Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application

Márcio Castro
Institute of Informatics
UFRGS, Brazil
mbcastro@inf.ufrgs.br

Emilio Francesquini
University of São Paulo, Brazil
University of Grenoble, France
emilio@ime.usp.br

Thomas M. Nguélé
University of Yaoundé, Cameroon
thomas.nguele@imag.fr

Jean-François Méhaut
University of Grenoble, France
CEA/DRT/LETI
mehaut@imag.fr

ABSTRACT

The exponential growth in processor performance seems to have reached a turning point. Nowadays, energy efficiency is as important as performance and has become a critical aspect to the development of scalable systems. These strict energy constraints paved the way for the development of multi and manycore processors. Research on the performance and the energy efficiency of numerical kernels on multicores are common but studies in the context of manycores are sparse. Unlike these works, in this paper we analyze a well-known irregular NP-complete problem, the Traveling-Salesman Problem (TSP). This study investigates two aspects of the TSP on multicore, NUMA, and manycore processors. First, we concentrate on the nontrivial task of adapting this application to a manycore, specifically the novel MPPA-256 manycore processor. Then, we analyze its performance and energy consumption on different platforms that comprise general-purpose and low-power multicores, a NUMA machine, and the MPPA-256 manycore. Our results show that applications able to fully use the resources of a manycore can have better performance and may consume 9.8 and 13 times less energy when compared to low-power and general-purpose multicore processors, respectively.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

Keywords

Manycore, Multicore, NUMA, energy, performance, TSP

IA^3 2013 November 17-21, 2013, Denver CO, USA

ACM 978-1-4503-2503-5/13/11...$15.00.
http://dx.doi.org/10.1145/2535753.2535757

1. INTRODUCTION

The demand for higher processor performance made chipmakers 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 are not enough to ensure the steep curve of performance improvement predicted by Moore's Law and expected from the users [8].

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 [4] was created to evaluate the use of such components in an HPC environment [10]. While these low-power multicore processors usually do not offer the same performance of their regular counterparts, they normally offer better energy-to-solution results.

Current highly-parallel processors are taking this paradigm even further, featuring hundreds or even thousands of cores, while being energy efficient. The execution model of these processors may follow two different approaches. Light-weight manycore processors, such as Tilera Tile-Gx [15] and Kalray MPPA-256 [3], offer autonomous cores and an execution model that supports POSIX threads to accomplish both data and task parallelism. This 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 Instruction, Multiple Data (SIMD) 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.

Research on the power efficiency and numerical kernel performances of both general purpose and low-power processors are quite common [12, 14]. However, in this work we are interested in the analysis of the code adaptations needed to go from multicores to manycores. Here we will be using a well-known irregular problem: the Traveling-Salesman Problem (TSP). The TSP is NP-complete and, for a large enough

instance, its solution can be parallelized to make use of an arbitrary number of threads assuring the complete use of the chosen platforms. While this problem is highly parallelizable, it also displays important issues related to imbalance and irregularity. The behavior of an execution for the same instance of the problem can drastically change depending on the order and the number tasks.

We consider two important aspects in this study. The first aspect concerns the programming issues and challenges encountered when adapting the TSP for MPPA-256. 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. The lessons learned give some insights on what can be faced when adapting parallel applications to manycores. The second aspect concerns the performance and energy consumption of multicores and manycores. Our tests were carried out on four different hardware platforms: Intel Xeon E5, SGI Altix UV 2000, Nvidia Tegra 3, and Kalray MPPA-256. The first two are composed of regular x86 processors, while the others are based on low-power processors. We compare the chip-to-chip performance of these architectures as well as their power efficiency and show that the energy-to-solution for the same instance of the problem can vary from 2.7kJ (MPPA-256) up to 35.4kJ (Xeon E5) while the time-to-solution varies from $325s$ (MPPA-256) to $4,495s$ (Tegra 3). Next, we compare the Altix UV 2000 and the MPPA-256 platforms. Although very different from each other, these platforms share some similarities that give us the opportunity to evaluate important aspects of their scalability. We concluded that both architectures scale well for the chosen problem and MPPA-256 may consume up to 12x less energy than Altix UV 2000 to solve the same instance of the problem.

