7

Using hardware resource allocation to balance HPC applications

Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla and Mateo Valero
Barcelona Supercomputing Center
Spain

1. Introduction

High Performance Computing (HPC) applications are usually *Single Process-Multiple Data* (SPMD) and are implemented using an MPI or an OpenMP library. In MPI applications, all the processes execute the same code on different data sets and use synchronization primitives (such as barriers or collective operations) to coordinate their work. Since the processes execute the same code, they are supposed to reach their synchronization points roughly at the same time.

However, this is not always the case, as many applications suffer from *imbalance*, where a parallel application has multiple inter-dependent tasks[1] and these tasks have to wait for others to complete in order to continue their execution (in Section 2 we will see some causes of applications' imbalance). During this waiting time, the CPUs of the waiting tasks are idle, thus, not performing any useful job. If one process has to complete its execution while all the other processes are waiting for it to reach the synchronization point; then several processors may be idle, resulting in a significant loss of performance and waste of resources.

In fact, imbalance is a very common problem that has been studied by many researchers. Since there are several different factors that may create or make variable imbalance, there is no trivial solution and no solution solves all application's imbalance. A more detailed survey about solutions for the problem of imbalance is presented at Section 5.

Most of the current Supercomputers use processors with some multi-threaded features (TOP500, 2007). In the last years, the performance achievable by traditional super-scalar processor designs has almost saturated due to the limitation imposed by Instruction-Level Parallelism (ILP). As a consequence, Thread-Level Parallelism (TLP) has become a common strategy for improving processor performance. Since it is difficult to extract more Instruction-Level Parallelism from a single program, MultiThreaded (MT) processors, that is, processors that execute multiple threads at the same time, obtain more parallelism by simultaneously executing several tasks. This strategy has led to a wide range of MT processor architectures, from Simultaneous Multi-Threaded processors (SMT) (Serrano et al., 1993; Tullsen et al., 1995; Marr et all, 2002), in which most processor resources are shared

[1]In this chapter, the term *task* refers to a software entity representing an MPI process, a *software thread* or simply a *process*.

among hardware threads[2], to Chip Multi-Processors (CMP) (Bossen et al, 2002), in which every hardware thread has its own dedicated processor resources, only sharing the highest levels of the memory hierarchy (for example the L2 cache), and a combination of both (Sinharoy et al., 2005; IBM et al. 2006; Le et al, 2007). Resource sharing makes multi-threaded processors have good performance/cost and performance/power consumption ratios (Alpert, 2003), two desirable characteristics for a supercomputer.

Usually, software has no control over how processor resources are distributed among the active hardware threads in multi-threaded processors. For example, in an SMT processor the *instruction fetch policy*, decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources are allocated to the threads. This is an undesirable characteristic that makes the execution time of programs unpredictable (Cazorla et al., 2006). In order to alleviate this problem, recently, some processor vendors have equipped their MT processors with mechanisms that allow the software to control processor's internals resource allocation, and thus, control application's speed.

There are several ways to reassign hardware resources in multi-threaded processors. In theory, every shared resource in a system can be partitioned or biased to satisfy a load-balancing target. For instance, cache replacement policy, processor fetch or decode cycles, power and several other split or shared resources can be controlled to improve the execution of a set of critical tasks in order to balance a parallel application.

In practice, currently, not every system allows such control over its hardware resources. For instance, dynamic voltage scaling can be used to save power for the slower tasks without sacrificing the performance of the critical tasks (the ones that limit the application's execution time), but it will not provide performance speedup. In cases where it is possible to give more resources to the critical tasks, increasing its speed, there is potential to decrease the overall program's execution time. These mechanisms open new opportunities to improve the performance of parallel applications.

The work presented in this chapter is a first step toward the use of hardware resource allocation to improve software targets: re-assigning hardware resources in a multi-threaded processor can reduce the imbalance in parallel applications, and hence improve their performance. In particular, this work presents a way to regain balance assigning more hardware resources to processes that compute the longer. The solution is transparent to the users and is implemented at the Operating System (OS) or run-time levels. In order to use it, users do not need to adapt their programming model or to know specific processor's implementation details when writing or compiling their applications.

In this chapter, the idea of load balancing through smart hardware resource allocation is explored experimentally on a real system with an MT processor, the IBM POWER5™ (Kalla et al., 2004). The POWER5 is a dual-core, 2-way SMT processor that allows us to change the way hardware resources are assigned to the core's SMT contexts by means of a *software-controlled hardware priority* (or hardware thread priority[3]) that controls the number of resources each context receives. This machine runs a Linux kernel that we modified in order to allow the HPC application to exploit the advantage of assigning the processor's resources.

[2]The terms *thread*, *hardware thread* and *context* are employed interchangeably to refer to a *hardware context* of an SMT processor.

[3]The hardware thread priorities mentioned here are independent of the operating system's concept of software thread priority.

As case studies, we performed several experiments with MPI applications, focused on the IBM POWER5. We present them in increasing order of complexity, that is, when their imbalance becomes more and more variable:

1. We started from a micro-benchmark (Metbench), developed at the Barcelona Supercomputing Center (BSC), where we introduce some artificial imbalance.
2. In the second experiment, we ran the widely used the NAS BT-MZ (NASA, 2009) benchmark; this version suffers of load imbalance, as shown in Section 4.2.
3. We demonstrate the effect of the proposal on a dynamic application (MetbenchVar), motivating the push for dynamic mechanisms that use hardware resource allocation, effectively using resource redistribution to perform load balancing.
4. Finally, we present a real application running on MareNostrum, SIESTA (SIESTA, 2009; Soler et al., 2002). With this specific input, SIESTA exhibits a very unpredictable imbalance.

Our results show that controlling hardware resources is a powerful tool that can significantly decrease applications' execution time. However, if used incorrectly, it may lead to significant performance loss. Moreover, non-HPC applications may benefit differently from re-assigning hardware resources.

The rest of this chapter is organized as follows: Section 2 shows the imbalance problem in HPC applications, classifying and discussing its sources; Section 3 introduces the concept of load-balancing based on smart allocation of hardware resources; we present the POWER5 processor and its prioritization mechanism, and the Linux kernel interface required to use the prioritization system. Section 4 shows our case-studies; Section 5 presents similar works in the same area; finally Section 6 provides our conclusion and future work.

2. Imbalance in HPC applications

HPC applications are usually SPMD, which means that every process executes the same code on different data. For example, let's assume that an HPC application is performing a matrix-vector multiplication and that each process receives a sub-matrix and the part of the vector required to compute the sub-matrix by vector multiplication. If the matrix can be divided into homogeneous parts (i.e., they require the same amount of time to be processed), all the processes in the parallel application would finish, ideally, at the same time.

However, the data set could be very different: let's suppose that, in the previous example, the matrix is sparse or triagonal, hence, the time required to process the data sub-set could vary as well. In this scenario the amount of time required to complete the sub-matrix by vector multiplication depends on the number of non-zero values present in the sub-matrix. In the extreme case, one process could receive a full sub-matrix while another gets an empty one. The former process requires much more time to reach the synchronization point than the latter.

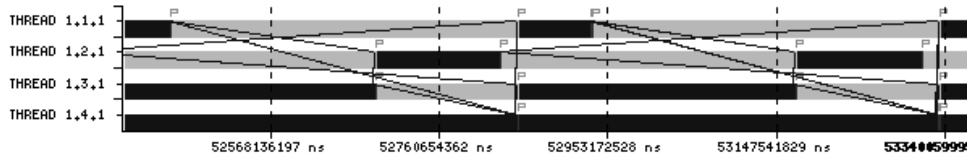


Fig. 1. Two iterations of NAS BT-MZ showing the message exchanges. In this trace, black areas represent computation, grey areas represent waiting time.

The NAS BT-MZ benchmark, explained in Section 4.2, is a clear example of an imbalanced application due to data distribution. As shown in Figure 1, each MPI process communicates with its two neighborhoods, exchanging data after each iteration. The processes get different amount of work and the process *P*4 gets to perform the largest part of the computations. At the end, because of the communications, all other processes are slowed down by *P*4 and have to wait for most of their time in order to allow *P*4 to complete its job.
We classify the sources of imbalance in two main classes: *intrinsic* and *extrinsic* factors of imbalance. Bellow we detail issues and possible reasons for both of the classes.

