Task Packing: efficient task scheduling in unbalanced parallel programs to maximize CPU utilization

Gladys Utrera[a], Montse Farreras[b], Jordi Fornes[a]

[a] *Computer Architecture Department*
Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3
08034 Barcelona, Catalunya, Spain
[b] *Battcock Centre for Experimental Astrophysics*
Cambridge University
JJ Thomson Avenue
Cambridge CB3 0HE, UK

Abstract

Load imbalance in parallel systems can be generated by external factors to the currently running applications like operating system noise or the underlying hardware like a heterogeneous cluster. HPC applications working on irregular data structures can also have difficulties to balance their computations across the parallel tasks. In this article we extend, improve and evaluate more deeply the *Task Packing* mechanism proposed in a previous work. The main idea of the mechanism is to concentrate the idle cycles of unbalanced applications in such a way that one or more CPUs are freed from execution. To achieve this, CPUs are stressed with just useful work of the parallel application tasks, provided performance is not degraded. The packing is solved by an algorithm based on the Knapsack problem, in a minimum number of CPUs and using oversubscription. We design and implement a more efficient version of such mechanism. To that end, we perform the Task Packing "in place", taking advantage of idle cycles generated at synchronization points of unbalanced applications. Evaluations are carried out on a heterogeneous platform using FT and miniFE benchmarks. Results showed that our proposal generates low overhead. In addition the amount of freed CPUs are related to a load imbalance metric which can be used as a prediction for it.

Email addresses: gutrera@ac.upc.edu (Gladys Utrera), mf582@cam.ac.uk (Montse Farreras), jfornes@ac.upc.edu (Jordi Fornes)

Preprint submitted to Journal of Parallel and Distributed Computing *9th July 2019*

Keywords: MPI, HPC, oversubscription, load balancing, Knapsack algorithm.

---

1. Introduction

In parallel computing, load imbalance is a well known source of efficiency loss. Not only load imbalance results in the application losing performance but also prevents an efficient use of the High Performance Computing (HPC) system as a whole, wasting CPU cycles and ultimately wasting energy. The problem of load imbalance is getting ever more relevant nowadays because of the growth in size and in energy consumption of the current HPC systems.

Despite load imbalance has received a decent amount of attention since the beginning of parallel programming there is still not a standard solution, which is not surprising due to the complexity of the problem.

Many HPC applications, ranging from models of the physical world to web search or graph clustering, are composed of irregular data structures or sparse data, which leads to load imbalance in both computation and communication. Programmers tune their parallel codes by hand to balance them, redistributing the data or even including some load-balancing code alongside their application code. This tunning is tedious and cumbersome and slows down productivity.

Moreover, even a well-balanced parallel application will show imbalance if running on an heterogeneous platform.

A heterogeneous platform is a platform that is not homogeneous and homogeneous platforms do not exist (or they have a short life anyway). Assuming that...

> A homogeneous platform used for parallel and distributed computing consists of a system where all its processors are identical and are connected in a network with identical latency and bandwidth among them. Besides, identical operating system, runtimes and compilers are used in creating and running the programs.

This definition makes almost impossible to create and keep homogeneous an HPC cluster. For example, a cluster that fits with the definition at the very beginning will at some point need some nodes to be repaired. Should the system manager change the same piece in every single node just to keep within homogeneity? In practice, the homogeneity is broken sooner or later.

According to a study by Dongarra et al. [1], we can neatly divide heterogeneous platforms into five groups: (a) Vendor-designed platforms, such as those based on coprocessors. We can further sub-divide this group based on the vendor architecture in Nvidia graphical processing units (GPUs) [2],

Intel Xeon Phi [3], ClearSpeed CSX700 [4] or Toshiba, IBM, Sony Cell Broadband Engine [5]; (b) Heterogeneous cluster, that is, a homogeneous cluster, design for parallel computing, where some characteristics have been relaxed (not identical processors or heterogeneous network topology or non batch scheduler, allowing several users running their applications in the same set of processors); (c) Local network of computers (LNC) are very similar to a heterogeneous cluster, but here the nodes are general-purpose computers; (d) Global network of computers (GNC) are like LNC, but nodes can be geographically distributed, and (e) grid-based systems are GNC where users can login from everywhere and find the same user-friendly environment. To this classification, we should add the cloud-based systems, technologically successors of the grid computing, however cloud systems rely on virtualization technologies rather than on batch systems and are much more market oriented. They are the next-generation data centers [6].

Programming a parallel application to run efficiently on an heterogeneous system requires specific knowledge and expertise making it a very difficult task for scientists (and programmers too) as system software evolves very slowly [7]. Nevertheless, heterogeneous systems are widespread and gaining importance as they are seen by some as a way to mitigate the slowdown of Moore's Law [8]. A lot of research targets heterogeneous systems so we expect the evolution of clusters will gravitate towards heterogeneous scenarios and we foresee load imbalance becoming even a bigger problem. Even well-balanced applications will show imbalance, hand tunning them to balance the work will be not only cumbersome but also not portable, being both application and platform dependent.

Our proposal is to compute an intelligent "packing" of tasks that keeps fewer CPUs more busy by making oversubscription of the CPUs, provided the performance of the program is not degraded or degraded up to a prefixed limit (e.g. no more than 20% of the total time execution). In this way we are balancing the number of CPU cycles across CPUs, or in other words we are balancing useful work across CPUs. This is a novel view of the load balancing definition: instead of reducing the total execution time using the same number of CPUs, we claim to reduce the number of CPUs keeping the same total execution time. As a consequence, we will have free CPUs that can be used for other purposes or for preserving energy consumption. Potential benefits of having CPU(s) of a node freed from execution during a parallel session has already been pointed out by other authors:

(i) Conservation of energy: most modern CPUs are capable of lowering their frequency (and even their voltage) during idle states for reducing its power dissipation and energy consumption [9], [10]. Our proposed scheduling mechanism helps to foster this by providing completely freed CPUs. At a first glance, throttling all the CPUs that are idly waiting due to load imbalance should yield the same power savings as throttling the freed CPU. However, in practice tasks may wait, for example, on a spin lock (depending on the communication system) and this busy waiting naturally consumes energy. Moreover, as lowering the voltage or even shutting down a whole CPU temporally requires much more preparation time (but yields better energy saving than just frequency throttling), completely freed CPUs would suit much better in these cases than sporadically idle ones.

(ii) The executing of alien tasks from other applications. Freeing CPUs would enable the co-scheduling of tasks stemming from other applications onto the same compute node, which would help to improve system utilization. Job scheduling techniques such as Backfilling, could be applied [11], [12]. This task co-scheduling should happen at job level granularity and it would require our task scheduling mechanism to interact with the job manager to notify of freed CPUs. The job manager could then co-schedule task from other applications, taking into account their resource demands. Most job managers can be configured to manage resources on a fine-grain basis to enable this co-scheduling (i.e. SLURM [13] Consumable Resource Allocation Plugin). The implementation of this co-scheduling is out of the scope of this paper.

(iii) Reduction of operating system (OS) noise. The freed CPUs could be used to execute OS work, which in turn may reduce the OS noise and eventually, as a side effect could, contribute to reduce the load imbalance of the application [14], [15], [16].

This work extends largely the proof of concept of an idea already published [17]. The main contributions of this paper are:

- We design, implement and evaluate a scheduler mechanism for parallel programs, the Task Packing algorithm, to reallocate tasks from a parallel program, within each node based on oversubscription, i.e., running an application with a number of OS level tasks larger than

the number of assigned CPUs. The reallocation is computed using a particular case of the well-known Knapsack algorithm.

The strategy is evaluated using FT from the NAS Parallel Benchmarks [18] and the miniFE Benchmark, a miniapplication from the Mantevo project [19]. The miniFE miniapplication, is similar to HPCG [20]. Like HPCG, miniFE is intended to be the best approximation to an unstructured implicit finite element or finite volume application, but in 8,000 lines or less. The HPCG (High Performance Conjugate Gradients) Benchmark project is an effort to create a more relevant metric for ranking HPC systems than the High Performance LINPACK (HPL) benchmark [21], which is currently used by the Top500 benchmark. The evaluation our our proposal is done on a heterogeneous platform, as we believe the growth in number and complexity of heterogeneous platforms increase the relevance of our work, however the study is equally valid for unbalanced applications running on homogeneous platforms.

- We avoid the overhead that our scheduler could produce, allocating the scheduler in the task that arrives first at the synchronization point at each node. In this way, that task which otherwise would have idle cycles while waiting for the other tasks to arrive (imbalanced behavior), computes the CPU allocation of the tasks at its node using the information gathered in previous iterations.

- We propose to use a load imbalance metric that serves to predict the benefit obtained, in terms of freed CPUs, after applying the Task Packing mechanism.

- We demonstrate that the Task Packing algorithm can free CPUs without performance degradation for our target applications.

We address to the imbalance intrinsic to the application or the architecture. These two causes of imbalance have predictable repetitive unbalanced behavior on the application, which is an important property that our work relies on.

The whole mechanism is transparent to the programmer and the user. To benefit from the Task Packing mechanism, the application does not have to be modified nor recompiled.

