

# A Dynamic Load Balancing approach with SMPSuperscalar and MPI

Marta Garcia<sup>1,2</sup>, Julita Corbalan<sup>1,2</sup>,  
Rosa Maria Badia<sup>1,3</sup>, and Jesus Labarta<sup>1,2</sup>

<sup>1</sup> Barcelona Supercomputing Center (BSC)

<sup>2</sup> Universitat Politecnica de Catalunya (UPC)

<sup>3</sup> Artificial Intelligence Research Institute (IIIA) Spanish National Research Council (CSIC)

**Abstract.** In this paper we are going to compare the performance obtained with the hybrid programming models MPI+SMPSs and MPI+OpenMP, specially when executing with a Dynamic Load Balancing (DLB) Library. But first we will describe the SMPSuperscalar programming model and how it hybridizes nicely with MPI. We are also explaining the load balancing algorithm for hybrid applications, LeWI, and how it can improve the performance of hybrid applications. We will also analyze why SMPSs is able to exploit the benefits of LeWI further than OpenMP. The performance results will show not only how the performance of hybrid applications can be improved with LeWI but also the benefit of using a hybrid programming model MPI+SMPSs for load balancing instead of MPI+OpenMP.

## 1 Introduction

In an HPC environment using a hybrid programming model is often a good approach to obtain a good performance when parallelizing an application. When we talk about hybrid programming models the first combination that comes to our minds is MPI+OpenMP.

The reason for MPI+OpenMP being the most used hybrid model is its success. Its success is not only because the performance obtained (which is the sum of the good performance obtained by the two programming models on their own). It is also because the flexibility it gives, being able to program clusters of shared memory nodes and the possibility to tackle the parallelization from different approaches, with more or less granularity, using shared memory or communication. Not only that but it has also shown how the use of OpenMP as the second level of parallelism can help load balance the MPI level [1].

In this paper we will talk about the Load Balancing Library, DLB, and a balancing algorithm, LeWI, that can improve the performance of hybrid applications. DLB can load balance an application at runtime without modifying nor analyzing the application. In a previous work [2] we showed the potential of DLB and LeWI when executed with MPI+OpenMP applications. But we also found

a limitation of OpenMP, that prevented the algorithm to obtain all the possible benefit. We are going to explain how using a hybrid programming model with SMPSuperscalar+MPI can overcome this limitation and obtain a better performance for most applications.

We will introduce the basics of SMPSuperscalar, a programming model aimed at shared memory systems. SMPSs is a programming model that has shown to obtain a performance comparable to OpenMP on its own [3]. Not only this but it can also hybridize with MPI offering a powerful hybrid model. We will explain how the MPI+SMPSs approach can help load balance applications with LeWI, and improve the performance obtained by the MPI+OpenMP version of the same application.

The paper is organized as follows: we will first explain the basics of the SMPSs programming model and the different techniques to combine it with MPI. In the next section we will introduce the DLB library and the LeWI algorithm and show how it can benefit from using MPI+SMPSs. In Section 4 we will present the performance evaluation comparing the performance of the MPI+OpenMP and MPI+SMPSs versions of an application with and without LeWI. Finally, we will conclude summing up the results presented in this paper and the future work.

## 2 Hybrid MPI+SMPSuperscalar Programming Model

### 2.1 SMPSuperscalar (SMPSs)

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

We can say that there are two kinds of code in an SMPSs application: the tasks and the serial code. The serial code is all the code that is outside a taskified function. SMPSs threads present a hierarchy. The master thread executes the serial code, creates the tasks and can execute tasks. The slave threads will only execute tasks.

In Figure 1 we can see an example of execution of an SMPSs application. In this example there is just one taskified function, *func\_task*. The master thread starts executing some user code until it reaches the start directive for SMPSs (*css\_start*), when the worker threads are created. At this point of the execution there are no tasks in the task graph yet so the worker threads are idle while

the master thread continues executing the user code. When the master thread reaches a call to a taskified function it will create the task and add it to the task graph. The worker threads will get the tasks that are ready to run from the graph and execute them always respecting the dependences. At some point of the execution the master thread can decide to execute a task (if it has reached a barrier or if there are too many of them in the graph). In the example the master thread executes task 3 after task 1 is finished. At the end of the execution the *css\_finish* is called to join all the threads.

