Dynamic Load Balancing of MPI+OpenMP applications

Julita Corbalán, Alejandro Duran, Jesús Labarta
CEPBA-IBM Research Institute
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain.
{*juli, aduran, jesus*}*@ac.upc.es*

Abstract

The hybrid programming model MPI+OpenMP are useful to solve the problems of load balancing of parallel applications independently of the architecture. Typical approaches to balance parallel applications using two levels of parallelism or only MPI consist of including complex codes that dynamically detect which data domains are more computational intensive and either manually redistribute the allocated processors or manually redistribute data. This approach has two drawbacks: it is time consuming and it requires an expert in application analysis. In this paper we present an automatic and dynamic approach for load balancing MPI+OpenMP applications. The system will calculate the percentage of load imbalance and will decide a processor distribution for the MPI processes that eliminates the computational load imbalance. Results show that this method can balance effectively applications without analyzing nor modifying them and that in the cases that the application was well balanced does not incur in a great overhead for the dynamic instrumentation and analysis realized.

Keywords: MPI, OpenMP, load balancing, resource management, parallel models, autonomic computing

1 Introduction

A current trend in high performance architecture is clusters of shared memory (SMP) nodes. MPP manufacturers are replacing single processors in their existing systems by powerful SMP nodes (small or medium SMPs are more and more frequent due to their affordable cost). Moreover, large SMPs are limited by the number of CPUS in a single system. Clustering them seems the natural way to reach the same scalability as distributed systems.

MPI and MPI+OpenMP are the two programming models that programmers can use to execute in clusters of SMP. When application is well balanced pure MPI programs usually results in a good application performace. The problem appears when application has internal static or dynamic load unbalance. If the load unbalance is static, there exists approaches that consist of statically analyze the application and perform the data distribution accordingly. If load unbalance is dynamic, complex code lines to analyze and redistribute data must be inserted in the application to solve this problem. In this case, programmers must spent a lot of time analyzing the application code and their behavior at run time. Moreover, it is not only a question of time, analyzing a parallel application is a complicated job.

In this work, we propose to exploit the OpenMP malleability to solve the load unbalance of irregular MPI applications. The goal is do that automatic and dynamically by the system (resource manager and runtime libraries) without *a priori* application analysis.

One of the key points of our proposal is to be conscious that there are several MPI processes, with OpenMP parallelism inside, that are collaborating to execute a single MPI+OpenMP job. Since resources are allocated to jobs, one processor initially allocated to a MPI process that compounds the job can be reallocated to another MPI process of the same job, as long as they are in the same SMP node, *helping* it to finish the work.

We present a Dynamic Processor Balancing (DPB) approach for MPI+OpenMP applications. The main idea is that the system dynamically measures the percentage of computational load imbalance presented by the different MPI processes and, according to that, it redistributes OpenMP processes among them. We have developed a runtime library that dynamically measures the percentage of load imbalance per MPI process and informs to the resource manager who controls the processor allocation in the SMP node. The resource manager redistributes processors trying to balance the computational power. Moreover, since the resource manager has a global view of the system, it could decide to move processors from a job to another if

this would increase the system throughput.

In this paper we present a preliminary study of the potential of this technique. Evaluations have been done assuming that there is only one MPI+OpenMP application simultaneously running and limiting the resource manager to only one SMP node.

In the next section we will introduce the related work. In section 3 the proposed technique is presented and its components explored. In section 4 some results are presented showing the potential of the technique. Finally, section 5 concludes the paper and shows future directions of research.

2 Related work

The proposal presented by Huang and Tafti [1] is the closest one to our work. They advocate for the idea of balancing irregular applications by modifying the computational power rather than using the typical mesh redistribution. In their work, the application detects the overloading of some of its processes and tries to solve the problem by creating new threads at run time. They observe that one of the difficulties of this method is that they do not control the operating system decisions which could opposite their own ones.