The rest of the paper is organized in the following way: Section II presents the related work; Section III describes the motivation, design and implement-

ation of our proposal; Section IV is dedicated to discuss the applicability of the proposal; Section V presents the experimental results. Finally in Section VI the conclusions and future work are presented.

2. Related work

Oversubscription has been demonstrated to be useful in a wide variety of situations. As reported by Iancu et al. [22], in their work, they evaluate the impact of executing MPI, UPC and OpenMP applications with task oversubscription and show that they could improve system throughput by up to 27% when applications share all the cores and are executed with multiple tasks per core. In the HPX-5 implementation of the of the ParalleX execution model [23] was common to assign more than one domain to one core. Oversubscription was used there to enable computation to overlap with communication effectively hiding network latency [24]. In a previous work, Utrera et al. [25], proposed a mechanism, the Load Balancing Detector (LDB), to classify applications dynamically, without any previous knowledge of it, depending on their balance degree and apply the appropriate process queue type to each job. LDB demonstrated to work especially well for the imbalanced jobs.

The knapsack problem has a long tradition of applications in cryptography [26, 27], production planning [28] and scheduling [29, 30] between others. In addition, there are implementation proposals to solve it parallel and efficiently even for large problems [31].

Some authors have given up to the MPI paradigm. They considered that when dealing with system noise, heterogeneity of processing units or variable completion times, you need something different than MPI. Recently, some authors are purposing programming paradigms based on task-based graph concept. One of the most promising is PaRSEC [32], the Parallel Runtime Scheduling and Execution Controller is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Although its goals are similar to ours, it has the vocation of replacing MPI, thus it has to cover a broad panorama. PaRSEC maps the tasks, detects data dependencies and schedules the tasks to achieve the maximum parallelism. Besides PaRSEC, other authors are exploring similar ideas based on data flow graphs [33, 34].

Years ago, a stochastic model that described the dynamics of an ant colony appeared [35]. It was the trigger for a whole family of algorithms dealing with well-known hard problems to appear in the scene. They were inspired in nature and implemented a metaphor about ants and their way of finding the correct path and broadcast that information via pheromones.

Since the first ACO (Ant Optimization Colony) algorithm [36], many

significant research results have been obtained. Soon this population-oriented search, that was successfully applied to NP-hard combinatorial optimization problems, attracted the attention of parallel programmers. Sequential ACO algorithms became ideal targets to speed up. First using MPI (Message Passing Interface), and more recently with Nvidia CUDA, a lot of effort has been made on them. As a consequence, nowadays is a perfect guinea pig to test runtime optimization ideas on both homogeneous and heterogeneous platforms. For a overview of recent research on the development of high performing algorithmic variants of ACO see [37].

There is a branch of ACO research that is related with our work. When the goal is balancing the workload in a heterogeneous environment running ACO implementations. Specially interesting is the work of [38]. In this article, authors presented a parallelization strategy for massive and heterogeneous parallel systems, using ACO as a case study and taking into account not only time, but also power consumption and accuracy.

Several load balancing strategies were evaluated also in [39]. In this case, authors explored the collaboration between OpenMP and Nvidia GPUs in a heterogeneous cluster using rCUDA[40] as a framework.

There are many works in the literature about load balancing in MPI jobs. In general they propose techniques that imply modifications in the code to balance at runtime the data between the different tasks of the application [41], or work just on shared memory architectures [42]. Others have proved to have acceptable performance and rely on checkpoint/restart-based scheme to work on distributed architectures [43]. Dinan et al. [41] compare work stealing and work sharing when implemented on top of the MPI interface for message passing by using the unbalanced tree-search benchmark. They find that both algorithms perform quite well but depend on parameters like system load, job scheduling and pre-emption policies. They extended their implementation to a cluster with distributed memory. Load imbalance deserves special attention, in particular the impact on real applications. Bohme et al. [44] analyses this kind of applications reporting that 12.5% of the time is spent in wait states, and that 70% of it comes from propagation of the wait state. Other alternatives like the one proposed by Bonetti et al. [45] propose changes in the OS to balance MPI applications through smart hardware resource allocation based on a prioritization mechanism on multithreaded processors. They also address to HPC iterative applications and apply oversubscription as part of the balancing mechanism. The load balancing strategy just works within a single node.

About how to take advantage of this unbalanced behavior there is a recent work [46], where the authors try to get benefit from the already unbalanced generated by the global synchronization. The work analyses the arrival pattern to develop an imbalance-tolerant hierarchical algorithm. They study collective operations such as the reduction and broadcast algorithm, and propose a kind of dynamic leader selection to select a leader node at very invocation of a collective. They support their supremacy in efficiency as they are the unique that take into account the hierarchical nature of the current systems. Intranode communication has to be taken into account in terms of latency as it has a different behavior as the internode communication latency.

Our work brings the opportunity to idle CPUs in a HPC cluster. The power benefits of lowering voltage levels to idle CPUs in a cluster have been already studied [47]. However, having a free CPU or core for different purposes is a traditional wish of many ideas to enable highly efficient message passing on many-core architectures. It was at design's foundations of the BlueGene/L supercomputers, where one of the two cores of every single compute-node could be used just for communication [48]. More recently, a technique for optimizing the MPI one-sided communication, called Casper, was based in the assumption that they could keep aside a small number of CPUs to handle asynchronous communication progress [49].

In the area of distributed exascale computing systems (DECS), a recent work, split in two papers, shows very promising results[50], [51]. Authors center their research in the load balancing problem for a exascale computing scenario. They are interested in a dynamic model for the system load management.They proposed a distributed model to estimate each node available capacity and needs. Authors introduce the concept of compensating factor (that include the communications delay and load transfer costs) in order to to calculate the optimum amount of load that should be transferred among nodes. Besides, the authors explore the possibility of an intelligent system, able to apply different algorithms for load balancing in execution time, based in the previous gain knowledge.

Our work use the base idea of packing tasks using oversubscription and doing the scheduling by applying the knapsack algorithm from a previous work [17]. That work was a proof of concept of the idea, and the evaluation was very simple. We extend that work and differ from it in the following ways:

- We make a very different and more efficient implementation of the

whole mechanism. We do not rely on shared memory in the local node, all the information needed to make the scheduling decisions is gathered through the message passing system making the mechanism more portable.

- In addition, and this makes the difference in performance, we execute the scheduling algorithm "in place", that is, one of the tasks from the parallel application performs the Task Packing algorithm, instead of having an extra CPU dedicated to execute just the Task Packing algorithm. In this work, the first task that arrives to the synchronization point at each node becomes the "local master" and performs the Task Packing algorithm. In this way the scheduling does not interfere with normal execution as it uses cycles eventually dedicated for synchronization. By executing inplace the Task Packing algorithm the application is able to take advantage of all the assigned CPUs.

- We make a deeper evaluation of the mechanism. To that end we use two well-known application benchmarks (FT and miniFE). In the previous work, the evaluation was done using an artificial benchmark: repeated executions of an implementation of the Mandelbrot algorithm.

- We consider as the root cause of load imbalance, the heterogeneity of the underlying platform rather than the application itself.

- Finally we propose to use a load imbalance metric that serves to predict the benefit obtained, in terms of freed CPUs, after applying the Task Packing mechanism.

3. Our approach: Task Packing

.

In this section we discuss objectives, requirements, design and implementation of our proposal.

The general idea is to improve load balancing by means of concentrating idle cycles of CPUs in such a way that one or more CPUs are freed. To achieve that we stress the CPUs with just useful work of the parallel application tasks, provided application performance is not degraded. We are not trying to improve the application performance, but we aim to keep the same execution time.

Our final objective is to improve the throughput of the system.

3.1. Targeted scenario

There are several reasons behind an application showing load imbalance. For the purpose of this work we classify them in three categories: (i) Unbalanced application; (ii) Heterogeneous platform; and (iii) Random imbalances.

Categories (i) and (ii) motivate our work and have already been introduced in section 1. In the (iii) category we include load imbalance due to factors external to the application which may happen occasionally (like operating system noise or network traffic congestion). In (iii) the imbalance happens randomly and its behavior is difficult to predict. On the contrary, both in (i) and (ii) the imbalance is intrinsic to either the application (the former) or the architecture (the later). These two causes of imbalance (aka *permanent* imbalance) we expect they will lead to a predictable repetitive unbalanced behavior on the application, which is an important property that our work relies on.

Concerning the application, tasks of parallel applications usually interact among them to exchange data and synchronize to progress execution. Those synchronization points usually happen at the end of each iteration and are the place where the imbalance becomes noticeable, as all the tasks have to wait for the slowest one (see Listing 1 and figure 1).

