HAL
open science

On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms

Emilio Francesquini, Márcio Castro, Pedro Henrique Penna, Fabrice Dupros, Henrique Cota De Freitas, Philippe Olivier Alexandre Navaux, Jean-François Méhaut

To cite this version:

Emilio Francesquini, Márcio Castro, Pedro Henrique Penna, Fabrice Dupros, Henrique Cota De Freitas, et al.. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. Journal of Parallel and Distributed Computing, 2015, 76, pp. 32-48. 10.1016/j.jpdc.2014.11.002 . hal-01092325

HAL Id: hal-01092325

https://brgm.hal.science/hal-01092325v1

Submitted on 8 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

*Manuscript
Click here to view linked References

On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms

Emilio Francesquini[a,b,g], Márcio Castro[c,d], Pedro H. Penna[e], Fabrice Dupros[f], Henrique C. Freitas[e], Philippe O. A. Navaux[c], Jean-François Méhaut[g]

[a] *Institute of Computing, University of Campinas (UNICAMP)*
Av. Albert Einstein, 1251 - Cidade Universitária- 13083-852 - Campinas, Brazil
[b] *Institute of Mathematics and Statistics, University of São Paulo (USP)*
Rua do Matão, 1010 - Cidade Universitária - 05508-090 - São Paulo - Brazil
[c] *Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)*
Av. Bento Gonçalves, 9500 - Campus do Vale - 91501-970 - Porto Alegre - Brazil
[d] *Department of Informatics and Statistics, Federal University of Santa Catarina (UFSC)*
Campus Reitor João David Ferreira Lima - Trindade - 88040-970 - Florianópolis - Brazil
[e] *Department of Computer Science, Pontifical Catholic University of Minas Gerais (PUC Minas)*
Avenida Dom José Gaspar, 500 - 30535-901 - Belo Horizonte - Brazil
[f] *Bureau de Recherches Géologiques et Minières (BRGM)*
BP 6009, 45060 Orléans Cedex 2, France
[g] *CEA-DRT - LIG Laboratory, University of Grenoble*
110 Avenue de la Chimie, 38400 Saint-Martin d'Hères, France

Abstract

Until the last decade, performance of HPC architectures has been almost exclusively quantified by their processing power. However, energy efficiency is being recently considered as important as raw performance and has become a critical aspect to the development of scalable systems. These strict energy constraints guided the development of a new class of so-called light-weight manycore processors. This study evaluates the computing and energy performance of two well-known irregular NP-hard problems — the Traveling-Salesman Problem (TSP) and K-Means clustering — and a numerical seismic wave propagation simulation kernel — Ondes3D — on multicore, NUMA, and manycore platforms. First, we concentrate on the nontrivial task of adapting these applications to a manycore, specifically the novel MPPA-256 manycore processor. Then, we analyze their performance and energy consumption on those different machines. Our results show that applications able to fully use the resources of a manycore can have better performance and may consume from 3.8x to 13x less energy when compared to low-power and general-purpose multicore processors, respectively.

Keywords: Manycore, multicore, NUMA, energy efficiency, performance, TSP, seismic wave propagation, k-means

Email addresses: `francesquini@ic.unicamp.br` (Emilio Francesquini), `marcio.castro@inf.ufsc.br` (Márcio Castro), `pedro.penna@sga.pucminas.br` (Pedro H. Penna), `f.dupros@brgm.fr` (Fabrice Dupros), `cota@pucminas.br` (Henrique C. Freitas), `navaux@inf.ufrgs.br` (Philippe O. A. Navaux), `jean-francois.mehaut@imag.fr` (Jean-François Méhaut)

1. Introduction

Demand for higher processor performance led chipmakers to include into their designs solutions that are a combination of brute-force and innovation. The increase of processors cache size, instruction-level parallelism and working frequency have been for the last decades their main tools to accomplish this mission. However, these approaches seem to have reached a point in which they, by themselves, are not sufficent to ensure the steep curve of performance improvement predicted by Moore's Law and expected by the users [1].

An exponential increase in power consumption related to a linear increase in the clock frequency [2] and a higher complexity to design new processors changed the course of development of these new processors. Power consumption has become a critical aspect to the development of both large and small scale systems. This concern is now enough to warrant the research on the use of embedded low-power processors to create the next generation of HPC systems. For instance, the European Mont-Blanc project [3] was created to evaluate the use of such components in an HPC environment [4]. While these low-power multicore processors usually do not offer the same performance as their regular counterparts, they normally offer better energy-to-solution results.

Current highly-parallel processors take this paradigm even further. They normally possess hundreds (sometimes thousands) of cores which execute with high energy efficiency. The execution model of these processors usually follows two different approaches. Light-weight manycore processors, such as Tilera Tile-Gx [5] and Kalray MPPA-256 [6], offer autonomous cores and a shared memory execution model. In this case, traditional tools such as POSIX threads are employed to accomplish both data and task parallelism. The use these tools may ease the paradigm shift from multicores to manycores, since several parallel applications developed for multicores rely on this model. Differently, Graphics Processing Units (GPUs) follow another approach based on a Single Program, Multiple Data (SPMD) model, relying on runtime APIs such as CUDA and OpenCL. Thus, considerable effort may be necessary to adapt parallel code originally developed for multicores to GPUs. Here we are interested in the former.

In this paper we describe three different irregular applications and the necessary adaptations to use them on four distinct hardware platforms. The Traveling Salesman Problem (TSP), the K-Means clustering (K-Means) algorithm, and a Seismic Wave Propagation kernel (Ondes3D). Solutions to the TSP and K-Means problems are NP-hard and, for a large enough instance, the algorithm can be parallelized to make use of an arbitrary number of threads, assuring the complete use of the chosen platforms. Ondes3D, on the other hand, employs a numeric seismic wave propagation simulation algorithm. These applications were chosen because they represent three different behaviors: CPU-bound (TSP), memory-bound (Ondes3D), and mixed (K-Means). However, while all of them are highly parallelizable, they also reveal important issues related to imbalance and irregularity: the execution course for the same instance of the problem can drastically change depending on the order and on the number of employed processor cores.

We consider two important aspects in this study. The first aspect concerns the programming issues and challenges encountered when adapting these irregular applications for the MPPA-256 manycore processor. The use of Network-on-Chip (NoC) for communication and the absence of cache coherence protocols are among the important factors that make the development of parallel applications on this processor not trivial. Additionally, processors such as MPPA-256 have important memory constraints, *e.g.*, limited amount of directly addressable memory (2 MB). Furthermore, efficient execution on this processor requires data transfers in conformance to the NoC topology to mitigate the, otherwise high, communication costs. The lessons learned give

some insights on what can be faced when adapting parallel applications to manycores.

The second aspect concerns the performance and energy consumption of multicores and manycores. Our experiments were carried out on four different hardware platforms: Intel Xeon E5, SGI Altix UV 2000, Samsung Exynos 5, and Kalray MPPA-256. The first two are composed of general-purpose processors, while the remaining two are based on embedded low-power processors. We compare the overall performance of these platforms as well as their power efficiency. Our results show that the energy-to-solution for the same instance of the problem can present important variations between the experimental platforms. For every application, MPPA-256 presented the best energy-to-solution, consuming from 3.8x to 6.9x less energy than the second best platform (Exynos 5). When compared to Xeon E5 and Altix UV 2000, MPPA-256 consumed respectively from 5.7x to 13.1x and from 8.5x to 12.3x less energy. The time-to-solution on the other hand was dominated by the Altix UV 2000 platform. MPPA-256 and Xeon E5 showed approximately equivalent performances, however with a clear advantage to Xeon E5 for memory-bound applications. With relation to the Altix UV 2000 platform, the execution of the applications were on average from 13.0x to 20.4x, 108.8x to 154.8x and 13.0x to 15.3x slower on the Xeon E5, Exynos 5 and MPPA-256 platforms respectively. Next, we compare the Altix UV 2000 and the MPPA-256 platforms. Although very different from each other, these platforms share some similarities that give us the opportunity to evaluate important aspects of their scalability and energy efficiency. We concluded that both architectures scale well for the chosen applications. While on low core counts MPPA-256 may have a higher energy-to-solution, it can quickly fill the gap as the increase in the number of cores results in a small increase in the average power consumption.

The remainder of this paper is organized as follows. Section 2 outlines the evaluated platforms. A high-level description of the TSP, K-Means and Ondes3D as well as their algorithms are detailed in Section 3. Next, Section 4 discusses the challenges encountered when passing from multicores to the MPPA-256 manycore processor. Then, Section 5 presents performance and energy efficiency evaluations. Finally, we discuss related works in Section 6 and conclude in Section 7.

2. Experimental Platforms

In this section we describe the experimental platforms used in this study. These platforms represent three different classes: general-purpose, embedded and manycore.

2.1. General-Purpose

Xeon E5. The Intel Xeon E5 is a 64-bit x86-64 processor. In this study we used a Xeon E5-4640 Sandy Bridge-EP, which has 8 physical cores running at 2.40 GHz. Each core has 32 KB instruction and 32 KB data L1 caches and 256 KB of L2 cache. All the 8 cores share a 20 MB L3 cache and the platform has 32 GB of DDR3 memory.

Altix UV 2000. SGI Altix UV 2000 (Figure 1) is a Non-Uniform Memory Access (NUMA) platform designed by SGI. The platform is composed of 24 NUMA nodes. Each node has a Xeon E5-4640 Sandy Bridge-EP processor (with the same specifications of the Xeon E5 platform) and 32 GB of DDR3 memory. This memory is shared in a ccNUMA fashion through SGI's proprietary NUMAlink6 (bidirectional). This high-speed interconnection provides a point-to-point bandwidth of 6.7 GB/s per direction. Overall, this platform has 192 physical cores.

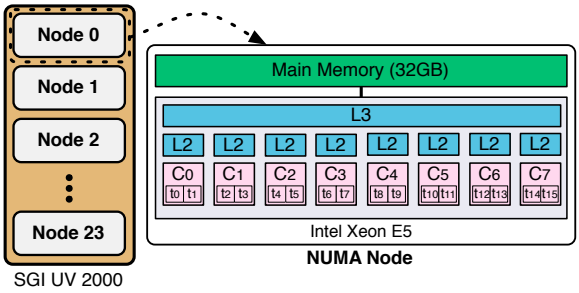

Figure 1: A simplified view of Altix UV 2000.

