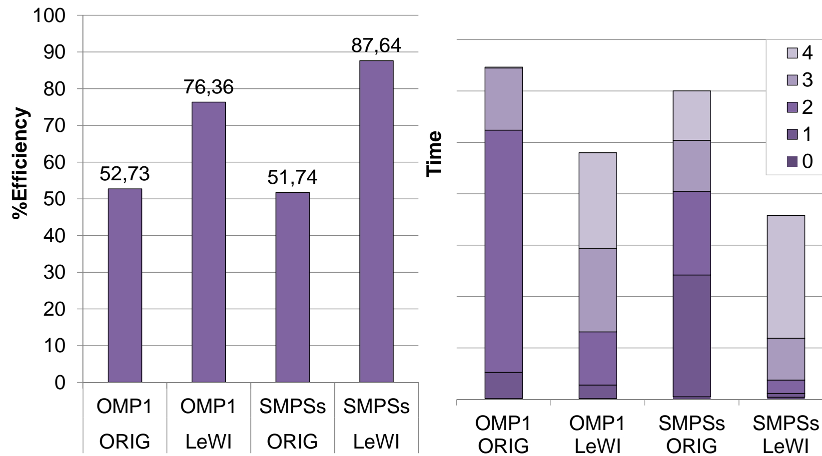

Figure 9: BT-MZ Efficiency and cpus used (1 node)

In Figure 9 we can see two charts both obtained from real executions of the application in one node, with 4 MPI processes and class A. The left hand chart shows the Efficiency obtained with every combination of programming models and load balancing. In the right side chart, we see the execution time of each combination, and amount of time that was running with the different number of CPUs.

The amount of time the application is running with the different number of CPUs gives us an interesting extra information. We can see how OMP1 ORIG version runs most of the time with 2 CPUs (and a negligible amount of time with 4 CPUs) and how that situation is improved by LeWI. In the case of OpenMP and LeWI, the application is running most of the time with 3 and 4 CPUs. With SMPSs and LeWI it is still better, running around the 75% of the time with 4 CPUs.

Even though results are better with SMPSs than with OpenMP, they are still far from the potential showed by PILS in the previous section. In section 7 we will show which parallelization characteristics can limit the performance improvements.

6.3. LUB

Figure 10 shows the LUB speedup when running in 1 node, we can compare the speedup of the different versions (OpenMP or SMPSs and Original or LeWI) combined with different block sizes. Results are quite similar to the presented in the previous section with BT-MZ application. Baseline configurations (ORIG) reaches similar speedup with OpenMP and SMPSs, but LeWI is able to improve SMPSs version more than the OpenMP version. While LeWI improves the speedup of OpenMP from 2,9 (in average) to 3,5, it is able to improve SMPSs from 2,9 (also in average) to close to 4.

In Figure 11 we can see the speedup obtained when running LUB in 2 and 4 nodes. We can see that, in multiple nodes, LeWI is also able to help SMPSs more than OpenMP, and the general comparison between the different versions is similar at the execution in one node. In absolute values, the speedup obtained is not close to the optimum as obtained when running in a single node. The best speedup in two nodes is close to 6 obtained by the SMPSs and LeWI version and

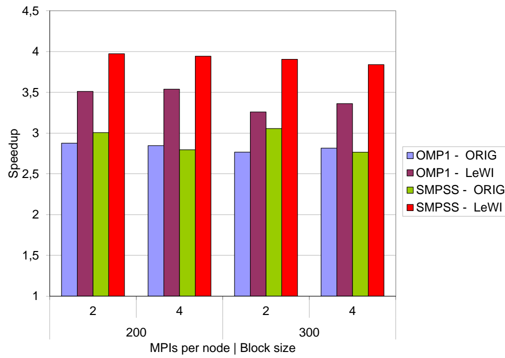

Figure 10: LUB running in 1 node

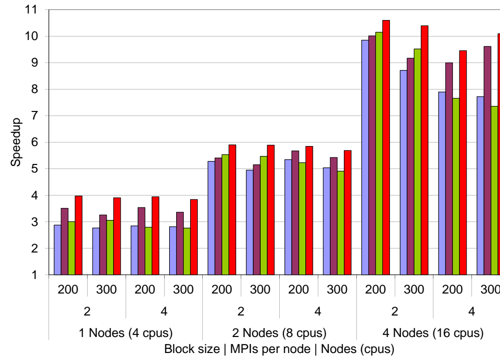

Figure 11: LUB running in 2 and 4 nodes

far from the ideal speedup of 8. In four nodes, the maximum speedup obtained by the SMPSs and LeWI version is 10 being only the 62% of the ideal speedup of 16.