Listing 1: Typical SPMD parallel pseudocode.

```
1 Initialize;
2 while (end_condition){
3         compute;
4         synchronize;
5 }
6 Finalize;
```

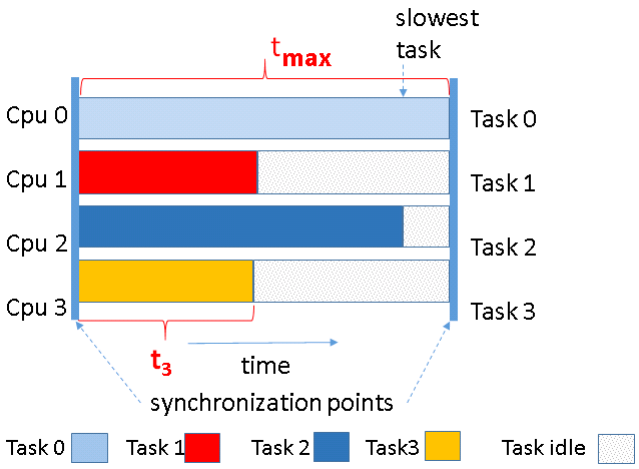

Figure 1: Unbalanced parallel task execution between two synchronization points.

Figure 1 shows an example of execution of tasks from a parallel unbalanced application. There are four tasks: Task 0, Task 1, Task 2 and Task 3 running on CPU 0, CPU 1, CPU 2 and CPU 3 respectively. The drawing shows the execution between two synchronization points, where Task 0 is the slowest task. The light grey color represents the period of time where the task is blocked at the synchronization point (idle CPU cycles) waiting for the slowest task to arrive, others colors like light blue, red, blue and yellow represent useful work of Task 0, Task 1, Task 2 and Task 4 respectively.

In our example in figure 1, if we could use CPU 1 to run Task 3 as well, CPU 3 would be free. Our Task Packing can do this provided that the unbalanced parallel application:

- shows a predictable repetitive unbalanced behavior. So we target *permanent* imbalance (Categories (i) and (ii) above); and
- follows an iterative pattern of computation phase followed by a global synchronization (Listing 1).

3.2. Task Packing mechanism design issues

We propose to use a scheduling strategy to reallocate tasks in CPUs in such a way that k tasks, each one with duration t_i and being t_{max} the largest task duration, can be bounded to a CPU if they satisfy the condition:

$$\sum_{i=0}^{k} t_i \leq t_{max} \quad (1)$$

This task "packing" ensures on one hand that a given CPU only has useful cycles, and on the other hand, that idle cycles are concentrated so that some CPUs can be freed from executing the application, therefore minimizing the number of CPUs used by the application.

The task reallocation solution for the packing is carried out by the Subset Sum algorithm, a particular case of the Knapsack problem [52]. The problem is defined as follows: given a set of integers, the algorithm tries to find a non-empty subset whose sum is a given number. The given number is in our case the elapsed time between iterations t_{max} (excluding synchronization time). Taking this number as an upper bound, we pack the tasks taking into account only their useful time t_i (time to arrive to the synchronization point).

The scheduling algorithm runs "in place", that is, one of the tasks from the parallel application performs the Task Packing algorithm, instead of having an extra CPU dedicated to execute just the Task Packing algorithm. In this work, the first task that arrives to the synchronization point at each node becomes the "local master" and performs the Task Packing algorithm. In this way the scheduling does not interfere with normal execution as it uses cycles eventually dedicated for synchronization. By executing in place the Task Packing algorithm the application is able to take advantage of all the assigned CPUs.

We call our implementation variation: the *Task Packing algorithm*.

The Subset Sum problem is NP-complete, so the solution that can be obtained in polynomial time may not be the optimal one. But we do not need the optimal solution to be able to apply Task Packing and get the benefits. In section 3.3 the algorithm is explained in more detail.

The Task Packing algorithm is computed at runtime and in a transparent way to the programmer. All the functionality of our Task Packing mechanism is interposed between the application and the parallel programming model runtime.

In this work we have chosen pure message passing library MPI (Message Passing Interface [53]) and its interposition mechanism to transparently add our Task Packing functionality. The same ideas could be use in other programming models that follow the Single Program Multiple Data (SPMD) paradigm (i.e. Unified Parallel C (UPC) [54]).

We work on parallel applications which are not malleable. The proposed mechanism applies task migrations between CPUs just inside a node. This restriction avoids new overheads due to application checkpointing, moving data across the network between nodes and disk storage management among others.

We set as a requirement that the scheduling algorithm and the migration of tasks between CPUs do not affect the performance of the application. We are not trying to improve the performance, but we aim to keep the same execution time for the application. This is the reason why the packing is done using as an upper bound the iteration time of the slowest task.

The steps to do the reallocation are:

- Discard the first measurement, as it will be contaminated with overheads related to first accessing data like cache misses and coherency memory protocols.

- Gather iteration times at each node. This information is broadcasted and collected by all the tasks at each node. Using this information the local master is selected at each node, as the task with the shortest iteration time. Each local master has all the necessary information to apply the Task Packing algorithm. This step is repeated the first iterations until the difference between iteration times at each task is within a confident interval (e.g. less than 5%). As soon as a task arrives to this conclusion, iteration times are broadcasted (the task publishes a special value to notify this). Previous iteration times from all task at each node are kept by all the tasks within the node.

- Find out the task with the longest iteration time, which corresponds to the slowest task. This iteration time is used later as an upper bound to perform the Subset Sum algorithm for task reallocation.

  This is the only operation that is performed globally, at the application level. The rest of the algorithm is applied within each node. In addition, as each iteration finishes with a global synchronization, we are not introducing any extra synchronizations. The extra overhead added comes from exchanging iteration times at application and node level. In this sense, we perform the synchronizations in an incremental way, first the local exchange and then the global one. Experiments showed us that this overhead is negligible and that the incremental synchroniz-

ation served to minimize the total overhead. For this reason scalability is not affected because of the Task Packing algorithm.

- Once the local master has been established and has collected the iteration times within its node. At the next iteration, the Task Packing algorithm is applied. The local master is in charge of doing such work as it will be the first to complete the iteration and arrive to the synchronization point. The parameters passed to the algorithm are: iteration times of tasks within its local node and the global upper bound previously computed. After that, the solution is broadcasted to all the tasks in the node. In this way, as soon as each task finishes its calculation, it is able to receive the task reallocation instructions and perform the migration if necessary.

Notice that as the task reallocation is done at node level, different nodes can have different Task Packing results. Furthermore, there may be nodes with no task reallocation because their CPUs have no idle cycles (e.g. all of their tasks have execution times close to the slowest one). Figure 2 shows an example of an unbalanced application running on two nodes with four tasks on each node. We can observe that while tasks on node 0 are well-balanced, tasks on node 1 are unbalanced, so tasks on node 0 will have to wait at the synchronization point for tasks on node 1. Consequently the whole application is unbalanced. After applying the Task Packing algorithm at each node and taking into account the slowest task (Task 4 on node 1) we arrive at a task reallocation showed in Figure 3. In that figure we can see that Task 2 and Task 3 in node 0 have been migrated from CPU 2 and CPU 3 to CPU 0 and CPU 1 respectively, freeing two CPUs on that node. In node 1, there was also a migration of Task 7 from CPU 3 to CPU 1, freeing one CPU in this case.

Finally, oversubscription has an impact depending on the application communication degree. If the application has a low communication degree, i.e., tasks have low or null interaction between them apart from the global synchronization point, their tasks can be executed almost sequentially. Only time slice context switching is performed, but with useful utilization of the CPU during all the execution. But if the application has a high communication degree, the tasks have to interact between them frequently (e.g. many point-to-point message exchange). In this case, to progress execution and to avoid wasting CPU cycles waiting for a message, immediate context switch-

ing is allowed. In this way, the time slice is aborted, and the CPU is yielded to another task with useful work to do.

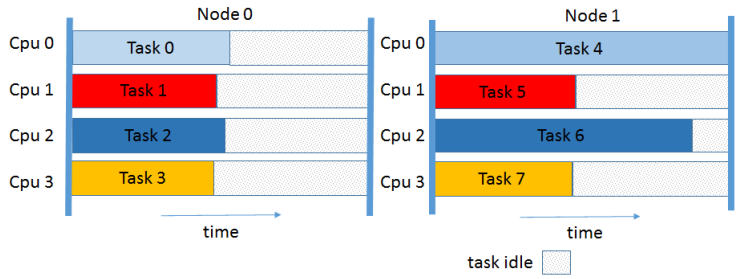

Figure 2: Unbalanced parallel application execution between two synchronization points running on two nodes.

3.3. Implementation details

We generate a shared library which is preloaded before running the applications. Global synchronization points (i.e. *MPI_Barrier*) expected at the end of each iteration, are intercepted and that is where our scheduling takes place.

Even though global collective operations like *MPI_Allreduce* do not synchronize (do not ensure that the tasks finish at the same time), this issue does not affect the iteration times measurement in our current implementation. We consider the imbalance between synchronizations points in a way that: we start the measurement after returning from the collective call (within the MPI interposition library), and we finish the measurement before calling it again.

The wrapper function of the synchronization operation encapsulates the coding of the packing algorithm (the code can be seen in listing 2).

