LeWI: A Runtime Balancing Algorithm for Nested Parallelism

Marta Garcia, Julita Corbalan and Jesus Labarta
Barcelona Supercomputing Center BSC-CNS and
Universitat Politecnica de Catalunya UPC
Barcelona, Spain
marta.garcia, julita.corbalan, jesus.labarta@bsc.es

***Abstract*—We present *LeWI*: a novel load balancing algorithm, that can balance applications with very different patterns of imbalance. Our algorithm can balance fine grain imbalances, non iterative applications and applications with irregular imbalance. To achieve this *LeWI* reassigns the computational resources of blocked processes to other processes more loaded.**

We have implemented *LeWI* within DLB a Dynamic Load Balancing Library developed by us. DLB helps parallel programming models to make the most of the computational power available with the minimum effort. It solves the imbalance among processes in applications with two levels of parallelism using the malleability of the inner level.

The performance evaluation shows that *LeWI*, the novel balancing algorithm we are presenting in this paper, together with DLB is able to improve the performance of a different range of unbalanced applications and when applied to well balanced applications it does not introduce significant overhead. Therefore we present a mechanism that can be used with any hybrid application without needing a programmer to analyze the application nor modify it.

***Keywords*-Dynamic, Load balancing, hybrid applications, MPI, OpenMP, Unbalance, Parallelism.**

I. Introduction

One typical source of inefficiency in parallel applications is the imbalance between processes. The usual way of attacking this problem is analyzing the application to find the source of imbalance then modify the source code to introduce some balancing code and test it. If the results are not satisfactory the process starts again and iterates until the performance achieved is satisfactory. This process must be done by an expert in the parallel programming model used by the application. The analyst may or may not know how the application works. At the end it is a time consuming and error prone process.

The solution we propose in this paper is a library (DLB) that can be loaded dynamically when running an application to improve its load balance at runtime. DLB can balance applications with two levels of parallelism where the inner level is malleable. We use the malleability of the inner level to balance the outer level. The application does not need to be modified nor analyzed.

The current version of the library has modules to balance hybrid MPI+OpenMP applications but it is designed to be easily extended for other programming models as long as the inner level is malleable. From now on we will talk about the MPI level and the OpenMP level to refer to the outer and inner level of parallelism of the application respectively.

Within the DLB library we have implemented and evaluated two different algorithms of load balancing: *LeWI* (*Lend When Idle*), the novel algorithm we are presenting in this paper, and *DWB* (*Dynamic Weight Balancing*), an algorithm based in a previous work by Duran et al. [1].

The *DWB* is based on the load balance model presented by Corbalan et al. [2] and the algorithm presented by Duran et al. [1]. This model exploits the iterative behavior of some applications and re-distributes the OpenMP threads based on the computational load of each MPI process per iteration.

We detected that this algorithm had several limitations and we tried to overcome them with *LeWI*. This novel algorithm distributes resources equally among MPI processes in a node while they are doing computation, and re-assigns resources of MPI processes while they are blocked in communication calls. Both algorithms will be further explained in the following sections.

The performance evaluation showed that the *LeWI* algorithm can improve the performance of different applications. Moreover we showed that our algorithm can balance irregular or non iterative applications where *DWB* algorithm fails.

II. Related work and Motivation

The load balancing problem is as old as parallel programming. Solutions for load balancing can be divided in two main groups: The ones that are applied before running the application and the ones that are applied at runtime. In the first group the most used approach is domain partitioning [3] [4]. The most representative and widely used is Metis [5]. These solutions can only be applied to problems with a certain data distribution (graph, mesh...) and must be calculated before each execution for each different set of input data.

In our work we are going to focus in solutions of the second group that are applied at runtime and do not need to modify the input datasets.

A typical approach to solve the issue of load balancing in parallel applications at runtime is to combine two levels of parallelism. Smith et al. [6] 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. [7] 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. [8] 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.

Although in general using two levels of parallelism can help balance applications most times is not enough. To solve this there are two main approaches in the literature. One option is to redistribute the data of the application so that it is better balanced. The other approach is to redistribute the computational power so that the imbalance in the data distribution is solved.