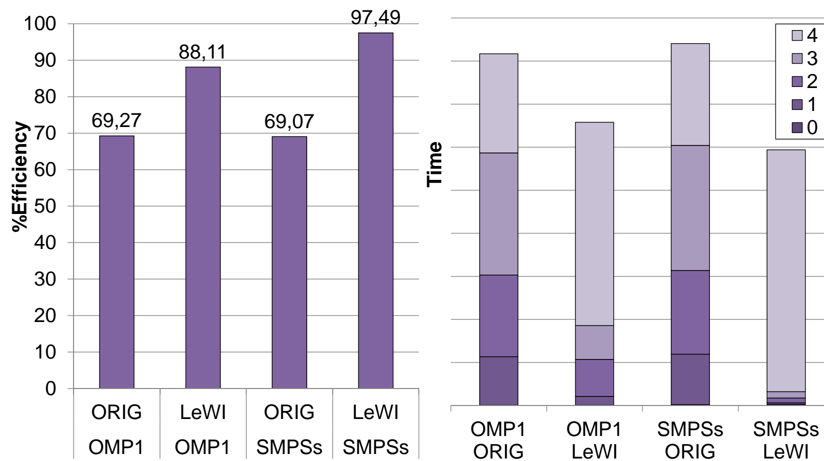

Figure 12: LUB Efficiency and Time(number of cpus used)

Figure 12 shows two charts obtained from executions of LUB in one node and block size 200. The left side chart 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 is better than the one reached with BT specifically the LeWI-SMPSs version (up to 97% compared with 87% in BT).

The right side chart of Figure 12 show the time each version is running with 0, 1, 2, 3, and 4 CPUs. In this application, we can see that the two original versions (SMPSs-ORIG and OMP1-ORIG) have a very similar behavior in the use of the CPUs (this is also reflected in the Efficiency measure). Rather, there is a significant difference between LeWI versions. When running with SMPSs and LeWI almost 95% of the time the application uses the 4 CPUSs available while in the execution with OpenMP and LeWI, the four CPUs are used 71% of the time.

In this application, we can observe how starting from the same base point LeWI is able to help the SMPSs programming model much more than the OpenMP programming model. In

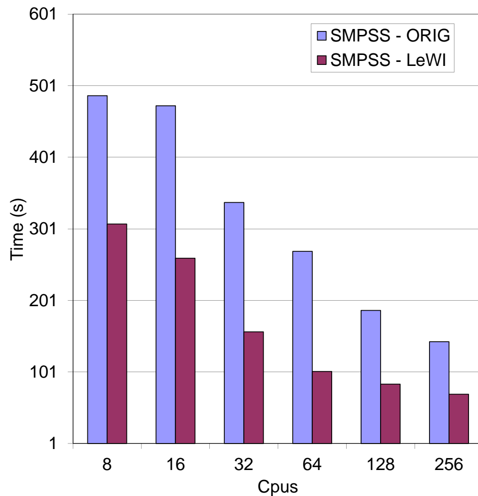

Figure 13: Gromacs in 4 to 256 cpus (1 to 64 nodes)

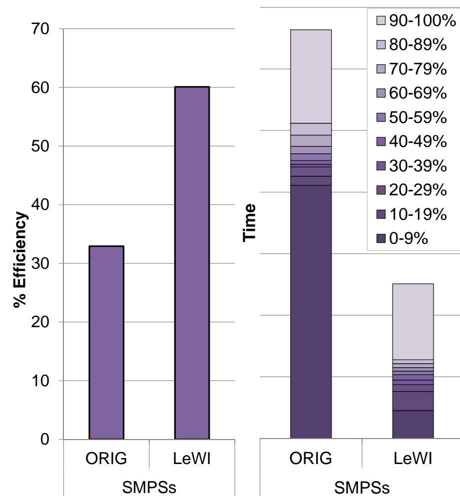

Figure 14: Gromacs Efficiency and number of cpus used

section 7.1 we will study the reason for this difference further.

6.4. Gromacs

In this section, we will compare the MPI+SMPSs version of the application when running with LeWI and without it (*ORIG*). It is important to notice that when running without LeWI the application will not be able to use the SMPSs parallelization, it will be like an MPI only execution.

In Figure 13 we can see the execution time of Gromacs when running in 4 to 256 cores (1 to 64 nodes). The LeWI version improves the execution of the original run in all cases. It can run in 65% less time in the case of 64 cores, or what is the same speed up the execution by almost 3 with the same number of computational resources.

In the left hand side of Figure 14 we can see the Efficiency obtained by the execution of Gromacs in 64 cores (16 nodes). We can see how the efficiency of the original execution is very low, only 32% and when running with LeWI it is almost doubled to 62% of efficiency.

In the right hand chart of Figure 14 is shown the percentage of the available CPUs that the application is using. The top color means that between the 90 and 100% of the CPUs are being used, we can see that the amount of time that we are using all the CPUs is very similar in the two versions. The main difference between the two executions is in the time that they are using between 0 and 9% (between 0 and 5 cores of the 64 available).

We can understand the behavior of the application and its performance better with Figures 15 and 16. In this charts, we can see the same information separated by nodes.