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

2. PLATFORMS

In this section we describe the multicore and manycore platforms used in this study. We first discuss the two platforms based on general-purpose processors and then we present the two platforms based on low-power processors.

2.1 General-Purpose Processors

Xeon E5. The Intel Xeon E5 is a 64-bit x86-64 processor. In this study, we used a Xeon E5-4640 Sandy Bridge-EP processor chip, which has 8 CPU cores (16 threads with Hyper-Threading support enabled) running at 2.40GHz. Each core has 32KB instruction and 32KB data L1 caches and 256KB of L2 cache. All the 8 cores share a 20MB L3 cache and the platform has 32GB of DDR3 memory.

Altix UV 2000. The SGI Altix UV 2000 (Figure 1) is a Non-Uniform Memory Access (NUMA) platform designed by SGI. The platform is composed of 24 NUMA nodes. Each node has a Xeon E5-4640 Sandy Bridge-EP processor (with the same specifications of the Xeon E5 platform) and 32GB of DDR3 memory. This memory is shared in a ccNUMA fashion through the SGI's proprietary NUMAlink6 (bidirectional). This high-speed interconnection provides a point-to-point bandwidth of 6.7 GB/s per direction. Overall, this platform has 192 CPU cores (384 threads with Hyper-Threading support enabled).

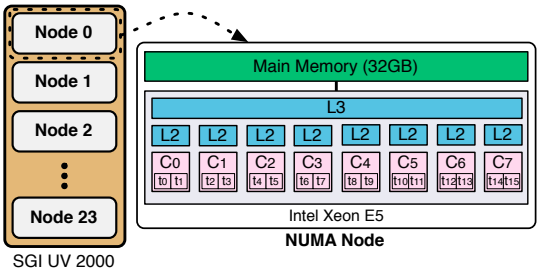

Figure 1: A simplified view of Altix UV 2000.

2.2 Embedded Processors

Carma. The Carma DevKit from SECO features a Nvidia Tegra 3 running at up to 1.3GHz, a Nvidia Quadro 1000M GPU with 96 CUDA cores and 2GB of LP-DDR2 memory. Tegra 3 is an embedded processor extensively used in mobile devices such as smartphones and tablets. It implements a Variable Symmetric Multiprocessing (vSMP) technology: it integrates a quad-core processor along with a 5th low-power companion core, built using a special low power silicon process that executes tasks at low frequency for active standby mode. All five CPU cores are identical ARM Cortex A9 CPUs, and are individually enabled and disabled (via aggressive power gating) based on the workload.

MPPA-256. The MPPA-256 is a single-chip manycore processor developed by Kalray that integrates 256 user cores and 32 system cores in 28nm CMOS technology running at 400MHz. These cores are distributed across 16 compute clusters and 4 I/O subsystems that communicate through data and control Networks-on-Chip (NoCs). This processor targets parallel applications whose programming models fall within the following classes: Kahn Process Networks (KPN), as motivated by media processing; Single Program Multiple Data (SPMD), traditionally used for numerical kernels; and time-triggered control systems [1, 3].

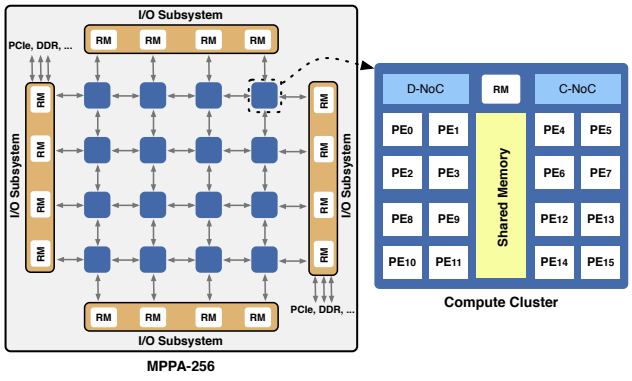

Figure 2: A simplified view of the MPPA-256.

Figure 2 shows the architecture overview of the MPPA-256. It features two types of cores: Processing Elements (PE) and Resource Managers (RM). Although RMs and PEs implement the same Very Long Instruction Word (VLIW) architecture, they have different purposes: PEs are dedicated to run user threads (one thread per PE) in non-interruptible and non-preemptible mode whereas RMs execute kernel rou-

tines and services of NoC interfaces. Operations executed by RMs vary from task and communication management to I/O data exchanges between either external buses (e.g. PCIe) or SDRAM. For this reason, RMs have privileged connections to NoC interfaces. Both PEs and RMs feature private 2-way associative instruction and data caches.