2.2. *Embedded*

Exynos 5. Samsung Exynos 5 is a Multiprocessor System-on-Chip (MPSoC) that implements the recent ARM big.LITTLE heterogeneous computing architecture. In this study we used the ODROID-XU+E board, which features a Samsung Exynos 5 5410 Octa processor. This processor integrates a Cortex-A15 quad-core running at 1.6 GHz with a Cortex-A7 quad-core running at 1.2 GHz on the same chip. Both CPUs are connected to 2 GB of LP-DDR3 memory. This processor uses a clustered model approach: the operating system scheduler is only aware of four out of the total of eight processing cores. If at any point in time the load on the four Cortex-A7 cores surpasses a pre-established threshold, the processor itself switches the execution to the Cortex-A15 cores. This is done in a way that is transparent to the operating system. The rationale here is that, while the Cortex-A15 cores provide better performance, they also incur in a higher energy utilization and by switching between the two sets of cores energy can be saved.

2.3. *Manycore*

MPPA-256. Kalray MPPA-256 is a single-chip manycore processor developed by Kalray that integrates 256 user cores and 32 system cores. It uses 28nm CMOS technology running at 400 MHz. These cores are distributed across 16 compute clusters and 4 I/O subsystems that communicate through data and control NoCs. This processor targets parallel applications whose programming models fall within the following classes: Kahn Process Networks (KPN), as motivated by media processing; SPMD, traditionally used for numerical kernels; and time-triggered control systems [7, 6].

Figure 2 shows the architecture overview of the MPPA-256. It features two types of cores: Processing Elements (PE) and Resource Managers (RM). Although RMs and PEs implement the same Very Long Instruction Word (VLIW) architecture, they have different purposes: PEs are dedicated to run user threads (one thread per PE) in non-interruptible and non-preemptible mode whereas RMs execute kernel routines and services of NoC interfaces. Operations executed by RMs vary from task and communication management to I/O data exchanges between either external buses (e.g. PCIe) or SDRAM. For this reason, RMs have privileged connections to NoC interfaces. Both PEs and RMs feature private 2-way associative instruction and data caches.

PEs and RMs are grouped within compute clusters and I/O subsystems. Each compute cluster features 16 PEs, 1 RM and a local shared memory of 2 MB, which creates a interconnection with high bandwidth and throughput between PEs. Each I/O subsystem relies on 4 RMs with a shared D-cache, static memory and external DDR access (2 GB). Contrary to the RMs available on

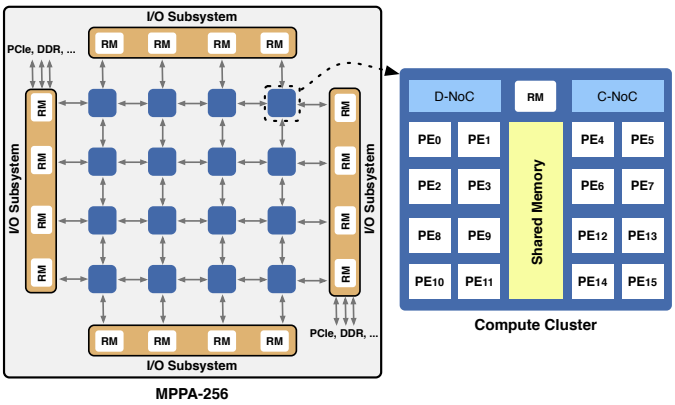

Figure 2: A simplified view of the MPPA-256.

compute clusters, the RMs of I/O subsystems also run user code. An important peculiarity of the MPPA-256 architecture is that it does not support cache coherence between PEs, even among those in the same compute cluster.

Parallel applications running on MPPA-256 usually follow the *master/worker* pattern. The *master* process runs on an RM of the I/O subsystem and it is responsible for spawning *worker* processes. These processes are then executed on compute clusters and each process may create up to 16 POSIX threads, one for each PE. In other words, the *master* process running on the I/O subsystem must spawn 16 *worker* processes and each process must create 16 threads in order to make full use of the 256 cores.

Compute clusters and I/O subsystems are connected by two parallel NoCs, the Data NoC (D-NoC) and the Control NoC (C-NoC). Both NoCs have bi-directional links, and there is one NoC node per compute cluster, which is controlled by the RM. I/O subsystems, on the other hand, have 4 NoC nodes, each one associated with a D-NoC router and a C-NoC router. The D-NoC is dedicated to high bandwidth data transfers whereas the C-NoC is dedicated to peripheral D-NoC flow control, power management and application messages.

2.4. Synthesis

As we previously mentioned, Xeon E5, Altix UV 2000, Exynos 5 and MPPA-256 represent different platform classes. Both Xeon E5 and Altix UV 2000 belong to the class of general-purpose platforms we usually find in servers. These platforms are tuned for performance rather than energy efficiency. Differently from those performance-centric platforms, Exynos 5 targets mobile devices in which power is one of the most important concerns. Finally, MPPA-256 belongs to the light-weight manycore platform class. It presents a high density of cores in a single chip but still is more energy efficient than general-purpose processors. In the next section we describe the three applications we used in this paper to analyze the performance and the energy efficiency of these platforms. These applications can also be categorized into distinct behavioral classes what allows us to carry out this study in a comprehensive yet simple manner.

3. Case Studies

Application execution performance can vary a lot depending on the hardware platform on which they are being executed. These pieces of software are normally categorized by their behavior taking into account the execution aspect that influences their performance the most. For instance, an application in which the time used for memory accesses is a performance bottleneck is said to be memory-bound. Similarly, an application in which the execution bottleneck is the computation time is said to be CPU-bound. As we show in Section 5, an application that on a hardware architecture is CPU-bound can become memory-bound on other architectures if the underlying hardware characteristics are not taken into consideration.

To highlight the impact different amounts of computation and communication can make on the execution performance and the energy efficiency of the experimental hardware platforms, we chose three applications with three distinct behaviors (CPU-bound, memory-bound and mixed). We now detail these applications further.

3.1. Traveling-Salesman Problem

The TSP consists in solving the routing problem of a hypothetical traveling- salesman. Such a route must pass through n cities, only once per city, return to the city of origin and have the shortest possible length. It is a very well studied NP-hard problem. More formally, the problem could be represented as a complete undirected graph $G = (V, E)$, $|V| = n$ where each edge $(i, j) \in E$ has an associated cost $c(i, j) \geq 0$ representing the distance from the city i to j (Figure 3a). The goal is to find a hamiltonian cycle with minimum cost that visits each city only once and finishes at the city of departure.

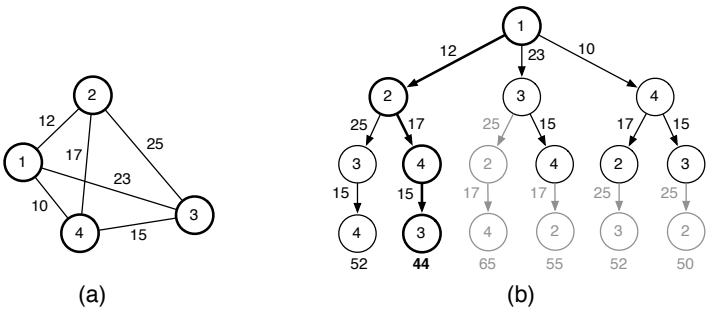

Figure 3: Example of TSP with 4 cities.

There are several different approaches to solve this problem [8]. These solutions normally employ brute force, simple or complex heuristics, approximation algorithms or a mix of them. We are not going to detail the different available approaches since we are after the evaluation and performance comparison of an embarrassingly parallel non-numerical application across different architectures. Therefore, we use a brute force exact algorithm based on a simple heuristic [9]. We first explain the sequential version of our algorithm, then we explain how we extended it to work with multiple threads. Finally, we present its distributed version.

3.1.1. Sequential Algorithm

The sequential version of the algorithm is based on the branch and bound method using brute force. Algorithm 3.1 outlines this solution. It takes as input the number of cities and a cost matrix, and outputs the minimum path length.

Algorithm 3.1: TSP SEQUENTIAL($n_cities, costs$)

```
global min_path
procedure TSP_SOLVE(last_city, current_cost, cities)
 if cities = ∅
   then return (current_cost)
 for each i ∈ cities
   do
 { new_cost ← current_cost + costs[last_city, i]
 { if new_cost < min_path
 {   then { new_min ← TSP_SOLVE(i, new_cost, cities\{i})
 {        { ATOMIC_UPDATE_IF_LESS(min_path, new_min)

main
 min_path ← ∞
 TSP_SOLVE(1, 0, {2, 3, ..., n_cities})
 output (min_path)
```

This algorithm does a depth-first search looking for the shortest path and has complexity $O(n!)$. It does not explore paths that are already known to be longer than the best path found so far, therefore discarding fruitless branches. Figure 3b shows this behavior. The shaded edges are those that the algorithm does not follow, since a possible solution that includes them would be more costly than the one it has already identified. This simple pruning technique greatly improves the performance of the algorithm. However, it also introduces irregularities into the search space. The search depth needed to discard one of the branches depends on the order in which the branches were searched.

3.1.2. Multi-threaded Algorithm

The multi-threaded version of the algorithm works by creating a queue of tasks from which each thread takes the jobs to be executed. A task is nothing more than one of the branches of the search tree. The generation of the tasks is done sequentially since the time needed to do it is negligible. As soon as one thread runs out of work, it takes a new task from the queue. The number of tasks to be generated is a function of the number of threads and is defined by the *max_hops* parameter. This is the minimum number of levels of the search tree that must be descended so that there is a minimum (parameterizable) number of tasks per thread. The total number of tasks as a function of levels l and cities n can be determined by the following recurrence relation (Equation 1) which is defined for $0 \leq l < n$.

$$t(l,n) = \begin{cases} 1 & l = 0 \\ t(l-1,n) * (n-l) & \text{otherwise} \end{cases} \tag{1}$$

Algorithm 3.2 shows the pseudo-code for this approach. This algorithm also receives as a parameter the number of threads *n_threads* to be used.

Algorithm 3.2: TSP MULTI-THREADED($n_cities, costs, n_threads, max_hops$)

```
global queue, min_path
procedure GENERATE_TASKS(n_hops, last_city, current_cost, cities)
 if n_hops = max_hops
   then { task ← (last_city, current_cost, cities)
          ENQUEUE_TASK(queue, task)
   else { for each i ∈ cities
          do { if last_city = none
               then last_cost ← 0
               else last_cost ← costs[last_city, i]
             new_cost ← curr_cost + last_cost
             GENERATE_TASKS(n_hops + 1, i, new_cost, cities\{i})

procedure DO_WORK()
 while queue ≠ ∅
   do { (last_city, current_cost, cities) ← ATOMIC_DEQUEUE(queue)
        TSP_SOLVE(last_city, current_cost, cities)

main
 min_path ← ∞
 GENERATE_TASKS(0, none, 0, {1, 2, ..., n_cities})
 for i ← 1 to n_threads
   do SPAWN_THREAD(DO_WORK())
 WAIT_EVERY_CHILD_THREAD()
 output (min_path)
```

3.1.3. Distributed Algorithm

The distributed algorithm is similar to the multi-threaded version. It receives as an additional parameter the number of distributed *peers* to be used. The number of peers and the number of threads define the total number of lines of execution. For each peer, *n_threads* will be created, thus totaling *n_threads*×*n_peers* threads. Inside each peer, the execution is nearly identical to that of the multi-threaded case. The only difference is that when the *min_path* is updated, this update is broadcasted to every other peer so they can also use it to optimize their execution. At the end of the execution, one of the peers (typically the 0-th) prints the solution. The final solution might have been discovered by any one of the peers, however all of them are aware of it due to the broadcasts of each discovered *min_path*.