There are several studies of the first approach like Charm++ [9], an object-oriented parallel programming language that employs object migration to achieve load balance. Adaptive MPI (AMPI) [10] is an implementation of MPI that uses the load balancing capabilities of Charm++. Balasubramaniam et al. [11] 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 more 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. We propose a flexible solution transparent to the applications and the programmer, where we do not need to modify the application.

There are also relevant works done to solve the imbalance redistributing the computational power. Zhang et al. [12] 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. [13] 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. [14] 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.

Closer to our approach are the works done by Spiegel et al. [15] and Duran et al. [1]. They both aim to balance applications with two levels of parallelism by redistributing the computational power of the inner level. And they both do it at runtime without modifying the application. The first one balances MPI+OpenMP hybrid applications. The second one balances OpenMP applications with nested parallelism.

Although their algorithms are different, they converge to the same solution. They both use previous iterations to detect the load of each process. The algorithm of Spiegel et al. needs several iterations to converge to a stable and optimal redistribution and their approach is focused to SMPs with a large number of CPUs.

We selected the algorithm presented by Duran et al. (Dynamic Weight Balancing Algorithm: DWB) to port it to our library because to our knowledge it presented the best results of dynamic load balancing at runtime and without the need to modify the application. We analyzed its performance and we found that it presents several limitations that does not make it suitable for all kind of hybrid applications. To understand better the contribution of our *LeWI* algorithm we are going to explain further the *DWB* algorithm in the next section.

A. DWB: Dynamic Weight Balancing

The *DWB* algorithm is targeted to iterative applications. The main loop of the application will be detected with a Dynamic Periodicity Detector (DPD) [16] so the *DWB* algorithm will know the time each MPI process spent computing and waiting in an iteration of the application. With this information the algorithm will assign to each MPI process a number of OpenMP threads proportional to its computational load (the algorithm will always assign at least 1 thread per MPI process). The algorithm will keep getting metrics of the application and change its decision at any moment if it thinks it will improve the performance of the application. For example because of a change of phase in the application that needs another distribution of threads.

The main problem of this algorithm is the granularity to solve the imbalance because it depends on the number of CPUs available per node.

We have done a synthetic study of the potential of the *DWB* algorithm when applied to applications with different levels of imbalance (for this study we supposed an ideal parallelization of the application, therefore the inefficiency can only come from the imbalance between processes). We wanted to see the impact of the number of CPUs available per node in the efficiency[1] of the algorithm when running with two MPIs processes per node. In Figure 1 we show the results obtained for the different number of processors per node. In the x axis is represented the imbalance of an application as a percentage between two processes (50% means a well balanced application, right most side, and 1% represents a very unbalanced application, left most side). In the y axis we show the maximum efficiency that can be

[1]Efficiency is a metric to show the percentage of time that the assigned CPUs are doing useful computation respect all the time the application is running. A perfect balanced application will have a 100% of Efficiency.

$$\text{Efficiency} = \frac{\sum_{num\_CPUs}^{x=1}(cpu\_time_x)}{time_app * num\_CPUs} * 100$$

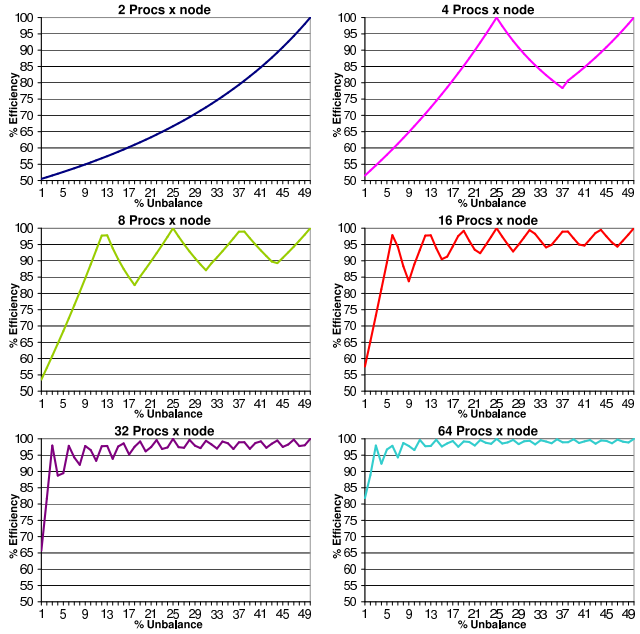

Figure 1. Imbalance/Efficiency ratio depending on number of CPUs when using DWB with two MPIs in one node

obtained when using the *DWB* algorithm (100% of efficiency means that the applications is using all the computational time, 50% means that half of the time that the application is running the CPUs are idle). The different lines represent the different number of CPUs per node. We can see that with few CPUs per node (i.e. 2, 4) the efficiency that can be obtained depends on the imbalance of the application. To get a good performance in almost all the cases we need a high number of cores per node (i.e. 16, 32,...).