Henty [2] compares the performance achieved by the hybrid model with the one achieved by a pure MPI, when executing a discrete element modeling algorithm. In that case they conclude that its hybrid model does not improve the pure MPI. Shan et al. [3] compare the performance between two different kinds of adaptive applications under the three programming models: MPI, OpenMP and Hybrid. They observe similar performance results for the three models but they also note that the convenience of using a particular model should be based on the application characteristics. Capello and Etiemble [4] arrive to the same conclusion. Dent et al. [11] evaluate the hybrid model and they conclude it is an interesting solution to applications such as IFS, where exists load balancing problems and a lot of overhead due the cost of the message passing when using a great number of processors. Smith , after evaluating the convenience of an hybrid model[5], believes such a model could contribute with the best from MPI and OpenMP models and it seems a good solution to those cases where MPI model does not scale well. He also concludes that the appropriate model should be selected depending of the particular characteristics of the application. Finally, some authors such as Schloegel et al [12] and Walshaw et al. [13][14] have been working on the opposite approach. They have been working of solving the load-balancing problem of irregular applications by proposing mesh repartitioning algorithms and evaluating the convenience of repartition the mesh or just adjust them.

3 Dynamic Processor Balancing

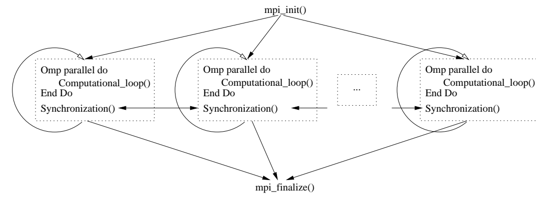


Figure 1. Basic MPI+OpenMP structure

MPI+OpenMp jobs are composed by a set of MPI processes that periodically synchronize. Each one of these processes opens loop parallelism, with OpenMP, inside them. Figure 1 shows the main structure of a two level MPI+OpenMP application. Computational loops can consume different time depending on the data each MPI process calculates. If the application is irregular, the amount of data to process can also vary during the application life for each MPI process.

Rather than redistributing data, processor balancing consists of redistributing computational power, that is the number of allocated processors, among collaborative MPI processes. Processor balancing can be done by the application itself or by the system. If the application performs this work itself three main problems arise: **(1)** the system can take decisions that unauthorize application decisions (this problem is also mentioned by Tafti in [6]), **(2)** the programmer has to introduce a complex implementation to dynamically evaluate the different computational percentages of each MPI group and redistribute OpenMP processes, and **(3)** power balancing could be runtime dependent and not *a priori* calculated. In any case, this is a complicated process that must be done by an expert and that requires to spent a lot of time.

Our proposal is that processor balancing can be done dynamically by the system transparently to the application and without any previous analysis. This approach has the advantage that it is totally transparent to the programmer, applications must not be modified depending neither on the architecture nor the data, and rescheduling decisions are taken considering not just the job but also the system workload. In this paper we will show that information extracted automatically at runtime is enough to reach a good balancing without modifying the original application at all.

Processor balancing is performed in several steps: Initially, the resource manager applies an Equipartition [15] policy. Once decided the initial distribution, each MPI process, while running normally, will measure the time dedicated to execute code and the time spent by communication such as barriers or sending/receiving messages. This computation will be automatically done by a run time library. This information will be sent periodically to the resource

manager, who will adjust the job allocation, moving processors from low loaded MPI processes to high loaded MPI processes (from the point of view of computation). This process will be repeated until a stable allocation is found.

Moreover, since the system has a global overview, it can detect situations such as applications that cannot be balanced, reallocating some of its processors to other running jobs where they could produce more benefit. Obviously, the final implementation should include some filters to avoid undesirable job behavior as ping-pong effects.

In next subsections the main elements introduced in the system are presented: the run time profiling library, the modifications done in the OpenMP run time to give support to this mechanism, and the Dynamic Processor Balancing policy (DPB) implemented inside the resource manager.

3.1 Profiling libray

Most scientific applications have the characteristic that they are iterative, that is, they apply the same algorithm several times to the same data. Data is repeatedly processed until the number of iterations reaches a fixed value, or until the value of some parameters reaches a certain value (for instance, when the error converges to a certain value). The profiling library exploits this characteristic to accumulate meaningful times for computation and communication usage.

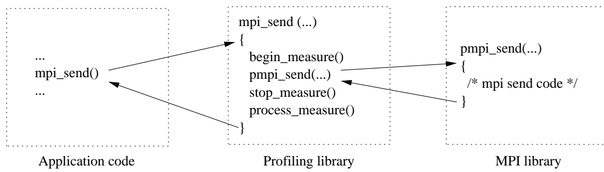


Figure 2. MPI profiling mechanism

MPI defines a standard mechanism to instrument MPI applications that consist of providing a new interface that it is called before the real MPI interface[16]. Figure 2 shows how the standard MPI profiling mechanism works. The application is instrumented using this profiling mechanism. When a MPI call is invoked from the application the library measures the time spent in the call and add its to a total count of time spent in MPI calls.