In Figure 15 is shown the efficiency in each node, we can see how the average efficiency per node in the original execution is 40% while the LeWI version can achieve a 60% of efficiency in average. We observe that there are two nodes that achieve a higher efficiency, when running with LeWI more than 80% of efficiency, they are nodes 6 and 7. From this, we can deduce that these two nodes have a higher computational load than the other nodes, and the low efficiency of the other nodes is because they do not have work to do and are waiting for nodes 6 and 7 to finish their computation.

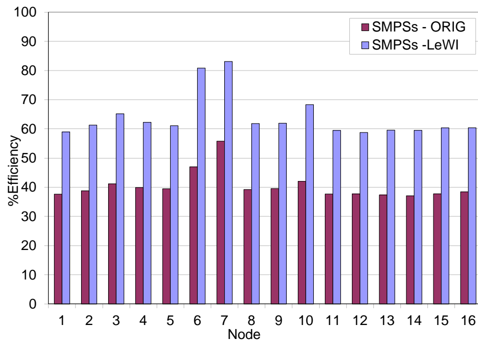

Figure 15: Gromacs Efficiency per node

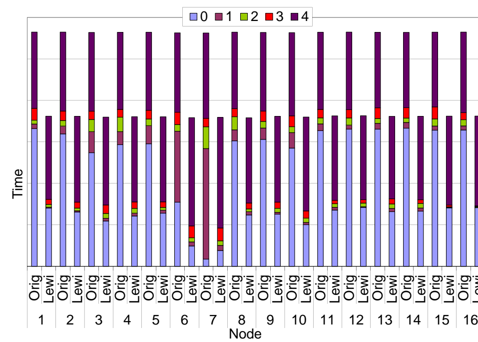

Figure 16: Gromacs Cpus used per node

If we look at Figure 16 at first sight we can see that almost all the nodes reduce at less than half the time they are doing nothing (using 0 CPUs). For all of them, there is still some time that they are not using any CPU. This time is when they are waiting for node 6 or 7. As we can see the nodes 6 and 7 spend much less time running with 0 CPUs than the others.

We can say that this application presents internode imbalance that LeWI cannot solve. We will analyze it further and try to improve it in section 8.2.

6.5. Gadget

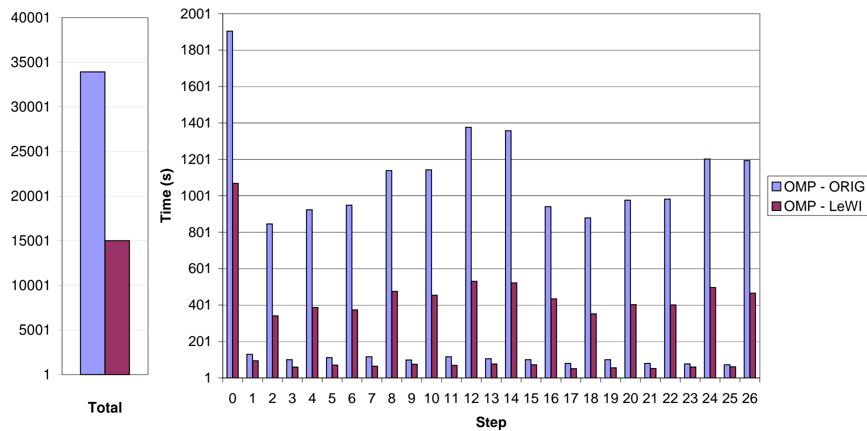

Figure 17: Gadget running in 800 cpus (200 nodes)

In Figure 17 we can see the execution time spent by the application when running with and without LeWI. In the left side of the figure, we can see the time in seconds of the whole application. In the right side, as the application is organized in time steps independent from each other, we can see the time spent in each time step.

The LeWI execution is able to speed up the application 2.5 times with the same number of computational resources.

7. Analyzing parallelization aspects that limit LeWI performance

In the previous section, while analyzing the performance of the applications, we pointed out different characteristics of the parallelization of the applications that could be limiting the performance when using the automatic load balancing.

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.

The three aspects that we have identified are the following:

- Parallelism Grain in OpenMP applications
- Task duration in SMPSs applications
- Distribution of MPI processes among computation nodes

7.1. Parallelism Grain in OpenMP applications

The Parallelism Grain is the amount of computation that can be done in parallel between MPI calls. Typically in parallel applications the higher the Parallelism Grain the better.

In Figure 2 we can see a schematic representation of a piece of hybrid MPI+OpenMP code. The usual way to parallelize this code (if possible) would be version *a*. Because it is the most efficient option. As we explained in section 4.1 LeWI improves the performance of applications changing the number of OpenMP threads during the execution. As OpenMP threads can only be changed outside parallel regions, in version *a* it wouldn't be possible to change the number of threads between the 2 MPI calls.