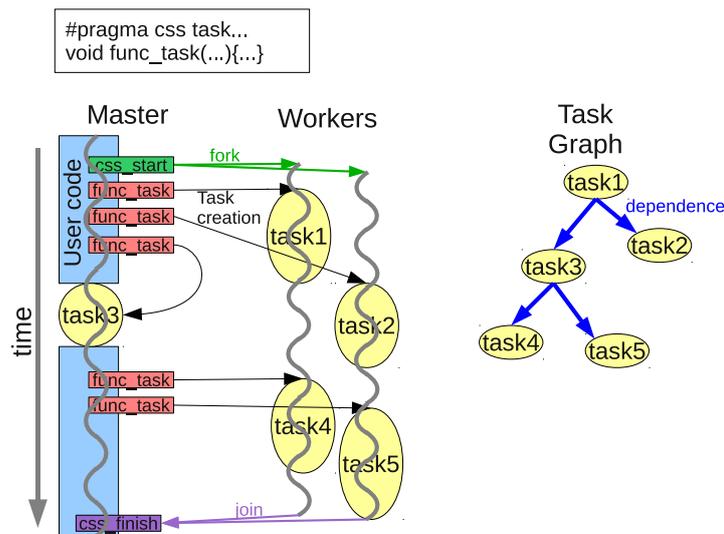


Fig. 1. SMPSSs execution model

The number of threads can be changed at any time during the execution, the only limitation is that a thread will never leave a task before finishing it.

The SMPSSs programming model does not need explicit synchronization between tasks, because the correctness of the execution is ensured by the dependences. But it is important to notice that the serial code, that will be executed in serie by the master thread, can run in parallel with tasks executed by slave threads. To avoid race conditions between tasks and the serial code we need some synchronization mechanisms:

**Barrier:** All the tasks created before the barrier must be finished before the execution can proceed further in the serial code.

**Wait on(variable):** Creates a dependence with the given variable at this point of the code. Therefore, the execution can not proceed from this point until the dependences for the given variable are fulfilled.

These are the basics of the programming model and the runtime. There are more features that we will not explain here, but you can find more details in the documentation [4].

## 2.2 Hybrid MPI+SMPSuperscalar

SMPSuperscalar hybridizes nicely with MPI. We can spawn MPI processes across nodes and exploit the node parallelism with SMPSs.

There are several ways of mixing MPI and SMPSs, the most obvious and easy one would be to have the MPI calls as serial SMPSs code. We must ensure that the data that we want to send or receive is ready with one of the synchronization mechanisms that SMPSs offers (*barrier, wait on...*).

When using MPI+SMPSs we can also encapsulate MPI calls inside SMPSs tasks, this approach allows to overlap communication and computation. In this case the dependences would be satisfied by the runtime if they were correctly set in the code as inputs and outputs of the task containing the MPI call. This approach presents some drawbacks that must be taken into account:

- **Possibility of several concurrent MPI calls:** need to use thread safe MPI
- **Reordering of MPI calls:** Can introduce a deadlock, need to control order of communication tasks.
- **Wasting a core while in a communication task.**

To avoid these limitations SMPSs includes a Communication Thread. The Communication Thread is a mechanism that allows to overlap computation and communication transparently for the programmer. The programmer must mark the tasks that include MPI communication calls with *device(comm\_thread)* and the runtime will handle them. There will be a special thread not counted among the general pool of threads that only executes Communication Tasks, and Communication Tasks can only be executed by the Communication Thread.

With this environment Communication Tasks are executed in order, therefore we ensure that there are no introduced deadlocks. And we do not need a thread safe MPI because only one thread will be executing Communication Tasks.

## 3 Dynamic Load Balancing (DLB) Library

The Dynamic Load Balancing (DLB) is a shared library that helps load balance applications with two levels of parallelism. The current version provides support for:

- MPI+OpenMP
- MPI+SMPSs

The aim of DLB is to balance the MPI level using the malleability of the inner parallel level. One of its main properties is that the load balancing will be done

at runtime without analyzing nor modifying the application previously. The algorithm that has showed better performance results is LeWI (Lend When Idle) [2]. And this is the algorithm that we are going to explain in the following section and use for the performance evaluation.