We have designed and developed the *LeWI* algorithm to solve this granularity problem when balancing applications at runtime in nodes with a moderated number of CPUs. Our proposal is more flexible than the current state-of-the-art in several ways:

- It does not need the application to be iterative.
- It can balance efficiently applications with different levels of imbalance independently of the number of CPUs available
- It can balance very irregular applications where the imbalance changes each iteration. Approaches like *DWB* can not balance this kind of applications because they use previous iterations to decide the new distribution.

III. The DLB (Dynamic Load Balancing) Library

In this section we describe the main characteristics of the DLB library and the main concepts of its architecture to understand how it works.

The DLB library aims to balance applications with two levels of parallelism, currently we have modules implemented to balance hybrid MPI+OpenMP applications where the MPI is the outer level and OpenMP the inner one. The OpenMP threads are not 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 to balance it but we just need to dynamically load the library when running the application.

We have developed the DLB in a modular way. This allows to easily:

- Enable the DLB library for different programming models, at both inner and outer level.
- Add new balancing algorithms.

Independent of the load balancing algorithm used the DLB library will balance the MPI processes redistributing the computational power of the OpenMP threads running underneath. This means that it will try to balance the load of the processes that are running in the same node (shared memory). But even balancing within the nodes we are able to improve the global performance of applications running in more than one node as we will show in the performance evaluation section. This version of the library will not be able to balance MPI processes that are running in different nodes. So we will need to run more than one MPI process per node to use our balancing library.

Within DLB we have implemented two different load balancing algorithms. The *DWB* algorithm based on a previous work [1] has been already explained in the previous section. The second algorithm implemented (called *LeWI*) is a novel idea that came from the study of the limitations of the first one and is further explained in the following section.

A. LeWI: Lend CPUs When Idle

The new balancing algorithm we are presenting is based in the following observation: the imbalance between processes implies that one (or more) process is blocked waiting for others. And while a process is blocked waiting the CPUs it has assigned to run are idle.

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

The main idea of the *LeWI* algorithm is to lend the OpenMP threads (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 has lent the CPUs gets out of the blocking call it will recover its CPUs and the process that was using them will be notified to stop using the lent threads.

In Figure 2 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 OpenMP threads. In Subfigure 2.a we can see the behavior of an unbalanced application. On the other side subfigure 2.b shows the

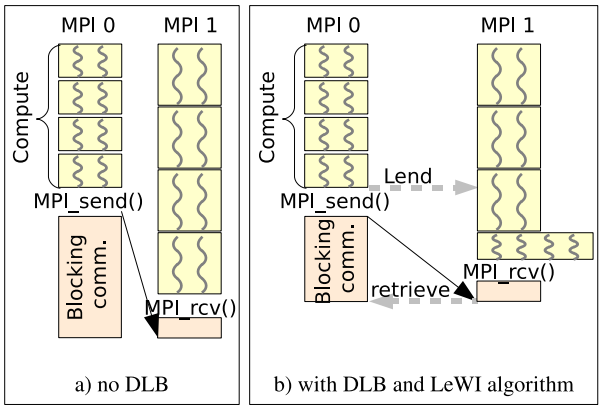

Figure 2. Example of LeWI algorithm

execution of the same application with the balancing library and the *Lend* algorithm. We can see that when the MPI process 0 gets into the blocking call it will lend its two OpenMP threads to the MPI process 1. The second MPI process 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

Both actions (lend and retrieve) can take some time until the changes are reflected in the execution due to restriction of the programming model used. For instance, in OpenMP, a change in the number of OpenMP threads is only applied when entering a new parallel region. This latency means that while retrieving CPUs the application can run with more threads than CPUs available. We have seen that this does not have a significant impact in the performance.