The iterative structure of the application is detected using a Dynamic Periodicity Detector library (DPD) [8]. DPD is called from the instrumented MPI call and it is feeded with a value that is a composition of the MPI primitive type (send, receive, ...), the destination process and the buffer address. With this value DPD will try to detect the pattern of the periodic behavior of the application. Once, a period is detected the profiling library keeps track of the time spent in executing the whole period.

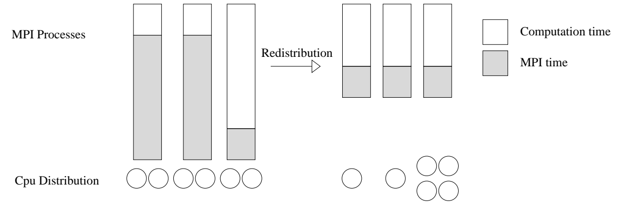


Figure 3. DPB example

These two values, mpi execution time and period execution time, are averaged from the values of a few periods and passed to the resource manager for feeding the allocation policy.

3.2 Dynamic Processor Balancing policy

The goal of the Dynamic Processor Balancing (DPB) policy is generating a processor distribution where all the MPI process spend the same amount of time in computation, reducing the computation imbalance as much as possible based on the data gathered by the profiling library.

Figure 3 shows an example of how DPB works. In the figure there is one process with more computation time than the others. This produces that global execution time increases as the other two processes (the ones in the left) are spending their time waiting at the synchronization point (send/receive, barrier, ...). In this case, DPB will take the decision of stealing a processor to each of the left processes and give them to the loaded one. This decision makes the two victim processes go slower but the global time is reduced due to a better utilization of the resources.

The policy is not working constantly but is only invoked when the resource manager has collected enough information for the policy to work (computation and MPI time from all processes of a job). Each time the policy is invoked it tries to improve one of the processes of the job, the one with highest computation time, by increasing its processor allocation. Those processors are stolen from a victim process. The chosen victim is the process with minimum computation time of those that have more than one processor (a process needs always at least one). Once the victim is selected, an ideal execution time for next allocation, t_{i+1} is computed for that process using the formula:

$t_{i+1}(highest) = t_i(highest) - (t_{mpi}(victim) - t_{mpi}(highest))/2$

This heuristic assumes that the MPI time of the process that has more computation time is the minimum MPI time that any of the process can have. Then, with this *future* time, the number of cpus that should be moved between the processes to obtain that time based on the last execution time, is calculated as follows:

$cpus = \frac{cpus_i(highest)*t_i(highest)}{t_{i+1}(highest)} - cpus_i(highest)$

Some restriction apply to the above:

- No process can have allocated less than one processor. So this means that sometimes the calculated cpus will not be possible. This case is treated giving the maximum possible.
- If the time t_{i+1} is estimated that will be worst that actual time the current allocation is maintained.

Even with this checks, sometimes a decision will lead to an increase in execution time. To recover from those situations the last allocation is always saved. When the policy detects that the last execution time was worst than the previous, it recovers the saved allocation. After there is any change in the allocation, the mpi and period time counters of theprofiling library are reset to zero to obtain new data from the job.

3.3 OpenMP runtime library modifications

When the policy decides a new allocation the resource manager informs the processes of their new processor availability, by leaving the new information in a shared memory zone of the process. After that, the OpenMP run time library should adjust it parallelism level (number of running threads) to comply with the system policy.

From the application point of view this can be done in two ways:

Synchronously Two rendezvous points are defined at the entrance and exit of parallel regions. When an application arrives at a synchronization point, it checks for changes in its allocation and adjusts its resources properly. So, this means that while the application is inside the parallel region could potentially run with more (or less) resources than those actually allocated to it.

Asynchronously In this version, the resource manager does not wait for the application to make the changes but it preempts immediately the processor stopping the running thread on it. As this can happen inside a parallel region, the run time needs the capability to recover the work that was doing or it has assigned that thread in order to exit the region. This is not an easy task as available resources can change several time inside a parallel region leading to deadlocks if not carefully planed. Further information of this approach can be found at Martorell et al. [10].

Our implementation, in the IBM's XL library, uses the first approach. As the parallel regions our work focuses are small enough, the time a process does not comply the allocation is so small that there are no significative difference between the results obtained with both approaches. So, the results obtained will be applicable to both scenarios as long as this restriction is maintained.