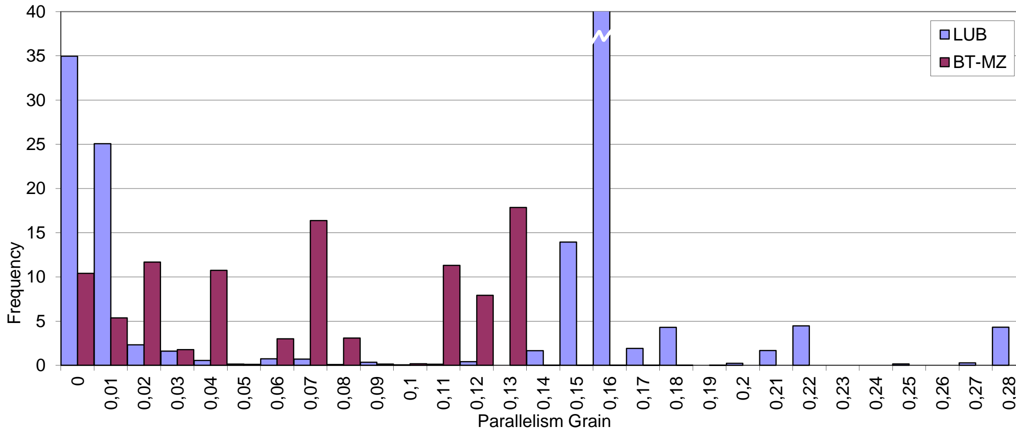

Figure 18: Parallelism grain of LUB and BT-MZ

If we divided the work in 2 parallel regions (version *b*, parallelism grain 0.5), the number of threads could be changed at point 2. If we had 4 parallel regions (version *c*, parallelism grain 0.25), there would be 3 points in the code (2, 3 and 4) where the number of threads could be updated. For this reason the lower parallelism grain in the parallelization the higher flexibility we have to improve the performance using the load balancing algorithm LeWI.

We analyzed the parallelism grain in LUB and BT-MZ applications and presented it in Figure 18 (This data was obtained from Paraver trace of a real execution). In the x axis, we can see the

parallelism grain that goes 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.

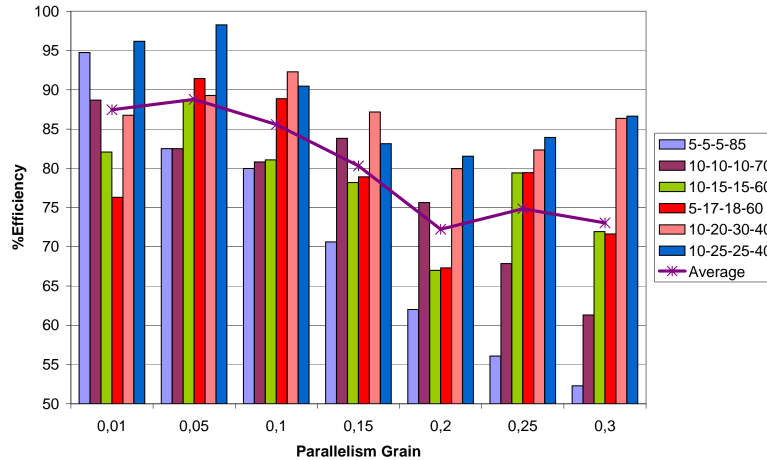

Figure 19: PILS efficiency depending on parallelism grain

In Figure 19 we show the Efficiency obtained with PILS 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 in section 5.3.2 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 more difficult 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 obtained with 5-17-18-60 distribution.

7.2. Task duration 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 small enough to have enough flexibility to load balance the execution.

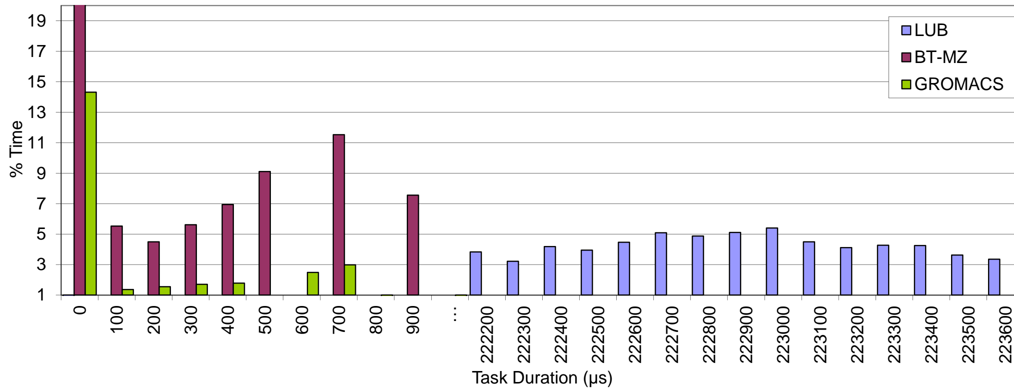

Figure 20: Task duration of LUB, BT-MZ and GROMACS

Figure 20 shows the duration of tasks in microseconds of LUB, BT-MZ and Gromacs (for LUB and BT-MZ executions in one node with 4 MPIs, for Gromacs in 16 nodes with 64 MPIs). 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 vary from less than 100 microseconds to 1000 microsecond, Gromacs' tasks duration are between 100 and 700 microseconds, but most of them are around 100 microseconds. LUB has a more coarse grain task duration between 222000 and 223000 microseconds.

