



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

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

► To cite this version:

Emilio Francesquini, Márcio Castro, Pedro Penna, Fabrice Dupros, Henrique 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*, Elsevier, 2015, 76, pp. 32-48. <10.1016/j.jpdc.2014.11.002>. <hal-01092325>

HAL Id: hal-01092325

<https://hal-brgm.archives-ouvertes.fr/hal-01092325>

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

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)

Preprint submitted to Journal of Parallel and Distributed Computing

August 21, 2014

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 sufficient 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 Tiler 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

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

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

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

74 2. Experimental Platforms

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

77 2.1. General-Purpose

78 **Xeon E5.** The Intel Xeon E5 is a 64-bit x86-64 processor. In this study we used a Xeon E5-
79 4640 Sandy Bridge-EP, which has 8 physical cores running at 2.40 GHz. Each core has 32 KB
80 instruction and 32 KB data L1 caches and 256 KB of L2 cache. All the 8 cores share a 20 MB
81 L3 cache and the platform has 32 GB of DDR3 memory.

82 **Altix UV 2000.** SGI Altix UV 2000 (Figure 1) is a Non-Uniform Memory Access (NUMA)
83 platform designed by SGI. The platform is composed of 24 NUMA nodes. Each node has a
84 Xeon E5-4640 Sandy Bridge-EP processor (with the same specifications of the Xeon E5 platform)
85 and 32 GB of DDR3 memory. This memory is shared in a ccNUMA fashion through SGI's
86 proprietary NUMalink6 (bidirectional). This high-speed interconnection provides a point-to-
87 point bandwidth of 6.7 GB/s per direction. Overall, this platform has 192 physical cores.

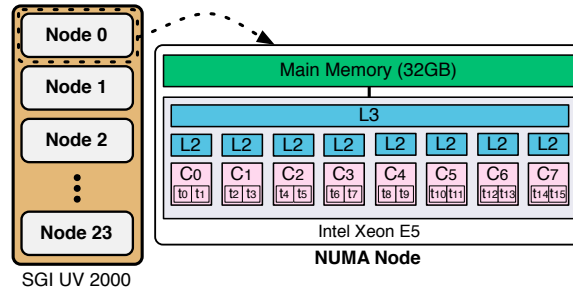


Figure 1: A simplified view of Altix UV 2000.

88 *2.2. Embedded*

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

100 *2.3. Manycore*

101 **MPPA-256.** Kalray MPPA-256 is a single-chip manycore processor developed by Kalray that
 102 integrates 256 user cores and 32 system cores. It uses 28nm CMOS technology running at
 103 400 MHz. These cores are distributed across 16 compute clusters and 4 I/O subsystems that
 104 communicate through data and control NoCs. This processor targets parallel applications whose
 105 programming models fall within the following classes: Kahn Process Networks (KPN), as moti-
 106 vated by media processing; SPMD, traditionally used for numerical kernels; and time-triggered
 107 control systems [7, 6].

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

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

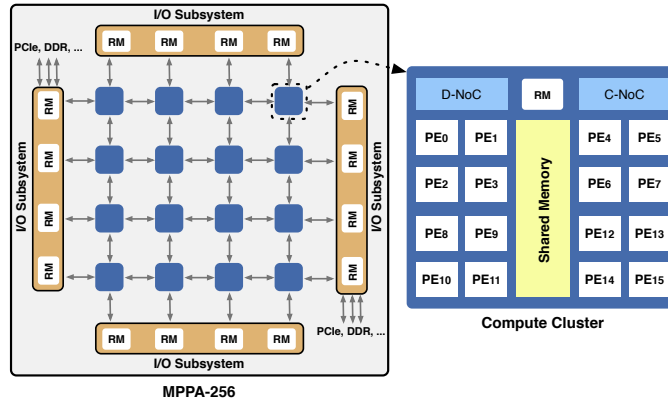


Figure 2: A simplified view of the MPPA-256.

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

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

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

135 2.4. Synthesis

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

146 **3. Case Studies**

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

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

159 *3.1. Traveling-Salesman Problem*

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

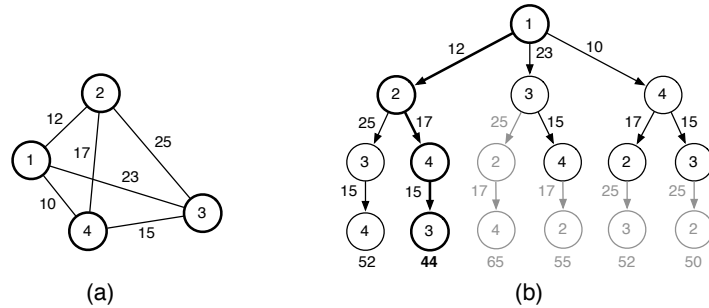


Figure 3: Example of TSP with 4 cities.