2.1. Intrinsic imbalance

We refer to *intrinsic imbalance* as the imbalance an application experiences because of data (for example a sparse matrix) or algorithm (as for instance, a branch and bound implementation where some branches may be cut much earlier than others and each task gets a set of branches). The causes for the intrinsic imbalance are internal to the application's code, input set or both. It could be caused by several factors; here we point some of them out:
Input set: As we already said, this scenario happens when a process has a small input set to work on while another has a large amount of data to process. One example of application that is strongly dependent on the input set is SIESTA (Soler et al., 2002) (described in better details in Section 4.4).
SIESTA analyzes materials at the atom level. Depending on the distribution and density of the atoms across the material, some processes may perform more work than others. Very homogeneous materials tend to be well balanced, although SIESTA may also present imbalance caused by the algorithm. Figure 2 shows the trace of SIESTA when processing atoms of graphite (C6). In this case, the four MPI processes execute, respectively for 92.82%, 91.44%, 91.81% and 91.68% of the time. In fact, if we discard the initialization phase, they all have more than 98.80% of CPU utilization.
In another case, shown in Figure 3, when processing PTCDA molecules (perylene-3,4,9,10-tetracarboxylic-3,4,9,10-dianhydride), it exhibits a highly imbalanced execution: the MPI processes show respectively 92.94%, 21.79%, 96.60%, 21.71% of utilization.
Domain: Iterative methods approximate the solution of a problem (for example, Partial Differential Equations, PDE) with a function in some domain starting from an initial condition. The domain is divided in several sub-domains and each process computes its part of the solution. At the end of every iteration, the error made in the approximation is computed and, eventually, another iteration is to be started. If the function in some part of the domain is smooth, only few iterations are required to converge to a good approximation. Conversely, if the function has several peaks in the sub-domain, more

iterations are necessary to find a good solution and/or more points in the domain have to be considered during the computing phase.

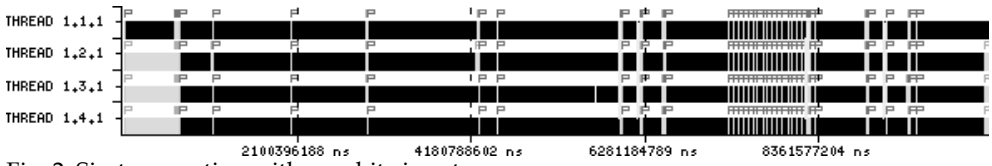


Fig. 2. Siesta execution with graphite input.

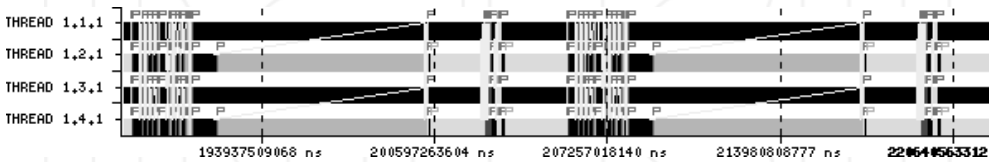


Fig. 3. Siesta execution with PTCDA input. Only part of the execution is pictured.

Data exchanging: During their execution, processes may require to exchange data among themselves. If the two peers are on the same node, the latency of the communication is small; if a process needs to exchange data with a neighbor on another node the latency is large, even larger if the destination process is far away in the network.
In all the previous cases, the application may result to be imbalanced.

2.2. Extrinsic imbalance

Even if both the application's algorithm and the input set are balanced, the execution of the parallel application can still be imbalanced. This is caused by external factors that slow some processes down (but not others). For example, the Operating System (OS) might decide to run another process (say a kernel daemon) in place of the MPI process running on one CPU. Since that MPI process is not able to run all the time while the others are running, it takes longer to complete, making all the other processes wait for it. Those external factors are the sources of *extrinsic imbalance*. There may be several causes for the imbalance:
OS noise: The CPU is used by the OS to perform services such as handling interrupts, page reclaiming, assigning memory on demand, etc. The OS noise has been recognized as one of the major source of extrinsic imbalance (Gioiosa et al., 2004; Petrini et al., 2003; Tsafrir et al., 2005). A classical example is the interrupt annoyance problem present in Intel processors: all the interrupts coming from external devices are routed to CPU0; therefore, the OS noise caused by executing the interrupt handlers on CPU0 is higher than the noise on the other CPUs.
User daemons: HPC systems often run profile or statistic collectors together with the HPC applications. These activities could steal computing power from one process, delaying its execution.
Network topology: Exchanging data between processes in the same sub-network is faster than exchanging data between processes in different sub-networks. In general, if the job scheduler has placed processes that need to communicate ``far away'', their communication latency could increase so much that the whole application will be affected.

Memory management: Even inside a single node, it is common to have NUMA (Non-Uniform Memory Access). A process that requests a large amount of memory may have it allocated in a memory region that is comparably slower than the memory allocated to the other processes of a parallel application (maybe because there is not enough memory close enough to this processor). In this case, the performance of this process will be significantly impacted and, depending on the application, this process may delay the execution of the entire program, making the others wait for its results.

An expert programmer could reduce the intrinsic imbalance in the application. However, this is not an easy task, as the imbalance can be caused by the algorithm, but it can also be caused by the input data set, changing distribution and intensity according to different inputs. Balancing a HPC application by hand is a time-consuming task and may require quite a lot of effort. In fact, the programmer has to distribute the data among the processes considering the way the algorithm has been implemented and the correctness of the application. Moreover, on many applications this work has to be done every time the input or the machine change.

Even worse is the case of extrinsic imbalance, as those factors are neither under the control of the application nor of the programmer and there is no straightforward way to solve this problem. Thus, it is clear that a mechanism that aims to solve the imbalance of an application should be transparent to the user, dynamic and independent from the programming model, libraries or input set. As we will see later, the proposal presented in this chapter is both transparent and independent from the programming model, libraries and input set.

3. Hardware Resource Allocation

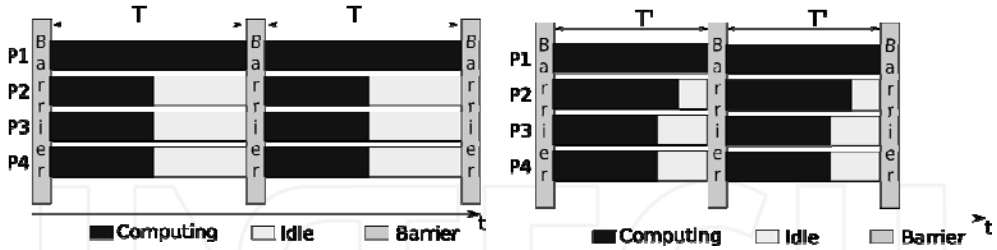
With the arrival of MT architectures, and in particular those that allow the software to control processor's resource allocation, new opportunities arise to mitigate the problem of imbalance in HPC applications. This is mainly due to the fact that the software is allowed to exercise a fine control over the progress of tasks, by allocating or deallocating processor resources to them. Such a fine-grain control cannot be achieved by previous solutions for load imbalance; in fact, even if a lot of processors with shared resources have been introduced in the market since early 90s, very few of them allow the software to control how internal resources are allocated. Allowing the software to control how to assign shared resources is a key factor for HPC systems. In this view, having MT processors able to provide such mechanism will be essential for improving the performance of HPC systems.

The solution presented in this chapter for balancing HPC applications, consists of assigning more hardware resources to the most compute-intensive processes (the bottleneck). Giving this process more hardware resource shall decrease its execution time and, since this process is the bottleneck of the application, the execution time of the whole MPI application.

Clearly the underlying processor has to support the capability of re-assigning processor resources among running contexts. Currently, multi-threaded processors like the IBM POWER5 (Kalla et al., 2004), the POWER6 (Le et al., 2007) or the Cell processor (IBM et al., 2006; IBM, 2008) provide such a capability with their hardware thread priority mechanisms. More details about the POWER5 prioritization mechanism are available in Section 3.1.

Even if in this chapter we focus on the IBM POWER5, the idea presented is general and can be applied to other MT processors that allow the OS to the control or influence the allocation of processor's resources (for example, partitioning a shared L2 cache in a multi-core CPU

(Moreto et al., 2008; Qureshi and Patt, 2006). The IBM POWER5 processor is used, among others, by ASC Purple, installed at the Lawrence Livermore National Laboratory[4].



(a) Imbalanced HPC application (b) More balanced HPC application

Fig. 4. Expected effect of the proposed solution ($T' < T$).