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. Each time a idle CPU is waiting to be assigned the algorithm calculates a load factor for each MPI process. This load factor is obtained by dividing the computational load of the MPI process by the number of threads it has assigned. The idle CPU will be assigned to the MPI process with the higher load factor. The CPUs will be assigned one by one. That is, if the process that is blocked and lending the CPUs had two threads, we will calculate to which process must be lent the first thread, once this is done we will calculate to which process must be lent the second thread, but taking into account the thread that was previously lent.

The way to obtain the computational load of the MPI processes depends on the type of application. If it is an iterative application (we use DPD to detect it) the computational load of each MPI process will correspond to their load during the last iteration. If the application is not iterative the computational load will be the time accumulated since the beginning of the application.

IV. Performance Evaluation

In this section we are going to present the results obtained from the performance evaluation of the algorithms and the library.

A. Environment

As our target was a clustered architecture, all our experiments have been run in Marenostrum. Marenostrum 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.

B. Methodology

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

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

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

- The original application.
- The application with DLB and *DWB* algorithm.
- The application with DLB and *LeWI* algorithm.
- The original application with 4 threads OpenMP each MPI process. In this case we are overloading the node and leaving the responsibility of balancing to the operating system scheduler. With this version we need to use a guided schedule for OpenMP (in all the other cases we are using a static schedule).

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

We are using the speedup to compare the performance of each experiment. The speedup has been calculated as $\frac{serial\_time}{execution\_time}$ where the serial_time is the time that took the

application to finish with one MPI process and 1 OpenMP thread, we have used this time because in most of the cases we do not have a serial version of the application to run.

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

C. Applications

In this section we are going to explain briefly the applications we have used for the performance evaluation.

In Table I we show the Efficiency and Load Balance for each application when executed with different number of MPI processes. Efficiency is a metric than shows the effective usage of the computational power (how to compute it is explained in Section II-A). Load Balance is a metric of global imbalance between processes that is calculated as follows: $LB = \frac{\sum_{num\_CPUs}^{x=1}(cpu\_time_x)}{Max_{num\_CPUs}^{x=1}(cpu\_time_x)*num\_CPUs} * 100$

Application	Num. MPIs	Efficiency	Load Balance
BT-MZ class A	2	0,664	0,666
	4	0,526	0,528
	8	0,344	0,351
SP-MZ class A	2	0,990	0,999
	4	0,978	0,998
	8	0,814	0,984
LUB	2	0,810	0,972
	4	0,568	0,914
	8	0,285	0,835
FLOWer	4	0,914	0,971
	6	0,781	0,893
	8	0,733	0,818
	12	0,499	0,557
	16	0,387	0,429
	24	0,174	0,274

Table I
APPLICATIONS' EFFICIENCY AND LOAD BALANCE

- **NAS Multizone benchmarks** We can find 3 applications within the NAS-MZ package [17]: BT, LU and SP. All the applications are iterative but the BT zone partition is done asymmetrically. This results in an unbalanced execution. SP and LU on the other hand do the zone partition in a symmetric way so their executions are expected to be balanced. We are only going to show the results obtained with BT-MZ and SP-MZ because are representative of a balanced and unbalanced application as can be seen in Table I.
- **LUB** computes a LU decomposition on a two dimensional matrix structure. The application we are using is a blocked version, so the different blocks are the units of work used to distribute the computation. This application is unbalanced but in an irregular way because the most loaded MPI process is not the same during all the execution. Therefore we can see in Table I that its Efficiency is low while its Load Balance is good this is because it does not present a global unbalance.
- **FLOWer** Is a flow solver developed at the German Aerospace Center (DLR) [18]. Its part of a project to simulate PHOENIX which is a small scale prototype of the Space Hopper. We did not know the type of imbalance that presented this application before evaluating it. This is an example of improving the performance of an application without analyzing it first. In Table I we can see that the more MPI processes it uses the worst Efficiency and Load Balance it presents.

D. Running in a single node

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

In Figure 3 we can see the speedup obtained with the BT-MZ application, this application is very unbalanced but at the same time presents a very regular imbalance that is maintained during all it execution. We can see that increasing the number of MPIs with the original version produces a decrease in the performance. On the other hand when using DLB with *LeWI* algorithm the performance improves as we increase the number of MPIs per node. The best performance is obtained with 4 MPIs per node with the *LeWI* algorithm. It improves by 13% the speedup with respect to the original application with the traditional configuration of 1 MPI per node. Compared to the original application when running with 4 MPIs per node *LeWI* obtains a 64% of improvement.