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

174 3.1.1. Sequential Algorithm

175 The sequential version of the algorithm is based on the branch and bound method using brute
 176 force. Algorithm 3.1 outlines this solution. It takes as input the number of cities and a cost
 177 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  $cities = \emptyset$ 
    then return ( $current\_cost$ )
  for each  $i \in cities$ 
    do
       $new\_cost \leftarrow current\_cost + costs[last\_city, i]$ 
      if  $new\_cost < min\_path$ 
        then  $\begin{cases} new\_min \leftarrow TSP\_SOLVE(i, new\_cost, cities \setminus \{i\}) \\ ATOMIC\_UPDATE\_IF\_LESS(min\_path, new\_min) \end{cases}$ 

main
 $min\_path \leftarrow \infty$ 
TSP_SOLVE(1, 0, {2, 3, ...,  $n\_cities$ })
output ( $min\_path$ )

```

178 This algorithm does a depth-first search looking for the shortest path and has complexity
 179 $O(n!)$. It does not explore paths that are already known to be longer than the best path found
 180 so far, therefore discarding fruitless branches. Figure 3b shows this behavior. The shaded edges
 181 are those that the algorithm does not follow, since a possible solution that includes them would
 182 be more costly than the one it has already identified. This simple pruning technique greatly
 183 improves the performance of the algorithm. However, it also introduces irregularities into the
 184 search space. The search depth needed to discard one of the branches depends on the order in
 185 which the branches were searched.

186 3.1.2. Multi-threaded Algorithm

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

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

196 Algorithm 3.2 shows the pseudo-code for this approach. This algorithm also receives as a
197 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  $\left\{ \begin{array}{l} task \leftarrow (last\_city, current\_cost, cities) \\ ENQUEUE\_TASK(queue, task) \end{array} \right.$ 
  else  $\left\{ \begin{array}{l} \textbf{for each } i \in cities \\ \quad \textbf{if } last\_city = none \\ \quad \quad \textbf{then } last\_cost \leftarrow 0 \\ \quad \quad \textbf{else } last\_cost \leftarrow costs[last\_city, i] \\ \quad \quad new\_cost \leftarrow curr\_cost + last\_cost \\ \quad \quad GENERATE\_TASKS(n\_hops + 1, i, new\_cost, cities \setminus \{i\}) \end{array} \right.$ 

procedure DO_WORK()
  while  $queue \neq \emptyset$ 
  do  $\left\{ \begin{array}{l} (last\_city, current\_cost, cities) \leftarrow ATOMIC\_DEQUEUE(queue) \\ TSP\_SOLVE(last\_city, current\_cost, cities) \end{array} \right.$ 

main
   $min\_path \leftarrow \infty$ 
  GENERATE_TASKS(0, none, 0, {1, 2, ...,  $n\_cities$ })
  for  $i \leftarrow 1$  to  $n\_threads$ 
  do SPAWN_THREAD(DO_WORK())
  WAIT_EVERY_CHILD_THREAD()
  output ( $min\_path$ )

```

198 3.1.3. Distributed Algorithm

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

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

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

224 3.2. The K-Means Clustering Problem

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

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

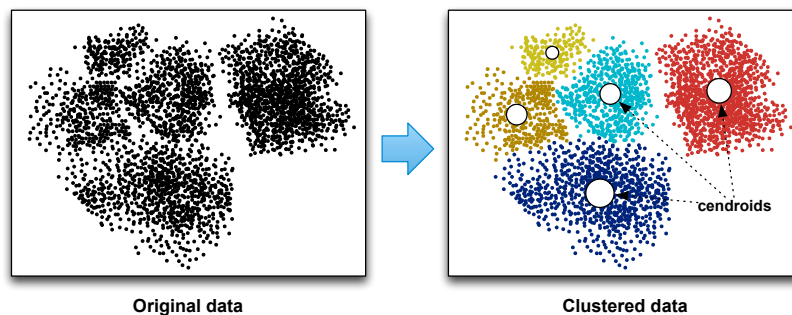


Figure 4: An example of K-Means with 5 partitions.

232 Several distinct heuristics have been proposed to address the K-Means clustering problem
 233 [12, 13]. One of the most widely employed is the Lloyd's algorithm [14], also known as K-
 234 Means algorithm. Such heuristic is based on an iterative strategy that finds a locally minimum
 235 solution for the problem. In our work we used this algorithm as a case study. In the following
 236 subsections, we first present the sequential version of the algorithm and then we introduce and
 237 explain our parallel and distributed versions.