We should point out that increasing the performance of one process by giving it more hardware resources, does not come for free. In fact, at the same time, the performance of the other process running on the same core, therefore sharing the resources with the former process, may reduce. Figure 4 shows a synthetic example that illustrates this case: in Figure 4(a), process $P1$ shares resources with $P2$, while $P3$ shares them with $P4$; $P2$, $P3$ and $P4$ take the same amount of time to reach their synchronization point but $P1$ takes much longer. As a result, $P2$, $P3$ and $P4$ are idle for a long time. In Figure 4(b), we increase the priority of $P1$, so it uses more hardware resources and its execution time decreases; $P2$'s execution time, instead, increases since it runs with less hardware resources. Since $P2$ is not the bottleneck and has enough "spare time", its slowdown does not affect the application's performance. On the other hand, the performance improvement of $P1$ directly translates into a performance improvement for the whole application, as it is possible to see comparing Figures 4(a) and 4(b).

No assumption is made on what kind of application, programming model or input set the programmer has to use. The only assumption made is that the underlying processor must provide a mechanism, visible at software level, to control the hardware shared resources. The solution for load balancing through hardware resource allocation works at OS level and is completely transparent to the users, who are free to use the MPI, OpenMP or any other programming model or library they wish. Moreover, the approach can be adjusted so the amount of resources assigned to a process can change according to the input set provided to the application.

It is important to notice that not all the POWER5 priorities are available from the user-level and a special kernel patch was needed to enable the use of the full spectrum of software-controlled hardware priorities. For the technique presented in the current chapter, we employ the same patch developed to perform the characterization in (Boneti et al., 2008a). The patch only provides a mechanism to set all the priorities (available at OS level) from user applications. It is the responsibility of the user applications (or run time systems) to balance the system load using this interface.

[4]The 3rd supercomputer in the Top500 list of 06/2006, the 11th at the list of 11/2007.

3.1. The IBM POWER5 processor

The IBM POWER5 (IBM, 2005a; IBM, 2005b; IBM, 2005c; Sinharoy et al., 2005) processor is a dual-core chip where each core is a 2-way SMT core (Kalla et al., 2004). Each core has its own private first-level data and instruction caches. The unified second- and third-level caches are shared between cores.

The forms of Multi-Threading implemented in the POWER5 are Simultaneous Multi-threading and Chip-Multiprocessing. The main characteristic of SMT processors is their ability to issue instructions from different threads in the same cycle. As a result, SMTs not only can switch to a different thread to use the idle issue cycles in a long-latency operation, like coarse-grain multi-threading, or in a short-latency operation, like in a fine-grain multi-threaded, but also fill unused issue slots in a given cycle.

What makes the IBM POWER5 ideal for testing our proposal is the capability that each core has to assign some hardware resources to one context rather than to the other. Each context in a core has a *hardware thread priority* (Boneti et al, 2008a; Gibbs et al., 2005; Kalla et al., 2003), an integer value in the range of 0 (the context is off) to 7 (the other context is off and the core is running in Single Thread (ST) mode), as illustrated in Table 1. As the hardware thread priority of a context increases (keeping the other constant) the amount of hardware resources assigned to that context increases too.

Priority	Priority level	Privilege level	or-nop inst.
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User	`or 1,1,1`
3	Medium-Low	User	`or 6,6,6`
4	Medium	User	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

Table 1. Hardware thread priorities in the IBM POWER5 processor

3.1.1. Thread priorities implementation

The way each core assigns more hardware resources to a given hardware thread is by decoding more instructions from that thread than from the other. In other words, the number of decode cycles assigned to each thread depends on its hardware priority. In general, the higher the priority, the higher the number of decode cycles assigned to the thread (and, therefore, the higher the number of shared resources held by the thread).

Let's assume two threads (ThreadA and ThreadB) are running on a POWER5 core with priorities X and Y, respectively. In POWER5 the decode time is divided in time-slices of *R* cycles: the lower priority thread receives 1 of those cycles, while the higher priority thread receives (*R*-1) cycles. *R* is computed as:

$$R = 2^{|X-Y|+1} \tag{1}$$

Table 2 shows the possible values of *R* and how many decode slots are assigned to the two threads as the difference between ThreadA's and ThreadB's priority moves from 0 to 4. In

fact, the amount of resources assigned to a thread is determined using the difference between the thread priorities, X and Y. For example, assuming that ThreadA has hardware priority 6 and ThreadB has hardware priority 2 (the difference is 4), then the core fetches 31 times from context0 and once from context1 (more details on the hardware implementation are provided in (Gibbs et al., 2005). It is clear that the performance of the process running on Context0 shall increase to the detriment of the one running on Context1. When any of the threads has priority 0 or 1, the behavior of the hardware prioritization mechanism is different, as shown in Table 3.

Priority difference (X-Y)	R	Decode cycles for A	Decode cycles for B
0	2	1	1
1	4	3	1
2	8	7	1
3	16	15	1
4	32	31	1

Table 2. Decode cycle allocation in the IBM POWER5 with different priorities.

Thread A	Thread B	Action
>1	>1	Decode cycles are given to the two threads as according with the thread's priorities.
1	>1	ThreadB gets all the execution resources; ThreadA takes what is left over.
1	1	Power save mode; both ThreadA and ThreadB receive 1 of 64 decode cycles.
0	>1	Processor in ST mode. ThreadB receives all the resources.
0	1	1 of 32 cycles are given to ThreadB.
0	0	Processor is stopped.

Table 3. Resource allocation in the IBM POWER5 when the priority of any of the threads is 0 or 1.

3.1.2. Hardware interface for priority management

The IBM POWER5 provides two different interfaces to change the priority of a thread: issuing an `or`-nop instruction or using the *Thread Status Register* (`TSR`). We used the former interface, in which case a thread has to execute an instruction like `or X,X,X`, where `X` is an specific register number (see Table 1). This operation does not do anything but changing the hardware thread priority. Table 1 also shows the privilege level required to set each priority and how to change priority using this interface. The second interface consists of writing the hardware priority into the local (i.e., per-context) `TSR` by means of a `mtspr` operation. The actual thread priority can be read from the local `TSR` using a `mfspr` instruction.

3.2. The Linux kernel interface to hardware priorities

By default, users can only set three hardware priorities: MEDIUM (4), MEDIUM-LOW (3) and LOW (2). This basically means that users are only allowed to reduce their priority, since the MEDIUM priority is the default case. If the user reduces the thread priority when a process

does not require lot or resources (for example because the process is waiting for a lock), the overall performance might increase (because the other thread receives more resources and, therefore, may go faster). Thus, it is recommended that the user reduces the thread priority whenever the thread processor is executing a low-priority operation (such as spinning for a lock, polling, etc.).
Modern Linux kernels running on IBM POWER5 processors make use of the hardware priority mechanism the chip provides. In this Section we will first explore the standard behavior of the Linux kernel when dealing with hardware priorities, and then present how we modified the standard kernel in order to solve the imbalance problem by means of the IBM POWER5 hardware prioritization mechanism.

3.2.1. The use of priorities in the standard Linux Kernel

The Linux kernel only exploits hardware priorities in a limited number of cases: the general idea is to reduce the priority of a process that is not performing any useful operation and to give more resources to the process running on the other context.
The standard Linux kernel makes use of the thread priorities in three cases:

1. The processor is spinning for a lock in kernel mode. In this case the priority of the spinning process is reduced (the process is not really advancing in its job).
2. The kernel is waiting for some operations to complete. This happens, for example, when the kernel wants a specific CPU to perform some operation by means of a `smp_call_function()` (for example, invalidating its TLB) and cannot proceed until the operation has completed. In this case the priority of the CPU is decreased until the operation completes.
3. The kernel is running the idle process because there is no other process ready to run. In this case the kernel reduces the priority of the idle CPU and, eventually, put the core in Single Thread (ST) mode.

In all these cases the kernel reduces the priority of the context, restoring the priority to MEDIUM when there is some job to perform. The hardware thread priority is also reset to MEDIUM as soon as the kernel executes an interrupt or an exception handler as well as a system call. In fact, since the kernel does not keep track of the current priority, it cannot restore the process' priority. Therefore, the kernel simply resets the priority to MEDIUM every time it starts to execute an interrupt handler (or a system call), so that it can be sure that those critical operations will be performed with enough resources.