To avoid two peers working on the same subproblem, each peer *peer_id* only works on the tasks which were assigned to it. To do so, we specify the desired number of partitions per peer. We also specify the percentage of the tasks that will be distributed in the beginning of the execution. Afterwards, as the peers run out of work, they will ask a master peer for more partitions. To reduce communication, the master peer sends sets of partitions of decreasing size at each request [10]. The rationale behind it is that, as the task sizes are irregular, distributing a smaller number of partitions during the end of the execution might decrease the imbalance between the peers. In this case, for each request the master peer sends a set of partitions S and the peer *peer_id* will work on the tasks such that $task_index \mod n_partitions \in S$. Since the task generation is done locally, the amount of transferred data can be minimized.

As a runtime optimization, only one thread per peer becomes responsible for asking for more partitions when the peer runs out of work. Once this thread receives a new partition from the master peer, it generates and populates the peer's task queue with new tasks. During the generation of these tasks, the remaining $n_threads - 1$ threads can begin to process tasks as soon as they are enqueued, without the need to wait for the end of the task generation. This behavior is further discussed in Section 5.5.

3.2. The K-Means Clustering Problem

Clustering analysis plays an important role in different fields, including data mining, pattern recognition, image analysis and bioinformatics [11]. In this context, a widely used and studied clustering approach is the K-Means clustering.

Formally, the K-Means clustering problem can be defined as follows. Given a set of n points in a real d-dimensional space, the problem is to partition these n points into k partitions, so as to minimize the mean squared distance from each point to the center of the partition it belongs to. Figure 4 illustrates an instance of this problem.

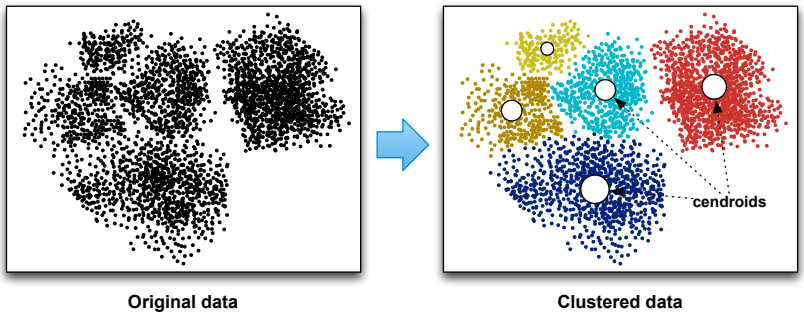

Figure 4: An example of K-Means with 5 partitions.

Several distinct heuristics have been proposed to address the K-Means clustering problem [12, 13]. One of the most widely employed is the Lloyd's algorithm [14], also known as K-Means algorithm. Such heuristic is based on an iterative strategy that finds a locally minimum solution for the problem. In our work we used this algorithm as a case study. In the following subsections, we first present the sequential version of the algorithm and then we introduce and explain our parallel and distributed versions.

3.2.1. Sequential Algorithm

The sequential version of K-Means is shown in Algorithm 3.3. The main idea is to use the notion of minimum Euclidean distance to iteratively partition the data points. The algorithm takes as input the set of data *points*, the number of partitions k, and the minimum accepted distance, *mindistance*, between each point and the partition's center (centroids). Upon completion, the algorithm returns the partitions themselves.

Algorithm 3.3: K-MEANS SEQUENTIAL($points, k, mindistance$)

```
global partitions

procedure POPULATE()
 for each pnt ∈ points
   do pnt.partition ← NEAREST(pnt, partitions)

procedure COMPUTE_CENTROIDS()
 for each part ∈ partitions
   do part.centroid ← COMPUTE_MEAN(part.population)

main
 RANDOM_POPULATE(partitions, points)
 COMPUTE_CENTROIDS()
 repeat
   { POPULATE()
   { COMPUTE_CENTROIDS()
 until HAS_CHANGED() and TOO_FAR()
 return (partitions)
```

The sequential K-Means algorithm works as follows. Initially, data points are evenly and randomly distributed among the k partitions, and the initial centroids are computed. Then, the data points are re-clustered into partitions taking into account the minimum Euclidean distance between them and the centroids — points are assigned to the nearest partition. Next, the centroid of each partition is recalculated taking the mean of all points in the partition, and the whole procedure is repeated until no centroid is changed and every point is farther than the minimum accepted distance.

It is worthy to note that this algorithm presents a natural irregularity: at any time during the execution, the number of points within each partition (population) may differ, implying different recalculation times for each partition's centroid.

3.2.2. Multi-threaded Algorithm

The multi-threaded version of the K-Means algorithm is presented in Algorithm 3.4. Compared to the sequential algorithm, it takes an additional parameter, t, that specifies the total number of execution flows. The strategy adopted is to assign to each thread an unique range of points and partitions, and split the algorithm in two phases. In the first phase, each thread re-clusters its

own range of points into the k partitions. In the second phase, each thread works in its own range of partitions, in order to recalculate centroids.

Algorithm 3.4: K-MEANS MULTI-THREADED($points, k, mindistance, t$)

```
global partitions

procedure DO_KMEANS(work)
 repeat
  { POPULATE(work.first_point, work.last_point)
  { COMPUTE_CENTROIDS(work.first_partition, work.last_partition)
 until HAS_CHANGED() and TOO_FAR()

main
 RANDOM_POPULATE(partitions, points)
 COMPUTE_CENTROIDS(partitions.first, partition.last)
 for i ← 0 to (t − 1)
      { work.first_point ← i × num_points ÷ t
      { work.last_point ← (i + 1) × num_points ÷ t
   do { work.first_partition ← i × k ÷ t
      { work.last_partition ← (i + 1) × k ÷ t
      { SPAWN_THREAD(DO_KMEANS(work))
 WAIT_EVERY_CHILD_THREAD()
 return (partitions)
```

The multi-threaded version of the algorithm still presents some important execution irregularities. Although the range of points and partitions are evenly distributed among the working threads, the amount of work for each thread may vary during each iteration, since for the duration of the second phase more populated partitions end up requiring more operations to have their centroids recalculated.

3.2.3. Distributed Algorithm

The distributed algorithm described in this section is widely used in practice [11, 15, 16] and a scalability analysis for this algorithm can be seen in the work by Rodrigues *et. al.* [17]. Compared to the multi-threaded algorithm, the distributed K-Means algorithm takes an additional parameter p that specifies the number of distributed peers to be used. Each peer by itself spawns t working threads, so the total number of threads equals to $p \times t$.

The strategy employed in this algorithm is to first distribute the data points and replicate the data centroids among peers, and then to loop over a two-phase iteration. In the first phase, partitions are populated, as in the multi-threaded algorithm, and in the second phase, data centroids are recalculated. For this recalculation, first each peer uses its local data points to compute partial centroids, *i.e.*, a partial sum of data points and population within a partition. Next, peers exchange partial centroids so that each peer ends up with the partial centroids of the same partitions. Finally, peers compute their local centroids and broadcast them.

One could argue that it would be possible to remove some of the irregularity from the multi-threaded version if we used the same partial centroid calculation technique used by the distributed implementation. The multi-threaded version of the algorithm splits the computation in two independent phases: populate partitions and compute centroids of partitions. The main advantage of using this approach is that it requires fewer thread synchronization structures, when compared to the distributed implementation. However, this multi-threaded implementation may introduce some irregularity in the application. Instead, if we adopted the same technique used by the distributed approach, we could split the overall work into smaller tasks and decrease some of the irregularity. However, the decreased irregularity would be achieved at the expense of an important increase in the cost of synchronization structures.

Nevertheless, the irregularity itself is closely related to the working data set: if partitions are too unbalanced, irregularity is strongly present; whereas if points are evenly distributed among partitions, irregularity is not so sharply presented. In both the multi-threaded and distributed algorithms, we work with a uniformly distributed random data set, and thus irregularity is not strongly present. Considering that, if we adopted the distributed approach in the multi-threaded implementation, we would further decrease irregularity, but the performance gains obtained by that strategy are quickly overcome by the additional synchronization procedures causing, in fact, performance degradation.

3.3. Seismic Wave Propagation

Understanding the wave propagation with respect to the structure of the Earth lies at the core of many analysis both in the oil and gas industry and for quantitative seismic hazard assessment. In this paper the earthquake process is described as elastodynamics and we use a finite-differences scheme for solving the wave propagation problem in elastic media [18]. This approach was first proposed in 1970 and since then it has been widely employed due to its simple formulation and implementation. In this section we describe the governing equations and discuss some of their standard sequential and parallel implementations.

The seismic wave equation in the case of an elastic material is:

$$\rho \frac{\partial v_i}{\partial t} = \frac{\partial \sigma_{ij}}{\partial j} + F_i \tag{2}$$

Additionally, the constitutive relation in the case of a isotropic medium is:

$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda \delta_{ij} \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right) + \mu \left(\frac{\partial v_i}{\partial j} + \frac{\partial v_j}{\partial i} \right) \tag{3}$$

Where indices i, j, k represent a component of a vector or tensor field in Cartesian coordinates (x, y, z), v_i and σ_{ij} represent the velocity and stress field respectively, and F_i denotes an external source force. ρ is the material density and λ and μ are the elastic coefficients known as Lamé parameters. A time derivative is denoted by $\frac{\partial}{\partial t}$ and a spatial derivative with respect to the i-th direction is represented by $\frac{\partial}{\partial i}$. The Kronecker symbol δ_{ij} is equal to 1 if $i = j$ and zero otherwise.

3.3.1. Sequential Algorithm

As mentioned before, the finite differences method is one of the most popular techniques to solve the elastodynamics equations and to simulate the propagation of seismic waves [18, 19]. One of the key features of this scheme is the introduction of a staggered-grid [20] for the discretization of the seismic wave equation.

Indeed, all the unknowns are evaluated at the same location for classical collocated methods over a regular Cartesian grid whereas the staggered grid leads to a shift of the derivatives by half a grid cell (Figure 5). The equations are rewritten as a first-order system in time and therefore the velocity and the stress fields can be simultaneously evaluated at a given time step.