238 3.2.1. *Sequential Algorithm*

239 The sequential version of K-Means is shown in Algorithm 3.3. The main idea is to use the
240 notion of minimum Euclidean distance to iteratively partition the data points. The algorithm takes
241 as input the set of data *points*, the number of partitions *k*, and the minimum accepted distance,
242 *mindistance*, between each point and the partition's center (centroids). Upon completion, the
243 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)
```

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

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

254 3.2.2. *Multi-threaded Algorithm*

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

259 own range of points into the k partitions. In the second phase, each thread works in its own range
 260 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  $i \leftarrow 0$  to  $(t - 1)$ 
    do {
       $work.first\_point \leftarrow i \times num\_points \div t$ 
       $work.last\_point \leftarrow (i + 1) \times num\_points \div t$ 
       $work.first\_partition \leftarrow i \times k \div t$ 
       $work.last\_partition \leftarrow (i + 1) \times k \div t$ 
      SPAWN_THREAD(DO_KMEANS( $work$ ))
    }
  WAIT_EVERY_CHILD_THREAD()
  return ( $partitions$ )
  
```

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

266 3.2.3. Distributed Algorithm

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

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

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

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

297 3.3. Seismic Wave Propagation

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

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

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

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

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

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

312 3.3.1. Sequential Algorithm

313 As mentioned before, the finite differences method is one of the most popular techniques
 314 to solve the elastodynamics equations and to simulate the propagation of seismic waves [18,
 315 19]. One of the key features of this scheme is the introduction of a staggered-grid [20] for the
 316 discretization of the seismic wave equation.

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

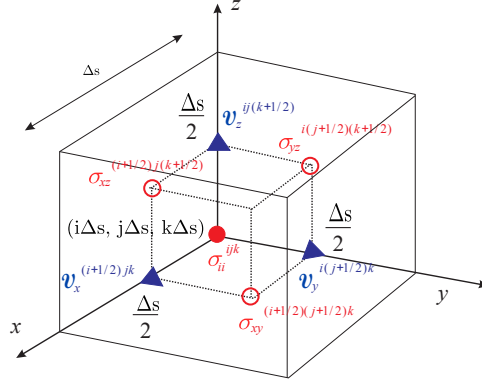


Figure 5: Elementary 3D cell of the staggered grid and distribution of the stress (σ) and the velocity (v) components.