Listing 2: Wrapper of the global synchronization MPI operation.

```
1  int global_synchronization_wrapper () {
2        int ret;
3        int solution[node_size][node_size]; // cpus x local ids
4
5        end_iter = MPI_Wtime();
6        iteration_time = end_iter - start_iter;
7
8        if (iteration_number == 2) {
9          // gather iteration times within each node
10         PMPI_Allgather(&iteration_time, 1, MPI_DOUBLE,
               local_iteration_times, 1, MPI_DOUBLE, comm_node);
```

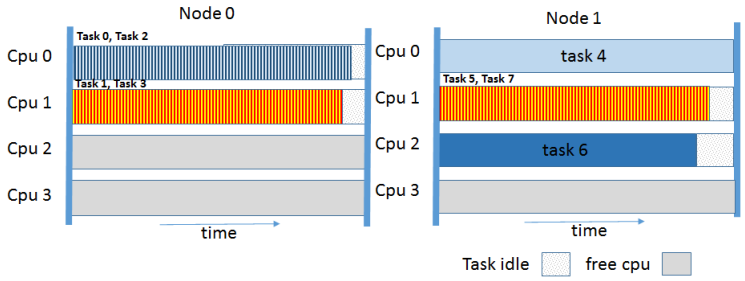

Figure 3: Unbalanced parallel application execution between two synchronization points running on two nodes after Task Packing algorithm is applied.

```
11
12          // find out longest iteration
13          PMPI_Allreduce(&iteration_time, &global_upper_bound, 1,
                MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
14
15          // find local master: task with shortest time iteration
16          local_master = find_min (local_iteration_times, node_size);
17
18        }
19        else if (iteration_number == 3) {
20          // perform task packing
21          if (local_id == local_master)
22                task_packing (local_iteration_times, node_size,
                    global_upper_bound, solution);
23
24          // broadcast packing solution to tasks within a node
25          PMPI_Bcast(solution, node_size*node_size, MPI_INT, local_master,
                comm_node);
26
27          // reallocate me
28          my_task_reallocation (solution, local_id, getpid());
29        }
30
31        ret = PMPI_global_synchronization();
32
33        start_iter = MPI_Wtime();
34        return ret;
35  }
```

We force immediate context switching rather than polling in the MPI communication system, to enable execution progress to tasks running on oversubscribed CPUs when performing MPI blocking operations [22]. In this way, if a task is blocked in an operation like waiting for a not yet arrived message, it is context switched immediately instead of polling for the message wasting CPU cycles. The immediate context switching allows another task to perform useful execution like sending the messages to blocked tasks.

In order to allow immediate context switching while a MPI communication blocking operation is performed the MPI Aggressive Mode is set. As this is an implementation specific feature below the MPI API, there is no standard way. However, most MPI implementations provide an environment variable for that purpose. If required, it could also be implemented as part or our interposition library, by intercepting the MPI blocking operations, and forcing the context switch (*sched_yield*) instead of polling. We have made some experiments obtaining negligible overhead.

Tasks running alone on CPUs do not need immediate context switching and this may be more energy consuming than polling. The amount of wasted CPU cycles in this context would depend on the application itself (i.e. communication pattern and degree) and the number of tasks running without oversubscription. We have made some preliminary experiments with the FT and miniFE benchmarks, measuring Energy consumption at job level (this is what our platform allows us) and the difference between both communication modes is negligible (less than 1%).

The mechanism is scalable since the packing is performed at node level. No knowledge about the global state of the application is needed. The packing algorithm is executed at each node and the migration decisions affect only to the local node.

We assume iterative and regular unbalanced behavior. The Task Packing algorithm is applied once and then tasks are bound to their assigned CPUs till the end of the execution. The Task Packing algorithm is not applied again. Notice that in case a Task Packing mechanism would be applied again, to measure iteration times fairly, the application should be first expanded to the freed CPUs.

The C code shown in listing 3 shows the implementation of the scheduler which invokes the Subset sum algorithm until there are no more tasks unassigned to CPUs.

Listing 3: Subset Sum algorithm invocation by the Scheduler.

```
1 int task_packing (int set[], int node_size, int sum, int solution[
      node_size][node_size]) {
2     int cpu=0; int unassigned_tasks=ntasks;
3     while (unassigned_tasks>0)
4         // sum is last elapsed time between two sync points
5         if (isSubSetSum (set, sum, &solution[cpu], &num)) {
6                 cpu++;
7                 update_unassigned_tasks(set, &solution[cpu]);
8                 unassigned_tasks=unassigned_tasks-num;
9         }
```

```
        else {
                sum--; // diminish upper bound
        }
}
```

At the beginning of execution, we wrap the *MPI_Init* operation to define MPI communication groups. In particular one subgroup for each node. Communication within a node is done exclusively through the communication library, making our mechanism more portable across platforms and MPI implementations.

Finally in listing 4 we show the implementation of the Subset Sum algorithm (*isSubsetSum*) in charge of packing tasks in one CPU.

Listing 4: Subset Sum algorithm code.

```

int isSubsetSum(int set[], int n, int sum, int subset[], int count, int
    sol[]){

  int i;

  if(sum == 0) {
       for(i =0; i < count; i++) {
                sol[i]=subset[i];
       }
       num=count;
       return true;
  }
  if(n < 0  && sum != 0)  return false;

  if (set[n]<0) return isSubsetSum(set, n-1, sum, subset, count , sol);

  subset[count] =  n;
  return isSubsetSum(set, n-1, sum-set[n], subset, count + 1, sol)
         + isSubsetSum(set, n-1, sum, subset, count , sol);
}
```

This algorithm proved to have negligible execution times for set sizes around 30 elements. After that, the execution time increments and the performance consequently of the Task Packing degrades. For this reason, in case we have set sizes larger than this limit we split the set into sizes that suits our performance limit and apply the packing in each of the subsets separately. This would not result in the optimal solution because it is not considered the whole set, but we can still find packing cases to apply. We believe that by splitting the original set in an intelligent way we can find a solution close to the optimal one. On the other hand, we observed that splitting the original set in subsets to apply Task Packing separately to each subset have an important advantage on NUMA nodes. When aligning the subsets to a

NUMA node, we avoid process migration between different NUMA nodes and the consequently remote data access costs for each process on the new location.

The mechanism identifies appropriate synchronization points transparently. We apply the mechanism just after several iterations have passed (in our experiments 10 iterations approximately). We assume that after observing similar behavior during a reasonable amount of iterations, this is the regular state (main loop) and apply the mechanism. In this way we avoid taking into consideration global synchronizations performed during initialization phases. To intercept the correct collective that performs the global synchronization call we use an environment variable (*MPIT_TASKPACKING*) which must be set with the name of the MPI operation that perform the synchronization at each iteration.

We are aware that there are applications that have more than one call to the same collective communications within one iteration, so by measuring iteration times, the mechanism will never consider there is a regular behavior. In this case, the only possible way to apply the mechanism is by inserting in the code two explicit API calls to our library: *MPIT_Begin_taskpacking()* to indicate the beginning of the iteration and the other one at the end (*MPIT_End_taskpacking()*). This one is in charge of performing the Task Packing.

We currently work just on the *MPI_COMM_WORLD* communicator. Extending it to all communicators is straightforward and just a few modifications are required. The knapsack algorithm return as a parameter the new task allocation as a result and this is the part that should be modified, as each previously defined local node communicator has its own set of CPUs, so the new allocation should use just this subset. This can be solved by keeping the information about the CPUs allocated to each local node communicator.

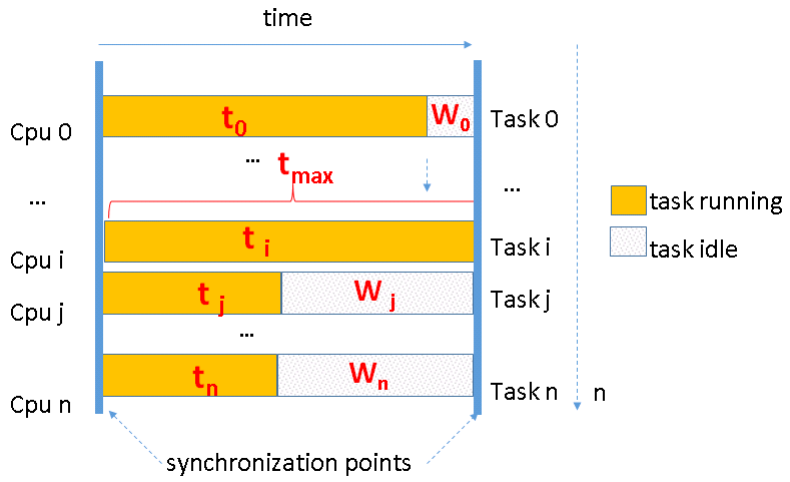

Figure 4: Representation of a parallel task execution between two synchronization points.

4. Applicability of the proposal

In section 3.1 we have discussed the targeted scenario and the requisites so that our mechanism can successfully run. In this section we would like to discuss the applicability of our solution.

We have stated that we target unbalanced applications with predictable unbalanced behavior and global synchronization points. And our goal is to free CPUs.

In order to free CPUs we will try to oversubscribe one or more CPUs with two or more tasks, so that the original CPU(s) where this(these) tasks were running can be free.