PEs and RMs are grouped within compute clusters and I/O subsystems. Each compute cluster features 16 PEs, 1 RM and a local shared memory of 2MB, which enables a high bandwidth and throughput between PEs. Each I/O subsystem relies on 4 RMs with a shared D-cache, static memory and external DDR access. Contrary to the RMs available on compute clusters, the RMs of I/O subsystems also run user code. An important difference of the MPPA-256 architecture is that it does not support cache coherence between PEs, even among those in the same compute cluster.

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

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

3. CASE STUDY: THE TSP

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

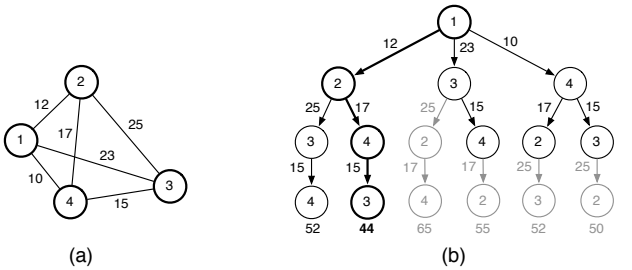

Figure 3: Example of TSP with 4 cities.

There are several different approaches for the resolution of this problem [7]. These solutions normally employ brute force, simple or complex heuristics, approximation algorithms or a mix of them. We are not going to detail the different available approaches since here we want to evaluate and compare the performance of an embarrassingly parallel non-numerical application across different architectures. Therefore, we use a brute force exact algorithm based on a simple heuristic. We first explain the sequential version of our algorithm, then we explain how we extended it to work with multiple threads. Finally, we present its distributed version.

3.1 Sequential Algorithm

The sequential version of the algorithm is based on the branch and bound method using brute force. Figure 4 outlines this solution. It takes as input the number of cities, and a cost matrix, and outputs the minimum path length.

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

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

Figure 4: Sequential version of TSP.

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

3.2 Multi-threaded Algorithm

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

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

Figure 5 shows the algorithm for this approach. This algorithm also receives as a parameter the number of threads *n_threads* to be used.

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

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

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

Figure 5: Multi-threaded version of TSP.

3.3 Distributed Algorithm

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

To avoid two peers working on the same subproblem, each peer *peer_id* only works on the tasks which were assigned to it. To do so, we specify the desired number of partitions per peer. We also specify the percentage of the tasks that will be distributed in the beginning of the execution. Afterwards, as the peers run out of work, they will ask a master peer for more partitions. To reduce communication, the master peer sends sets of partitions of decreasing size at each request [9]. The rationale behind it is that, as the task sizes are irregular, distributing a smaller number of partitions during the end of the execution might decrease the imbalance between the peers. In this case, for each request the master peer sends a set of partitions S and the peer *peer_id* will work on the tasks such that $task\_index \mod n\_partitions \in S$. Since the task generation is done locally, the amount of transferred data can be minimized.

As an implementation improvement, only one thread per peer becomes responsible for asking more partitions when the peer runs out of work. Once this thread receives a new partition from the master peer, it generates and populates the peer's task queue with new tasks. During the generation of these tasks, the remaining $n\_threads - 1$ threads can begin to process tasks as soon as they are enqueued, without the need to wait for the end of the task generation. This behavior is further discussed in Section 5.3.3.

4. ADAPTING THE TSP FOR MANYCORES

We presented in Section 3 insights into the algorithms for the resolution of the TSP. However, efficiently passing from multicores to manycores might be a nontrivial task. There are several reasons for that, being the most evident the natural architectural differences between these platforms. These differences usually force us to make adaptations to the code. In this section, we discuss these architectural aspects and adaptations as well as the rationale behind them.

POSIX threads are supported by the four tested platforms, thus the multi-threaded code for Xeon E5, Altix UV 2000, and Carma is exactly the same. In this version of the code, the global variable *min_path* defined by the algorithm depicted in Figure 4 is implemented using a simple shared variable that is accessed by every thread. The atomic function ATOMIC_UPDATE_IF_LESS(*current_value*, *new_value*) is implemented using a regular POSIX lock.