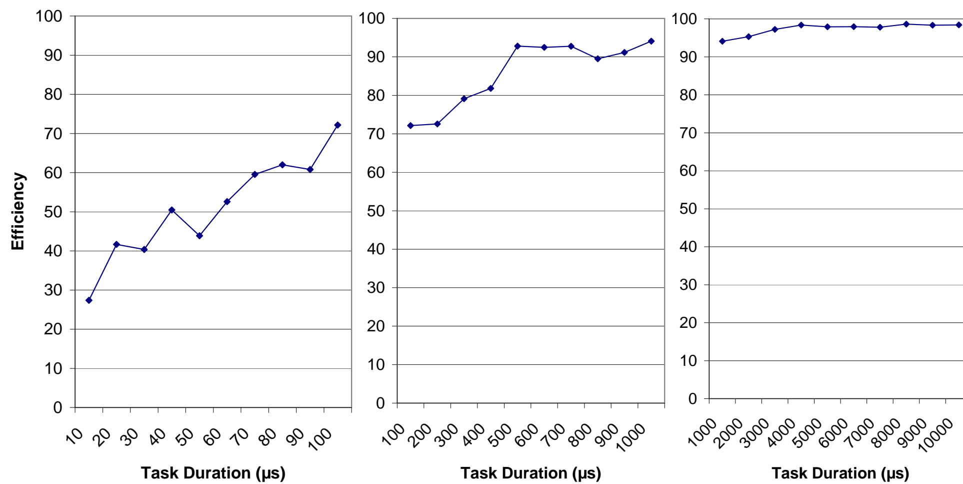

Figure 21: PILS efficiency depending on the task duration

We can compare the efficiency obtained with PILS when running with similar task durations in Figure 21. 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 is reached with task duration of around 4000 microseconds obtaining an efficiency of 98%.

If we compare the efficiency obtained with BT-MZ in section 5.3.2 with the results obtained with PILS, we see that BT-MZ SMPSs version obtained an efficiency of 87% we can see that it is the average efficiency obtained by PILS with tasks between 100 and 900 microseconds (the duration of BT-MZ tasks). 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 section 5.3.2 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.

Finally, Gromacs had an efficiency of 62% when running with LeWI, and we saw that most of its tasks had a duration between 0 and 100 microseconds. Checking the first chart in Figure 21 we can see that this efficiency correspond to executions of PILS with tasks durations between 70 and 90 microseconds. It is interesting to observe that the inefficiency obtained in Gromacs executions is a result of its task granularity, not only to internode imbalance as we pointed in our preliminary analysis.

8. Hints to improve the performance when using LeWI

In this section we will modify some of the applications analyzed, We will adjust the parallelization aspects that we identified in the previous section that could limit the performance of the automatic load balancing.

8.1. Parallelism Grain in OpenMP applications

Parallelism grain is the main aspect that can limit the performance of the automatic load balancing for MPI+OpenMP applications. It is specially sensitive because what the automatic load balancing mechanism needs is against the main believe of parallel programmers.

The tendency in OpenMP parallelization is to decrease the number of parallel regions to improve the performance of the programming model. While LeWI needs parallel regions to improve the efficiency of the application.

In the previous section, we analyzed the parallelism grain of different applications. We saw that the parallelism grain of BT-MZ and GROMACS was small enough to satisfy the load balancing mechanism needs. For the LUB application, the parallelism grain was limiting the efficacy of the automatic load balancing.

We modified the LUB parallelization by parallelizing some inner loops instead of the outer ones, with this change the parallelism grain of the application decreases.

In Figure 22 we can see the parallelism grain of the original parallelization and the modified one for the LUB application. The series labeled as OMP1 is the original LUB application while the series labeled OMP2 is the modified version. We can see how the parallelism grain of LUB has clearly been reduced. With the new version, the parallelism grain is 0,01 or less most of the time.

Figure 23 show the performance obtained with LUB when running in a single node. We can see how the best performance is obtained when running with LeWI the modified version (series