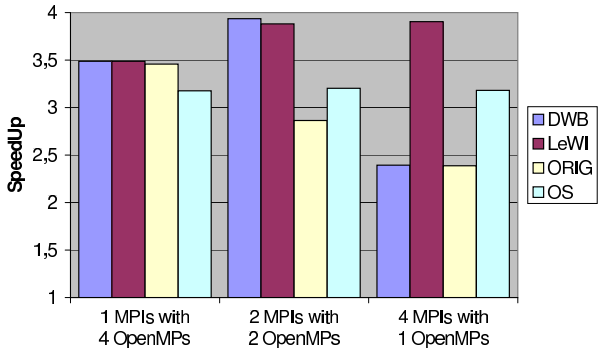

Figure 3. BT-MZ Class A in one node (4 cores)

The BT-MZ application with two MPIs per node is the only case where the performance of the *DWB* is close to the one obtained by *LeWI*. The reason is that the imbalance in

this case corresponds to a perfect partition of the processors (1-3)

The SP-MZ performance is shown in Figure 4. As we said this is a balanced application and we can see that there is no significant overhead introduced by DLB. The performance loss experimented by the OS serie is due to the use of the guided schedule for OpenMP loops.

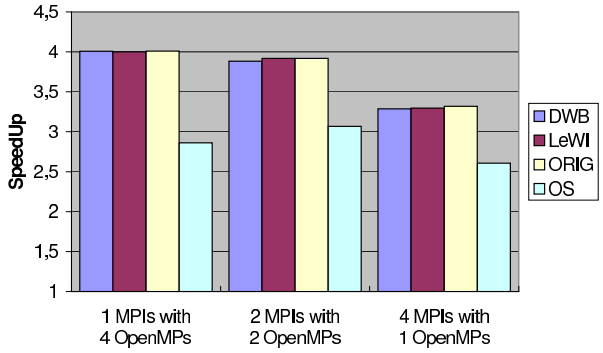

Figure 4. SP-MZ Class A in one node (4 cores)

The speedup of the LUB application is shown in Figure 5. The imbalance of this application is very irregular because the most loaded MPI process changes each iteration. In this case we can see that the *LeWI* algorithm improves the performance of the application. And again the best performance is obtained by *LeWI* running with 4 MPIs per node. This configuration improves by 27% the speedup of the original application running with 1 MPI per node. The *DWB* algorithm, on the other hand, is not able to improve the performance of the original application because one of the limitations of this algorithm is that it needs the same imbalance during successive iterations.

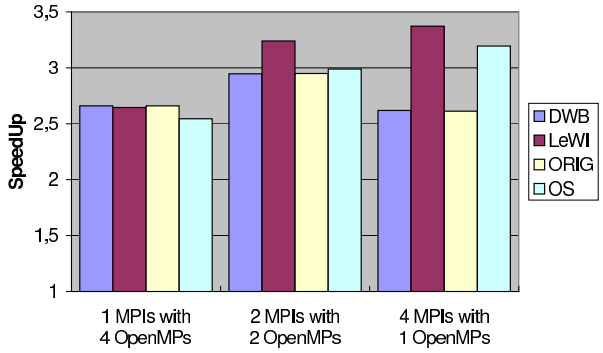

Figure 5. LUB in one node (4 cores)

We can see the performance results for the FLOWer application in Figure 6. Running with less than 8 MPI processes this application does not present a significant imbalance. When running with 4 MPIs, *LeWI* improves the performance by 9% with respect to the original application with 1 MPI per node and a 4,5% respect the original application running with 4 MPIs per node.

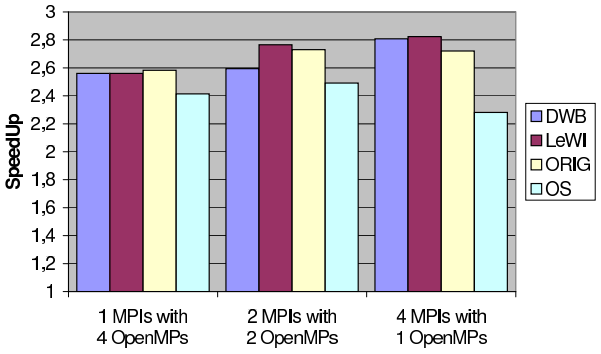

Figure 6. FLOWer in one node (4 cores)