### 3.1 LeWI Algorithm

The philosophy of LeWI is based on the fact that when an MPI process is waiting in an MPI blocking call none of its threads is doing useful work. Therefore, we have one or several CPUs that are not being used. LeWI aims to use these CPUs to speedup other MPI processes running in the same node. The usual behavior of an MPI application is that if a process is blocked in an MPI call it is waiting for one or several other processes to finish. Speeding up processes that are more loaded helps to load balance the application and speedup the whole application.

In Figure 2 we can see the behavior of the LeWI algorithm when balancing an unbalanced application. On the left, Figure 2.a shows an unbalanced hybrid application with 2 MPI processes and 2 threads per process. In this example MPI process 1 is more loaded than MPI process 0 and this makes that MPI process 0 must wait in an *MPI\_Send* for some time.

In the center, Figure 2.b shows the behavior of the same application when executed with the LeWI algorithm. When an MPI process reaches a blocking MPI call it will lend its CPUs to the other MPI processes running in the same node. With the lent CPUs the more loaded MPI processes will be able to finish its computation faster and the MPI process 0 will be less time waiting in the MPI call. The use of the computational resources will be better and the application will perform better.

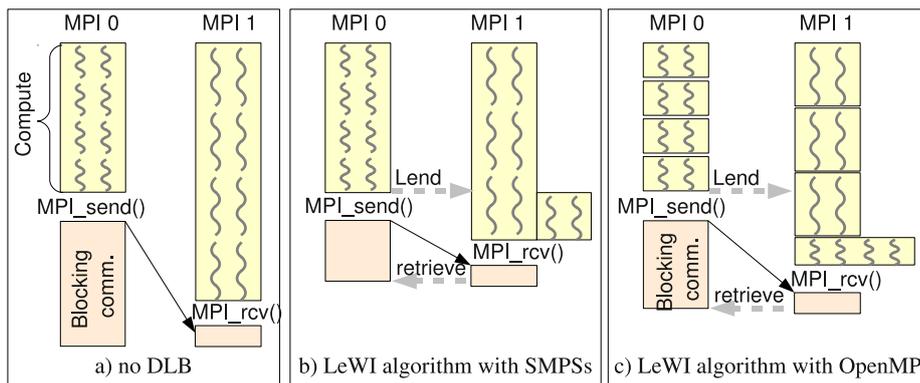


Fig. 2. LeWI behavior

When an MPI process that has lent its CPUs reaches the end of the blocking call it will retrieve the CPUs that it had before lending them. And it will be able to continue its computation with its threads.

### 3.2 SMPSs Potential for Load Balancing

The limitation that we detected in this algorithm is the fact that OpenMP can only 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 2.c; when MPI 0 lends its CPUs to MPI 1, MPI process 1 is already executing its third parallel loop. Therefore, MPI 1 is not able to use the lent CPUs until the fourth loop starts.

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.

In Figure 2 we show the difference between using OpenMP or SMPSs when running with LeWI algorithm and how it can impact the performance. Figure 2.c shows the limitation in OpenMP that can not start to use the threads until it reaches a new parallel region. In Figure 2.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 using SMPSs instead of OpenMP.

## 4 Performance Evaluation

### 4.1 Environment

The experiments have been executed on Marenostrom. Marenostrom has 10240 PowerPC 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 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.

### 4.2 Methodology

In this section we will use the speedup to compare the performance of each experiment. The speedup has been computed as the serial time divided by the parallel execution time.

The serial time used to compute the speedup 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 SMPSSs. By using this baseline we exclude from the computation of the speedup the overhead introduced by the MPI runtime.

In the following charts we will be comparing several configurations for each application. Their meaning is as follows:

**OMP - ORIG:** Execution of the original MPI+OpenMP application without any load balancing.

**OMP - LeWI:** Execution of the MPI+OpenMP application with DLB and LeWI algorithm.

**SMPSSs - ORIG:** Execution of the original MPI+SMPSSs application without any load balancing.

**SMPSSs - LeWI:** Execution of the MPI+SMPSSs application with DLB and LeWI algorithm.

All the executions have been run in Marenostrom (4 cores per Node). We will present different configurations, running with 2 MPI processes per node or 4 MPI processes per node. Although it is not usual to run more than one MPI process in the same node sometimes it is done because the application is MPI only or for performance reasons. In our case it is necessary because we balance between MPI processes running in the same node. Furthermore, we showed in a previous work that running several MPI processes in the same node could improve the performance of applications when combined with the DLB library.

In the following sections we will present the results obtained with different applications.