3.2.2. Modification to the Linux kernel

In order to use the hardware prioritization for balancing the HPC application, we modified the original kernel code for two reasons:

1. Every time the CPU receives an interrupt, the interrupt handler sets the priority back to MEDIUM, regardless of the current priority. We want to maintain the given priority even after an interrupt is received or during the interrupt handler itself; thus, we removed the code that makes use of the hardware thread priority capabilities from the handlers.
2. Only hardware priorities 2 (LOW), 3 (MEDIUM-LOW) and 4 (MEDIUM) can be set by a user-level program. Priorities 1 (VERY LOW), 5 (MEDIUM-HIGH) and 6 (HIGH) can only be set by the Operating System (OS). Priorities 0 (context off) and 7 (VERY HIGH, ST mode) can only be set by the Hypervisor. We developed an interface that allows

the user to set all the possible priorities available in kernel mode. A user who wants to set priority `N` to process `<PID>` shall simply write to a `proc` file, like:

```
echo N > /proc/<PID>/hmt_priority
```

This patch provides a mechanism to set all the priorities from user applications. It is developed for several standard kernel versions (2.6.19, 2.6.24, 2.6.28, etc) in a way that it is not intrusive and has no impact on the performance of our experiments. With this patch, it is the responsibility of the user applications, system scheduler or run time systems to balance the system load. It is the building block that can be used for other mechanisms, like the transparent load balancer proposed in (Boneti et al., 2008b).

4. Case Studies on the IBM POWER5 processor

In this section, we present some experiments on an IBM OpenPower 710 server, with one POWER5 processor. Since MPI is the most common protocol, the test cases in this section are MPI applications (in the experiments we used the MPI-CH 1.0.4p1 implementation of MPI protocol).

We present four different cases: Section 4.1 shows how the IBM POWER5 priority mechanism works using our micro-benchmark (Metbench); Section 4.2 provides details on how the hardware priorities can be used to balance a widely used benchmark (NAS BT-MZ) and improve its performance. Section 4.3 presents a different version of Metbench that presents dynamic behavior and, thus, variable imbalance. Finally, 4.4 shows how the hardware prioritization improves the performance of a real application frequently executed on MareNostrum (SIESTA). In this case, SIESTA receives an input that makes it exhibit a variable behavior and imbalance.

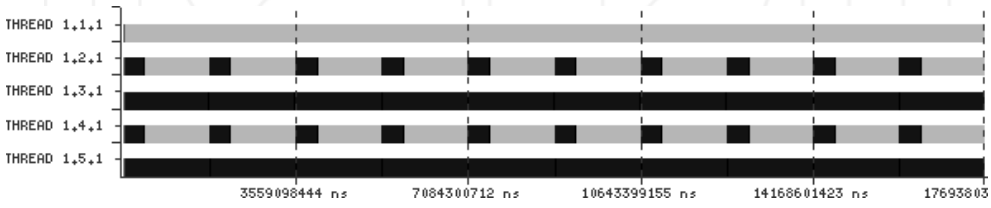
In order to present experiments in a simple way, we use as metric the total execution time of the application. We use PARAVER (Labarta et al., 1996), a visualization and performance analysis tool developed at CEPBA, to collect data and statistics and to show the trace of each process during the tests.

4.1. Metbench

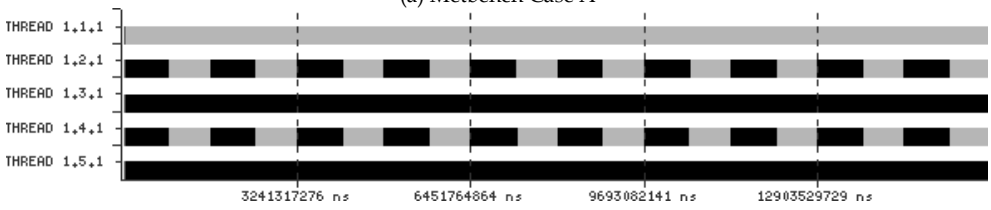
Metbench (Minimum Execution Time Benchmark) is a suite of MPI micro-benchmarks developed at BSC whose structure is representative of the real applications running on MareNostrum. Metbench consists of a *framework* and several *loads*. The framework is composed by a *master* process and several *workers*: each worker executes its assigned load and then waits for all the others to complete their task. The role of the master is to maintain a strict synchronization between the workers: once all the workers have finished their tasks, the master eventually starts another iteration (the number of iterations to perform is a run-time parameter). The master and the workers only exchange data during the initialization phase and use an `mpi\_barrier()` to get synchronized. In the traces shown in this section, the master process corresponds to the first process and is not balanced as it will be always idle, waiting for the conclusion of all worker processes.

One of the goals of Metbench is to allow researchers at BSC to understand the performance and capabilities of a processor or a cluster. In order to do that, we developed several loads, each one stressing a different processor resource (for example, the Floating Point Unit, the L2 cache, the branch predictor, etc) for a given amount of time.

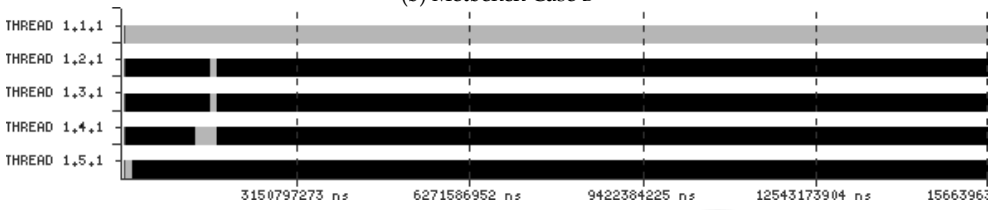
In this experiment we introduce imbalance in the MPI application by assigning to a worker a larger load than the one assigned to the worker on the same core. In this way, the faster worker will spend most of its time waiting for the slower worker to process its load. As we will see in Section 4.2 this scenario is quite common for both standard benchmarks and real applications. Figure 5 shows the effect of the hardware resource allocation on Metbench. Each horizontal line represents the activity of a process and each color represents a different state: dark bars show computing time while grey bars show waiting time. In this example, processes $P1$ (the master), $P2$, and $P3$ are mapped to the first core of the POWER5, while processes $P4$ and $P5$ are mapped to the other core. The x-axis represents time.



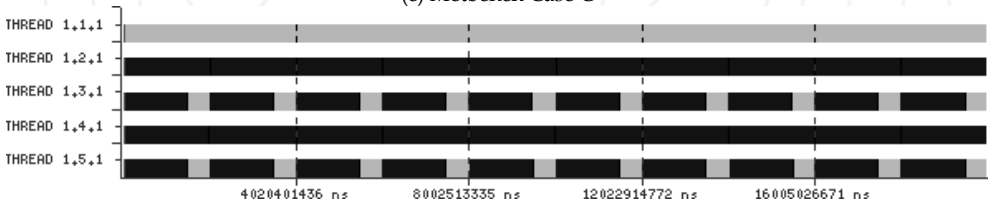
(a) Metbench Case A



(b) Metbench Case B



(c) Metbench Case C



(d) Metbench Case D

Fig. 5. Effect of the hardware thread prioritization on Metbench. Each trace represents only some iterations of the application.

Case A: Figure 5(a) represents our reference case, i.e., the MPI application is running with default priorities (4). As we can see from Figure 5(a) Metbench shows a great imbalance:

more specifically, processes *P*2 and *P*4 spend about 75.6% of their time waiting for processes *P*3 and *P*5 to complete their computing phase.

Case B: Using the software-controlled hardware prioritization, we increased the priority of *P*3 and *P*5 (the most computing intensive processes) up to 6, while the priority of *P*2 and *P*4 are set to 5 (remember that what really matters is the difference between the thread priorities, here *P*2 and *P*4 are running with less priority than in Case A).

Figure 5(b) shows how the imbalance has been reduced, also reducing the total execution time (from 81.64 sec to 76.98 sec, 5.71% of improvement).

Case C: We increased again the amount of hardware resources assigned to *P*3 and *P*5 in order to speed them up.

Indeed, we obtained an even more balanced situation where all the processes compute for (roughly) the same amount of time. The total execution time reduces to 74.90 sec (8.26% of improvement over Case A).

Case D: Next, we increased again the amount of resources given to *P*3 and *P*5. As we can see from Figure 5(d) we reversed the imbalance, i.e., now *P*3 and *P*5 are much faster than *P*2 and *P*4 and spend most of their time waiting. As a result the execution time (95.71 sec) increases.