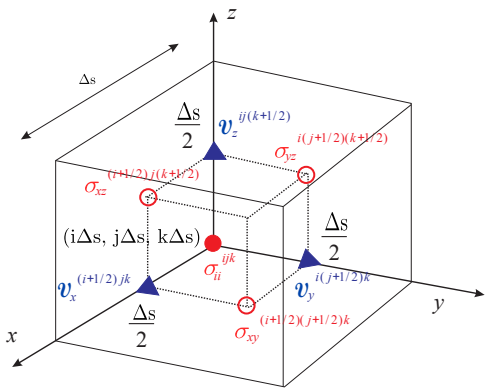

Figure 5: Elementary 3D cell of the staggered grid and distribution of the stress (σ) and the velocity (v) components.

The computational procedure is described in Algorithm 3.5. Inside the time step loop, the first triple nested loop is devoted to the computation of the velocity components, and the second loop reuses the velocity results of the previous time step to update the stress field. For instance, the stencil applied for the computation of the velocity component in the x-direction is given by Equation 4. Exponents i, j, k indicate the spatial direction, $\sigma^{ijk} = \sigma(i\Delta s, j\Delta s, k\Delta s)$, Δs corresponds to the space step, Δt to the time step and a_1, a_2 and a_3 are defined as three constants.

Algorithm 3.5: SEQUENTIAL SEISMIC WAVE PROPAGATION KERNEL(σ, v)

for $x \leftarrow 1$ **to** *x_dimension*
  do { **for** $y \leftarrow 1$ **to** *y_dimension*
    do { **for** $z \leftarrow 1$ **to** *z_dimension*
      do { COMPUTE_STRESS($\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{xz}, \sigma_{yz}$)
for $x \leftarrow 1$ **to** *x_dimension*
  do { **for** $y \leftarrow 1$ **to** *y_dimension*
    do { **for** $z \leftarrow 1$ **to** *z_dimension*
      do { COMPUTE_VELOCITY(v_x, v_y, v_z)

One particularity of the three-dimensional simulation of seismic wave propagation is the consideration of a finite computing domain whereas the physical problem is unbounded. Additional numerical conditions are then required to absorb the energy at the artificial boundaries. At the lateral and bottom edges of the three-dimensional geometry, a specific set of equations is implemented. For instance, the classical Perfectly Matched Layer (PML) relies on the implemen-

tation of a sponge numerical function that provides exponential attenuation in the nonphysical region [21]. A fixed size of ten grid points is chosen for the thickness of this layer (represented in gray color in Figure 6) and the CPU-cost ratio observed between a boundary grid point and a physical domain point varies from two to four.

$$
\begin{aligned}
v_x^{(i+\frac{1}{2})jk}\Big(l+\tfrac{1}{2}\Big) &= v_x^{(i+\frac{1}{2})jk}\Big(l-\tfrac{1}{2}\Big) + a_1 F_x^{(i+\frac{1}{2})jk} \\
&+ a_2\Big[\frac{\sigma_{xx}^{(i+1)jk} - \sigma_{xx}^{ijk}}{\Delta x} + \frac{\sigma_{xy}^{(i+\frac{1}{2})(j+\frac{1}{2})k} - \sigma_{xy}^{(i+\frac{1}{2})(j-\frac{1}{2})k}}{\Delta y} + \frac{\sigma_{xz}^{(i+\frac{1}{2})j(k+\frac{1}{2})} - \sigma_{xz}^{(i+\frac{1}{2})j(k-\frac{1}{2})}}{\Delta z}\Big] \\
&- a_3\Big[\frac{\sigma_{xx}^{(i+2)jk} - \sigma_{xx}^{(i-1)jk}}{\Delta x} + \frac{\sigma_{xy}^{(i+\frac{1}{2})(j+\frac{3}{2})k} - \sigma_{xy}^{(i+\frac{1}{2})(j-\frac{3}{2})k}}{\Delta y} + \frac{\sigma_{xz}^{(i+\frac{1}{2})j(k+\frac{3}{2})} - \sigma_{xz}^{(i+\frac{1}{2})j(k-\frac{3}{2})}}{\Delta z}\Big] \qquad (4)
\end{aligned}
$$

This numerical kernel leads to several challenges when considering its implementation on parallel architectures. The load imbalance must be tackled with different strategies adapted to shared or distributed architectures. This first level of irregularity could be worsened by the memory-bound nature of this numerical stencil and advanced strategies must therefore be used to maximize the performances on hierarchical platforms.

3.3.2. Multi-threaded Algorithm

On shared-memory architectures, a popular way to extract parallelism is to exploit the triple nested loops coming from the three dimensions of the problem under study. This allows a very straightforward use of OpenMP directives. However, two levels of irregularity should be considered with this straightforward implementation. Firstly, the imbalance coming from the absorbing boundary conditions could be partially addressed by using a dynamic schedule across the loop iterations. This leads to significant improvements in the distribution of the load [22]. This solution comes at the expense of introducing an irregular access of the data with higher NUMA penalties on hierarchical platforms. In this paper we tackled the imbalance by exploiting a static strategy along with an intelligent memory allocation policy. Basically, we guarantee that the memory accessed by each thread is allocated close to the thread. This reduces considerably the latencies in NUMA platforms. In the same sense, advanced runtime systems also provide good results in order to improve threads and memory mapping [23].

3.3.3. Distributed Algorithm

On distributed memory architectures, most standard parallel implementations of the elastodynamics equations are based on cartesian grid decomposition. Although our code is different, our approach is very similar to that used by *Cui et al.* [24] and *Furumura and Chen* [25]. Our distributed algorithm works by decomposing the computational domain into sub-domains D_i in such a way that each sub-domain is mapped to one *peer*. Inside each peer, the execution is nearly identical to that of the multi-threaded case. The difference is that peers need to communicate with their neighbors to exchange boundary data. Figure 6 shows this decomposition with 3×3 subdomains with an equal number of grid points in each.

This strategy can be optimized by using non-blocking communications among peers and by overlapping communications and computations. For instance, we first compute the boundary grid points located between neighbors. Then, these values are exchanged between neighboring peers using non-blocking communications. During this exchange phase, each peer computes its inner points in parallel.

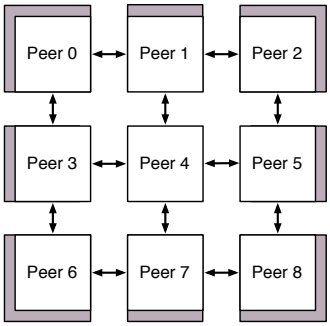

Figure 6: Distributed implementation of the seismic wave propagation kernel with a 3×3 decomposition. The gray regions represent the irregular absorbing boundary condition layers.

This decomposition also implies an irregular load between peers, due to the absorbing boundary condition layers. The irregularity of the distributed algorithm is investigated by *Tesser et al.* [26]. In this case, quasi-static decomposition, adaptive mesh refinement or parallel mesh partitioning are standard techniques that intend to balance computations. In this paper we applied the former technique [27].

3.4. General Considerations

In this section we described the three applications we use as case studies. We presented TSP, a CPU-bound application, in which only a small amount of data needs to be kept during the execution. In the multi-threaded algorithm threads rarely communicate, and when they do so they only exchange small amounts of data. Conversely, Ondes3D is a memory-bound application that accesses vast amounts of memory during its execution. Threads on the multi-threaded version often communicate to exchange information about the borders of the current data set on which they are working. K-Means is halfway between these two applications: the time used for data accesses is well balanced with computation time. In the multi-threaded version each thread has its own set of data that is synchronized at the end of each iteration.

Both K-Means and TSP present strong irregularity which translates to unpredictable execution times. The irregularity itself is linked to the problem (and to the input) and the chosen algorithm can only do so much to try to alleviate this issue during runtime [28]. These kind of applications have a strong need for dynamic load balancing to manage the irregularity. Weak irregularity is also present in these applications and it is normally associated to the chosen algorithms, data structures, and load-balancing strategy. With varying degrees of effort (and success) these weak irregularities might lightened by distinct implementation choices.

Contrary to TSP and K-Means, Ondes3D is an application that presents only weak irregularity. It is a predictable application in the sense that the total number of operations and communications is known in advance. For this application, given the input, it is even possible with some effort to statically perform the load balancing before the actual execution. However, in practice, this kind of approach seems to be left aside in favor of a dynamically load-balanced stencil based approach. In this kind of solution the irregularity arises from the data and their shape (at the border of computation domain and inside the computation domain) and not from the load-balancer. The load-balancer is, in fact, the responsible for trying to ensure a more regular execution.

By taking into consideration these three applications and the three distinct hardware architectures used in our experimental evaluations, we can draw comprehensive (yet straightforward) conclusions about execution performance and energy efficiency of these applications on the distinct hardware platform classes.

Manycores have several distinctive features that must be considered so that applications can achieve good performance. In the next section we present the adaptations we did to the applications we just presented in order to achieve an efficient execution.

4. Adapting Irregular Applications for Manycores

The adaptation of existing distributed applications to manycores such as the MPPA-256 can be straightforward, as in the case of K-Means. On the other hand, some applications as the TSP and Ondes3D demand much more effort. In this section we describe the adaptations these applications had to go through to efficiently execute on the MPPA-256 manycore.

4.1. TSP

Section 3.1 presented some insights into the algorithms for the resolution of the TSP. However, efficiently passing from multicores to manycores might be a nontrivial task. There are several reasons for that, the most evident being the natural architectural differences between these platforms. These differences usually force us to make adaptations to the code. In this section, we discuss these architectural aspects and adaptations as well as the rationale behind them.

POSIX threads are supported by all experimental platforms. This allowed us to execute the application on Xeon E5, Altix UV 2000, and Exynos 5 using exactly the same code. In our implementation, the global variable *min_path*, defined by Algorithm 3.1, is implemented using a simple shared variable that is accessed by every thread. The function ATOMIC_UPDATE_IF_LESS() is therefore implemented using a regular POSIX lock.

Unfortunately, this common solution is not appropriate to the MPPA-256 platform since it does not possess coherent caches. Despite the fact that the update of *min_path* works as it should (on the MPPA-256 platform the POSIX lock implementation invalidates the whole cache) and the final path length is correct, each one of the worker threads might be using a stale value of the *min_path* variable for a long time (in the worst case until the end of its execution) and wasting time on fruitless branches of the search tree. This means that, although correct, the execution might be severely slowed down. To correct it, we have used platform specific instructions that allow us direct access to the local memory of the cluster, bypassing the cache (`__builtin_k1_lwu` and `__builtin_k1_swu` to load/store data from/to the local memory, respectively). The cost of reading the variable in this manner is clearly higher than using the value stored in the caches (reading from memory takes 8 cycles whereas reading from cache takes at most 2 cycles). Yet, the performance improvement due to the better pruning of the search tree largely outweighs the additional cost.