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

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

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

5. EXPERIMENTAL RESULTS

In this section, we present a performance and energy efficiency evaluation of the platforms when running the parallel and distributed versions of the TSP. We start by presenting our measurement methodology along with the metrics used to analyze the results on all platforms. Then, we compare the energy and computing performance of each multicore/ manycore processor (Section 5.2). Finally, in Section 5.3, we do a more thorough comparison between MPPA-256 and Altix UV 2000 when varying the number of processing units.

5.1 Measurement Methodology

We use two important metrics to compare the energy and computing performance of different multicore and manycore platforms: *time-to-solution* and *energy-to-solution*. Time-to-solution is the time spent to reach a solution for a given problem. In our case, this is the overall execution time of the parallel/distributed TSP. Energy-to-solution is the amount of energy spent to reach a solution for a problem. It can be computed by multiplying the average power consumed while running the application by the time-to-solution.

Table 1: Power consumption of the 4 processors.

	Xeon E5	Altix UV 2000	Carma	MPPA-256
Power (W)	68.6	1,418.4	5.88	8.26
Method	Sensors	Sensors	Spec	Sensors

Table 1 shows the overview of the power consumed by each one of the platforms used in our experiments during the execution of the parallel and distributed versions of the TSP. Even though the Altix UV 2000 features 24 Xeon E5 processors, it consumes less than 24 times the power observed on Xeon E5. This is an expected phenomenon because Xeon E5 runs a multi-threaded version of TSP whereas Altix UV 2000 runs its distributed counterpart. The distributed version experiences periods of low processor usage as, for example, those during the task request/response cycle and those related to load imbalance (we discuss it further in Section 5.3).

The power consumed by each processor was obtained using two different approaches. Both Xeon E5 and Altix UV 2000 feature Intel Sandy Bridge microarchitecture, which has Running Average Power Limit (RAPL) energy sensors. This allows us to measure the power consumption of CPU-level components through Machine-Specific Registers (MSRs). We used this approach to obtain the energy consumption of the whole CPU package including cores and cache memory (named RAPL PKG domain). Power measurements using this approach are very accurate as shown in [13, 5]. Similarly, MPPA-256 also features sensors to measure the power consumption of the entire chip, *i.e.*, 16 compute clusters and 4 I/O subsystems. Finally, we used the power consumption specification for Carma, since it does not feature any hardware sensor. In this case, we excluded the energy consumption of the GPU and the LP-DDR2 to make a fair comparison with other platforms.

Concerning the software stack, we compiled the TSP with the same major revision of GCC (4.7) on all platforms along with the optimization flag -O3. However, MPPA-256 features a modified version of GCC to build binaries to its specific platform. All platforms except MPPA-256 run Linux v3.0. MPPA-256 runs two different operating systems. The Real Time Executive for Multiprocessor System (RTEMS)[1] runs on the I/O subsystems whereas the NodeOS (a proprietary operating system developed by Kalray) runs on the RM of each computing cluster.

We also defined three input problem sizes for the TSP: *small* (16 cities), *medium* (18 cities) and *large* (20 cities). We used a small problem size when running the TSP with low thread counts in order to obtain the results in a reasonable time[2]. In all experiments, we have used the same instance of the problem to guarantee the same execution path among different runs. Each experiment was repeated at least 20 times to guarantee a confidence level of 95%.

5.2 Chip-to-Chip Comparison

Figure 6 compares both time-to-solution (right y-axis) and energy-to-solution (left y-axis) metrics on all processors (results obtained on Altix UV 2000 are shown in Section 5.3). In these experiments, we executed the TSP with a large problem size and we used every core from each processor. Differently from the tests executed on the other processors, we used two times more threads than the number of physical cores on the Xeon E5 platform. Although this processor has 8 physical cores, it features Hyper-Threading (HT) which doubles the number of logical cores, allowing the execution of 16 threads concurrently. HT was beneficial in our case, improving the performance of the multi-threaded TSP.