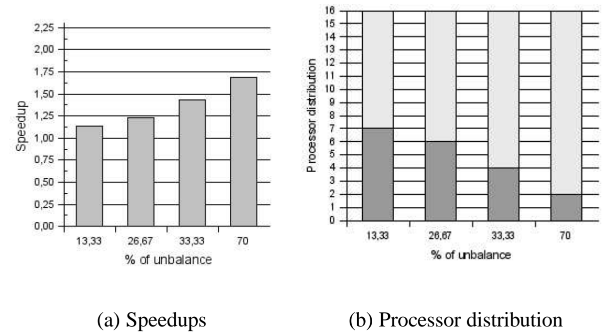
4 Evaluation

4.1 Architecture

The evaluation has been performed in a single node of an IBM RS-6000 SP with 8nodes of 16 Nighthawk Power3 @375Mhz (192 Gflops/s) with 64 Gb RAM of total memory. A total of 336Gflops and 1.8TB of Hard Disk are available. The operating system was AIX 5.1. MPI library was configured to use shared memory for message passing inside the node.

4.2 Synthetic case

Firstly, a synthetic application was used to discover the potential of the technique presented. The synthetic application includes a simple external loop with two internal loops. Two MPI processes execute the external loop and internal loops are parallelized with eigth OpenMP threads. At the beginning and the end of each external iteration there is a message interchange to synchronize MPI processes. So, it is a simple case that follows the structure shown in Figure 1. This synthetic job allows giving a specific workload to each of the MPI processes, allowing to use different imbalance scenarios.



(a) Speedups (b) Processor distribution

Figure 4. Results for synthetic imbalances

Four different scenarios have been tested: 13,33%, 26,67%, 33,33% and 70% of imbalance. The speedups obtained (the version without the balancing mechanism was taken as reference) are summarized in the Figure 4(a). This results show that the technique is able to cope perfectly with different imbalance situations so it can becoming interesting policy in order to balance, in a transparent way, hybrid applications. It obtains, at least, the same gain that the imbalance that has been introduced. In figure 4(b) the processor distribution that DPB used is shown. There it can be seen that, in fact, the percentage of processor unbalance of the allocation closely reassembles the imbalance of the scenario.

4.3 Applications

To verify our approach in a more complex situation we executed some MPI+OpenMP applications in a single SMP node. Each job made use of all the processors of a node distributed among the different MPI processes.

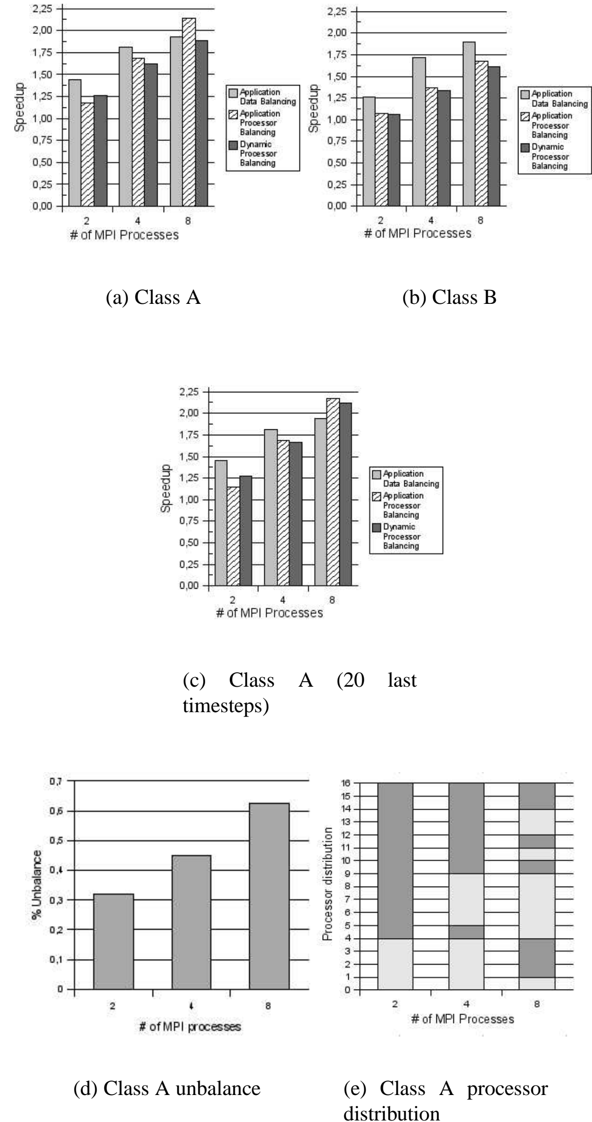
The applications, selected from the NAS Multizone benchmark suite [9], were: BT, LU and SP with input data classes A and B. These benchmarks solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions. BT zone partition is done asymmetrically so an equipartition of the zones (the default approach in all benchmarks) will result in an unbalanced execution. SP and LU on the other hand do the zone partition in a symmetric way so their execution are expected to be balanced.