Figure 4 represents a parallel task execution between two global synchronization points. If we define t_i as the running time of task i between two synchronization points and w_i the time that task i is idle waiting for the other tasks to finish running, then the time between synchronization points (t_{max}) can be defined as follows:

$$t_{max} = \max_{0 \leq i < n} t_i \tag{2}$$

This time t_{max} defines the limit or threshold used by the Task Packing algorithm to compute the task reallocation. The following equation is true:

$$\forall_i(t_i + w_i = t_{max}) \tag{3}$$

For the Task Packing algorithm to be successful, that is to be able to allocate two tasks in the same CPU, at least we need the following to be true:

$$\begin{aligned} \exists_{i,j} \quad & (\quad t_i \leq w_j) \\ & (\quad t_i \leq t_{max} - t_j) \\ & (\quad t_i + t_j \leq t_{max}) \end{aligned}$$

If we generalize equation 4 for more than two tasks we have the following equation, which determines the condition:

$$t_{max} \geq \sum_{i \geq 2}^{n-1} t_i \tag{4}$$

The problem that the task packing algorithm resolves can be seen as follows: Given a set of n bins, the CPUs (all of the same size, which in turn corresponds to the runtime of the slowest task in a parallel session t_{max}) as well as a set of n items, the Tasks (of size t_i, where all t_i is smaller or equal to the size of the bins t_{max}), the algorithm determines a distribution of the items into the bins so that the number of still empty bins is as large as possible. A necessity for the algorithm to be successful in finding a distribution with one or more still empty bins is that two or more items fit in one bin. As stated in equation4, the size of at least two items must be smaller than the size of the bin.

This may seem a difficult condition to satisfy at the first glance. However, we argue that: (i) Heterogeneous platforms can exhibit a degree of work unbalance that satisfy our condition, as we show in our evaluation section. (ii) Applications in the way programmers think of them are naturally unbalanced. The unbalance becomes bigger as the application scales up and applications tend to be bigger and bigger as computers evolve and more computational power is available and scientists want to take advantage of it.

An example of this situation is happening in the SKA project [55]. The Square Kilometer Array (SKA) project is an international effort to build the worlds largest radio telescope, with eventually over a square kilometer of collecting area. This huge telescope will generate vasts amounts of data that need to be processed and analyzed to create images and other products to be distributed to astronomers. The project is now at the design phase. The parallel codes being developed at this stage of the project are very preliminary and mainly intended as a proof-of-concept. In order to further support the design decisions the team has developed a parametric model [56] to ascertain the computational demands of the software. Based on the these data, load balancing is already foreseen to be a problem and decisions are being taken to alleviate it: the scientific algorithms are being re-designed, and the team is moving away from rigid SPMD paradigms towards a less rigid model where global synchronization points are kept at a minimum.

Other big applications with load balancing problems (i.e. GROMACS and Gadget) follow the computation-synchronization model. They are written in MPI and cannot avoid the use of global synchronization points. GROMACS [57] hand tunes the process distribution to appaliate the load balancing issues, while Gadget [58] provides some extra code alongside their application code to tackle the balancing issues [59]. Both approaches slow down productivity of parallel programmers and scientists that have to diverge from their scientific problems to handle the load balancing of their application.

We claim that providing a mechanism where this can be done automatically would improve productivity of scientists writing/designing their scientific parallel applications.

Finally, another important point to consider is that scientific parallel applications do not follow a structure as simple as the one pictured in Listing 1. Applications have usually more than one phase of computation-synchronization which may exhibit different behavior, therefore the packing computed for one phase may not work for the next phase. We are planning an adaptive Task Packing algorithm that would extend our implementation by dynamically monitoring the unbalance during execution and if at one point a core passes the threshold t_{max} the task scheduling should be recomputed. This may not be simple, as to recompute iterations times, the application should expand to the previously freed CPUs, which possibly have been reused.

5. Results and evaluation

We evaluated our *Task Packing* mechanism on two benchmarks, *miniFE* and *FT*, and compared their execution times with and without the mechanism. In the following sections, we first describe our experimental platform, then provide details about the applications and their load imbalance and finally present our evaluation results.

5.1. Execution framework

The experiments were run in the MarenostrumIII supercomputer which is based on Intel SandyBridge processors with iDataPlex Compute Racks and running Linux Operating System. The interconnection networks are 10Gigabit Ethernet for disk accesses and Infiniband FDR10 for communication purposes. The machine had 48,896 Intel SandyBridge-EP E52670 cores at 2.6 GHz (3,056 compute nodes) with 103.5 TB of main memory, where 42 of these nodes were heterogeneous compute nodes with 8 x 8G DDR3-1600 DIMMs (4GB/core) and 2 x Xeon Phi 5110P accelerators [60].

In this work we call "Xeon node", the host of the heterogeneous node; and we call "Xeon Phi nodes" to the accelerators of the heterogeneous node. In this work, we do not make distinction between core and CPU. For simplicity we use the term CPU when referring to CPUs and cores.

The MPI library used is Intel MPI 4.1.3.049. The C compiler is gcc/4.7.2. The performance analysis was made using the Extrae library and Paraver tool [61].

5.2. Applications

For the experiments we use the following benchmarks:

- *miniFE*, a miniapplication from the Mantevo suite project. This benchmark performs a sparse matrix vector multiplication which is widely used in linear algebra and graph algorithms and a good representative of HPC applications that work on irregular structures. The miniFE miniapp benchmark, is similar to HPCG [20] but provides a much more complete vertical coverage of the steps in this class of applications. Like HPCG, miniFE is intended to be the best approximation to an unstructured implicit finite element for finite volume application, but in 8,000 lines or less. The benchmark is an iterative application. At each iteration there is first a point-to-point communication phase,

then the calculation phase composed by a matrix vector product and finally a global reduction (*MPI_Allreduce* operation). We run miniFE with a 200x200x200 matrix size and the following configuration: 16 tasks on each Xeon node and 24 tasks on each Xeon Phi node.

- FT, from the NAS Parallel Benchmarks which performs a 3-D partial differential equation solved using FFTs. It is a rigorous test of heavy long-distance communication performance. It is iterative, and the synchronization is achieved through an exchange of data from all tasks to all tasks (*MPI_Alltoall* operation)[18]. We run FT class C, with the following configuration: 16 tasks on the Xeon node and 8 tasks on each Xeon Phi Node.

We selected those benchmarks because they represent commonly found HPC applications that follow iterative pattern of computation phase followed by a global synchronization. The first benchmark, $miniFE$, is an example of applications with high communication degree and point-to-point communication type while the second one, FT, only does calculation during each iteration and the communication is heavily concentrated in the synchronization points.

For all the results presented, we defined a confidence coefficient of 95% and ran each experiment multiple times to reduce the standard error. We assumed experiments to be independent, therefore the formulas associated with a normal distribution apply [62].

5.3. Analysis of load imbalance on the heterogeneous platform

We show first the execution times of a typical iteration. The execution times are taken between two consecutive synchronization points ($MPI\_Alltoall$ in FT and $MPI\_Allreduce$ in miniFE). The experiments evidence the load imbalance generated when running on a heterogeneous platform.

We can see the results for the execution of FT in figure 5 running on 4 heterogeneous nodes. The iteration times range from 0.2 to 0.9 seconds. Notice that values that correspond to the Xeon nodes (ranks 0-15, 32-48, 64-80 and 96-111) are definitely less than the values from the Xeon Phi nodes (rest of the ranks). The iteration times of the FT benchmark are composed of just calculation, there is no communication at all during this phase. This explain the almost constant values of iteration times in CPUs with the same characteristics. Applying the Task Packing algorithm in this context is straightforward.

In figure 6 we can see the results for the iteration times for miniFE executed on 4 heterogeneous nodes. The iteration times range from 5 to 80 milliseconds approximately. The iterations times are far from being constant as happened with the FT results. However, values that correspond to the Xeon nodes (ranks 0-15, 64-79, 128-142 and 192-207) tend to be less than values from the Xeon Phi nodes (rest of the ranks). The point-to-point communication phase included in these iteration times add this variability mostly due to the limited bandwidth between Xeon and Xeon Phi nodes. Despite this variability the difference is big enough to take advantage of the Task Packing algorithm.

Although the Task Packing mechanism is applied on every node, in these experiments the task migration occurs just within Xeon nodes (the ones with smaller execution times). At these nodes, the condition showed in equation 1 is satisfied, which means that there exist k iterations (i) where their sum of execution times (t_i) is less than the upper bound given by the slowest task (t_{max}). The slowest task as seen in the "iteration times" figures belongs to a Xeon Phi node.