### 4.3 PILS

PILS is a synthetic benchmark that we have developed to help us evaluate Load Balancing techniques. What we aim to reproduce with PILS is not a whole application but the parallel region between MPI calls of an application.

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 SMPSSs version this function will be taskified, meaning that each call to this function is a task. 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.

In Figure 3 we can see a schematic representation of PILS. The amount of work load is given at the beginning of the execution to each MPI process. This work load is computed in the parallel regions. The number of parallel regions

depends on the parallel grain parameter. A parallel region in the OpenMP version corresponds to a parallel loop. In the SMPSSs version a parallel region is a loop that creates several tasks and finishes with a SMPSSs barrier. At the end of the iteration there is an MPI barrier to synchronize all the processes.

The PILS benchmark has several configurable parameters that allows us to reproduce the behavior of different types of applications. The different parameters are the following:

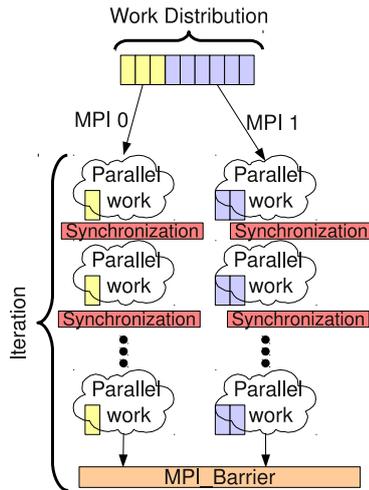


Fig. 3. PILS benchmark

Parallel Grain	Parallel Regions
1	1
0,5	2
0,1	10

Fig. 4. Parallelism Grain and Parallel Regions

**Programming Model:** We can compile three different versions of the benchmark:

- MPI
- MPI+OpenMP
- MPI+SMPSSs

**Work Distribution:** We can introduce the load balance of the application as the different loads for each MPI process. For all the executions the sum of the loads will be the same (giving always the same amount of work to the application).

**Parallelism Grain:** Represents the amount of computation that is parallel for the iteration. The value can go from 1 (everything is parallel) to 0 (nothing is parallel). You can see the relationship between Parallelism Grain and the number of parallel regions in Figure 4.

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

In the following experiments we have executed PILS with the parameter *iterations* always 1. We want to compare the performance of the different programming models with different works loads and parallelism grain.

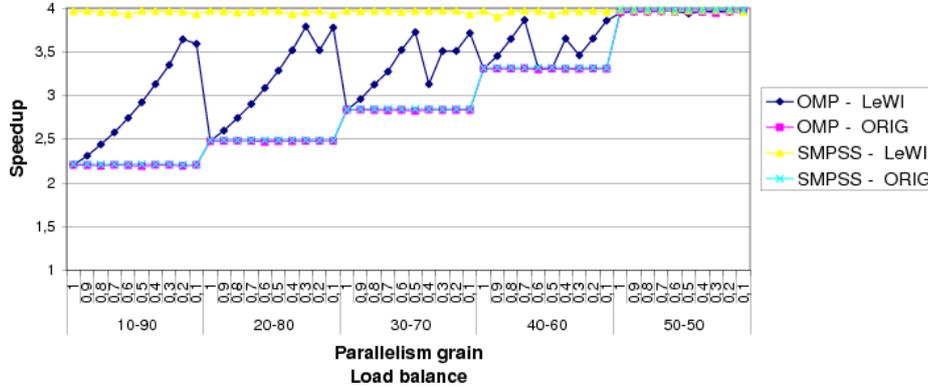


Fig. 5. PILS 2 MPIs per Node in Marenostrium

In Figure 5 we can see the speedup obtained for PILS with the 4 different versions when running with 2 MPI processes in the same node. We executed with five configurations for the work load that go from a very unbalanced application (10-90) to a well balanced application (50-50) and for each of the imbalance configurations we used 10 different parallelism grain from 1 to 0,1.

We can see how the executions without LeWI (SMPSSs-ORIG and OMP-ORIG) have the same performance, this means that the baseline for both are the same. But we can see the difference when running with LeWI (SMPSSs-LeWI and OMP-LeWI) while SMPSSs obtains for all the executions almost the ideal speedup of 4, the performance of the version with LeWI and OpenMP depends on the Parallelism Grain (or equivalent, the number of parallel regions between MPI calls).