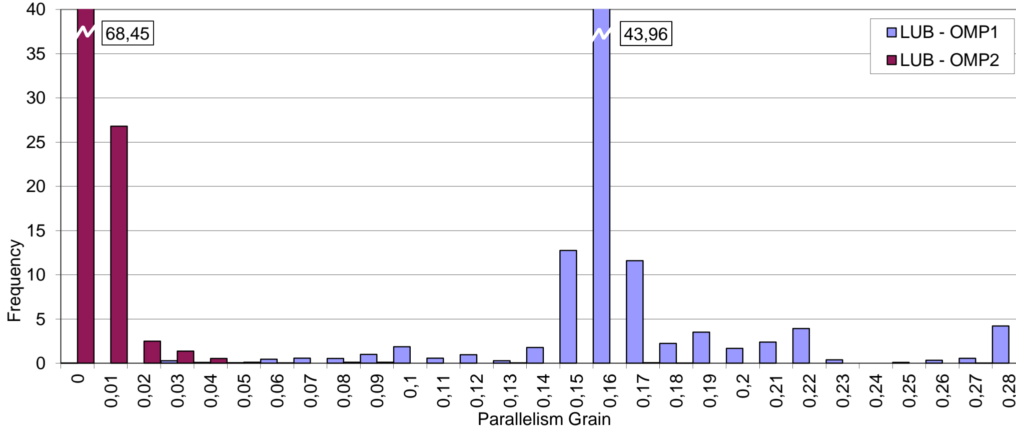

Figure 22: Parallelism grain of LUB modified

labeled OMP2 - LeWI). The new version obtains a speedup close to the optimum one (4) in all cases.

The new parallelization we suggest decreases the performance of the application because it stresses out the programming model. When running with LeWI, it can overcome that loss. The speedup obtained by the OMP2-LeWI version is better than the original application (OMP1-ORIG) and even the original application with load balancing (OMP1-LeWI)

8.2. MPIs distribution in MPI applications running in several nodes

When analyzing the efficiency obtained by applications running in several nodes, we identified the distribution of MPI processes among the nodes as a factor that can affect the performance. In this case, it is not a parallelization decision but an execution decision. This makes it even easier to change it and as we will see with an important impact in the efficient use of the resources.

Usually by default the applications will be executed with consecutive assignment of MPIs (i.e. an application running in 2 nodes with 8 MPI processes will run MPI processes 0, 1, 2 and 3 in node 0 and MPI processes 4, 5, 6, and 7 in node 1). Moreover, the communication pattern of some applications is designed to work with this kind of distribution.

In almost all HPC systems, we can choose a different distribution or even decide one by hand. We are using a common distribution known as CYCLIC that distributes the MPI processes in a cyclic way among the nodes (i.e. an application running in 2 nodes with 8 MPI processes will run MPI processes 0, 2, 4 and 6 in node 0 and MPI processes 1, 3, 5 and 7 in node 1).

All the executions shown until this point have been done using a consecutive assignment of MPI processes.

In Figure 24 we can see how the distribution of MPI processes affect the performance of the BT-MZ application when running in 2 and 4 nodes. We see 4 series representing the 2 MPI distributions and the application running with and without LeWI. We can observe that the speedup of the application is much better when running with LeWI and the cyclic distribution. It is important to notice that the original execution is not improved when using the cyclic distribution (ORIG-CYCLC compared with ORIG-BASE). This means that the communication pattern of this application is not affected by the MPI distribution. It is just the automatic load balancing mechanism that is benefited with the MPI distribution.

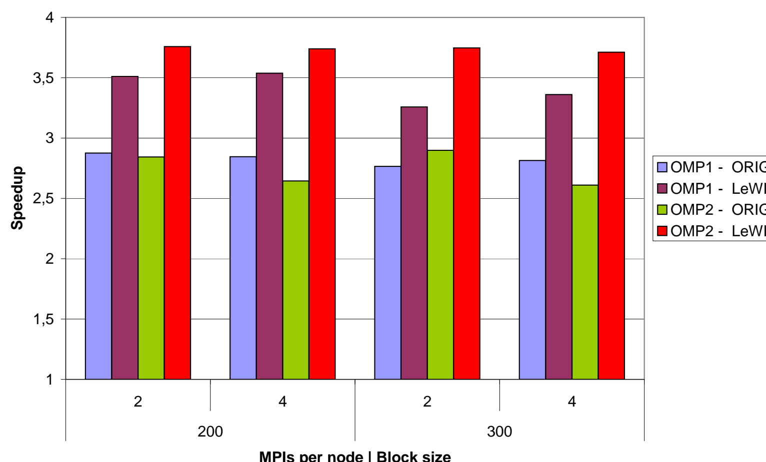

Figure 23: LUB performance with modified parallel grain

The impact of the MPI distribution in the performance of Gromacs can be seen in Figure 25. In this case, the data that are shown is execution time in seconds. We can see that, in this application, the cyclic distribution is not always beneficial for the application. In almost all cases, the performance is improved with the cyclic distribution and specially in executions with a high number of nodes. In Gromacs, we can see that the original application is also affected by the distribution of MPI processes. This means that the communication pattern is decisive in the performance of the application.

In Figure 26 we can see the execution time of Gadget with different MPI distributions with and without load balancing. Gadget in general is benefited with the cyclic distribution. Although, in the detail of the time steps, we can see that some of them are affected negatively by the cyclic distribution.

9. Conclusions and Future Work

The load balancing algorithm LeWI offers an easy and flexible way of improving the performance of hybrid applications. LeWI can be used out of the box to speedup hybrid applications with imbalance problems.