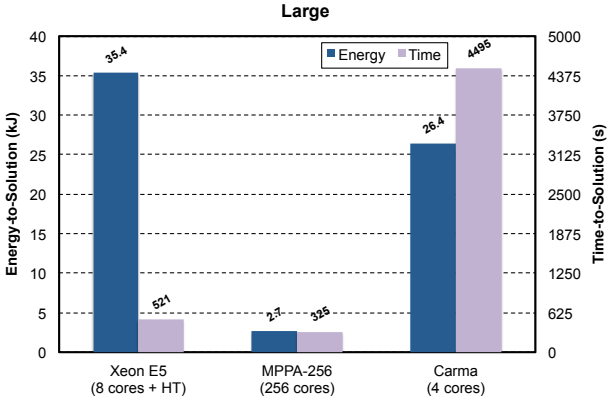

Figure 6: Time and energy-to-solution comparison between multicore and manycore processors.

Time-to-Solution. As expected, the TSP on Carma presented the highest execution times among all processors, being 8.6x slower than Xeon E5. The reason for that is threefold: (i) it has considerably lower clock frequency than Xeon E5; (ii) Xeon E5 is a performance-centric processor that is tuned far more for speed than for low power consumption; and (iii) Xeon E5 profits from its higher parallelism, since our parallel TSP scales considerably well as we increase the number of threads. Surprisingly, MPPA-256 presented the best execution time among all processors, executing the TSP 1.6x faster than than Xeon E5. Even though the clock frequency of MPPA-256 PEs is lower than that of the Xeon E5 cores, this embedded processor achieved better performance. Once again, this is due to the inherent characteristic of our TSP implementation. As we previously discussed in Section 3, there are few message exchanges between peers in

[1]RTEMS is available at `http://www.rtems.org`

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

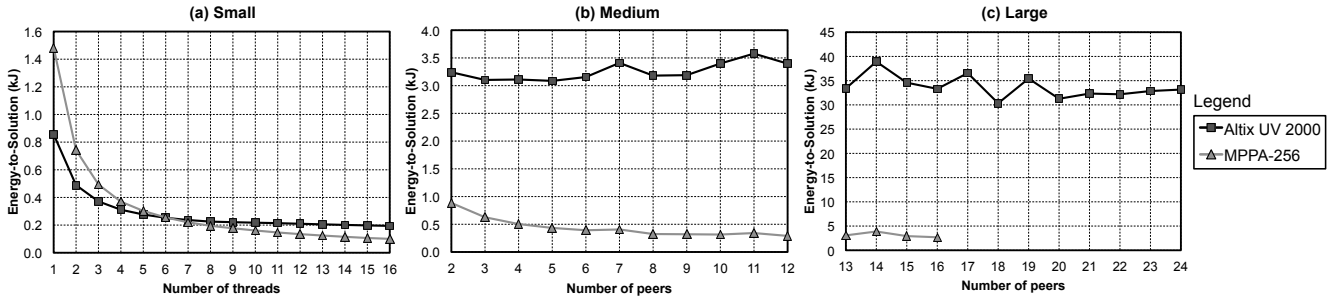

Figure 7: Energy-to-solution comparison between Altix UV 2000 and MPPA-256 with three problem sizes.

the distributed algorithm. The broadcast communication between clusters was implemented using asynchronous messages throughout the D-NoC, allowing compute clusters to overlap communication and computation.

Energy-to-Solution. Both embedded processors presented better energy-to-solution than Xeon E5. The low degree of parallelism available on the ARM processor was a clear disadvantage for Carma. Even though this processor consumes less power than the others, it ends up executing the TSP during a longer period of time. In our experiments, Carma consumed 25.2% less energy than Xeon E5. Among all processors used in our experiments, MPPA-256 showed the best energy-to-solution, reducing the energy consumption of Xeon E5 by approximately 92%.

5.3 MPPA-256 vs. Altix UV 2000

In the previous section, we showed that MPPA-256 presented the best results for both energy-to-solution and time-to-solution metrics. The main reason for that comes from the fact that MPPA-256 offers a much higher parallelism than other processors and yet has a low power consumption. In this section, we intend to extend the previous analysis by comparing the energy performance of MPPA-256 to another platform which also features the same level of parallelism, the Altix UV 2000. In other words, we intend to compare a manycore processor to a NUMA platform composed of several multicore processors.