In order to efficiently exploit the MPPA-256 platform, we needed to use every cluster of the chip. These clusters do not have a global memory space hence the need for the distributed version of the algorithm. Conversely, Altix UV 2000 platform has a global memory space, however, as the communications between the NUMA nodes are done through the NUMAlink6 interconnection, we can make a better use of this system by keeping the memory near the threads that use it and avoid using the link to perform anything but global synchronizations and *min_path* propagation. The distributed algorithm fits perfectly in this scenario.

In general, the distributed algorithm used by MPPA-256 and Altix UV 2000 is the same. Peers in the MPPA-256 platform take the form of compute clusters while in the Altix UV 2000 platform each peer is represented by a NUMA node. The difference lies on the implementation of the *min_path* broadcast and the task distribution. On the Altix UV 2000 platform, the implementation is based on shared memory using locks and condition variables. On the other hand, the implementation for the MPPA-256 platform is more complex. Since there is no shared memory between clusters, we employ asynchronous message exchanges. These message exchanges take the form of remote memory write operations. This can be done using proprietary MPPA-256 low-level system calls that allow a thread in a cluster to write to the memory of any other cluster on the chip. In both cases, the local value of the *min_path* variable is updated atomically. However, due to the time needed to broadcast a new value, some threads might use a stale value for a short time until the broadcast is completed.

4.2. K-Means

Section 3.2 presented the K-Means problem as well as three different approaches to solve it. In this section we discuss how the solution of this problem was adapted to the manycore architecture used in this work.

Xeon E5, Altix UV 2000 and Exynos 5 are platforms in which all cores have access to a global shared memory space. Additionally, these platforms support OpenMP. Therefore, for these platforms we employed the multi-threaded solution presented in Algorithm 3.4 using OpenMP for parallelization. Unfortunately this same solution is not appropriate for the MPPA-256 platform. Even though MPPA-256 supports OpenMP, cores in the MPPA-256 platform are grouped into 16-core clusters. Cores in the same cluster have access to the local shared memory but have no access to memory present on the remaining clusters. For this reason we had to embrace the distributed version of the application in order to explore the full computational power provided by this platform.

Despite the distributed algorithm presented in Section 3.2.3 being more appropriate to the MPPA-256 platform than the multi-threaded version, it has some characteristics that limit its direct use on this platform. Local memory available to each cluster (2 MB, of which 500 KB are used by the operating system) creates a strong constraint on the number of points that can be dealt with by each cluster. Even though the 32 MB (16×2 MB) of memory available in the computing clusters could store a reasonably sized workload, a static distribution of points at the initialization of the algorithm would totally disregard the 2 GB of memory available at the I/O subsystem. Therefore, we employed a dynamic solution for the distribution of points to be able to work with a number of points that is only limited by the amount of memory available at the I/O subsystem.

In order to do so, we implemented a variation of the distributed algorithm using a dynamic one-level tiling strategy. In this solution the I/O subsystem keeps a copy of all the points and partitions. At each iteration, during the *populate partitions* and *compute centroids* phases, each computing cluster repeatedly downloads chunks of points from the I/O subsystem. These chunks are small enough to fit into the available local memory. After these points are processed, they are discarded to make space for the next chunk. This download/process/discard process is repeated at each iteration until all points are processed. At this point the results for the current iteration are uploaded to the I/O subsystem. Then, the I/O subsystem broadcasts the partial results to every computing cluster and the next iteration begins.

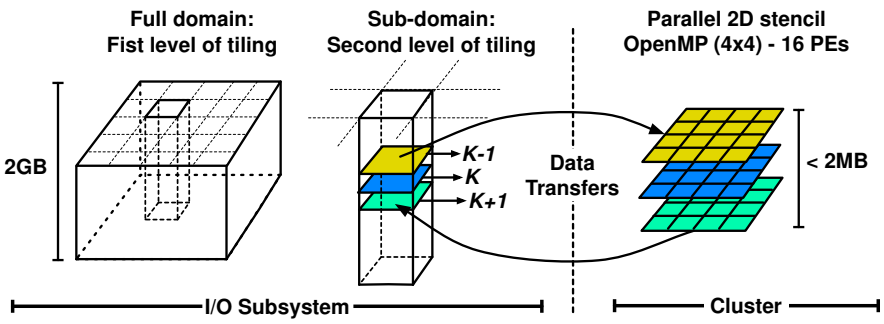

Figure 7: Multi-level tiling strategy to exploit the memory hierarchy of MPPA-256.

4.3. Seismic Wave Propagation

Performing stencil computations on the MPPA-256 processor is a challenging task. This class of numerical kernels has an important demand for memory bandwidth. This makes the efficient use of the low-latency memories distributed among compute clusters indispensable. In contrast to standard x86 processors, in which it is not uncommon to find last-level cache sizes of tens of megabytes, the MPPA-256 has only 32 MB of low-latency memory divided into 2 MB chunks spread throughout the 16 compute clusters.

The 3D data required for seismic wave modeling do not fit in those low-latency memories. Therefore, we need to design efficient master-to-slave and slave-to-master communications to make use of the 2 GB of memory available on the I/O subsystem and carefully overlap communications with computations to mask communication costs. We implement a two-level algorithm that decomposes the problem with respect to the memory available on both the I/O subsystem and the compute clusters. Figure 7 shows the algorithm.

The three dimensional structures corresponding to the velocity and stress fields are allocated on the I/O subsystem to maximize the problem size that can be simulated. Next, we divide the global computational domain into several subdomains corresponding to the number of compute clusters involved in the computation. This decomposition is performed along the horizontal direction providing a first level of data-parallelism. To respect the width of the stencil (fourth-order), we maintain an overlap of two grid points in each direction. These regions, called ghost zones, are updated at each stage of the computation with point-to-point communications between neighboring clusters. This decomposition is rather similar to the description provided in section 3.3.3. Unfortunately, this first level of decomposition is not sufficient as three-dimensional tiles do not fit into the 2 MB of memory available on each compute clusters.

A second level of decomposition is therefore required. This is performed along the vertical direction as we tile each three-dimensional subdomain into 2D slices. This leads to a significant reduction in memory consumption for each cluster but requires maintaining a good balance between the computation and communication. Indeed the procedure relies on a sliding window algorithm that traverses the 3D domains using 2D planes and overlaps data transfers with computations. This could be viewed as an explicit prefetching mechanism as the 2D planes required for the computation at one step are brought to the clusters during the computation performed at previous steps. Additionally, this vertical tiling strategy allows us to benefit from the symmetry of the domain in the horizontal directions. The costly absorbing boundary conditions grid

points located at the bottom of the domain are therefore evenly distributed among the computing clusters.

The number of planes prefetched in advance is parameterizable and its maximum value depends on the problem dimensions and the amount of available memory on each compute cluster. To better exploit the NoC, we carefully select the NoC node on the I/O subsystem with which the compute cluster will communicate. This choice is based on the NoC topology and aims at reducing the number of hops necessary to deliver a message. Moreover, the prefetching scheme also allows us to send less messages containing more data, which has been empirically proven to be more efficient than sending several messages of smaller size. OpenMP directives are employed by clusters to compute 2D problems with up to 16 PEs in parallel.

5. Experimental Results

In this section, we present performance and energy efficiency evaluations for the experimental platforms. These evaluations were conducted by the execution of parallel and distributed versions of the presented applications. We begin by introducing our energy consumption measurement methodology along with the metrics used to analyze the results on all platforms. Then, we compare their energy and computing performance.

5.1. Measurement Methodology

We use two important metrics to compare the energy and computing performance of different multicore and manycore platforms: *time-to-solution* and *energy-to-solution*. Time-to-solution is the time spent to reach a solution for a given problem. In our case, this is the overall execution time of the parallel/distributed version of the applications. Energy-to-solution is the amount of energy spent to reach a solution for a problem. Thus, the ratio between energy-to-solution and time-to-solution yields the average power consumed during the application execution.

Table 1 lists the average power consumed by each one of the platforms used in our experiments during the execution of the parallel and distributed versions of the applications. Even though the Altix UV 2000 features 24 Xeon E5 processors, it consumes less than 24 times the power observed on Xeon E5. This is an expected phenomenon because Xeon E5 runs a multithreaded version of the applications whereas Altix UV 2000 runs their distributed counterparts. Distributed versions experience periods of low processor usage as, for example, those during the task request/response cycle and those related to load imbalance. This will be further discussed in Section 5.5).

	Xeon E5	**Altix UV 2000**	**Exynos 5**	**MPPA-256**
TSP	67.9 W	1,418.4 W	5.3 W	8.3 W
K-Means	61.5 W	1,420.3 W	5.2 W	9.6 W
Ondes3D	57.5 W	1,353.0 W	4.6 W	8.4 W

Table 1: Average power consumption of the 4 processors while running the applications.

The power consumed by each processor was obtained using the same approach. Both Xeon E5 and Altix UV 2000 feature Intel Sandy Bridge microarchitecture, which has Running Average Power Limit (RAPL) energy sensors. This allows us to measure the power consumption of CPU-level components through Machine-Specific Registers (MSRs). We used this approach to

obtain the energy consumption of the whole CPU package including cores and cache memory (named RAPL PKG domain). Similarly, MPPA-256 and Exynos 5 also possess hardware sensors to measure power consumption of the entire chip. Power measurements using this approach are very accurate as shown in [29, 30].

	Small	**Medium**	**Large**
TSP	16 cities	18 cities	20 cities
K-Means	16,384 points 512 centroids	32,768 points 512 centroids	131,072 points 512 centroids
Ondes3D	16x16x16 grid points	48x64x48 grid points	128x128x128 grid points

Table 2: Problem sizes.

We also defined three input problem sizes for all applications (Table 2). These problem sizes were chosen based on the execution time on all platforms and amount of memory needed. For instance, we used a small problem size when running the applications with low thread counts in order to obtain the results in a reasonable time[1]. Each experiment was repeated as many times as needed to ensure a relative error inferior to 2% with 95% statistical confidence using Student's t-distribution.

5.2. Overall Results

Figure 8 compares both time-to-solution (right y-axis) and energy-to-solution (left y-axis) metrics on all processors. Since we used every core of each processor in these experiments, we executed the applications with *large* problem sizes.