We have analyzed in detail the performance of LeWI with very different applications. On one hand, 3 benchmarks that allowed us to test the behavior changing several parameters. On the other hand, 2 production codes that allowed us to see how LeWI can be applied to real applications with imbalance problems.

The experiments showed that LeWI can handle very different situations, e.g. executions in one node, in several nodes or in a high number of nodes (Gadget with 800 CPUs, 200 nodes).

We have also seen that LeWI can work with different programming models (in this paper MPI+OpenMP and MPI+SMPSs), but more important that it can be easily extended to be used with other programming models.

Although LeWI improves the performance of all the applications tested, we detected that, in some cases, the efficiency obtained was not close to the theoretical maximum. In this paper, we have focused in finding the sources of efficiency loss when using LeWI.

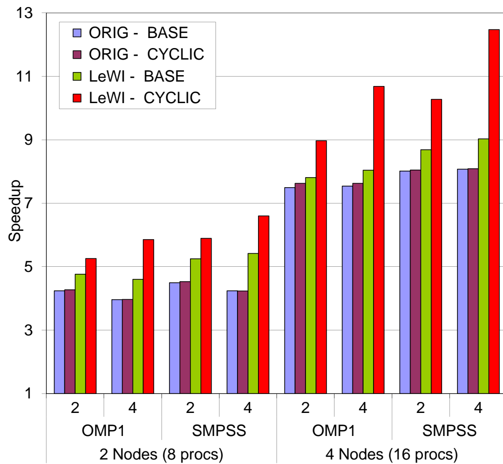

Figure 24: MPIs' distribution impact in BT-MZ

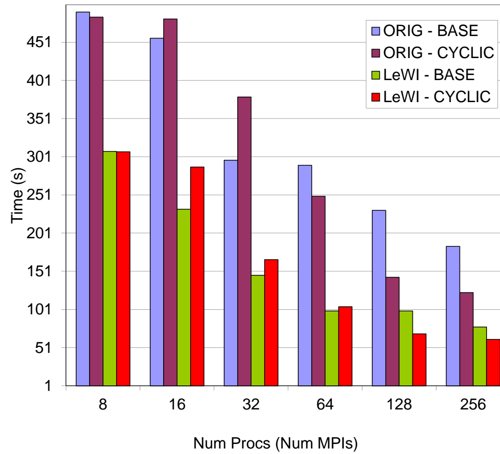

Figure 25: MPIs' distribution impact in Gromacs

We have identified several characteristics in the application structure that can limit the efficiency obtained when using LeWI. Also, some programming practices that can improve the performance of the application.

We will summarize these findings of the paper as the following hints to the parallel application developers.

The first advice and maybe the most important one is to forget the previous prejudices. Our indications will improve the execution of applications when running with LeWI, but in some cases they can result in a worse performance of the application without LeWI. For example, the BT-MZ application in one node with 2 MPI processes and 2 SMPSs threads per process obtains a speedup of 2.5 and with 4 MPI processes and 1 SMPSs thread per process the speedup is 2.2. When running with LeWI, the best configuration is to run 4 MPI processes and 1 SMPSs thread per process (the SMPSs level is used only for load balancing purposes) obtaining a speedup of 3.5 compared to 3.2 that is the speedup obtained when running with 2 MPI processes and 2 SMPSs threads per process.

For parallel application developers will be interesting to know that a very good improvement in performance can be obtained when using LeWI without needing to hybridize the whole application. For instance, both Gadget and Gromacs applications that originally were MPI only applications were partially parallelized with OpenMP and SMPSs respectively. We were able to obtain improvements in the performance of more than 40% in both cases when using LeWI.

A decision that can have a great impact in the performance of the application when using LeWI is the programming model used, we have seen that the more malleable the programming model, the best results we can obtain. In our examples, SMPSs obtains better results than OpenMP because it is more malleable.

Regarding the parallelization decisions, 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 67%, it was increased to 80% using LeWI, but modifying the parallelism grain when using LeWI we obtained an efficiency up

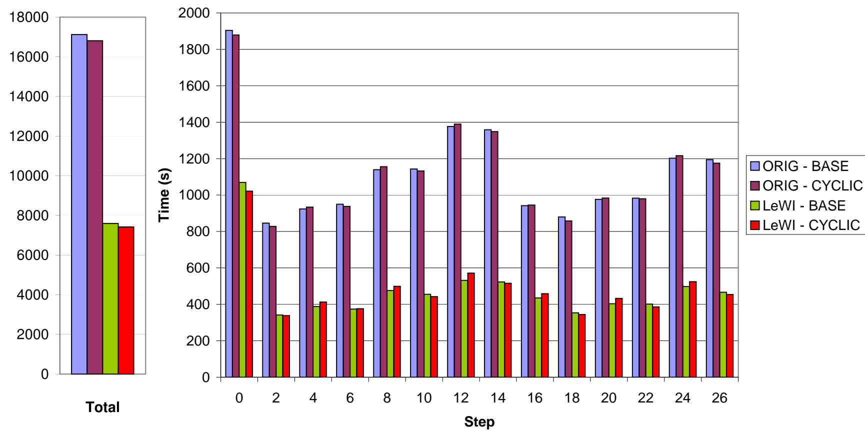

Figure 26: Gadget: Distribution of MPIs impact

to 92.5%. 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 obtain a good efficiency, but it would be recommendable to have tasks of more than 500 microseconds of duration.