Although both platforms differ in several aspects, they have a similar conceptual hierarchy. As we previously explained in Section 2, the Altix UV 2000 has 24 Intel Xeon E5 processors, each one with 16 logical cores with HT enabled. On the other hand, MPPA-256 has 16 compute clusters, each with 16 PEs. Taking this architectural similarity in mind, we proceeded as follows to compare their energy performances. First, we compare the energy-to-solution of both platforms when varying the number of threads from 1 to 16. In this case, we compare the performance of a single processor of Altix UV 2000 against a single compute cluster of MPPA-256, since we associate one thread per logical core/PE. For these tests, we used a small problem size due to time constraints. Then, we compare the energy-to-solution of both platforms when varying the number of peers (*i.e.*, processors on Altix UV 2000 and computing clusters on MPPA-256), while fixing the number of threads per processor/cluster to 16. We used a medium problem size to compare the energy-to-solution from 2 to 12 peers and a large problem size for more than 12 peers. Note that the MPPA-256 architecture is limited to 16 peers, therefore we only show the results for more than 16 peers on Altix UV 2000 platform. Figure 7 compares their energy-to-solution measurement results.

5.3.1 Varying the Number of Threads

The energy performance of both Altix UV 2000 and MPPA-256 may significantly vary depending on the number of threads used (Figure 7a). When comparing the energy efficiency of a single Altix UV 2000 processor against a single MPPA-256 cluster, we noticed that Altix UV 2000 outperformed MPPA-256 on low thread counts (it consumed 42% and 34% less energy with one and two cores/PEs, respectively). For more than 6 threads, however, the MPPA-256 cluster outperformed the Altix UV 2000 processor, consuming up to ∼48% less energy with 16 threads. After a deeper analysis, we concluded that this comes from the fact that the power consumed by a single Altix UV 2000 processor considerably increased as we increased the number of used cores whereas the power consumed by a single MPPA-256 cluster remained practically unchanged. More precisely, we noticed that the power consumption of a single Altix UV 2000 processor increased from 27W (single thread) up to 68.6W (16 threads) whereas it varied from 3.73W (single thread) up to 3.98W (16 threads) on the MPPA-256 cluster. This means that parallel applications running on MPPA-256 should exploit the full potential of PEs in order to achieve better energy performance than state-of-the-art Intel processors.

5.3.2 Varying the Number of Peers

The gap between the energy consumed by Altix UV 2000 and MPPA-256 became more important as we increased the number of peers. From 2 to 12 peers (Figure 7b), MPPA-256 consumed ∼8.3x less energy than Altix UV 2000 on average (medium problem size). This gap was even larger from 13 to 16 peers with a large problem size (Figure 7c): in this case, MPPA-256 consumed on average ∼11.3x less energy than Altix UV 2000. Once again, the rationale behind that comes from the high energy cost associated to the Altix UV 2000 processors: adding one Xeon E5 processor usually increases the overall power consumption of Altix UV 2000 by approximately 60W whereas adding one MPPA-256 cluster increases the overall power consumption of MPPA-256 by 0.29W. This means that MPPA-256 has a better tradeoff between computing/power performance for a fairly parallelizable application such as the TSP.

So far, we have only compared the energy-to-solution of MPPA-256 and Altix UV 2000, showing that the former consumed far less energy than latter to solve the same problem. Figure 8 illustrates the time-to-solution gap between them for a medium problem size when considering an equal number of resources (peers). Overall, our distributed version of TSP scaled considerably well and execution times showed similar trends on both platforms. However, Altix UV 2000 was approximately 9x faster than MPPA-256. This result was expected, since peers mean processors running at full speed (2.4GHz) on Altix UV 2000 whereas they represent

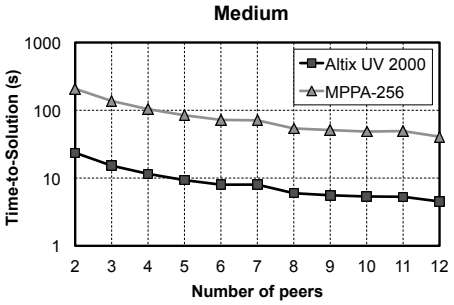

Figure 8: Time-to-solution comparison between Altix UV 2000 and MPPA-256.

blocks (compute clusters) of the MPPA-256 processor running at 400MHz. In other words, we are comparing entire processors on Altix UV 2000 against subsets of a single MPPA-256 processor. We also observed similar performance gaps with other problem sizes.