Figure 6 shows the performance of PILS when running with 4 MPI processes in the same node. In this case we can see 7 different configurations of work loads with different levels of imbalance between the MPI processes. Again the original executions without LeWI of the two models have the same performance but when running with LeWI we can see how SMPSSs performs much better. While SMPSSs with LeWI can obtain an almost ideal speedup for all the configurations, OpenMP depends on the Parallelism Grain and almost never can reach the same performance as SMPSSs.

#### 4.4 LUB

LUB is a kernel performing a LU matrix factorization. The structure is a two dimensional matrix organized by blocks. The data is distributed by blocks of

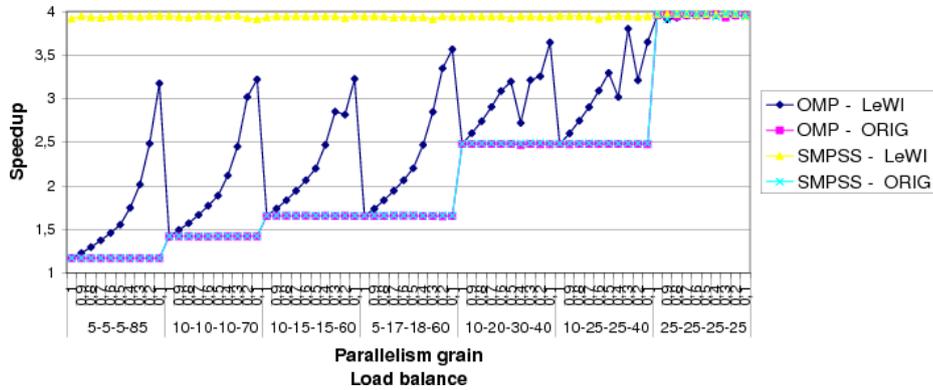


Fig. 6. PILS 4 MPIs per Node in Marenstrum

rows among the different MPI processes. In figure 7 we can see an schematic representation of the LUB kernel. 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 depend on the *fwd* block in the same column and the *bdiv* block in its row.

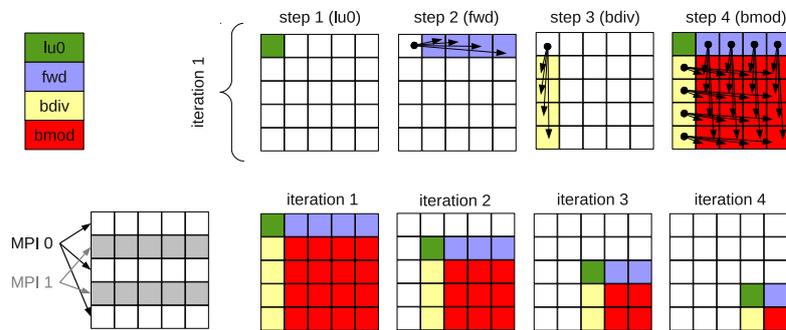


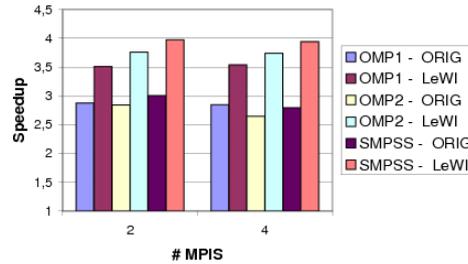
Fig. 7. LUB behavior

We have two OpenMP parallelizations for this application the first one (labeled as *OMP1* in the charts) is the “natural” one that parallelizes the outer possible level. But this parallelization is not the optimal for LeWI because it presents few loops between MPI calls. So it gives low malleability to change the number of threads. The second OpenMP parallelization (labeled as *OMP2*) parallelizes inner loops of the application giving more malleability to LeWI to

change the number of threads. We can say that *OMP2* is a parallelization tuned to obtain the best of LeWI but that introduces overhead.

The SMPSSs parallelization considers each algorithm applied to a block as a task which is the natural way of parallelizing this application with a task oriented programming model.

In the experiments we have worked with a matrix of 5000 x 5000 elements with block size of 200 x 200 elements.