In tables 1 and 2 we show the load imbalance calculated on the Xeon nodes. This load imbalance gives us an idea about the percentage of CPUs that could be freed after applying the Task Packing mechanism. We define the load imbalance at a particular node, as the ratio between two areas: the one determined by the sum of CPU times (t_i) in a node; over the upper bound (t_{max}) multiplied by the number of tasks as shown in equation 5.

$$Load\_imbalance = \frac{(\sum_{i=0}^{node\_size} t_i)}{t_{max} * node\_size} \tag{5}$$

Table 1: FT imbalance on Xeon nodes

number of MPI tasks	**32**	**64**	**128**	**256**
imbalance fraction	0.70	0.80	0.80	0.80

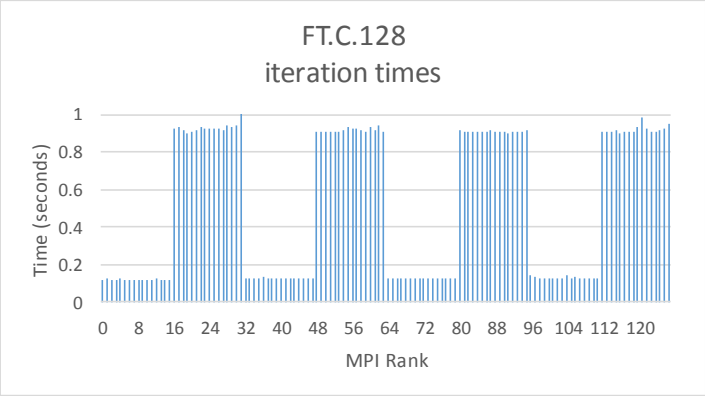

Figure 5: Execution times of one iteration between two consecutive synchronization points for each task in the miniFE benchmark on four Xeon nodes (ranks 0-15, 32-48, 64-80, 96-111) and eight Xeon Phi nodes (ranks 16-24, 25-31, 49-57, 58-63, 112-119, 120-127).

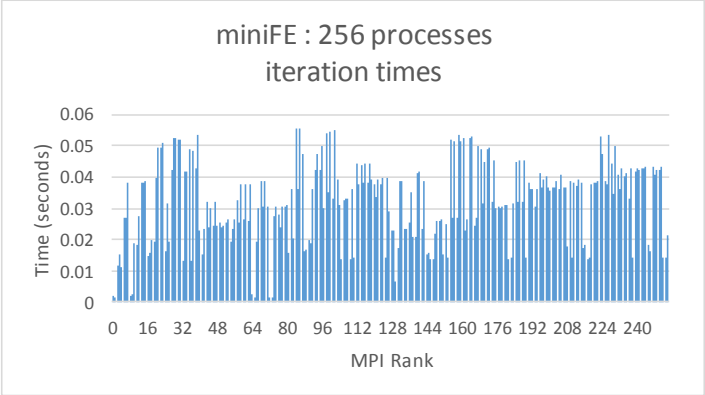

Figure 6: Execution times of one iteration between two consecutive synchronization points for each task in the miniFE benchmark on four Xeon nodes (ranks 0-15, 64-79, 128-142, 192-207) and eight Xeon Phi nodes (ranks 16-40, 31-64, 80-103, 104-127, 144-167, 168-191, 208-231, 232-255).

Table 2: miniFE imbalance on Xeon nodes

number of MPI tasks	**64**	**128**	**256**	**512**
imbalance fraction	0.71	0.47	0.39	0.31

5.4. Evaluation results

Figures 7 and 8 show the execution times when running FT and miniFE on heterogeneous nodes (from 1 to 8). The labels "NORMAL" and "TASK PACKING" correspond to the execution without applying and applying the Task Packing mechanism, respectively.

The execution times obtained with both task allocations are very close. However there is a tendency to increment the execution time with the number of tasks when applying Task Packing. The worst case is observed in miniFE with 512 tasks with a slowdown of Task Packing allocation with respect to Normal allocation of about 13%.

Taking a close look at the miniFE internal behavior, surprisingly enough, we found that point-to-point MPI operations are not the main source of performance degradation when running communicating tasks on oversubscribed CPUs. For example, in a normal execution, miniFE with 64 tasks, dedicates close to 0% of its execution time to point-to-point operations. If oversubscribing 4 tasks per CPU (16 CPUs assigned), the increment in time spent by point-to-point operations is about 15%. We found instead that *MPI_Allreduce* collective deals badly with CPUs oversubscription. This is not the case with other collectives like *MPI_Alltoall* (used in the FT benchmark). This collective in a normal execution of miniFE, i.e. without CPUs oversubscription, spends 35% of the total execution time when running with 16 tasks and goes up to approximately 65% when running with 128 tasks.

In addition, this collective has a hierarchical design adapted to the target platform. The reduction is solved following a specific rank ordering in this hierarchical design [63] generating dependencies between intermediate calculations. This fact combined with several tasks sharing one CPU (oversubscription) may degrade performance proportional to the size of the application.

The Task packing algorithm adds a global collective operation to get the maximum iteration time across nodes. Despite its cost increments with the number of tasks, this is done only once, and it does not affect the overall

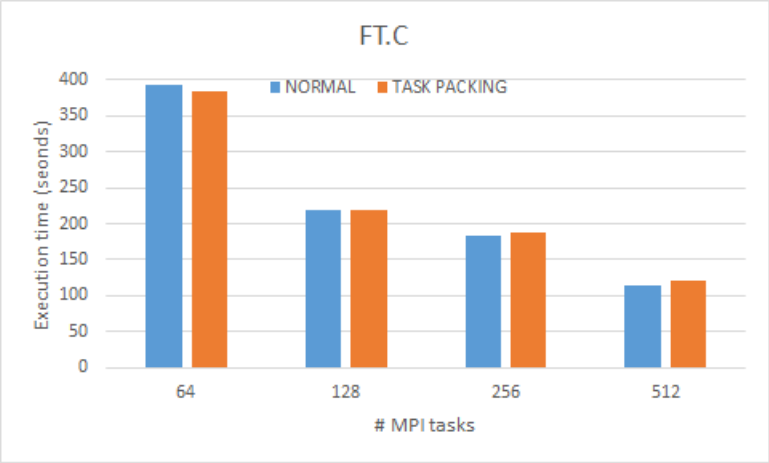

Figure 7: Execution times of FT, class C, run on 1, 2, 4 and 8 heterogeneous nodes with CPUs default allocation and after applying Task Packing.

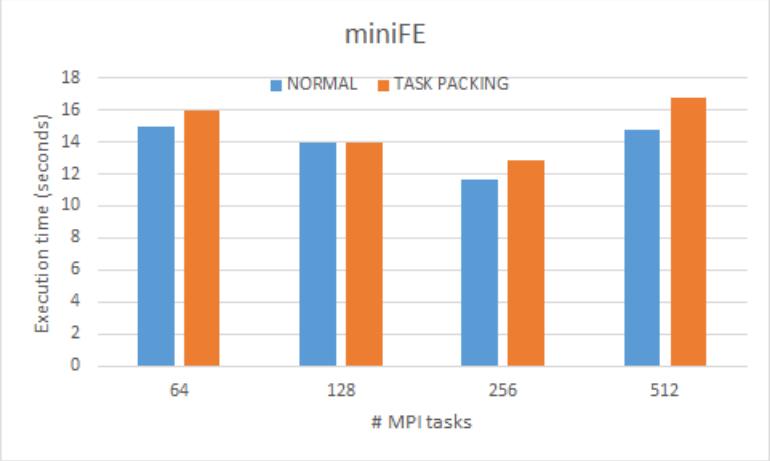

Figure 8: Execution times of miniFE, with matrix size 200x200x200 run on 1, 2, 4 and 8 heterogeneous nodes CPUs default allocation and after applying Task Packing.

performance.

In order to quantify the benefit obtained from our proposal, figures 9 and 10 show the relation between freed and assigned CPUs to the application after applying the Task Packing mechanism.

Recall that in our experiments, in FT, 16 tasks were configured to run on

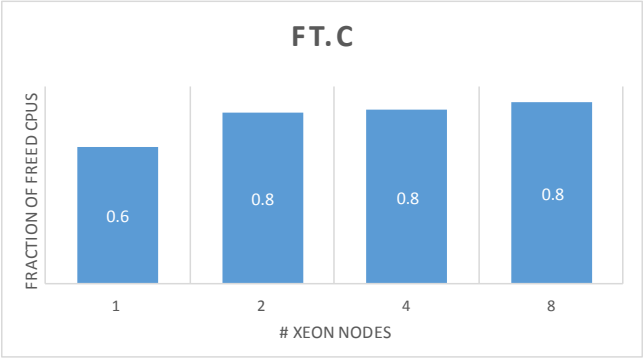

Figure 9: Freed CPUs on Xeon nodes after applying the Task Packing mechanism to the execution of FT on different number of nodes.

Xeon CPUs and 16 tasks were configured to run on Xeon Phi CPUs, at each heterogeneous node. For this reason, only 50% of the total assigned CPUs are eligible to be reassigned (i.e. the ones from the Xeon Phi nodes). The discussion about freed CPUs that follows takes into account just this 50% of the total number of assigned CPUs.

We can observe that in FT, the number of freed CPUs goes from 60% for one node to an impressive 80% of the total number of CPUs on Xeon nodes, when running on 4 nodes. This huge margin for oversubscription is explained because of the time spent in synchronization, especially when incrementing the number of tasks. The FT benchmark performs $MPI\_alltoall$ operation as global synchronization. This operation stresses the memory bandwidth, a critical point in the communication between Xeon and Xeon Phi nodes.