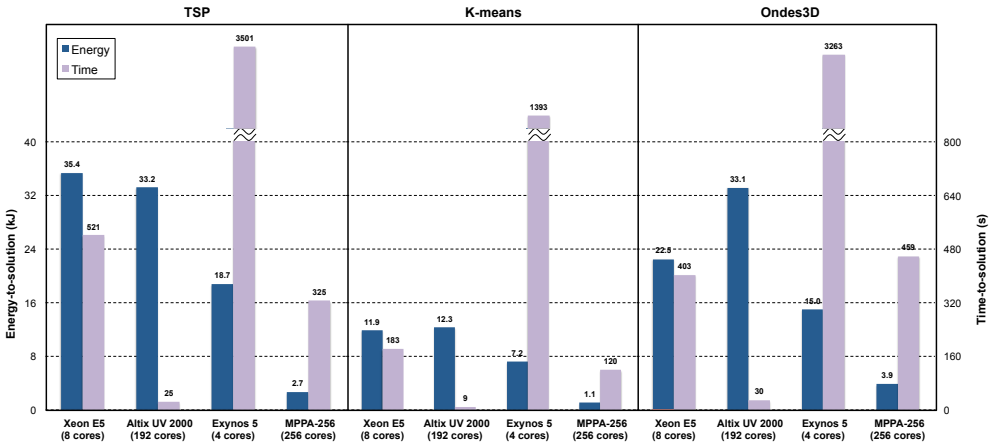

Figure 8: Time and energy-to-solution comparison between multicore, NUMA and manycore processors.

[1]The large problem size along with very low thread counts takes several hours on embedded processors due to their low clock frequency.

Time-to-Solution. As expected, applications on Exynos 5 presented the highest execution times among all platforms, being from 6.7x (TSP) up to 8.1x (Ondes3D) slower than Xeon E5. The reason for that is threefold: (i) it has considerably lower clock frequency than Xeon E5; (ii) Xeon E5 is a performance-centric processor that is tuned far more for speed than for low power consumption; and (iii) Xeon E5 profits from its higher parallelism, since all applications scale considerably well as we increase the number of threads. MPPA-256 presented better execution times than Xeon E5 on TSP and K-Means, being 1.6x and 1.5x faster respectively. Even though the clock frequency of MPPA-256 PEs is lower than that of the Xeon E5 cores, this embedded processor achieved better performance. Once again, this is due to the inherent characteristic of these applications. On TSP, peers only need to broadcast data when a new shortest path is found. On K-Means, peers communicate more often but this application still performs more computation than communication.

An optimized implementation of the seismic wave propagation algorithm has been considered as a baseline for our evaluations. As detailed in Section 3.3, the shared-memory implementation relies on efficient data and thread mapping strategies in order to reduce both the NUMA penalty and the load imbalance. It is well known that stencil-based computations like finite differences method applied to seismic wave propagation achieve a low fraction of the peak performance on standard processors such as x86. This is mainly due to the huge demand for memory bandwidth typical for this class of algorithms. On average, 30% of peak performance is reported for such implementations [31]. A detailed characterization of this behavior taking into consideration both the architecture and the algorithms is given by the roofline model [32]. Nonetheless, a more detailed discussion on the peak performance on the MPPA-256 architecture would require revisiting the roofline model which is out the scope of this paper. Our analysis confirmed our expectation that an important share of Ondes3D execution time is spent in communications. Although the prefetching scheme considerably hides the communication costs on MPPA-256, the latency and bandwidth of the NoC still hurts its performance, resulting in an execution time approximately 10% worse on MPPA-256 compared to Xeon E5. Not surprisingly, Altix UV 2000 platform presented the best execution times, since it has 24 performance optimized general- purpose multicore processors. We further discuss the scalability results on Altix UV 2000 and MPPA-256 in Section 5.4.

Energy-to-Solution. Both Exynos 5 and MPPA-256 presented better energy-to-solution than the other platforms. However, the low degree of parallelism available on the ARM processor was a clear disadvantage for Exynos 5. Even though this processor consumes less power than the others, it ends up executing the applications during a longer period of time. This results in a higher energy consumption compared to MPPA-256. Overall, MPPA-256 achieved the best energy-to-solution results, reducing the energy consumed by other platforms on TSP, K-Means and Ondes3D in at least 6.9x, 6.5x and 3.8x, respectively.

5.3. Energy Efficiency

In the previous section, we showed that MPPA-256 presented the best energy-to-solution results among all platforms. The main reason is that MPPA-256 offers a high parallelism and yet has a low power consumption. In this section, we intend to look in more detail at the energy efficiency of all platforms when we vary the number of cores. We first compare the energy-to-solution of all applications when varying the number of cores from 1 to the maximum number of cores available in each processor (Figure 9a). In other words, we compare the energy-to-solution obtained with a single processor of Altix UV 2000 (which is actually the Xeon E5), Exynos 5 and a single compute cluster of MPPA-256 (in this case, we vary the number of PEs). For these tests,

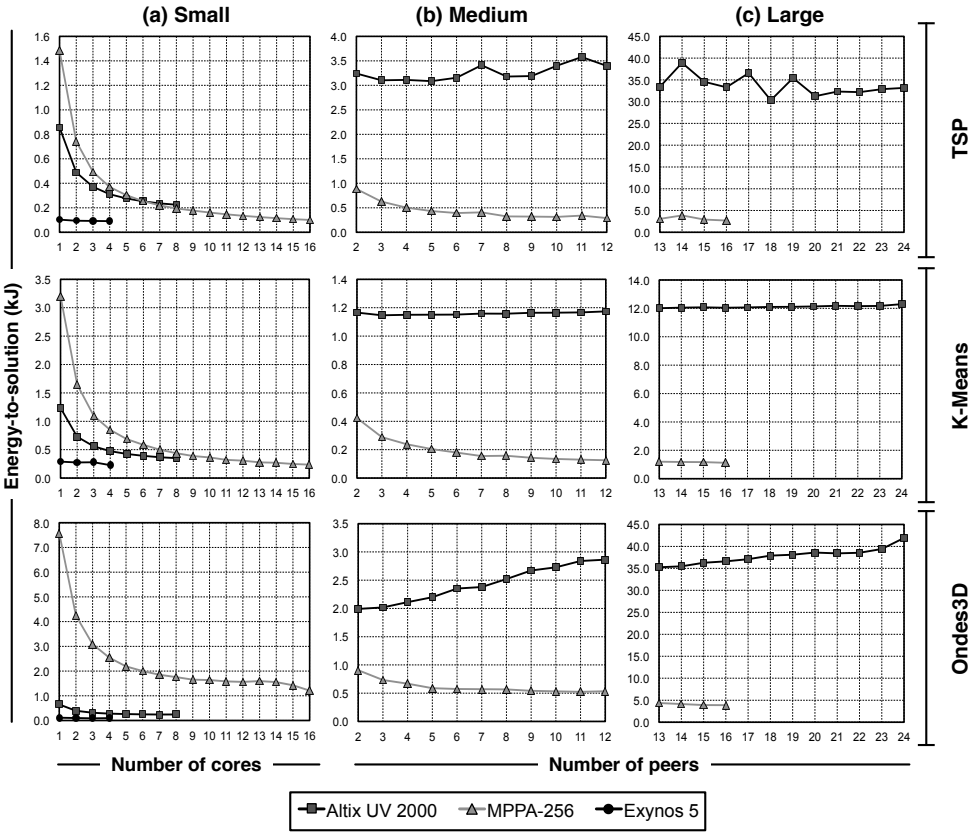

Figure 9: Energy-to-solution comparison on all platforms with three problem sizes.

we used a small problem size due to time constraints. Then, we compare the energy-to-solution on Altix UV 2000 and MPPA-256 when varying the number of peers (*i.e.*, processors on Altix UV 2000 and compute clusters on MPPA-256), while using the maximum number of cores available on each Altix UV 2000 processor (8 cores) and MPPA-256 clusters (16 PEs). We used a medium problem size to compare the energy-to-solution from 2 to 12 peers (Figure 9b) and a large problem size for more than 12 peers (Figure 9c). Note that the MPPA-256 architecture is limited to 16 peers, therefore we only show the results for more than 16 peers on Altix UV 2000.

Varying the Number of Cores. Exynos 5 achieved the best energy-to-solution for the small problem size in all applications. Figure 9a shows this behavior. The reasons for that are twofold. First, Exynos 5 is the least power hungry processor among all the experimental platforms. Second, the problem size used in this experiment is too small to scale past 4 cores. When we compare the energy efficiency of a single Altix UV 2000 processor against a single MPPA-256 cluster, we notice that Altix UV 2000 outperformed MPPA-256 with low core counts on TSP and K-Means. For more than 8 cores, however, the MPPA-256 cluster outperformed the Altix UV 2000 processor. This comes from the fact that the power consumed by a single Altix UV 2000 processor considerably increased as we increased the number of used cores whereas the power consumed by a single

MPPA-256 cluster remained practically unchanged. The only exception occurred on Ondes3D. In this case, MPPA-256 consumed much more energy than the Altix UV 2000 processor because communications on MPPA-256 could not be overlapped with computations using small problem sizes on Ondes3D. Moreover, the NoC bandwidth achieved in this case is poor, since we only perform 1-to-1 communications between the I/O subsystem and a single cluster.

Varying the Number of Peers. The gap between the energy consumed by Altix UV 2000 and MPPA-256 became more important as we increased the number of peers. From 2 to 12 peers (Figure 9b), MPPA-256 consumed at least 2.3x less energy than Altix UV 2000. This gap was even larger from 13 to 16 peers with a large problem size (Figure 9c): in this case, MPPA-256 consumed on average ∼11x less energy than Altix UV 2000. Once again, the rationale behind that comes from the high energy cost associated to the Altix UV 2000 processors: adding one Xeon E5 processor usually increases the overall power consumption of Altix UV 2000 by ∼60 W on average whereas adding one MPPA-256 cluster increases the overall power consumption of MPPA-256 by ∼0.3 W.

5.4. Scalability

So far, we have only compared the energy-to-solution of MPPA-256 and Altix UV 2000, showing that the former consumed far less energy than latter to solve the same problems. Figure 10 illustrates the time-to-solution gap between them for a *medium* problem size when considering an equal number of resources (peers) as well as the comparative speedup between the architectures. The speedup calculation was based on the effect that an increase on the number of peers has on performance. For that reason, and to maintain consistency throughout our comparisons, we employed as the baseline the execution time of the multi-threaded algorithms using a single peer. In other words, we compared the performance of the distributed algorithm using different numbers of fully utilized peers to that of a parallel version using all the resources of single peer, *i.e.*, with no inter-peer communications. We measure, therefore, the scalability of the distributed version of the algorithms and not that of the of the multi-threaded version. Detailed scalability evaluation analysis for the multi-threaded algorithms can be found in the base works presented in Section 3.