321 The computational procedure is described in Algorithm 3.5. Inside the time step loop, the
 322 first triple nested loop is devoted to the computation of the velocity components, and the second
 323 loop reuses the velocity results of the previous time step to update the stress field. For instance,
 324 the stencil applied for the computation of the velocity component in the x -direction is given
 325 by Equation 4. Exponents i, j, k indicate the spatial direction, $\sigma^{ijk} = \sigma(i\Delta s, j\Delta s, k\Delta s)$, Δs
 326 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( $v_x, v_y, v_z$ )
    }
  }

```

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

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

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

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

341 3.3.2. Multi-threaded Algorithm

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

354 3.3.3. Distributed Algorithm

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

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

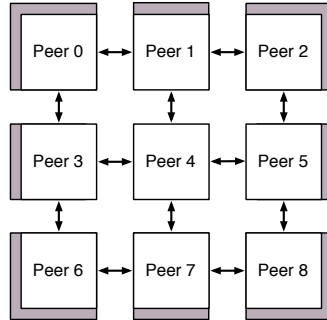


Figure 6: Distributed implementation of the seismic wave propagation kernel with a 3×3 decomposition. The gray regions represent the irregular absorbing boundary condition layers.

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

373 3.4. General Considerations

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

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

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

398 By taking into consideration these three applications and the three distinct hardware archi-
399 tectures used in our experimental evaluations, we can draw comprehensive (yet straightforward)
400 conclusions about execution performance and energy efficiency of these applications on the dis-
401 tinct hardware platform classes.

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

405 4. Adapting Irregular Applications for Manycores

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

410 4.1. TSP

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

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

421 Unfortunately, this common solution is not appropriate to the MPPA-256 platform since it
422 does not possess coherent caches. Despite the fact that the update of *min_path* works as it should
423 (on the MPPA-256 platform the POSIX lock implementation invalidates the whole cache) and the
424 final path length is correct, each one of the worker threads might be using a stale value of the
425 *min_path* variable for a long time (in the worst case until the end of its execution) and wasting
426 time on fruitless branches of the search tree. This means that, although correct, the execution
427 might be severely slowed down. To correct it, we have used platform specific instructions that
428 allow us direct access to the local memory of the cluster, bypassing the cache (`__builtin_k1_`
429 `lwu` and `__builtin_k1_swu` to load/store data from/to the local memory, respectively). The cost
430 of reading the variable in this manner is clearly higher than using the value stored in the caches
431 (reading from memory takes 8 cycles whereas reading from cache takes at most 2 cycles). Yet,
432 the performance improvement due to the better pruning of the search tree largely outweighs the
433 additional cost.

434 In order to efficiently exploit the MPPA-256 platform, we needed to use every cluster of the
435 chip. These clusters do not have a global memory space hence the need for the distributed version
436 of the algorithm. Conversely, Altix UV 2000 platform has a global memory space, however, as the
437 communications between the NUMA nodes are done through the NUMALink6 interconnection,
438 we can make a better use of this system by keeping the memory near the threads that use it and
439 avoid using the link to perform anything but global synchronizations and *min_path* propagation.
440 The distributed algorithm fits perfectly in this scenario.

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

453 4.2. *K-Means*

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

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

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

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

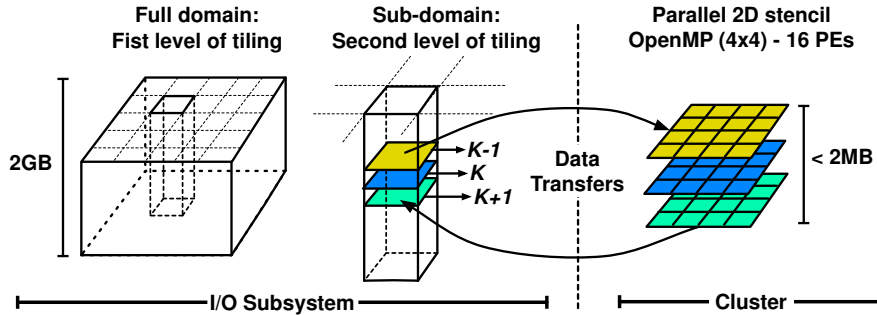