Finally, we have seen that the MPIs distribution between the nodes can have a big impact in the performance of the application when using LeWI. Changing the distribution of MPIs can increase the efficiency from 50% when running the original application and 56% when running with LeWI to 78% when running with LeWI and using a cyclic distribution. It is difficult to generalize if changing the MPI distribution will improve the performance of applications or not, as it depends highly on the communication pattern of each application. 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 cyclic distribution of MPI processes obtains better results than the base distribution.

Some of the advices we are giving here can be applied without modifying the application, other ones can be considered when starting to parallelize the application, but in either case they are not a time consuming process neither do need a previous analysis of the application behavior. In all cases will resume in a more efficient use of the computational resources by the application if used in conjunction with LeWI.

Acknowledgments

We thankfully acknowledge the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

References

[1] M. Garcia, J. Corbalan, J. Labarta, LeWI: A Runtime Balancing Algorithm for Nested Parallelism, in: Proceedings of the International Conference on Parallel Processing (ICPP09), 2009.

[2] G. Karypis, V. Kumar, MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0, 1995.

[3] L. V. Kale, S. Krishnan, Charm++: Parallel Programming with Message-Driven Objects, in: Parallel Programming using C++, 1996, pp. 175–213.

[4] M. A. Bhandarkar, L. V. Kalé, E. de Sturler, J. Hoeflinger, Adaptive load balancing for mpi programs, in: ICCS '01: Proceedings of the International Conference on Computational Science-Part II, 2001.

[5] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. P. Pabico, R. 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.

[6] S. Meraji, C. Tropper, Optimizing techniques for parallel digital logic simulation, Parallel and Distributed Systems, IEEE Transactions on 23 (2012) 1135–1146.

[7] L. Smith, M. Bull, Development of mixed mode mpi / openmp applications, Scientific Programming 9 (2001).

[8] 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.

[9] F. Cappello, D. 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.

[10] Y. Zhang, M. Burcea, V. Cheng, R. Ho, M. Voss, An adaptive openmp loop scheduler for hyperthreaded smps, in: ISCA PDCS, 2004, pp. 256–263.

[11] O. Sievert, H. Casanova, A simple mpi process swapping architecture for iterative applications, International Journal of High Performance Computing Applications 18 (2004) 341–352.

[12] K. E. Maghraoui, B. Szymanski, C. 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.

[13] A. Spiegel, D. an Mey, C. H. Bischof, Hybrid parallelization of cfd applications with dynamic thread balancing, in: PARA, 2004, pp. 433–441.

[14] A. Duran, M. Gonzàlez, J. Corbalán, Automatic thread distribution for nested parallelism in openmp, in: Proceedings of the 19th annual international conference on Supercomputing (ICS), 2005, pp. 121–130.

[15] 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.

[16] R. Rabenseifner, G. Hager, 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, IEEE Computer Society, 2009, pp. 427–436.

[17] L. Smith, M. Bull, Development of mixed mode mpi/openmp applications, Scientific Programming 9 (2001) 83–98.

[18] OpenMP Architecture Review Board, OpenMP application program interface version 2.5, `http://www.openmp.org/mp-documents/spec25.pdf` - Last accessed Nov. 2012, 2005.

[19] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: Proceedings of the IEEE International Conference on Cluster Computing, 2008.

[20] V. Marjanović, J. Labarta, E. Ayguadé, 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.

[21] Barcelona Supercomputing Center, Marenostrum, `http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture/` - Last accessed Nov. 2012.

[22] Barcelona Supercomputing Center, Extrae instrumentation package, `http://www.bsc.es/paraver` - Last accessed Nov. 2012.

[23] V. Pillet, J. Labarta, T. Cortés, S. Girona, Paraver: A tool to visualize and analyze parallel code, Transputer and occam Developments (1995) 17–32.

[24] H. Jin, R. F. V. der Wijngaart, Performance characteristics of the multi-zone nas parallel benchmarks, J. Parallel Distrib. Comput. 66 (2006) 674–685.

[25] E. Lindahl, B. Hess, D. van der Spoel, Gromacs 3.0: a package for molecular simulation and trajectory analysis, Journal of Molecular Modeling 7 (2001) 306–317. 10.1007/s008940100045.

[26] V. Springel, N. Yoshida, S. D. White, Gadget: a code for collisionless and gasdynamical cosmological simulations, New Astronomy 6 (2001) 79 – 117.