**Fig. 8.** LUB in 1 Node in Marenostrom

In Figure 8 we can see the performance obtained when running LUB in a single node of Marenostrom (4 cores). We have executed with 2 or 4 MPI processes in the same node. We can see how the 3 original versions (*OMP1*, *OMP2* and *SMPSSs*) have a similar performance, and in general all of them have a worst speedup when running with 4 MPI processes per node. The reason for this is that the imbalance of the application grows linearly with the number of MPI processes. But in all the cases the best performance close to the ideal one is obtained by the SMPSSs version when running with LeWI. Not even the modified OpenMP version (*OMP2*) implemented to give a chance to the OpenMP version compared to the SMPSSs version.

It is important to notice that with the OpenMP version *OMP1* LeWI is able to increase the speedup of the application from 2.8 to 3.5. This means executing the application without modifying it nor analyzing it previously. The tuned version *OMP2* was implemented to give a chance to the OpenMP version compared to the SMPSSs version.

The performance of LUB executed on 2 and 4 nodes of Marenostrom can be seen in Figure 9. We have also executed with 2 and 4 MPI processes per node each version. In this case we can see how the performance obtained is not so close to the ideal one. The two reasons for this performance drop are on one hand the more MPI processes the more imbalance that this application presents. And on the other LeWI can only balance between MPI processes running in the same node. This means that when the imbalance is present between processes running in different nodes LeWI can not improve its performance. But still the version SMPSSs with LeWI is the one that obtains the best speedup, even better than the OpenMP version modified to run with LeWI (*OMP2*).

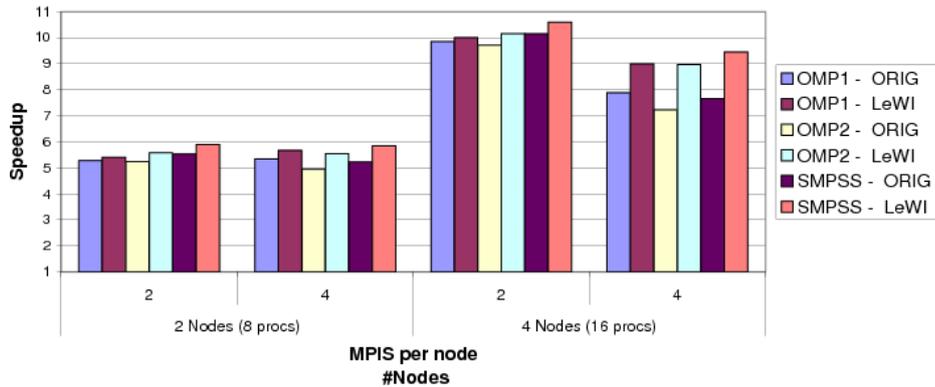


Fig. 9. LUB in 2 and 4 Nodes in Marenostrum

#### 4.5 BT-MZ (Nas Benchmarks)

The BT application is one of the benchmarks in the NAS Multizone suite [5]. 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 SMPSSs (labeled *SMPSSs*), and we have also modified the original MPI+OpenMP version because in some parts of the code it presented several *Loops* inside a single *Parallel*. This prevents LeWI to change the number of threads often enough. We have labeled this tuned version for LeWI *OMP2*.

The NAS benchmarks can run different classes that correspond to the size of the problem that it is solving. In our experiments we have executed classes A, B and C (with the following relationship of size:  $A < B < C$ )