Test	Proc	Core	% Comp	Priority	Exec. Time
A	P1	1	0.02	4	81.64s
	P2	1	24.32	4	
	P3	1	98.99	4	
	P4	2	24.31	4	
	P5	2	99.99	4	
B	P1	1	0.02	4	76.98s
	P2	1	51.16	5	
	P3	1	99.82	6	
	P4	2	51.18	5	
	P5	2	99.98	6	
C	P1	1	0.03	4	74.90s
	P2	1	98.96	4	
	P3	1	98.56	6	
	P4	2	97.01	4	
	P5	2	98.37	6	
D	P1	1	0.02	4	95.71s
	P2	1	99.87	3	
	P3	1	73.25	6	
	P4	2	99.72	3	
	P5	2	73.25	6	

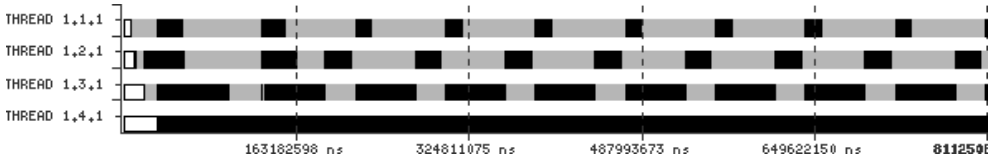
Table 4. Metbench balanced and imbalanced characterization

Case D shows an interesting property of the IBM POWER5 hardware priority mechanism: the hardware thread priority implementation is a powerful tool but the performance of the penalized process can be reduced more than linearly (in fact, exponentially) (Boneti et al. 2008a), thus, *P*2 and *P*4 can become the new bottlenecks.

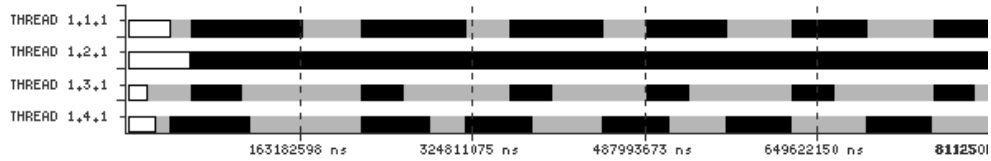
4.2. BT-MZ

Block Tri-diagonal (BT) is one of the NAS Parallel Benchmarks (NPB) suite. BT solves discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions, operating on a structured discretization mesh. BT Multi-Zone (BT-MZ) (Jin and der Wijngaart, 2006) is a variation of the BT benchmark which uses several meshes (named *zones*) for, in realistic applications, a single mesh is not enough to describe a complex domain.

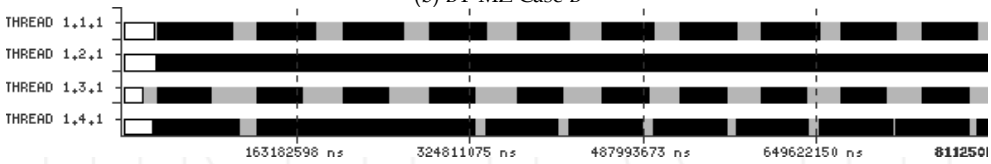
Besides the complexity of the algorithm, BT-MZ shows a behavior very similar to our Metbench benchmark: every process in the MPI application performs some computation on its part of the data set and then exchanges data with its neighbors asynchronously (using `mpi_isend()` and `mpi_irecv()`); after this communication phase (which lasts for a very short time, around 0.10% of the total execution time) each process waits (with a `mpi_waitall()` function) for its neighbors to complete their communication phases. In this way, each process gets synchronized with its neighbors (note that this does not mean that each process gets synchronized with all the other processes). Once a process has exchanged all the data it had to exchange, a new iteration can start and the previous behavior repeats again until the end of the application (in our experiments we used BT-MZ with default values: class A with 200 iterations).



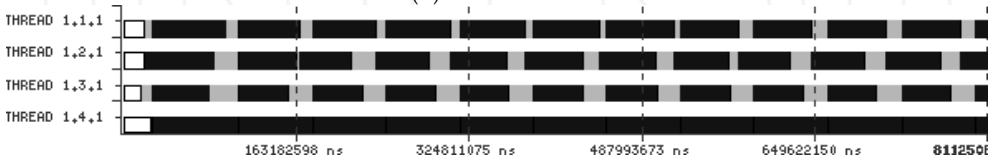
(a) BT-MZ Case A



(b) BT-MZ Case B



(C) BT-MZ Case C



(D) BT-MZ Case D

Fig. 6. Effect of the hardware thread prioritization on BT-MZ. Each trace represents only some iterations of the application. Communication has been removed to increase clearness

Case A: Figure 6(a) shows the BT behavior in the reference case, i.e. when process P_i is assigned to CPU_i and the priority of all the processes is 4. After an initialization phase (white bars at the beginning of the execution of each task), all the processes reach a barrier (synchronization point). From this point on, the real algorithm starts: during every iteration, each process alternate computing phases (black) with synchronization phases (grey).
It is easy to see from Figure 6(a) that BT-MZ shows a great imbalance[5]. The imbalance is caused by the fact that some processes (for example process *P*1) have a small part of the data to work on, while other processes (for example, processes *P*4) have a large amount of data to take care of. It is also clear that process *P*4 is the bottleneck of the application and that speeding up this process will improve overall performance.

Test	Proc	Core	% Comp	Priority	Exec. Time
ST	P1	1	49.33	7	108.32s
	P2	2	99.46	7	
A	P1	1	17.63	4	81.64s
	P2	1	28.91	4	
	P3	2	66.47	4	
	P4	2	99.72	4	
B	P1	1	52.33	3	127.91s
	P2	2	99.64	3	
	P3	2	28.87	6	
	P4	1	46.26	6	
C	P1	1	65.32	4	75.62s
	P2	2	99.68	4	
	P3	2	53.78	6	
	P4	1	85.88	6	
D	P1	1	82.73	4	66.88s
	P2	2	73.68	4	
	P3	2	66.40	5	
	P4	1	99.72	6	

Table 5. BT-MZ balanced and imbalanced characterization

Case B: In order to solve the imbalance introduced by data repartition in BT-MZ, we ran process *P*1 and *P*4 on the same core and assigned more hardware resources to the latter, improving its performance while decreasing *P*1's performance. This mapping seems reasonable, as our goal is to increase the performance of *P*4 (the most computing intensive process) and we know that, with this operation, we will reduce the performance of the process running on the same core with *P*4. We chose *P*1 because it is the process with the shortest computation phase.
In our first attempt to reduce the imbalance we assigned priority 3 to processes *P*1 and *P*2 and priority 6 to processes *P*3 and *P*4. Figure 6(b) shows how the imbalance has been inverted: process *P*1 now takes longer than *P*4 and the new bottleneck is now process *P*2, which is also running with priority 3. As a consequence, the total execution time increases

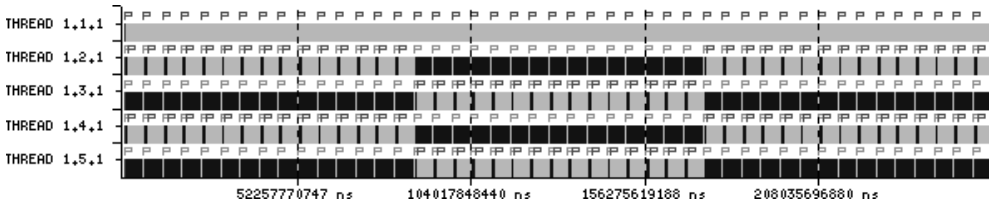
[5]Even if the goal of this chapter is not to show whether SMT processors are useful in HPC or not, the table also shows the ST mode performance (only one process per core) of the application.

(127.91 sec instead of 81.62 sec), which means the new bottleneck runs for much longer than the previous one.

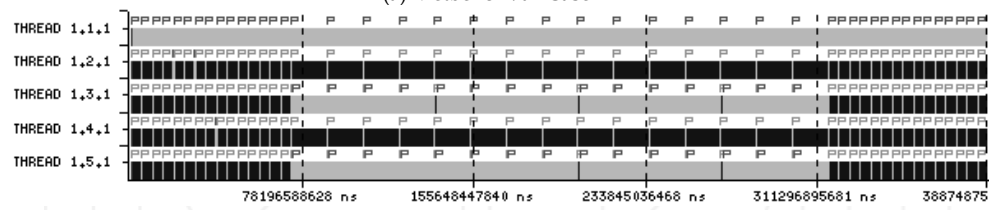
Case C: In order to restore the original relative behavior between process *P*1 and *P*4 we incremented the resources assigned to process *P*1 and *P*2. Figure 6(c) shows that *P*1 now runs for less time than *P*4, as in Case A. In addition, giving more resource to *P*2 (which is again the bottleneck) reduced the total execution time to 75.62 sec, with a 7.37% of improvement with respect to Case A.