E. Running in several nodes

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

In Figure 8 and 7 we can see the speedup obtained with the BT-MZ and the SP-MZ applications when running in 2 nodes (8 cores). The results of *LeWI* are similar to the ones obtained when running in one node but less speedup is achieved because we can only balance the processes running in the same node. Even so we improve the global performance of the BT-MZ original application while no overhead is introduced in the execution of the SP-MZ application. As we can not migrate processes across nodes in the current implementation, in the case of BT-MZ, to show the potential of DLB, we have mapped high loaded processes and light loaded ones in the same node.

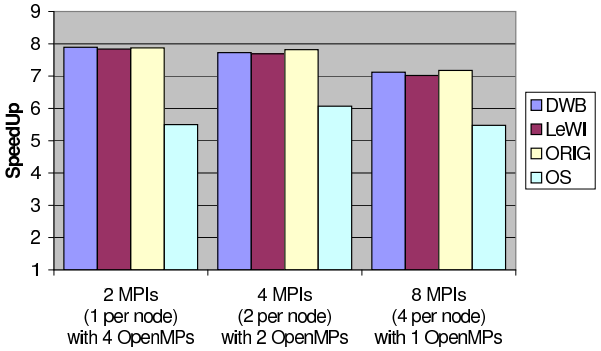

Figure 7. SP-MZ Class A in two nodes (8 cores)

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

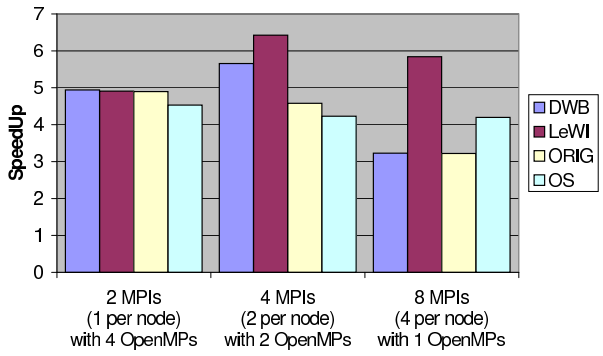

Figure 8. BT-MZ Class A in two nodes (8 cores)

We have executed the FLOWer application in 4 and 6 nodes and we can see the results obtained in Figures 9 and 10. As we said the imbalance of this application increases as the number of MPI processes increase. Because of this the performance drops as we increase the number of MPIs per node. We can see that the *LeWI* algorithm can still improve the performance of the application when running with 2 MPIs per node or 4 MPIs per node. The best execution of *LeWI* is a 2,5% better than the best execution of the original application.

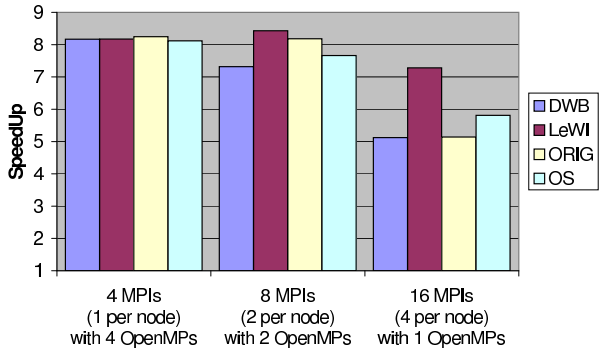

Figure 9. FLOWer in four nodes (16 cores)

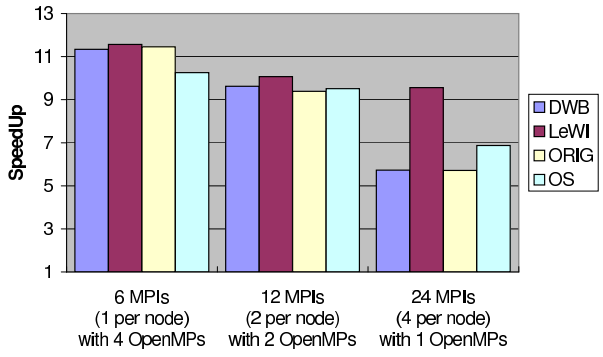

Figure 10. FLOWer in six nodes (24 cores)

It is interesting to see how in Figure 9 when running in four nodes the *LeWI* algorithm obtains a better performance when running with 2 MPIs per node than when running the original application with 1 MPI per node. This means that it is obtaining a better performance than a less unbalanced execution.