Taking a look at miniFE numbers, the percentage of CPUs eligible to be assigned constitutes 25% of the total number of assigned CPUs. From this 25%, the amount of freed CPUs goes from 60% on one heterogeneous node execution to 20% when running on 4 heterogeneous nodes.

The amount of freed CPUs correlates with the load imbalance calculated previously in equation 5 and showed in tables 1 and 2. So, the load imbalance can serve as a prediction about the benefit the Task Packing mechanism can achieve.

We succeeded in freeing up to 20% of the total number of assigned CPUs

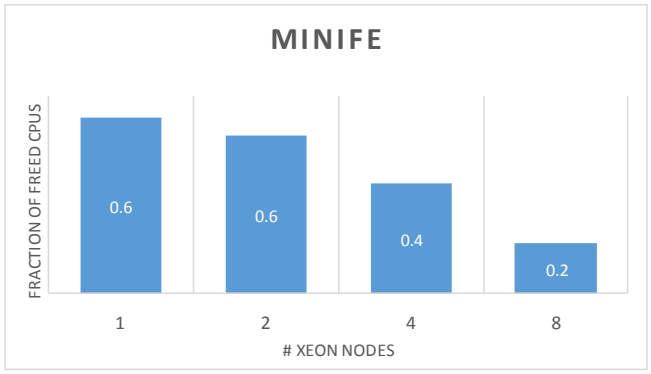

Figure 10: Freed CPUs on Xeon nodes after applying Task Packing mechanism to the execution of miniFE on different number of nodes.

in miniFE and up to 40% in FT on executions on heterogeneous platforms. When applying Task packing on application with communication phases between synchronization, extra overheads arises, leading to a performance degradation of up to 13% in the worst case.

6. Conclusions and future work

Imbalance behavior of applications is an undesirable property. There are cases where this imbalance comes from the application itself, as for example when working on irregular data structures. But sometimes the reasons are external to the applications, so even though the application is well-balanced, it can show an unbalanced behavior. The causes are for example when running on heterogeneous architectures, or due to operating system maintenance tasks (operating system noise), or due to network traffic congestion.

We present an extension of a previous work. The previous work presented a proof of concept of an idea about concentrating the CPU cycles in a smaller number of CPUs by using oversubscription, freeing CPUs for other purposes providing the execution time of the application was not degraded. At runtime, the Subset Sum algorithm, which is a particular case of the Knapsack algorithm tries to find a solution to reallocate tasks. Some of them are eventually bound to the same CPU. The task migration are only within a node. The algorithm just takes into account the execution time of each task till it arrives to the next synchronization point (and consequently where the unbalance becomes apparent). The execution time between two consecutive synchronization points is the upper bound parameter for applying the Task Packing algorithm.

In this work we provide a more efficient implementation, where the scheduling algorithm is executed at each node by a task from the unbalanced application. The manager task is selected by being the shortest (the one with the smallest execution time between two consecutive synchronization points). In this way, we take advantage of waiting cycles to make useful work instead, without degrading the application performance and without wasting an extra CPU for that purpose. In addition we gather all the information needed for the scheduling decisions through the message passing system, making the mechanism more portable.

The implementation of the mechanism is transparent to the programmer and the user. Our implementation is scalable as the Task Packing is applied at node level, and task migration decisions are made locally.

We have evaluated the mechanism using two well-known benchmarks: FT and miniFE. The first one with no communication at all apart from the regular synchronization points and the second one a point-to-point high communication degree application. Despite the applications are well-balanced, we have shown that they exhibit a high level of imbalance when executed

on an heterogeneous platform. We argue that heterogeneous platforms are increasing, both in number and complexity, and that load balancing will become even more problematic. This adds relevance to our work. All experiments presented in this paper have been done in an heterogeneous platform to emphasize this point.

Our experiments demonstrate that our Task Packing mechanism has negligible overhead on iterative applications with predictable unbalanced behavior. We detected an overhead up to 13% in applications that use a reduction collective because of its rank ordering algorithm. Without detriment of the application our Task Packing mechanism is able to free up to 40% of the total assigned CPUs in presence of 70% of imbalance. We have analyzed our results and observed that the benefit in terms of freed CPUs, depends on factors like: the communication degree of the application (high communication degree make the application more unpredictable, and scheduling decisions may not be always the accurate ones) and the number of eligible CPUs to be reassigned. This number is closely related to the node with the fastest CPUs. The greater the number of CPUs of that node, the greater the number of eligible CPUs. In addition, we have proposed a metric of load imbalance as an estimate of the amount of free CPUs that could be obtained after applying the Task Packing mechanism.

We considered in this work just iterative and regular applications. The load imbalance of an application has to be predictable in order to apply the Task Packing mechanism. Once the packing is applied there will be some nodes with CPUs oversubscribed. After that, if new load imbalance arises (e.g. current load imbalance disappear) it can be detected, but to apply the Task Packing algorithm we need the real iteration times of each of the tasks, that is to say, without oversubscription. One solution would be to expand the oversubscribed CPUs (one task per CPU) and start all over again. However, to expand the application CPUs should be available and this may not be possible: consider the case the freed CPUs were recycled by the job scheduler for backfilling purposes. We need to reformulate the strategy to be able to measure load imbalance after Task Packing was already applied and analyse the cases where our approach would be beneficial.

Finally, we believe that the conservation of energy and the improvement of system utilization are important topics that would benefit from our Task Packing mechanism. The freed CPU(s) could be either shutdown to save energy or used by tasks from other applications. Although most modern CPUs are capable of lowering their frequency for saving power consumption

during idle states. We argue that shutting down a whole CPUs temporally, despite requiring more preparation time it yields better energy saving than just throttling all the CPUs during sporadically idle states. We would have liked to provide an energy consumption evaluation but unfortunately we could not check energy consumption in our experimental platform. We will definitely make this measurements as soon as we are able to.

7. Acknowledgements

The authors acknowledge the support of the BSC (Barcelona Supercomputing Centre). We would like to thank the anonymous reviewers for their comments, which helped us to improve the manuscript. This work has been supported by the Spanish Ministry of Education (TIN2012-34557 and TIN2015-65316-P), the Spanish Ministry of Economy and Competitiveness (HAR2014-57776-P) and the Generalitat de Catalunya (2014-SGR-1051).

References

[1] J. Dongarra, A. L. Lastovetsky, High performance heterogeneous computing, Vol. 78, John Wiley & Sons, 2009.

[2] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, Nvidia tesla: A unified graphics and computing architecture, IEEE micro 28 (2).

[3] J. Jeffers, J. Reinders, Intel Xeon Phi coprocessor high-performance programming, Newnes, 2013.

[4] V. Kindratenko, Novel computing architectures, Computing in Science Engineering 11 (3) (2009) 54–57. doi:10.1109/MCSE.2009.56.

[5] T. Chen, R. Raghavan, J. N. Dale, E. Iwata, Cell broadband engine architecture and its first implementation—a performance view, IBM Journal of Research and Development 51 (5) (2007) 559–572.

[6] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation computer systems 25 (6) (2009) 599–616.

[7] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Commun. ACM 52 (10) (2009) 56–67. doi:10.1145/1562764.1562783.
URL http://doi.acm.org/10.1145/1562764.1562783

[8] L. Eeckhout, Is moores law slowing down? whats next?, IEEE Micro 37 (4) (2017) 4–5. doi:10.1109/MM.2017.3211123.

[9] M. Etinski, J. Corbalan, J. Labarta, M. Valero, Utilization driven power-aware parallel job scheduling, Computer Science-Research and Development 25 (3-4) (2010) 207–216.

[10] D. A. Ellsworth, A. D. Malony, B. Rountree, M. Schulz, Dynamic power sharing for higher job throughput, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 80:1–80:11. doi:10.1145/2807591.2807643.
URL http://doi.acm.org/10.1145/2807591.2807643

[11] S. Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan, Characterization of backfilling strategies for parallel job scheduling (2002) 514–519doi:10.1109/ICPPW.2002.1039773.

[12] G. Utrera, J. Corbalán, J. Labarta, Another approach to backfilled jobs: Applying virtual malleability to expired windows, in: Proceedings of the 19th Annual International Conference on Supercomputing, ICS '05, ACM, New York, NY, USA, 2005, pp. 313–322. doi:10.1145/1088149.1088191.
URL http://doi.acm.org/10.1145/1088149.1088191

[13] SchedMD, SLURM job scheduling system, https://slurm.schedmd.com/ [cited 01/10/2018].
URL https://slurm.schedmd.com/

[14] F. Petrini, D. J. Kerbyson, S. Pakin, The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q, in: Supercomputing, 2003 ACM/IEEE Conference, IEEE, 2003, pp. 55–55.

[15] K. B. Ferreira, P. G. Bridges, R. Brightwell, K. T. Pedretti, The impact of system design parameters on application noise sensitivity, Cluster Computing 16 (1) (2013) 117–129. doi:10.1007/s10586-011-0178-3.
URL http://dx.doi.org/10.1007/s10586-011-0178-3

[16] T. Hoefler, T. Schneider, A. Lumsdaine, Characterizing the influence of system noise on large-scale applications by simulation, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11. doi:10.1109/SC.2010.12.
URL http://dx.doi.org/10.1109/SC.2010.12

[17] G. Utrera, M. Farreras, J. Fornes, Task packing: Getting the best from mpi unbalanced applications, in: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2017, pp. 547–550. doi:10.1109/PDP.2017.51.