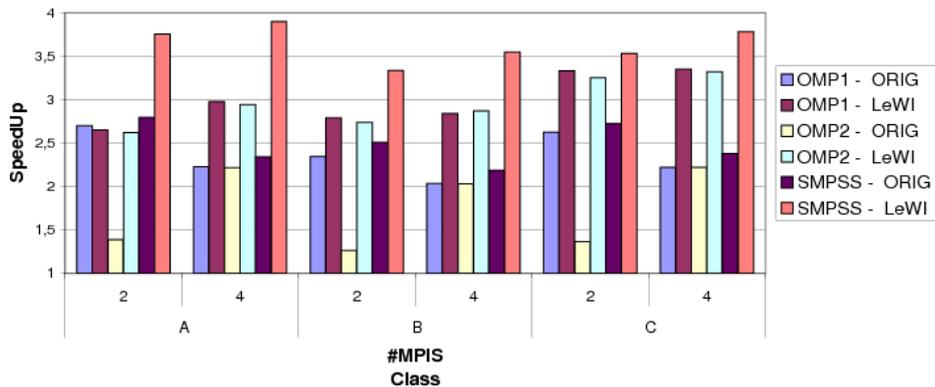


Fig. 10. BT-MZ in 1 Node in Marenostrum

In Figure 10 we can see the performance obtained with BT-MZ when running in a single node in Marenostrom 3 different classes, A, B and C. In this case we see a big difference between the performance of the original OpenMP version (*OMP1*) and the tuned version for LeWI (*OMP2*). The reason of this difference is because the granularity of the loops is too small for *OMP2* and some data locality is lost. When running with 4 MPI processes in the same node we do not see this effect because the OpenMP level is only used to load balance, therefore, the performance is the same for both versions (*OMP1* and *OMP2*).

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 SMPSSs version with LeWI being close to the ideal performance in some cases.

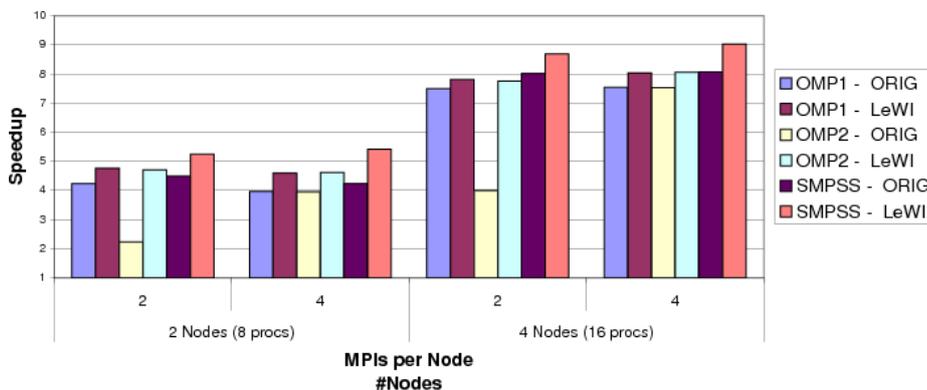


Fig. 11. BT-MZ class C in 2 and 4 Nodes in Marenostrom

In Figure 11 we can see the speedup obtained by the different executions of BT-MZ class C in 2 and 4 nodes. Again the versions with LeWI improve the performance of the original executions. We can notice that the improvement in the performance when running in several nodes is less than when running in a single node. The reason is that LeWI can only balance MPI processes running in the same node.

## 5 Conclusions

In this paper we have explained the basics of SMPSuperscalar. SMPSuperscalar is a programming model for shared memory systems that hybridizes nicely with MPI. We have explained why MPI+SMPSSs is a powerful hybrid programming model and shown it in the performance evaluation section.

We have also presented a load balancing algorithm for hybrid applications called LeWI. The LeWI algorithm will improve the load balance and performance of hybrid applications. To achieve this it will redistribute the computational

resources assigned to the application between the different MPI processes of the application.

The current version of LeWI supports MPI+OpenMP and MPI+SMPSs applications. We detected and explained a limitation of the algorithm when load balancing MPI+OpenMP applications related to the malleability of OpenMP. And we have seen that this limitation is avoided when using SMPSs instead of OpenMP.

In the performance evaluation we have executed three different applications with different imbalance patterns. For all the applications we have a MPI+OpenMP version and a MPI+SMPSs version. In the performance results obtained we have seen that SMPSs can obtain the same performance as OpenMP when combined with MPI in all the applications we have tested.

We have also shown how the LeWI algorithm improves the performance of all the applications executed and does not penalize the performance of balanced application. And we can use it without analyzing nor modifying the original application.

But the most interesting remark from the performance evaluation is that for all the applications the best performance is obtained with the MPI+SMPSs version of the application running with LeWI. The malleability of SMPSs allows LeWI to improve the performance of unbalanced applications and obtains a higher speedup than the same application with MPI+OpenMP and LeWI.

## Acknowledgments

We thankfully acknowledge the support of the European Commission through the TEXT project (FP7-261580), the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

## References

1. D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.
2. M. Garcia, J. Corbalan, and J. Labarta. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In *International Conference on Parallel Processing (ICPP09)*, 2009.
3. J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, 2008.
4. SMPSuperscalar official site and documentation: <http://www.bsc.es/smppsuperscalar>.
5. R. Van der Wijngaart and H Jin. Nas parallel benchmarks, multi-zone versions. Technical report, NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.