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

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

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

V. Conclusions and Future Work

In this paper we have presented a load balancing algorithm, *LeWI*, that has been implemented within a dynamic library, Dynamic Load Balancing (DLB).

The DLB library allows us to balance applications with two levels of parallelism without modifying the application or studying the imbalance it presents. The current version of the library can balance hybrid MPI+OpenMP applications.

We have shown, in the performance evaluation, that the *LeWI* algorithm can balance applications with different kinds of imbalance (even irregular or non iterative applications). We obtain improvements in the performance between 9% and 27% when running with *LeWI* and 4 MPIs per node respect to executions with a single MPI per node. When applied to balanced applications the evaluation shows that it does not introduce significant overhead.

And what is more interesting, we have been able to improve the performance of applications running in more than one node by load balancing the processes inside each node with our approach. In the evaluation we showed that *LeWI* can increase the global performance of an application up to a 31%.

The solution we are proposing can be used with any application without the need to analyze its behavior. Because we have shown that it does not introduce overhead when can not improve the performance. We think it could even be integrated with the runtime of the parallel programming model of choice.

As future work, we want to port the library to other parallel programming models. We would like to build a module for Cell Superscalar (CSs) [19]. Because Cell is a promising new architecture and the task approach used by CSs could work very well with our balancing policies.

We will implement a hybrid balancing algorithm that combines the two algorithms presented in this paper (DWB and LeWI) with this new algorithm we aim to obtain the benefits of both: the flexibility of LeWI and the stability of DWB.

If we find a suitable testbed we would like to test the performance of the algorithm in nodes with more than 4 CPUs. In this case we could experiment with the factor of running more MPIs in the same node or having more OpenMP threads to distribute between the processes. In the first case the decision to which process you lend the CPUs becomes more important. In the second case the delay until the decisions are applied can become crucial.

We would like to be able to detect the cases where the performance of the application can not be further improved even with the balancing algorithm. And, in these cases the runtime could predict if running with less threads does not affect the performance. Then a CPU could be powered-off and save energy without penalizing the execution time.

However our final objective would be to balance applications across nodes to maximize the usage of resources. To achieve this goal we plan to migrate automatically processes from heavy loaded nodes to lighter loaded ones

ACKNOWLEDGMENT

We would like to thank RWTH Aachen University, and in particular, Christian Terboven for the FLOWer application.

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050) and the MareIncognito project under the BSC-IBM collaboration agreement.

REFERENCES

[1] A. Duran, M. Gonzàlez, and 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.

[2] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of mpi+openmp applications," in *Proceedings of the International Conference on Parallel Processing (ICPP2004)*, 2004.

[3] C. W. A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, "Dynamic multi-partitioning for parallel finite element applications," in *Parallel Computing: Fundamentals and Applications (ParCo)*, 2000, pp. 259–266.

[4] K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper)," *Lecture Notes in Computer Science*, vol. 1900, 2001.

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

[6] L. Smith and M. Bull, "Development of mixed mode mpi / openmp applications," *Scientific Programming*, vol. 9, no. 2-3, 2001.

[7] 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.

[8] F. Cappello and 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.

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

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

[11] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. P. Pabico, and 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.

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

[13] O. Sievert and H. Casanova, "A simple mpi process swapping architecture for iterative applications," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 341–352, 2004.

[14] K. E. Maghraoui, B. Szymanski, and 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.

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

[16] F. Freitag, J. Corbalan, and J. Labarta, "A dynamic periodicity detector: Application to speedup computation," in *15th International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[17] H. Jin and R. F. V. der Wijngaart, "Performance characteristics of the multi-zone nas parallel benchmarks," *J. Parallel Distrib. Comput.*, vol. 66, no. 5, pp. 674–685, 2006.

[18] M. Hesse, B. U. Reinartz, and J. Ballmann, "Inviscid flow computation for the shuttle-like configuration phoenix," *Notes on Numerical Fluid Mechanics and Multidisciplinary Desing*, 2004.

[19] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: making it easier to program the cell broadband engine processor," *IBM Journal of Research and Development*, pp. 593–604, 2007.