Case D: Finally, we can argue that *P*2 and *P*3 execute their operation on a similar amount of data, therefore the amount of resources given to each process should not be as different as for *P*1 and *P*4. In our last test, we still assigned priority 4 to *P*1 and 6 to *P*4, as in the previous case, but we assigned priority 5 to *P*2 and 6 to *P*3, sharing resources between these two processes running on the same core more equally. Figure 6(d) shows that the imbalance has been reduced again with respect to Case C, in fact, now *P*2 and *P*3 compute more or less for the same amount of time. Also the new bottleneck is *P*4, which is much shorter than *P*2 in Case C. Table 5 shows how the total execution time has also been reduced to 66.88 sec, with an 18.08% of improvement over the reference Case A.

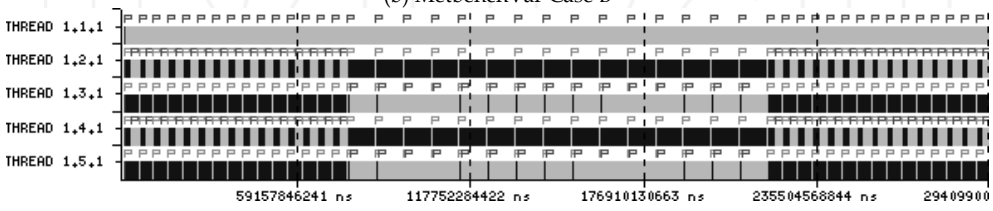
4.3. MetbenchVar



(a) MetbenchVar Case A



(b) MetbenchVar Case B



(c) MetbenchVar Case C

Fig. 7. Effect of the hardware thread prioritization on MetbenchVar

MetbenchVar is a slightly modified version of Metbench where the workers change their behavior after *k* iteration. Figure 7(a) shows the standard execution of MetbenchVar with

k=15: at the beginning *P*2 and *P*4 execute a small load while *P*3 and *P*5 a large load. At the 15th iteration, *P*2 and *P*4 start to execute the large load while *P*3 and *P*5 perform their task on the small load. In this way, we reverse the load imbalance at run time making the application's behavior dynamic. At the 30th iteration, we switch again the behavior of the tasks. Recall that, as it was the case for Metbench (Section 4.1), *P*1 does not perform any job and presents no significant impact on performance, as it only waits for *P*2 to *P*5 to finish their execution.

Figure 7(b) shows how the static prioritization works in this case: the application is perfectly balanced in the first (iterations 1-15) and third period (iteration 31-45) but the imbalance is reversed in the second period (iterations 16-30), as a result, in the second period the application performs worst than in the standard case. Furthermore, for this workload, the negative impact of applying the wrong prioritization is extremely high and, although for two thirds of the cases the benchmark runs with the right priorities (4,6), the performance degradation of running with the wrong priorities is by far more important. Overall, for this program, the static prioritization presents 50% of performance degradation when compared to the standard case of this benchmark.

Figure 7(c) shows that trying to decrease the priority difference between *P*2 and *P*3, and between *P*4 and *P*5 does not improve the baseline either. In this case, when comparing to the standard execution, statically applying a hardware prioritization still degrades performance by 13.20%.

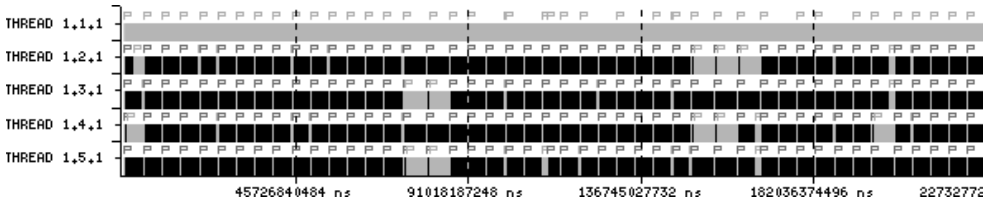


Fig. 8. Effect of the HPCSched on MetbenchVar.

The case where the application presents a dynamic behavior makes a strong motivation for dynamic mechanisms. In fact, dynamic mechanisms proposed in (Boneti et al., 2008b) are able to transparently balance this application and improve its execution time by 12.5%. Figure 8 shows the trace of MetbenchVar when running with HPCSched's uniform prioritization mechanism. The key of the improvement is the ability to change the priorities during the application's execution time, following the changes in its behavior.

Another very interesting point is that, for applications with very variable behavior, using the overall relative computational time (or utilization) of a task can be tricky. For instance, if we refer to the case A in Table 6, we can see that process *P*2 computes for 49.34% of the time, while *P*3 processed for 74.65% of the time. It becomes intuitive that we should always prioritize *P*2. However, let's take a look at the utilization per phase: during the first phase, the utilizations are 24.17%, 100.00%, 24.16%, 99.97%, during the second, they are 100%, 23.65%, 99.94%, 23.65%, finally, the third iteration has the same behavior as the first one. It becomes clear why a constant prioritization is not good, and furthermore, that the overall utilization is not a good indicator of imbalance for this application.

On Case B of Table 6, the measured overall utilization is also misleading. We may believe that the imbalance is not so different from the baseline Case A, however, for initial and final phases the utilizations are: 99.63%, 99.90%, 98.52%, 99.94% and for the middle phase: 99.95%, 4.90%, 99.87%, 4.89%. On the previous cases, as the imbalance was constant, it was not necessary to use per-phase utilization. Clearly, in the case of MetbenchVar, if the utilization is used as a metric, it must be evaluated for each of the phases of the program.

Test	Proc	Core	% Comp	Priority	Exec. Time
A	P1	1	0.01	4	259.79s
	P2	1	49.34	4	
	P3	1	74.65	4	
	P4	2	49.31	4	
	P5	2	76.63	4	
B	P1	1	0.00	4	388.75s
	P2	1	99.43	4	
	P3	1	40.65	6	
	P4	2	99.35	4	
	P5	2	40.64	6	
C	P1	1	0.01	4	294.10s
	P2	1	75.36	4	
	P3	1	56.34	5	
	P4	2	75.32	4	
	P5	2	56.35	5	
HPCSched	P1	1	0.01	-	227.33s
	P2	1	90.11	-	
	P3	1	93.95	-	
	P4	2	89.28	-	
	P5	2	93.75	-	

Table 6. MetbenchVar balanced and imbalanced characterization

4.4. Siesta

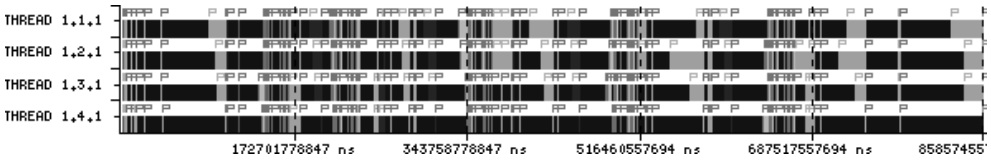
Our last experiment consists of running SIESTA as an example of real application. SIESTA (SIESTA, 2009; Soler et al., 2002) is a method for *ab initio order-N materials simulation*, specifically it is a self-consistent density functional method that uses standard norm-conserving pseudo-potentials and a flexible, numerical linear combination of atomic orbitals basis set, which includes multiple-zeta and polarization orbitals.

The application presents an imbalance caused by both the algorithm and the input set. For this very interesting input set, a nanoparticle of barium titanate, SIESTA behavior is not constant during each iteration, as can be seen in Figure 9(a); this makes our static balancing solution not as good as for the BT-MZ case. Yet, we achieved an improvement of 8.1% of execution time reduction with respect to the reference case (Case A).