Overall, the distributed version of the applications scaled considerably well and execution times showed similar trends on both platforms. However, Altix UV 2000 was from 9x up to 13x faster than MPPA-256. This result was expected, since peers mean processors running at full speed (2.4 GHz) on Altix UV 2000 whereas they represent blocks (compute clusters) of the MPPA-256 processor running at 400 MHz. In other words, we are comparing sets of entire processors on Altix UV 2000 against subsets of a single MPPA-256 processor. We also observed similar performance gaps with other problem sizes.

The near-linear speedups of TSP and K-Means on both architectures show that, although the actual implementations of the evaluated applications were adapted to accommodate each platform's idiosyncrasies, they in fact display good and similar scalability. The exception of Ondes3D can be explained by the amount communications performed by this algorithm. While TSP and K-Means are CPU-bound and communicate at regular but not so frequent intervals, communication on the Ondes3D is much more intensive. The weak scalability past six peers demonstrates the toll imposed by these communications to the NUMA interconnections on Altix UV 2000 and to the NoC on MPPA-256.

Moreover, in order to avoid NUMA effects on the Altix UV 2000 platform and ensure good execution performance, we had to employ some additional runtime optimizations. For all applications we employed thread-pinning [33]. Since the TSP was implemented using POSIX threads,

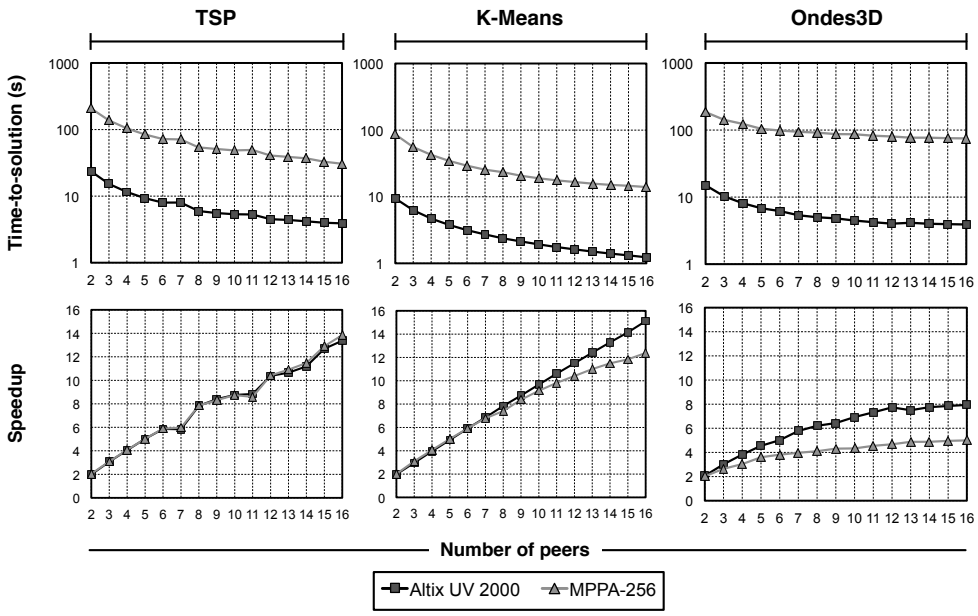

Figure 10: Time-to-solution and speedup comparison between Altix UV 2000 and MPPA-256.

we used Linux's specific system calls to ensure that threads would not be migrated during the execution of the application. K-Means and Ondes3D used OpenMP to implement parallelism, therefore in these cases we employed the `GOMP_CPU_AFFINITY` environment variable to ensure threads were correctly bound to the available cores. Additionally, we specially modified the initialization phase of the applications so that the first-touch strategy (Linux's default) could suitably place the allocated memory on the NUMA nodes. For that, each application would make each thread initialize (either with the actual value or with a dummy value when initialization had to be centralized) each private region of memory.

5.5. Irregularity

In the last sections, we analyzed the energy-to-solution and the scalability of the distributed versions of all applications on both Altix UV 2000 and MPPA-256. The results showed that our distributed solutions scaled well, which indicates that the inherent irregularities of each application were satisfactorily handled.

However, Figure 9a and Figure 9b reveal some points where the energy-to-solution abruptly increases (*e.g.,* from 6 to 7 and from 13 to 14 peers) in the TSP. In these cases, the addition of a peer incurred performance losses (higher execution times). In order to investigate this peculiar behavior, we traced the execution of the distributed version of TSP. Figure 11 shows the execution traces obtained on Altix UV 2000 while running the TSP with 14 peers.

Figure 11a shows a global view of the execution aggregated per peer. At the beginning (Figure 11b), one thread in each peer asks a master peer for partitions and starts populating the local pool of tasks. As tasks become available, other threads in the same peer can start the computation. Once the thread assigned to populate the local pool of tasks finishes its job, it also starts the computation. Afterwards, as the peers run out of work, they ask a master peer

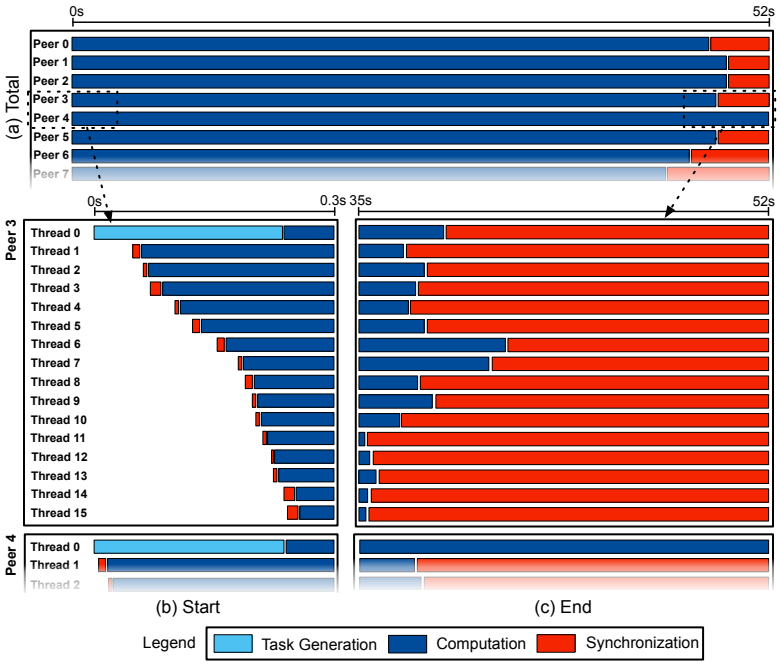

Figure 11: Execution traces of TSP on Altix UV 2000.

for more partitions. This strategy works fairly well throughout most of the execution. However, in some cases the load may become imbalanced at the end of the execution. Figure 11c shows what happens inside the peers. In this case, only peer 4 keeps processing for a long time while other peers are running out of tasks. This happens because the last task associated to Thread 0 from Peer 4 takes much longer to be completed. This problem can be reduced with a work-stealing strategy inside each peer, so that threads can steal segments of this big task to improve parallelism. However, we leave this optimization for a future work.

6. Related Work

Many works have been focusing on analyzing the performance and energy efficiency of low-power multicore processors. *Padoin et al.* [34] compared an ARM Cortex-A9 1.0 GHz dual-core processor from Texas Instruments to two multiprocessors: one composed of two quad-core 2.4 GHz Intel Xeon E5 processors and the other composed of four 2.0 GHz 8-core Xeon X7 processors. They analyzed different metrics such as time-to-solution, peak power and energy-to-solution using 6 benchmarks from the NAS Parallel Benchmarks (NPB). Their results showed that the ARM processor outperforms Xeon X7, considering the energy-to-solution metric, for most of the analyzed benchmarks. However, the Xeon E5 had the best energy-to-solution among the three processors.

Göddeke et al. [3] also conducted a comparison between ARM and x86 architectures using different classes of numerical solution methods for partial differential equations. They evaluated

weak and strong scalability on a cluster of 96 ARM Cortex-A9 dual-core processors and demonstrated that the ARM-based cluster can be more efficient in terms of energy-to-solution compared to a cluster of Intel Xeon X5550 processors. Similarly, *Ou et al.* [35] compared the energy efficiency of a ARM-based cluster against an Intel X86 workstation on three applications: a web server, an in-memory database and a video transcoding. They concluded that energy/efficiency ratio of the ARM cluster against the Intel workstation depend on the application and may vary from 1.21 up to 9.5.

Recent works are aiming to assess whether light-weight manycore processors can be used as basic blocks for future HPC architectures. *Totoni et al.* [36] compared the power and performance of Intel's Single-Chip Cloud Computer (SCC) to other types of CPUs and GPUs. The analysis was based on a set of parallel applications implemented with the Charm++ programming model. They showed that there is no single best solution that always achieves the best trade-off between power and performance. However, the results obtained with the Intel SCC suggest that manycores are an opportunity for the future. *Morari et al.* [37] proposed an optimized implementation of radix sort for the Tilera TILEPro64 manycore processor. The results showed that the their solution for TILEPro64 provides much better energy efficiency than an general-purpose multicore processor (Intel Xeon W5590) and comparable energy efficiency with respect to a GPU NVIDIA Tesla C2070. *Gharaibeh et al.* [38] showed how a synergistic use of CPUs and GPUs can improve the overall energy-to-solution on large-scale graph processing. In particular their approach is similar to our seismic wave simulation application in the sense that they map the problem (the input graph) to the interconnection topology of the underlying hardware platform.

Castro et al. [9] showed that MPPA-256 can be very competitive considering both performance and energy efficiency for a fairly parallelizible problem: the Traveling- Salesman Problem (TSP). The results indicated that the MPPA-256 may achieve better performance than an Intel Xeon processor with 8 CPU cores (16 threads with Hyper-Threading) running at 2.40GHz while consuming approximately 13 times less energy. Using a slightly different approach, *Aubry et al.* [7] compared the performance of an Intel Core i7-3820 processor with the MPPA-256. Their application, an H.264 video encoder, was implemented using a dataflow language that offers direct automatic mapping to MPPA-256. Their findings show that the performance of these traditional processors is on par with the performance of the MPPA-256 embedded processor which provides 6.4 times better energy efficiency.

Unlike the previous works, we have focused on the passage from multicores to manycores from the perspective of three irregular applications. We pointed out some of the programming issues that must be considered when developing parallel applications to manycores. Moreover, we analyzed the performance and energy consumption of these applications on a set of state-of-the-art multicore and manycore platforms, ranging from low-power processors to general-purpose processors.

7. Conclusion and Future Work

In this work we analyzed the performance and the energy-efficiency of four different hardware platforms. For that we employed applications with three different behaviors. The experimental results obtained during this research corroborated the widely accepted practice on the high-performance research domain that considers an appropriate appreciation of the underlying hardware idiosyncrasies essential to obtain good performance and energy efficiency.