Figure 7: Multi-level tiling strategy to exploit the memory hierarchy of MPPA-256.

484 4.3. *Seismic Wave Propagation*

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

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

497 The three dimensional structures corresponding to the velocity and stress fields are allocated
 498 on the I/O subsystem to maximize the problem size that can be simulated. Next, we divide the
 499 global computational domain into several subdomains corresponding to the number of compute
 500 clusters involved in the computation. This decomposition is performed along the horizontal
 501 direction providing a first level of data-parallelism. To respect the width of the stencil (fourth-
 502 order), we maintain an overlap of two grid points in each direction. These regions, called ghost
 503 zones, are updated at each stage of the computation with point-to-point communications between
 504 neighboring clusters. This decomposition is rather similar to the description provided in section
 505 3.3.3. Unfortunately, this first level of decomposition is not sufficient as three-dimensional tiles
 506 do not fit into the 2 MB of memory available on each compute clusters.

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

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

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

526 5. Experimental Results

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

532 5.1. Measurement Methodology

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

539 Table 1 lists the average power consumed by each one of the platforms used in our exper-
540 iments during the execution of the parallel and distributed versions of the applications. Even
541 though the Altix UV 2000 features 24 Xeon E5 processors, it consumes less than 24 times the
542 power observed on Xeon E5. This is an expected phenomenon because Xeon E5 runs a multi-
543 threaded version of the applications whereas Altix UV 2000 runs their distributed counterparts.
544 Distributed versions experience periods of low processor usage as, for example, those during the
545 task request/response cycle and those related to load imbalance. This will be further discussed in
546 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.

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

551 obtain the energy consumption of the whole CPU package including cores and cache memory
 552 (named RAPL PKG domain). Similarly, MPPA-256 and Exynos 5 also possess hardware sensors
 553 to measure power consumption of the entire chip. Power measurements using this approach are
 554 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.

555 We also defined three input problem sizes for all applications (Table 2). These problem sizes
 556 were chosen based on the execution time on all platforms and amount of memory needed. For
 557 instance, we used a small problem size when running the applications with low thread counts in
 558 order to obtain the results in a reasonable time¹. Each experiment was repeated as many times as
 559 needed to ensure a relative error inferior to 2% with 95% statistical confidence using Student's
 560 t-distribution.

561 5.2. Overall Results

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

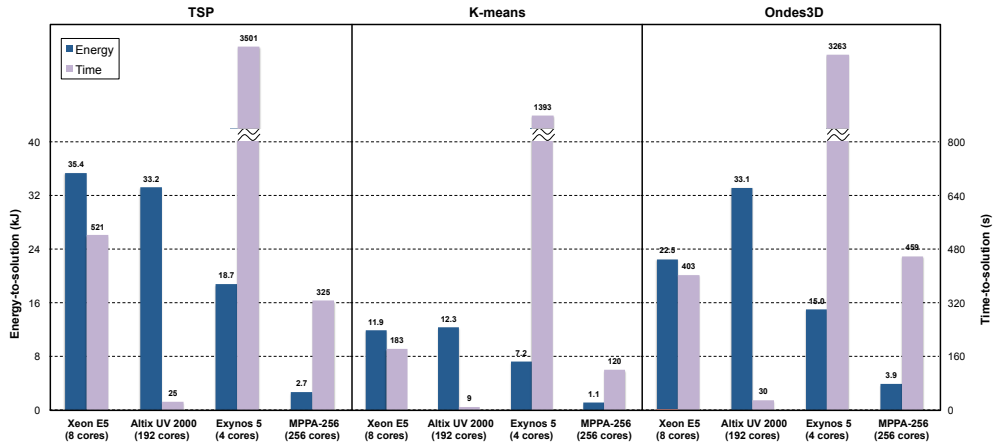


Figure 8: Time and energy-to-solution comparison between multicore, NUMA and manycore processors.

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

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

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

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

602 5.3. Energy Efficiency

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

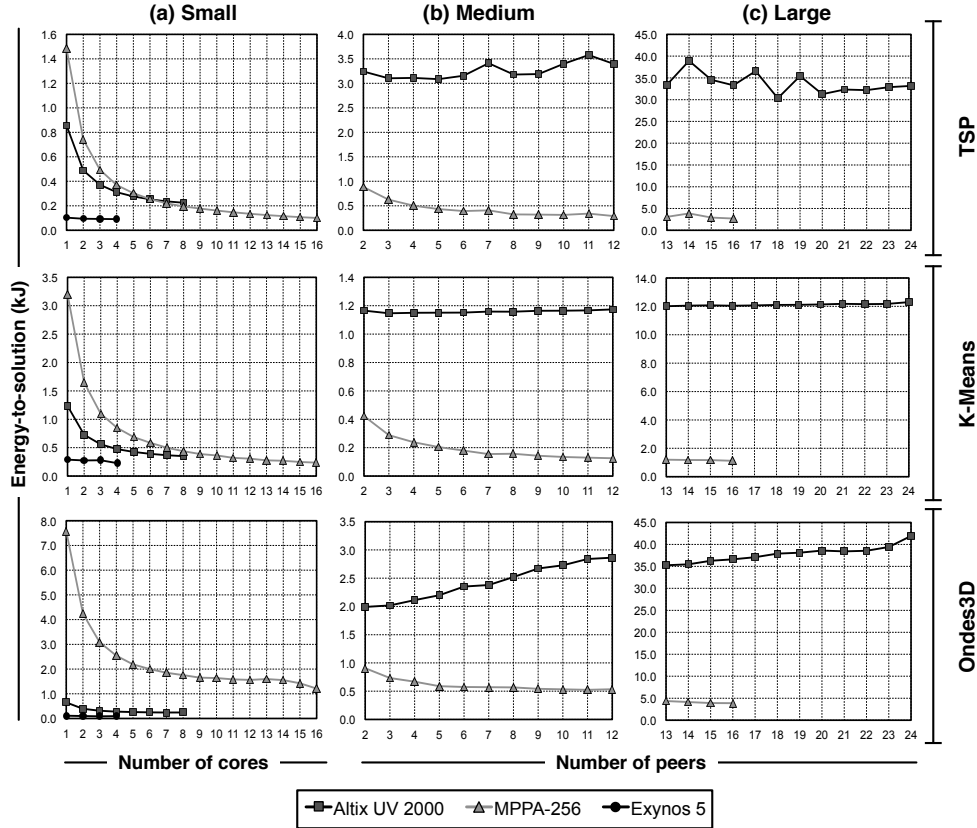


Figure 9: Energy-to-solution comparison on all platforms with three problem sizes.

611 we used a small problem size due to time constraints. Then, we compare the energy-to-solution
 612 on Altix UV 2000 and MPPA-256 when varying the number of peers (*i.e.*, processors on Altix UV
 613 2000 and compute clusters on MPPA-256), while using the maximum number of cores available
 614 on each Altix UV 2000 processor (8 cores) and MPPA-256 clusters (16 PEs). We used a medium
 615 problem size to compare the energy-to-solution from 2 to 12 peers (Figure 9b) and a large
 616 problem size for more than 12 peers (Figure 9c). Note that the MPPA-256 architecture is limited to 16