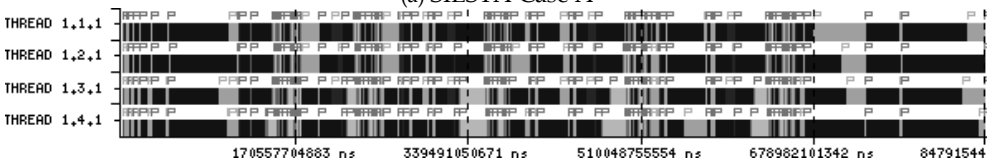
Case A: Like for BT-MZ, Case A is the reference case, i.e., where process P_i is assigned to CPU_i and the priority of all the processes is set to 4. Figure 9(a) shows the trace for this reference case. The program starts with an initialization phase (11.99% of the total time) at the end of which each process in the application must reach a barrier. The initialization phase already presents some little imbalance, which evidences how the input set makes

SIESTA imbalanced. In the internal parts, each process exchanges data only with a subset of the other processes in the application, and then reaches a synchronization point (`WaitAll()`), waiting for all the others to complete their jobs. In the last part, the processes finalize their work (13.41% of the total time): after the last barrier, each process computes its function on its sub-set of data and then ends. A complete execution of the program in this configuration takes 858.57 secs.

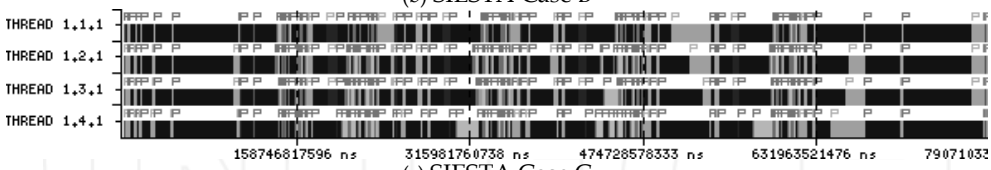
Case B: As we can see from the trace in Figure 9(a) is not easy to understand how to balance the application and whether our balancing approach is worth. However, Table 7 shows some more information about SIESTA (hard to retrieve from the trace): processes *P*1 and *P*2 spend a considerable amount of time waiting for *P*3 and *P*4 to reach the barrier. Thus, the first hint would be to put *P*1 and *P*3 on one core and *P*2 and *P*4 on the other and then play with priority. We tried this case but then we realized that *P*2 and *P*3 have almost the same amount of data to work on. Thus, in Case B we put *P*2 and *P*3 on the first core and *P*1 and *P*4 on the second one and increased the priority of *P*3 and *P*4 to 5. In this case we achieved a little improvement of 1.24% (the total execution time is 847.91 sec). Figure 9(b) shows that, in this new configuration, *P*2 is the new bottleneck of the finalization part.



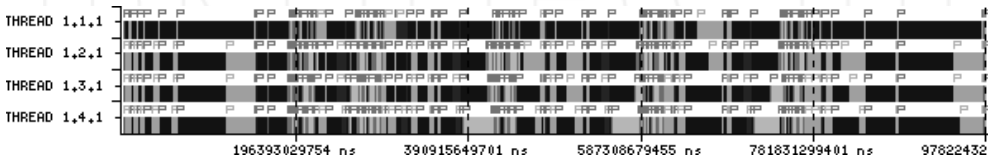
(a) SIESTA Case A



(b) SIESTA Case B



(c) SIESTA Case C



(d) SIESTA Case D

Fig. 9. Effect of the hardware thread prioritization on SIESTA

Case C: In the previous case we obtained a little improvement, still the application results quite imbalanced. We realized that, since *P*2 and *P*3 work, more or less, on the same amount of data, using a different priority for these two processes may introduce even more imbalance. Figure 9(b) shows that, indeed, this is the case. In Case C we restored the original relative behavior between process *P*2 and *P*3 setting both their priority to 4 (i.e., the difference is 0). Figure 9(c) shows how the application is now more balanced. For example, looking at the initialization and the finalization part, it is possible to see that the processes are much more balanced than in Case A and Case B. In fact, re-balancing SIESTA reduces the total execution time to 798.20 sec, an improvement of 8.1% with respect to the reference case.

Case D: Following the same idea of the previous case (i.e., leave *P*2 and *P*3 with the same priority and play with *P*1 and *P*4), we increased the amount of resources assigned to *P*4, penalizing *P*1. Figure 9(d) shows how we reverse the imbalance: SIESTA is again imbalanced, though in a different way than in the reference case. In Case D, *P*1 (the process with less hardware resources) is the bottleneck (in the initialization, finalization and most of the internal phases) and the total execution time increases to 976.35 sec, with a loss of 13.72%.

Test	Proc	Core	% Comp	Priority	Exec. Time
ST	P1	1	81.79	7	1236.05s
	P2	2	93.72	7	
A	P1	1	75.94	4	858.57s
	P2	1	75.24	4	
	P3	2	82.08	4	
	P4	2	93.47	4	
B	P1	2	79.57	4	847.91s
	P2	1	87.06	4	
	P3	1	72.04	5	
	P4	2	77.73	5	
C	P1	2	83.04	4	789.20s
	P2	1	79.66	4	
	P3	1	80.78	4	
	P4	2	78.74	5	
D	P1	2	90.76	4	976.35s
	P2	1	65.74	4	
	P3	1	68.08	4	
	P4	2	63.95	6	

Table 7. SIESTA balanced and imbalanced characterization

BT-MZ and SIESTA are two cases of non-balanced HPC applications, though their imbalance is quite different. BT-MZ executes several iterations, all of them similar from the execution time, CPU utilization and imbalance point of view. SIESTA also executes several iterations, but each iteration is not necessarily similar to the previous or the next one. In particular, the process that computes the most is not the same across all the iterations. For example, in the *i*-th iteration *P*1 could be the bottleneck while in the *(i+1)*-th the most computing process could be *P*4. This behavior suggests that a good balancing mechanism

would prioritize *P*1 in the *i*-th and *P*4 in the *i+1*-th iteration. Our static approach does not allow us to play in this way as we assign the priority at the beginning of the execution and never change them during the execution. We argue that a dynamic mechanism is required to correctly set priorities for applications that change their behavior throughout their execution.

5. Related work

Traditional solutions to attack the problem of load imbalance in HPC applications typically use dynamic data re-distribution. For OpenMP applications load balancing may be performed using some of the existing loop scheduling algorithms that assigns iterations to software threads dynamically (Aygade et al., 2003). MPI applications are much more complex because data communications are defined explicitly in the algorithm by programmers. Static approaches for distributing data using sophisticated tools have been proposed: for example, METIS (METIS, 2009) analyzes data and tries to find the best data distribution. These approaches achieve good performance results but have the drawback that they must be repeated for each input data set and architecture. Dynamic approaches have also been proposed in the literature (Schloegel et al., 2000) and (Walshaw and Cross, 2002). The authors try to solve the load-balancing problem of irregular applications by proposing mesh repartitioning algorithms and evaluating the convenience of repartitioning the mesh or adjusting it.

Processing re-distribution is another approach that consists of assigning more resources to those processes that compute for longer. In the case of OpenMP, this can be useful when using nested parallelism, assigning more software threads to those groups with high load (Duran et al., 2005). The case of MPI is much more complex because the number of processes is statically determined when starting the job (in case of malleable jobs), or when compiling the application (in case of rigid jobs). This problem has been also approached through hybrid programming models, combining MPI and OpenMP. Huang and Tafti (Huang and Tafti, 1999) balance 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 software 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 oppose their own ones.

Concerning the use of SMT architectures for HPC applications, several studies (Curtis-Maury and Wang, 2005; Celebioglu et al, 2004) show that Hyper-Threading (the SMT implementation of Intel Processors) improve performance for some workloads. However, for other workloads there are many conflicts when accessing shared resources, creating a negative impact on the performance. In (Curtis-Maury and Wang, 2005) the study is performed for MPI applications while in (Celebioglu et al, 2004) the study focuses in OpenMP applications. In (Celebioglu et al, 2004) the authors propose a mechanism that, given a multiprocessor machine with Hyper-Threading processors, dynamically deactivates the Hyper-Threading in some processors in order to improve the performance of the workload under study.

The solution presented in this chapter is orthogonal to both the software thread re-distribution and the dynamically activating Hyper-Threading. Let's assume that we want to run an HPC application on a cluster having several IBM POWER5 processors. The proposal in (Celebioglu et al, 2004) can be used to determine in which cores SMT has to be

deactivated. For those cores with the SMT feature active, hardware prioritization can be used to select the appropriate hardware priority to reduce imbalance. Compared with software thread-distribution, hardware prioritization can be seen as low level solution for load balancing.