Manycore processors seem to be the trend in the development of faster energy-efficient processors. The efficient use of a light-weight manycore processor demands adaptations to the

application code so that it can efficiently use the whole chip. Often these modifications are not trivial. For instance, the MPPA-256 platform has a strong constraint on the amount of available local memory. For this reason we had to implement specific tiling mechanisms to be able to deal with real-world scenarios (Ondes3D) and arbitrary problem sizes (K-Means). In the case of Ondes3D, we also needed to implement a prefetching mechanism to overlap communications with computation. On TSP, on the other hand, modifications were similar to those needed to port an application to the MPI paradigm. However, the absence of a coherent cache considerably increased the implementation complexity, requiring the use of full memory barriers or proprietary system calls designed to completely bypass the cache. On the Altix UV 2000, we had to employ thread pinning and memory placement to ensure performance.

As it is often the case for parallel applications, such modifications tend to introduce redundant computations and extra communications in order to improve the parallelism of the whole solution. Not every application is suitable to this kind of modification and, in the worst case scenario, a strictly serial application might be limited to the performance of a single core. For these three classes of applications (CPU-bound, memory- bound and mixed) we showed that highly-parallel platforms can be very competitive, even if the application is irregular in nature. Our results showed that MPPA-256 may achieve better performance than a traditional general-purpose multicore processor (Xeon E5) on CPU-bound and mixed workloads. For a memory-bound workload (Ondes3D) Xeon E5 performed better than MPPA-256. Although Altix UV 2000 presented the best performance results among all platforms it also presented a higher energy consumption when communication became more important (K-Means and Ondes3D), however it still showed an energy efficiency similar to Xeon E5. MPPA-256 presented the best energy efficiency among all platforms, reducing the energy consumed on TSP, K-Means and Ondes3D by at least 6.9x, 6.5x and 3.8x, respectively.

This work can be extended in two directions. First, we compared the energy efficiency of state-of-the-art Intel-based platforms (Xeon E5 and Altix UV 2000) to other low-power platforms (MPPA-256 and Exynos 5). These specific Intel-based platforms are optimized for performance, not for low energy consumption. As future work, we plan to compare the performance of these low-power processors to those based on low-power Intel processors such as the Intel Atom and the mobile versions of the Sandy Bridge architecture. Next, we intend to compare the performance and energy efficiency of lightweight manycore processors such as MPPA-256 to other manycore processors such as GPUs and the Intel Xeon Phi.

Acknowledgments

The authors would like to thank CAPES for funding this research through project CAPES/Cofecub 660/10 and through a PNPD/CAPES scholarship. This work was done in the context of LICIA and Mont-Blanc project (funded from the European Union's Seventh Framework Programme under grant agreement #288777), being partially supported by CNPq, FAPEMIG, FAPERGS and INRIA.

References

[1] J. Larus, Spending Moore's Dividend, Communications of the ACM 52 (2009) 62–69.

[2] D. Brooks, P. Bose, S.E. Schuster *et. al*, Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors, IEEE Micro 20 (2000) 26–44.

[3] D. Göddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, A. Ramirez, Energy Efficiency vs. Performance of the Numerical Solution of PDEs: An Application Study on a Low-power ARM-based Cluster, J. Comput. Physics 237 (2013) 132–150.

[4] N. Rajovic *et. al*, The Low-Power Architecture Approach Towards Exascale Computing, in: Workshop on Scalable Algorithms for Large-Scale Systems (ScalA), ACM, New York, USA, 2011, pp. 1–2.

[5] T. Fleig, O. Mattes, W. Karl, Evaluation of Adaptive Memory Management Techniques on the Tilera TILE-Gx Platform, in: International Conference on Architecture of Computing Systems (ARCS), VDE VERLAG, Luebeck, Deutschland, 2014, pp. 88–96.

[6] C. L. Benoît Dupont de Dinechin and Pierre Guironnet de Massasa, Guillaume Lagera, B. Orgogozoa, J. Reyberta, T. Strudela, A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor, in: Intl. Conference on Computational Science (ICCS), volume 18, Elsevier, Barcelona, Spain, 2013, pp. 1654–1663.

[7] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bobin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, R. Sirdey, Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor, in: International Conference on Computational Science (ICCS), volume 18, Elsevier, Barcelona, Spain, 2013, pp. 1624–1633.

[8] G. Laporte, The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms, European Journal of Operational Research 59 (1992) 231–247.

[9] M. Castro, E. Francesquini, T. M. Nguélé, J.-F. Méhaut, Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application, in: Workshop on Irregular Applications: Architectures & Algorithms (IA^3) - Supercomputing Conference (SC), ACM, Denver, EUA, 2013, p. Article No. 5.

[10] H. Li, H. L. Sudarsan, M. Stumm, K. C. Sevcik, Locality and Loop Scheduling on NUMA Multiprocessors, in: International Conference on Parallel Processing (ICPP), volume 2, IEEE Computer Society, Syracuse, USA, 1993, pp. 140–147.

[11] R. Xu, I. Wunsch, D., Survey of clustering algorithms, Neural Networks, IEEE Transactions on 16 (2005) 645–678.

[12] L. Kaufman, P. J. Rousseeuw, Finding groups in data: an introduction to cluster analysis, John Wiley and Sons, New York, 1990.

[13] A. K. Jain, R. C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[14] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, A. Wu, An efficient k-means clustering algorithm: analysis and implementation, Pattern Analysis and Machine Intelligence, IEEE Transactions on 24 (2002) 881–892.

[15] I. Dhillon, D. Modha, A data-clustering algorithm on distributed memory multiprocessors, in: M. Zaki, C.-T. Ho (Eds.), Large-Scale Parallel Data Mining, volume 1759 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2000, pp. 245–260.

[16] S. Rao, E. V. Prasad, N. B. Venkateswarlu, A scalable k-means clustering algorithm on multi-core architecture, in: Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on, pp. 1–9.

[17] L. Rodrigues, L. Zarate, C. Nobre, H. Freitas, Parallel and distributed kmeans to identify the translation initiation site of proteins, in: Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on, pp. 1639–1645.

[18] P. Moczoa, J. O. A. Robertssonb, L. Eisnerc, The Finite-difference Time-domain method for Modeling of Seismic Wave Propagation, in: Advances in Wave Propagation in Heterogeneous Media, volume 48 of *Advances in Geophysics*, Elsevier - Academic Press, 2007, pp. 421–516.

[19] H. Aochi, T. Ulrich, A. Ducellier, F. Dupros, D. Michea, Finite difference simulations of seismic wave propagation for understanding earthquake physics and predicting ground motions: Advances and challenges, in: Journal of Physics: Conference Series, volume 454, IOP Publishing, p. 012010.

[20] R. Madariaga, Dynamics of an expanding circular fault, Bulletin of the Seismological Society of America 66 (1976) 639–666.

[21] F. Collino, Perfectly matched absorbing layers for the paraxial equations, Journal of Computational Physics 131 (1997) 164–180.

[22] F. Dupros, H.-T. Do, H. Aochi, On scalability issues of the elastodynamics equations on multicore platforms, in: International Conference on Computational Science (ICCS), volume 18 of *Procedia Computer Science*, Elsevier, Barcelona, Spain, 2013, pp. 1226–1234.

[23] F. Dupros, C. Pousa, A. Carissimi, J.-F. Méhaut, Parallel Simulations of Seismic Wave Propagation on NUMA Architectures, in: International Parallel Computing conference (ParCo), volume 19 of *Advances in Parallel Computing*, IOS Press, Lyon, France, 2010, pp. 67–74.

[24] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, P. Maechling, Scalable earthquake simulation on petascale supercomputers, in: High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, pp. 1–20.

[25] T. Furumura, L. Chen, Parallel simulation of strong ground motions during recent and historical damaging earthquakes in tokyo, japan, Parallel Computing 31 (2005) 149 – 165. Parallel Graphics and Visualization.

[26] R. K. Tesser, L. L. Pilla, F. Dupros, P. O. A. Navaux, J.-F. Méhaut, C. Mendes, Improving the performance of seismic wave simulations with dynamic load balancing, in: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE Computer Society, Turin, Italy, 2014, pp. 196–203.

[27] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, J. Roman, Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation, in: International Conference on Computational Science and Engineering, São Paulo, Brazil, pp. 253–260.

[28] A. Gursoy, Data decomposition for parallel k-means clustering, in: R. Wyrzykowski, J. Dongarra, M. Paprzycki, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics, volume 3019 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2004, pp. 241–248.

[29] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, D. Rajwan, Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge, IEEE Micro 32 (2012) 20–27.

[30] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring Energy Consumption for Short Code Paths Using RAPL, ACM Sigmetrics Performance Evaluation Review 40 (2012) 13–17.

[31] C. Andreolli, P. Thierry, L. Borges, C. Yount, G. Skinner, Genetic Algorithm Based Auto-Tuning of Seismic Applications on Multi and Manycore Computers, in: EAGE Workshop on High Performance Computing for Upstream, Amsterdam, Netherlands. September, 2014. (To Appear).

[32] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, Optimization and performance modeling of stencil computations on modern microprocessors, SIAM Review 51 (2009) 129–159.

[33] R. Love, K. Korner, CPU Affinity, Linux Journal, (111), 2003.

[34] E. L. Padoin, D. A. G. de Oliveira, P. Velho, P. Navaux, Time-to-Solution and Energy-to-Solution: A Comparison between ARM and Xeon, in: Workshop on Applications for Multi-Core Architectures (WAMCA), IEEE Computer Society, New York, USA, 2012, pp. 48–53.

[35] Z. Ou, B. Pang, Y. Deng, J. Nurminen, A. Ylä-Jääski, P. Hui, Energy and Cost-Efficiency Analysis of ARM-Based Clusters, in: IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE Computer Society, Ottawa, Canada, 2012, pp. 115–123.

[36] E. Totoni, B. Behzad *et. al*, Comparing the Power and Performance of Intel's SCC to State-of-the-Art CPUs and GPUs, in: IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE Computer Society, New Brunswick, Canada, 2012, pp. 78–87.

[37] A. Morari, A. Tumeo, O. Villa, S. Secchi, M. Valero, Efficient sorting on the Tilera manycore architecture, in: IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE Computer Society, New York, USA, 2012, pp. 171–178.

[38] A. Gharaibeh, E. Santos-Neto, L. B. a. Costa, M. Ripeanu, The energy case for graph processing on hybrid cpu and gpu systems, in: Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms, IA3 '13, ACM, New York, NY, USA, 2013, pp. 2:1–2:8.