All the NAS-MZ benchmarks come with two load balancing algorithms, which can be selected at compile time. This algorithms represent slightly more than a 5% of the total code. Their objective is to overcome the possible imbalances from the default equipartion of the zones. First one, maps zones to MPI process trying that all them have a similar amount of computational work. It also, tries to minimize communication between MPI processes by taking into account zone boundaries. Second one, assigns a number a processors to each MPI process based on the computational load of the zones assigned to it from the initial equipartition. Both methods are calculated before the start of the computation based on knowledge of the data shape and computational weight of the application so we will refer to the first one as Application Data Balancing and to the second as Application Processor Balancing. Our objective is to obtain a similar performance to Application Processor Balancing but done in a dynamic and application independent way without modifiying the source code at all.

We have executed the NAS-MZ benchmarks with these two approaches, with DPB, and with a simple equipartition version, which has been used as reference value for the calculation of the speedup in the following sections.

4.3.1 Irregular zones (BT-MZ)

The situation that the data domain is irregular either in its geometry or in its computational load is very frequent on scientific codes mainly because of the nature of the entities being modelled (weather forecasting, ocean flow modelling, . . .). So the BT-MZ benchmark will evaluation will help to see if DPB can help to improve those codes.



(a) Class A (b) Class B

(c) Class A (20 last timesteps)

(d) Class A unbalance (e) Class A processor distribution

Figure 5. Results for BT-MZ

Figure 5(a) shows the speedup for the different load balancing strategies with data class A. As it can be seen DPB is closely tied with the Application Processor Balancing algorithm. For a two MPI processes execution, DPB gets some more performance (1,26 vs 1,18 of speedup), with four MPI

processes DPB is just a 4% behind. For the eight processes execution this difference is really high (14%). This is due to the warmup time of DPB (time to detect the application structure, obtain the first measures, find a stable allocation, ...). So, for longer executions DPB will be as good option as the Application Processor Balacing algorithm. This hypothesis is confirmed if we take a look at the speedups obtained for the last 20 time steps of the benchmark (figure 5(c)). There, not only the difference for the eight processes case decreases until a 2%, that is due to the dynamic profiling, but also the difference in the two processes case is bigger. For further confirmation, if we look the speedups of the data class B, which have a longer execution time, we can see that the differences between the two method remain similar.

When comparing with Application Data Balancing, this method obtains better performance in most cases. This is because data distribution allows finer movements than processor distribution. Even so, for the class A with 8 MPI processes both processor balancing techniques obtain better speedup (see figure 5(c)) so it is not a clear option in all the situations and worst of all it can not be performed transparently as DPB.

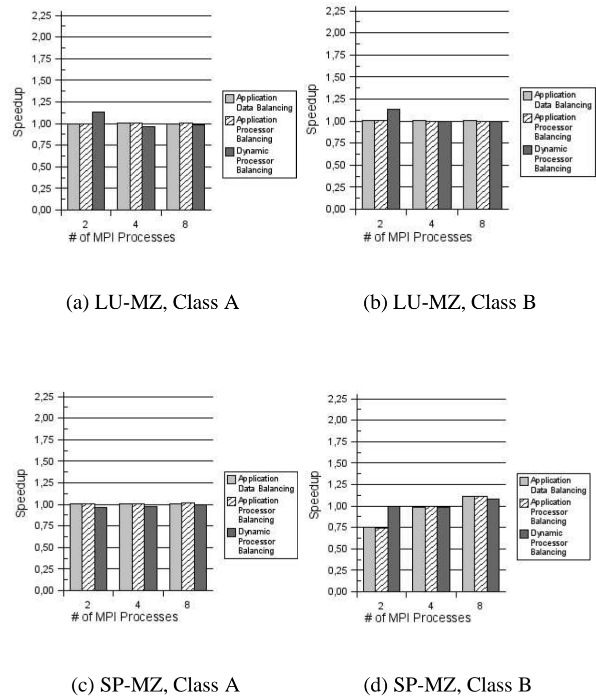
In figure 5(d) shows the processor distribution that DPB chooses for the different MPI processes configuration for data class A. It can be seen that the distributions found by the algorithm are quite complex.