6. Summary

In this chapter we present the problem of imbalance in HPC applications. In fact, some applications show an imbalanced behavior, i.e., some processes require more time to complete their computing phase while all the other processes are waiting at some synchronization point and cannot move forward. We show the reasons for imbalance and some examples where the application is imbalanced because of data distribution (NAS BT-MZ), or because of the application's input (SIESTA).

We also present the idea of using software controlled allocation of the hardware resources to perform load-balance of HPC applications. Experimental cases show how using a modified Linux kernel to control a processor capable to dynamically assign processor resources to running contexts (the IBM POWER5 in this case), reduces the application imbalance and, therefore, improves overall performance. The experiments performed show an improvement up to 18% for a widely used BT-MZ benchmark and up to 8.1% for a real application (SIESTA). These results do not require putting the burden of balancing the application on the programmer and are independent from the used programming model. In addition, we show cases where the application presents variable behavior. We discuss on why it motivates the use of automatic load-balancers based on software-controlled hardware resource allocation.

From the case studies presented, it is possible to conclude that the hardware resource allocation in multithreaded processors is an important tool that allows to load-balance HPC applications, improving significantly their performance.

7. References

Alpert, D. (2003). Will microprocessor become simpler? *Microprocessor Report*.

Ayguade, E., Blainey, B., Duran, A., Labarta, J., Martinez, F., Martorell, X., and Silvera, R. (2003). Is the schedule clause really necessary in openMP? *In Proceedings of the 4th International Workshop on OpenMP Applications and Tools (WOMPAT'03)*, volume 2716 of Lecture Notes in Computer Science (LNCS), pages 147–159, Toronto, Canada. Springer-Verlag (New York).

Boneti, C., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Cher, C.-Y., and Valero, M. (2008a). Software-controlled priority characterization of POWER5 processor. *In Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*, Beijing. ACM SIGARCH.

Boneti, C., Gioiosa, R., Cazorla, F. J., and Valero, M. (2008b). A dynamic scheduler for balancing HPC applications. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, Austin, TX. IEEE/ACM.

Bossen, D. C., Tendler, J. M., and Reick, K. (2002). Power4 system design for high reliability. *IEEE Micro*, 22(2):16–24.

Cazorla, F. J., Knijnenburg, P. M. W., Sakellariou, R., Fernandez, E., Ramirez, A., and Valero, M. (2006). Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785– 799.

Celebioglu, O., Saify, A., Leng, T., Hsieh, J., Mashayekhi, V., and Rooholamini, R. (2004). The performance impact of computational efficiency on HPC clusters with hyper-threading technology. *In Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEOPDS'04)*, Santa Fe, New Mexico, USA. IEEE Computer Society (Los Alamitos, CA).

Curtis-Maury, M. and Wang, T. (2005). Integrating multiple forms of multithreaded execution on multi-SMT systems: A study with scientific applications. *In Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 199–209, Torino, Italy. IEEE Computer Society.

Duran, A., Gonzalez, M., Corbalan, J., Martorell, X., Ayguade, E., Labarta, J., and Silvera, R. (2005). Automatic thread distribution for nested parallelism in OpenMP. *In Proceedings of the 19th ACM International Conference on Supercomputing (ICS'05)*, pages 121–130, Cambridge, Massachusetts, USA.

Gibbs, B., Atyam, B., Berres, F., Blanchard, B., Castillo, L., Coelho, P., Guerin, N., Liu, L., Maciel, C. D., Sosa, C., and Thirumalai, R. (2005). *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook. IBM, International Technical Support Organization, Austin, TX, USA.

Gioiosa, R., Petrini, F., Davis, K., and Lebaillif-Delamare, F. (2004). Analysis of system overhead on parallel computers. *In Proceedings of the 4th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'04),* pages 387–390, Rome, Italy.

Huang, W. and Tafti, D. (1999). A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. *In Proceedings of the Parallel Computational Fluid Dynamics (PCFD'99)*.

IBM (2005a). *User Instruction Set Architecture version 2.02*. Number 1 in PowerPC Architecture books.

IBM (2005b). *PowerPC Operating Environment Architecture version 2.02*. Number 3 in PowerPC Architecture books.

IBM (2005c). *PowerPC Virtual Environment Architecture version 2.02*. Number 2 in PowerPC Architecture books.

IBM (2008). *Cell broadband engine programming handbook v1.11*.

IBM, Sony, and Toshiba (2006). *Cell broadband engine architecture v1.01*.

Jin, H. and der Wijngaart, R. F. V. (2006). Performance characteristics of the multi-zone NAS parallel benchmarks. *Journal of Parallel and Distributed Computing*, 66(5):674– 685.

Kalla, R. N., Sinharoy, B., and Tendler, J. M. (2003). SMT implementation in POWER5. *In Hot Chips*, volume 15.

Kalla, R. N., Sinharoy, B., and Tendler, J. M. (2004). IBM POWER5 Chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47.

Labarta, J., Girona, S., Pillet, V., Cortes, T., and Gregoris, L. (1996). DiP: A parallel program development environment. *In Proceedings of the 2nd International Conference on Parallel Processing (Euro-Par'96)*, volume II of Lecture Notes in Computer Science, pages 665–674, Lyon, France. Springer.

Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W., Schwarz, E. M., and Vaden, M. T. (2007). *IBM POWER6 microarchitecture. IBM Journal of Research and Development*, 51(6):639– 662.

Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. (2002). *Hyper-threading technology architecture and microarchitecture*. Intel Technology Journal, 6(1):4–15.

Metis - family of multilevel partitioning algorithms (2009). http://glaros.dtc.umn.edu/gkhome/views/metis.

Moreto, M., Cazorla, F. J., Ramirez, A., and Valero, M. (2008). MLP-aware dynamic cache partitioning. *In Proceedins of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'08)*, volume 4917 of Lecture Notes in Computer Science, pages 337–352, Goteborg, Sweden. Springer.

NASA. NAS parallel benchmarks (2009). http://www.nas.nasa.gov/Resources/Software/npb.html

Petrini, F., Kerbyson, D. J., and Pakin, S. (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8, 192 processors of ASCI Q. *In International Conference for High Perfromance Computing, Networking, Storage and Analysis (SC'03)*, page 55. IEEE/ACM SIGARCH.

Qureshi, M. K. and Patt, Y. N. (2006). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *In Proceedings of the 39th International Symposium on Microar- chitecture (MICRO'06)*, pages 423–432. IEEE Computer Society.

Schloegel, K., Karypis, G., and Kumar, V. (2000). Parallel multilevel algorithms for multi-constraint graph partitioning. *In Proceedings of the 6th International Conference on Parallel Processing (Euro-Par'00)*, volume 1900 of LNCS, pages 296–310. Springer-Verlag, Berlin.

Serrano, M. J., Wood, R. C., and Nemirovsky, M. (1993). A study on multistreamed superscalar processors. *Technical Report 93-05*, University of California Santa Barbara.

SIESTA: A linear-scaling density-functional method (2009). http://www.uam.es/siesta/

Sinharoy, B., Kalla, R. N., Tendler, J. M., Eickemeyer, R. J., and Joyner, J. B. (2005). POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505– 521.

Soler, J. M., Artacho, E., Gale, J. D., Garcia, A., Junquera, J., Ordejon, P., and Sanchez-Portal, D. (2002). The SIESTA method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14(11).

The TOP500 Supercomputing Sites (2007). http://www.top500.org/lists/2007/06.

Tsafrir, D., Etsion, Y., Feitelson, D. G., and Kirkpatrick, S. (2005). System noise, os clock ticks, and fine- grained parallel applications. *In Proceedings of the 19th International Conference on Supercomputing (ICS '05)*, pages 303–312, New York, NY, USA. ACM Press.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 392–403.

Walshaw, C. H. and Cross, M. (2002). 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

Parallel and Distributed Computing

Edited by Alberto Ros

ISBN 978-953-307-057-5
Hard cover, 290 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

The 14 chapters presented in this book cover a wide variety of representative works ranging from hardware design to application development. Particularly, the topics that are addressed are programmable and reconfigurable devices and systems, dependability of GPUs (General Purpose Units), network topologies, cache coherence protocols, resource allocation, scheduling algorithms, peertopeer networks, largescale network simulation, and parallel routines and algorithms. In this way, the articles included in this book constitute an excellent reference for engineers and researchers who have particular interests in each of these topics in parallel and distributed computing.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821