 617 peers, therefore we only show the results for more than 16 peers on Altix UV 2000.

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

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

632 **Varying the Number of Peers.** The gap between the energy consumed by Altix UV 2000 and
633 MPPA-256 became more important as we increased the number of peers. From 2 to 12 peers
634 (Figure 9b), MPPA-256 consumed at least 2.3x less energy than Altix UV 2000. This gap was
635 even larger from 13 to 16 peers with a large problem size (Figure 9c): in this case, MPPA-256
636 consumed on average $\sim 11x$ less energy than Altix UV 2000. Once again, the rationale behind that
637 comes from the high energy cost associated to the Altix UV 2000 processors: adding one Xeon E5
638 processor usually increases the overall power consumption of Altix UV 2000 by ~ 60 W on average
639 whereas adding one MPPA-256 cluster increases the overall power consumption of MPPA-256 by
640 ~ 0.3 W.

641 5.4. Scalability

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

655 Overall, the distributed version of the applications scaled considerably well and execution
656 times showed similar trends on both platforms. However, Altix UV 2000 was from 9x up to 13x
657 faster than MPPA-256. This result was expected, since peers mean processors running at full speed
658 (2.4 GHz) on Altix UV 2000 whereas they represent blocks (compute clusters) of the MPPA-256
659 processor running at 400 MHz. In other words, we are comparing sets of entire processors on Altix
660 UV 2000 against subsets of a single MPPA-256 processor. We also observed similar performance
661 gaps with other problem sizes.

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

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

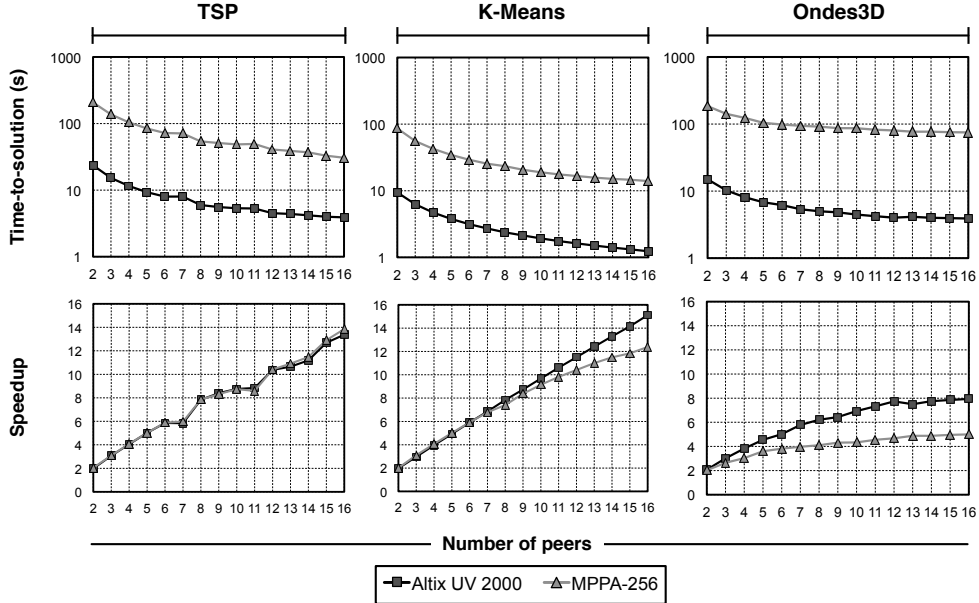


Figure 10: Time-to-solution and speedup comparison between Altix UV 2000 and MPPA-256.

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

681 5.5. Irregularity

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

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

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

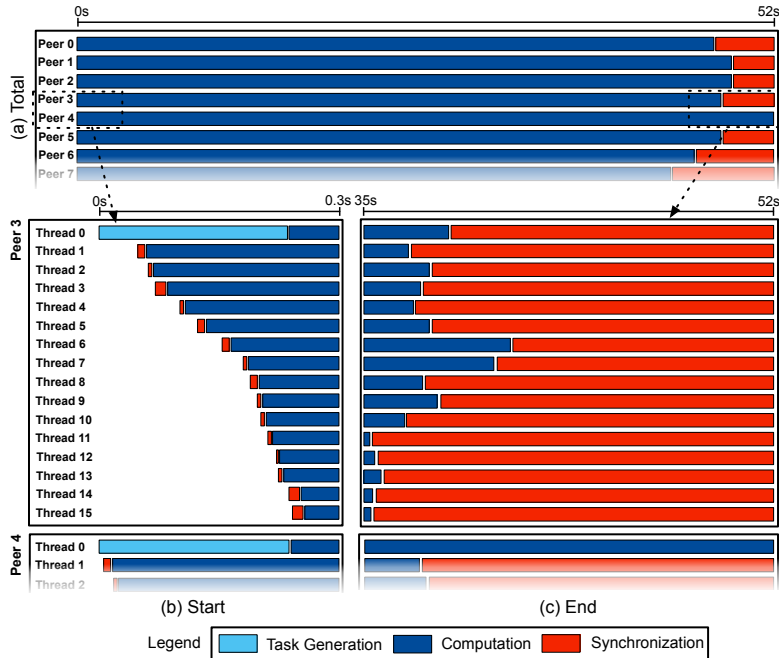


Figure 11: Execution traces of TSP on Altix UV 2000.

696 for more partitions. This strategy works fairly well throughout most of the execution. However,
 697 in some cases the load may become imbalanced at the end of the execution. Figure 11c shows
 698 what happens inside the peers. In this case, only peer 4 keeps processing for a long time while
 699 other peers are running out of tasks. This happens because the last task associated to Thread
 700 0 from Peer 4 takes much longer to be completed. This problem can be reduced with a work-
 701 stealing strategy inside each peer, so that threads can steal segments of this big task to improve
 702 parallelism. However, we leave this optimization for a future work.

703 6. Related Work

704 Many works have been focusing on analyzing the performance and energy efficiency of low-
 705 power multicore processors. *Padoin et al.* [34] compared an ARM Cortex-A9 1.0 GHz dual-
 706 core processor from Texas Instruments to two multiprocessors: one composed of two quad-core
 707 2.4 GHz Intel Xeon E5 processors and the other composed of four 2.0 GHz 8-core Xeon X7
 708 processors. They analyzed different metrics such as time-to-solution, peak power and energy-to-
 709 solution using 6 benchmarks from the NAS Parallel Benchmarks (NPB). Their results showed
 710 that the ARM processor outperforms Xeon X7, considering the energy-to-solution metric, for
 711 most of the analyzed benchmarks. However, the Xeon E5 had the best energy-to-solution among
 712 the three processors.

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

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

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

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

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

752 **7. Conclusion and Future Work**

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

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

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

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

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

792 Acknowledgments

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

798 References