4.3.2 Regular domains (LU-MZ, SP-MZ)

Evaluating a benchmark that is already balanced will show the overheads introduced by the profiling and constant monitoring of DPB.

Figure 6 shows the different speedups obtained for the LU-MZ and for the SP-MZ benchmarks for both, data class A and B. Surprisingly enough, DPB seems to improve the execution for both data classes of the LU-MZ benchmark with two MPIs (1,13 of speedup) when the benchmark was supposed not be unbalanced. Actually, the improvement is not due to a better processor distribution but because the implementation of the original benchmark spent to much time yielding and that affected their execution time. The same thing happens for the two MPIs case of the SP-MZ class B (see figure 6(d)) but here the reference execution was not affected by the yielding effect and the other seem to scale down.

If we concentrate on the other cases, we can see that none of the methods gets an improvement. And most important we can see that DPB doesn't have an important impact on their executions (a maximum of a 3% overhead) which it means it has a fairly good instrumentation method that it is almoust negligible.



(a) LU-MZ, Class A

(b) LU-MZ, Class B

(c) SP-MZ, Class A

(d) SP-MZ, Class B

Figure 6. Speedups for LU-MZ and SP-MZ

4.4 Effect of node size

We have evaluated the effect in the improvement achieved by DPB when varying the node size. Results presented in the previous sub-sections have been calculated with a single node with 16 cpus. Since we don't have a distributed version of the resource manager, we have performed some experiments to give an idea about the reduction in the speedup introduced when node size are small.

These experiments consist of the concurrent execution of two and four instances of the resource manager, each one managing a subset of the node: eight processors when sim-

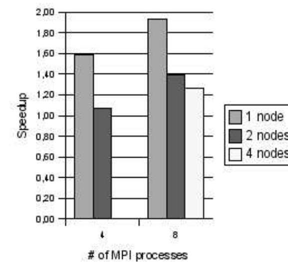


Figure 7. BT-MZ Class A for different nodes sizes

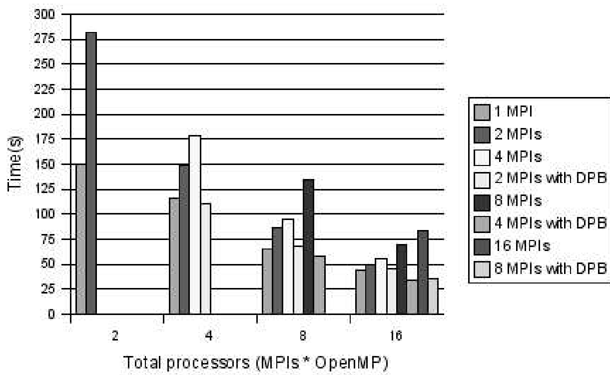


Figure 8. BT Class A. MPI vs MPI+OpenMP

ulating two nodes and four processors when simulating four nodes.

Figure 4.4 shows the speedup achieved by BT.A when executing in 1 node with 16 cpus, 2 nodes with eight cpus each one, and 4 nodes with 4 cpus. In all the experiments there are 16 cpus. The speedup is calculated comparing with the execution time of BT.A without load balancing.

With small nodes, DPB has less chances to move processors between MPI processes and this results in a reduction in the speedup. For instance, in the case of 8 mpi processes, the speedup goes from 1.9 when running in a node with 16 cpus to 1.3 when executing in 4 nodes (2 mpi processes per node).

4.5 Comparing MPI vs MPI+OpenMP

In figure 4.5 is shown a comparative of the execution times for different processor availability of combinations of MPI and Openmp: ranging from pure OpenMP (only one MPI process) to pure MPI (only one OpenMP in each) going throught hybrid combinations. There it can be seen than using more MPIs only increases the execution time while using just one gets the lowest time. When our dynamic policy is used even lowest execution times are achieved with hybrid configurations.