[18] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S.

Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, The nas parallel benchmarks summary and preliminary results, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91), 1991, pp. 158–165. doi:10.1145/125826.125925.

[19] Mantevo suite benchmarks, MiniFE from Mantevo suite benchmarks, http://mantevo.org/.

[20] HPCG, The High Performance Conjugate Gradients (HPCG) Benchmark, http://hpcg-benchmark.org.

[21] J. Dongarra, P. Luszczek, LINPACK Benchmark, Springer US, Boston, MA, 2011, pp. 1033–1036. doi:10.1007/978-0-387-09766-4_155.
URL http://dx.doi.org/10.1007/978-0-387-09766-4_155

[22] C. Iancu, S. Hofmeyr, F. Blagojevi, Y. Zheng, Oversubscription on multicore processors, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1–11.

[23] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, W. Zhu, Parallex: A study of a new parallel computation model, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, IEEE, 2007, pp. 1–6.

[24] T. Sterling, M. Anderson, P. K. Bohan, M. Brodowicz, A. Kulkarni, B. Zhang, Towards Exascale Co-design in a Runtime System, Springer International Publishing, Cham, 2015, pp. 85–99. doi:10.1007/978-3-319-15976-8_6.
URL http://dx.doi.org/10.1007/978-3-319-15976-8_6

[25] G. Utrera, J. Corbalan, J. Labarta, Scheduling parallel jobs on multicore clusters using cpu oversubscription, The Journal of Supercomputing 68 (3) (2014) 1113–1140. doi:10.1007/s11227-014-1142-9.

[26] R. C. Merkle, M. E. Hellman, Hiding information and signatures in trapdoor knapsacks, Information Theory, IEEE Transactions on 24 (5) (1978) 525–530.

[27] A. M. Odlyzko, The rise and fall of knapsack cryptosystems, Cryptology and computational number theory 42 (1990) 75–88.

[28] V. C. Camargo, L. Mattiolli, F. M. Toledo, A knapsack problem as a tool to solve the production planning problem in small foundries, Computers & Operations Research 39 (1) (2012) 86–92.

[29] M. Vasquez, J.-K. Hao, A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite, Computational Optimization and Applications 20 (2) (2001) 137–157.

[30] H. Kellerer, V. A. Strusevich, Fully polynomial approximation schemes for a symmetric quadratic knapsack problem and its scheduling applications, Algorithmica 57 (4) (2010) 769–795.

[31] A. Goldman, D. Trystram, An efficient parallel algorithm for solving the knapsack problem on hypercubes, Journal of Parallel and Distributed Computing 64 (11) (2004) 1213 – 1222. doi:https://doi.org/10.1016/j.jpdc.2002.10.001.
URL http://www.sciencedirect.com/science/article/pii/S0743731504000966

[32] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. J. Dongarra, Parsec: Exploiting heterogeneity to enhance scalability, Computing in Science & Engineering 15 (6) (2013) 36–45.

[33] J. Planas, R. M. Badia, E. Ayguadé, J. Labarta, Hierarchical task-based programming with starss, International Journal of High Performance Computing Applications 23 (3) (2009) 284–299.

[34] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.

[35] S. Goss, S. Aron, J.-L. Deneubourg, J. M. Pasteels, Self-organized shortcuts in the argentine ant, Naturwissenschaften 76 (12) (1989) 579–581.

[36] M. Dorigo, Optimization, learning and natural algorithms, PhD Thesis, Politecnico di Milano.

[37] M. Dorigo, T. Stützle, Ant colony optimization: overview and recent advances, in: Handbook of metaheuristics, Springer, 2019, pp. 311–351.

[38] A. Llanes, J. M. Cecilia, A. Sánchez, J. M. García, M. Amos, M. Ujaldón, Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization, Cluster Computing 19 (1) (2016) 1–11.

[39] B. Imbernon, J. Prades, D. Gimenez, J. M. Cecilia, F. Silla, Enhancing large-scale docking simulation on heterogeneous systems: An mpi vs rcuda study, Future Generation Computer Systems 79 (2018) 26–37.

[40] C. Reaño, F. Silla, G. Shainer, S. Schultz, Local and remote gpus perform similar with edr 100g infiniband, in: Proceedings of the Industrial Track of the 16th International Middleware Conference, ACM, 2015, p. 4.

[41] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, C. W. Tseng, Dynamic load balancing of unbalanced computations using message passing, in: 2007 IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1–8.

[42] G. Utrera, J. Corbalán, J. Labarta, High-Performance Computing: 6th International Symposium, ISHPC 2005, Nara, Japan, September 7-9, 2005, First International Workshop on Advanced Low Power Systems, ALPS 2006, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, Ch. Dynamic Load Balancing in MPI Jobs, pp. 117–129. doi:10.1007/978-3-540-77704-5_10.

[43] C. Huang, O. Lawlor, L. V. Kalé, Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, Ch. Adaptive MPI, pp. 306–322. doi:10.1007/978-3-540-24644-2_20.

[44] D. Bohme, F. Wolf, B. R. De Supinski, M. Schulz, M. Geimer, Scalable critical-path based performance analysis, in: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, IEEE, 2012, pp. 1330–1340.

[45] C. Boneti, R. Gioiosa, F. J. Cazorla, M. Valero, A dynamic scheduler for balancing hpc applications, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ,

USA, 2008, pp. 41:1–41:12.
URL http://dl.acm.org/citation.cfm?id=1413370.1413412

[46] B. S. Parsons, V. S. Pai, Exploiting process imbalance to improve mpi collective operations in hierarchical systems, in: Proceedings of the 29th ACM on International Conference on Supercomputing, ACM, 2015, pp. 57–66.

[47] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, T. Mattson, A 48-core ia-32 message-passing processor with dvfs in 45nm cmos, in: 2010 IEEE International Solid-State Circuits Conference - (ISSCC), 2010, pp. 108–109. doi:10.1109/ISSCC.2010.5434077.

[48] N. R. Adiga, G. Almási, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, et al., An overview of the bluegene/l supercomputer, in: Supercomputing, ACM/IEEE 2002 Conference, IEEE, 2002, pp. 60–60.

[49] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, Y. Ishikawa, Casper: An asynchronous progress model for mpi rma on many-core architectures, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 665–676.

[50] S. L. Mirtaheri, L. Grandinetti, Dynamic load balancing in distributed exascale computing systems, Cluster Computing 20 (4) (2017) 3677–3689.

[51] S. L. Mirtaheri, S. A. Fatemi, L. Grandinetti, Adaptive load balancing dashboard in dynamic distributed systems, Supercomputing Frontiers and Innovations 4 (4) (2017) 34–49.

[52] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of the ACM (JACM) 21 (2) (1974) 277–292.

[53] R. L. Graham, The MPI 2.2 Standard and the Emerging MPI 3 Standard, in: Proceedings of the 16th European PVM/MPI Users' Group

Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 2–2. doi:http://dx.doi.org/10.1007/978-3-642-03770-2_2.

[54] T. A. El-Ghazawi, W. W. Carlson, J. M. Draper, UPC Language Specifications, v1.1.1 Edition (October 2003).

[55] Square Kilometre Array, The SKA Project, http://www.skatelescope.org.

[56] B. Nikolic, P. Wortmann, P. Alexander, Parametric models of SDP compute requirements, http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000013_05_rep_sdp_performance_model_view_part_1_-_signed.pdf.

[57] E. Lindahl, B. Hess, D. van der Spoel, GROMACS 3.0: A package for molecular simulation and trajectory analysis, J. Mol. Mod. 7 (2001) 306–317.

[58] V. Springel, N. Yoshida, S. D. White, Gadget: a code for collisionless and gasdynamical cosmological simulations, New Astronomy 6 (2) (2001) 79 – 117. doi:https://doi.org/10.1016/S1384-1076(01)00042-2.
URL http://www.sciencedirect.com/science/article/pii/S1384107601000422

[59] M. G. Garcia, Dynamic load balancing for hybrid applications, Ph.D. thesis, Universitat Politecnica de Catalunya (2017).
URL http://hdl.handle.net/2117/108227

[60] Barcelona Supercomputing Center, Marenostrum 3, https://www.bsc.es/support/MareNostrum3-ug.pdf.

[61] Barcelona Supercomputing Center, Paraver: a flexible performance analysis tool, http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview.

[62] H. W.G., H. J.S., Statistics for Experimenters: Design, Innovation and Discovery, Box, G.E. Wiley, New York, 2005.

[63] J. M. Álvarez Llorente, J. C. Díaz-Martín, J. A. Rico-Gallego, Formal modeling and performance evaluation of a run-time rank remapping

technique in broadcast, allgather and allreduce mpi collective operations, in: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 963–972. doi:10.1109/CCGRID.2017.32. URL https://doi.org/10.1109/CCGRID.2017.32