- 799 [1] J. Larus, Spending Moore's Dividend, *Communications of the ACM* 52 (2009) 62–69.
800 [2] D. Brooks, P. Bose, S.E. Schuster *et. al.*, Power-Aware Microarchitecture: Design and Modeling Challenges for
801 Next-Generation Microprocessors, *IEEE Micro* 20 (2000) 26–44.

- 802 [3] D. GÖddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, A. Ramirez, Energy Efficiency vs.
803 Performance of the Numerical Solution of PDEs: An Application Study on a Low-power ARM-based Cluster, *J.*
804 *Comput. Physics* 237 (2013) 132–150.
- 805 [4] N. Rajovic *et. al.*, The Low-Power Architecture Approach Towards Exascale Computing, in: *Workshop on Scalable*
806 *Algorithms for Large-Scale Systems (ScalA)*, ACM, New York, USA, 2011, pp. 1–2.
- 807 [5] T. Fleig, O. Mattes, W. Karl, Evaluation of Adaptive Memory Management Techniques on the Tiler TILE-Gx
808 Platform, in: *International Conference on Architecture of Computing Systems (ARCS)*, VDE VERLAG, Luebeck,
809 Deutschland, 2014, pp. 88–96.
- 810 [6] C. L. Benoît Dupont de Dinechin and Pierre Guironnet de Massasa, Guillaume Lagera, B. Orgogozoa, J. Reyberta,
811 T. Strudela, A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor, in:
812 *Intl. Conference on Computational Science (ICCS)*, volume 18, Elsevier, Barcelona, Spain, 2013, pp. 1654–1663.
- 813 [7] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bobin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D.
814 de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H.
815 Nguyen, X. Raynaud, R. Sirdey, Extended Cyclostatic Dataflow Program Compilation and Execution for an Inte-
816 grated Manycore Processor, in: *International Conference on Computational Science (ICCS)*, volume 18, Elsevier,
817 Barcelona, Spain, 2013, pp. 1624–1633.
- 818 [8] G. Laporte, The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms, *European*
819 *Journal of Operational Research* 59 (1992) 231–247.
- 820 [9] M. Castro, E. Franceschini, T. M. Nguélé, J.-F. Méhaut, Analysis of Computing and Energy Performance of
821 Multicore, NUMA, and Manycore Platforms for an Irregular Application, in: *Workshop on Irregular Applications:*
822 *Architectures & Algorithms (IA³) - Supercomputing Conference (SC)*, ACM, Denver, EUA, 2013, p. Article No.
823 5.
- 824 [10] H. Li, H. L. Sudarsan, M. Stumm, K. C. Sevcik, Locality and Loop Scheduling on NUMA Multiprocessors, in:
825 *International Conference on Parallel Processing (ICPP)*, volume 2, IEEE Computer Society, Syracuse, USA, 1993,
826 pp. 140–147.
- 827 [11] R. Xu, I. Wunsch, D., Survey of clustering algorithms, *Neural Networks, IEEE Transactions on* 16 (2005) 645–678.
- 828 [12] L. Kaufman, P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*, John Wiley and Sons,
829 New York, 1990.
- 830 [13] A. K. Jain, R. C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- 831 [14] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, A. Wu, An efficient k-means clustering algorithm:
832 analysis and implementation, *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24 (2002) 881–892.
- 833 [15] I. Dhillon, D. Modha, A data-clustering algorithm on distributed memory multiprocessors, in: M. Zaki, C.-T.
834 Ho (Eds.), *Large-Scale Parallel Data Mining*, volume 1759 of *Lecture Notes in Computer Science*, Springer Berlin
835 Heidelberg, 2000, pp. 245–260.
- 836 [16] S. Rao, E. V. Prasad, N. B. Venkateswarlu, A scalable k-means clustering algorithm on multi-core architecture, in:
837 *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pp.
838 1–9.
- 839 [17] L. Rodrigues, L. Zarate, C. Nobre, H. Freitas, Parallel and distributed kmeans to identify the translation initiation
840 site of proteins, in: *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pp. 1639–
841 1645.
- 842 [18] P. Moczo, J. O. A. Robertsson, L. Eisner, The Finite-difference Time-domain method for Modeling of Seis-
843 mic Wave Propagation, in: *Advances in Wave Propagation in Heterogeneous Media*, volume 48 of *Advances in*
844 *Geophysics*, Elsevier - Academic Press, 2007, pp. 421–516.
- 845 [19] H. Aochi, T. Ulrich, A. Ducellier, F. Dupros, D. Michea, Finite difference simulations of seismic wave propagation
846 for understanding earthquake physics and predicting ground motions: Advances and challenges, in: *Journal of*
847 *Physics: Conference Series*, volume 454, IOP Publishing, p. 012010.