These results show point that when an application is unbalanced is better to use and hybrid approach that allows to overcome the unbalance either with a hardcoded algorithm, as those shown in 4.3.1, or ,even better, automatically like the proposal of this paper. When the application is well balanced, on the other hand, it may be worth it to use a pure MPI approach [4].

5 Conclusions and Future work

This papers investigates the feasibility of having a Dynamic Processors Balancing algorithm that helps to reduce the imbalance presented on MPI+OpenMP applications. Our proposal works at the system level changing the resource allocation of the jobs for improving their utilization and reducing total execution time. We have shown, as with a simple low overhead profiling mechanism, enough information can be collected to perform this task properly. In fact, results show close performance (sometimes even better) with some other application specific handcrafted techniques. Those other techniques require a good knowledge of the data geometry of the application and spending a considerable effort in analyzing and tuning of the applications. Also, the fact that our proposal does not need any modification in the application combined with the low impact that has on already balanced applications, because of its low overhead, makes it a good candidate for a default system component.

Future work will follow three directions. First of all, the developed technique will be used to evaluate a broader range of benchmarks and applications, and also evaluate them in bigger configurations. The second line of work it is to expand the current platform for using it for workloads. This means researching policies to maximize some system metric (like throughput) as well as application specific metrics. This new platform will be distributed, dealing with jobs that span through more than one SMP node. And the last line will be trying to get better resource usage by allowing processor sharing between processes. To be able to achieve that coordination between the three levels (system,MPI and OpenMP) will required.

6 Acknowledgements

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC2001-0995-C02-01, the ESPRIT Project POP (IST -2001-33071) and by the IBM CAS Program. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

References

[1] W. Huang and D. Tafti. ”A parallel Computing Framework for Dynamic Power Balancing in Adaptive Mesh Refinement Applications”. Proceedings of Parallel Computational Fluid Dynamics 99.

[2] D.S. Henty. ”Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling”. Proc. of the Supercomputing (SC). 2000.

[3] H. Shan, J.P. Shingh, L. Oliker and R. Biswas. ”A comparison of Three Programming Models for Adaptive Applications on the Origin2000”. Proc. Of Supercomputing (SC) 2000.

[4] F. Cappello and D. Etiemble. ”MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks”. Proc. of the Supercomputing (SC). 2000

[5] L.A. Smith. ”Mixed Mode MPI/OpenMP Programming” . Edimburg Parallel Computing Centre, Edinburgh, EH9 3JZ

[6] D.K.Tafti. ”Computational Power Balancing, Help for the overloaded processor”. http://www.me.vt.edu/people/faculty/tafti.html

[7] D.K. Tafti, G. Wang. ”Application of Embedded Parallelism to Large Scale Computations of Complex Industrial Flows”. http://www.me.vt.edu/people/faculty/tafti.html

[8] F. Freitag, J. Corbalan, J. Labarta. ”A Dynamic Periodicity Detector: Application to Speedup Computation”. IPDPS 2001.

[9] R.F. Van der Wijngaart, H.Jin, ”NAS Parallel Benchmarks, Multi-Zone Versions”. NAS Technical Report NAS-03-010. July 2003.

[10] X. Martorell, J. Corbalan, D. Nikolopoulos, N. Navarro, E. Polychronopoulos, T. Papatheodorou and J. Labarta. ”A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager”. Proc. of the 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP’2000), in conjunction with the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS’2000), May 2000.

[11] D. Dent, G. Mozdzynski, D. Salmond and B. Carruthers. ”Implementation and Performance of OpenMP in ECMWF’s IFS Code”. Fifth European SGI/Cray MPP Workshop, September 1999.

[12] K. Schloegel, G. Karypis and V. Kumar. ”Parallel multilevel algorithms for multi-constraint graph partitioning”. Technical Report #99-031. 1999. University of Minnesota, Minneapolis.

[13] C. Walshaw, A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten.”Dynamic multi-partitioning for parallel finite element applications”. In Proc. of Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo’99. August 1999. Imperial College Press. pages 259-266, 2000.

[14] C. Walshaw and M. Cross. ”Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes”. In Computational Mechanics Using High Performance Computing, pages 79-94. Saxe-Coburg Publications, Stirling, 2002.

[15] C. McCann, R. Vaswani, and J. Zahorjan. ”A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors”. ACM Transactions on Computer Systems, 11(2):146–178, May 1993.

[16] Message Passing Interface Forum. ”The MPI message-passing interface standard”. http://www.mpi-forum.org, May 1995.