- 848 [20] R. Madariaga, Dynamics of an expanding circular fault, *Bulletin of the Seismological Society of America* 66
849 (1976) 639–666.
- 850 [21] F. Collino, Perfectly matched absorbing layers for the paraxial equations, *Journal of Computational Physics* 131
851 (1997) 164–180.
- 852 [22] F. Dupros, H.-T. Do, H. Aochi, On scalability issues of the elastodynamics equations on multicore platforms, in:
853 *International Conference on Computational Science (ICCS)*, volume 18 of *Procedia Computer Science*, Elsevier,
854 Barcelona, Spain, 2013, pp. 1226–1234.
- 855 [23] F. Dupros, C. Pousa, A. Carissimi, J.-F. Méhaut, Parallel Simulations of Seismic Wave Propagation on NUMA
856 Architectures, in: *International Parallel Computing conference (ParCo)*, volume 19 of *Advances in Parallel Com-*
857 *puting*, IOS Press, Lyon, France, 2010, pp. 67–74.
- 858 [24] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque,
859 S. M. Day, P. Maechling, Scalable earthquake simulation on petascale supercomputers, in: *High Performance*
860 *Computing, Networking, Storage and Analysis (SC)*, 2010 International Conference for, pp. 1–20.

- 861 [25] T. Furumura, L. Chen, Parallel simulation of strong ground motions during recent and historical damaging earth-
862 quakes in tokyo, japan, *Parallel Computing* 31 (2005) 149 – 165. Parallel Graphics and Visualization.
- 863 [26] R. K. Tesser, L. L. Pilla, F. Dupros, P. O. A. Navaux, J.-F. Méhaut, C. Mendes, Improving the performance
864 of seismic wave simulations with dynamic load balancing, in: *Euromicro International Conference on Parallel,
865 Distributed and Network-Based Processing (PDP)*, IEEE Computer Society, Turin, Italy, 2014, pp. 196–203.
- 866 [27] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, J. Roman, Exploiting Intensive Multithreading for the Effi-
867 cient Simulation of 3D Seismic Wave Propagation, in: *International Conference on Computational Science and
868 Engineering*, São Paulo, Brazil, pp. 253–260.
- 869 [28] A. Gursoy, Data decomposition for parallel k-means clustering, in: R. Wyrzykowski, J. Dongarra, M. Paprzycki,
870 J. Waśniewski (Eds.), *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer
871 Science*, Springer Berlin Heidelberg, 2004, pp. 241–248.
- 872 [29] E. Rotem, A. Naveh, A. Ananthkrishnan, E. Weissmann, D. Rajwan, Power-Management Architecture of the Intel
873 Microarchitecture Code-Named Sandy Bridge, *IEEE Micro* 32 (2012) 20–27.
- 874 [30] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring Energy Consumption for Short Code Paths Using RAPL,
875 *ACM Sigmetrics Performance Evaluation Review* 40 (2012) 13–17.
- 876 [31] C. Andreolli, P. Thierry, L. Borges, C. Yount, G. Skinner, Genetic Algorithm Based Auto-Tuning of Seismic
877 Applications on Multi and Manycore Computers, in: *EAGE Workshop on High Performance Computing for
878 Upstream*, Amsterdam, Netherlands. September, 2014. (To Appear).
- 879 [32] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, Optimization and performance modeling of stencil
880 computations on modern microprocessors, *SIAM Review* 51 (2009) 129–159.
- 881 [33] R. Love, K. Korner, CPU Affinity, *Linux Journal*, (111), 2003.
- 882 [34] E. L. Padoin, D. A. G. de Oliveira, P. Velho, P. Navaux, Time-to-Solution and Energy-to-Solution: A Comparison
883 between ARM and Xeon, in: *Workshop on Applications for Multi-Core Architectures (WAMCA)*, IEEE Computer
884 Society, New York, USA, 2012, pp. 48–53.
- 885 [35] Z. Ou, B. Pang, Y. Deng, J. Nurminen, A. Ylä-Jääski, P. Hui, Energy and Cost-Efficiency Analysis of ARM-Based
886 Clusters, in: *IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE Computer
887 Society, Ottawa, Canada, 2012, pp. 115–123.
- 888 [36] E. Totonì, B. Behzad *et. al*, Comparing the Power and Performance of Intel’s SCC to State-of-the-Art CPUs and
889 GPUs, in: *IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer
890 Society, New Brunswick, Canada, 2012, pp. 78–87.
- 891 [37] A. Morari, A. Tumeo, O. Villa, S. Secchi, M. Valero, Efficient sorting on the Tiler manycore architecture, in:
892 *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE
893 Computer Society, New York, USA, 2012, pp. 171–178.
- 894 [38] A. Gharaibeh, E. Santos-Neto, L. B. a. Costa, M. Ripeanu, The energy case for graph processing on hybrid cpu
895 and gpu systems, in: *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms,
896 IA3 ’13*, ACM, New York, NY, USA, 2013, pp. 2:1–2:8.