5.3.3 Load Imbalance

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

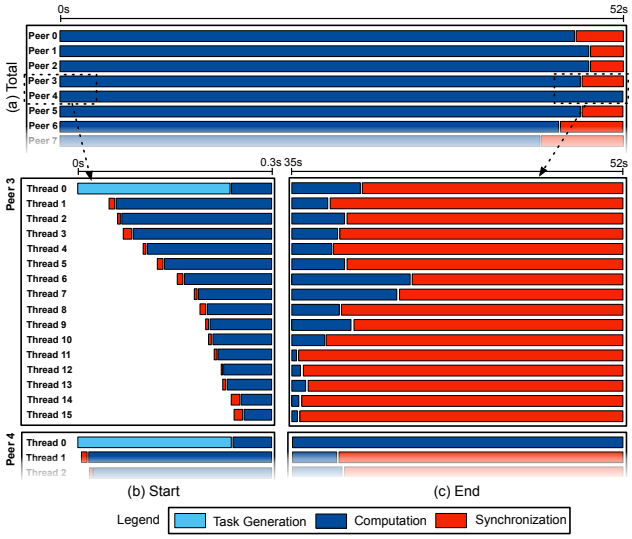

Figure 9: Execution traces of TSP on Altix UV 2000.

Figure 9a shows a global view of the execution aggregated per peer. At the beginning (Figure 9b), one thread in each peer asks a master peer for partitions and starts populating the local pool of tasks. As tasks become available, other threads in the same peer can start the computation. Once the thread assigned to populate the local pool of tasks finishes its job, it also starts the computation. Afterwards, as the peers run out of work, they ask a master peer for more partitions. This strategy works fairly well during almost the whole execution. However, in some cases the load may become imbalanced at the end of the execution. Figure 9c shows what happens inside the peers. In this case, only peer 4 keeps processing for a long time while other peers are running out of tasks. This happens because the last task associated to Thread 0 from Peer 4 takes much longer to be completed. This problem can be reduced with a work-stealing strategy inside each peer, so that threads can steal segments of this big task to improve parallelism. However, we leave this optimization as a future work.

6. RELATED WORK

Totoni et al. [16] compared the power and performance of Intel's Single-Chip Cloud Computer (SCC) to other types of CPUs and GPUs. The analysis was based on a set of parallel applications implemented with the Charm++ [6] programming model. They showed that there is no single best solution that always achieves the best trade-off between power and performance. However, the results obtained with the Intel SCC suggest that manycores are an opportunity for the future.

Padoin et al. [12] compared an ARM Cortex-A9 1.0GHz dual-core processor from Texas Instruments to two multiprocessors: one composed of two quad-core 2.4GHz Intel Xeon E5 processors and the other composed of four 2.0GHz 8-core Xeon X7 processors. They analyzed different metrics such as time-to-solution, peak power and energy-to-solution using 6 benchmarks from the NAS Parallel Benchmarks (NPB). Their results showed that the ARM processor outperforms Xeon X7, considering the energy-to-solution metric, for most of the analyzed benchmarks. However, the Xeon E5 had the best energy-to-solution among the three processors.

Göddeke et al. [4] also conducted a comparison between ARM and x86 architectures using different classes of numerical solution methods for partial differential equations. They evaluated weak and strong scalability on a cluster of 96 ARM Cortex-A9 dual-core processors and demonstrated that the ARM-based cluster can be more efficient in terms of energy-to-solution compared to a cluster of Intel Xeon X5550 processors. Similarly, *Ou et al.* [11] compared the energy efficiency of a ARM-based cluster against an Intel X86 workstation on three applications: a web server throughput, an in-memory database and a video transcoding. They concluded that energy/efficiency ratio of the ARM cluster against the Intel workstation depend on the application and may vary from 1.21 up to 9.5.

Using a slightly different approach, *Aubry et al.* [1] compared the performance of an Intel Core i7-3820 processor with the MPPA-256. Their application, an H.264 video encoder, was implemented using a dataflow language that offers direct automatic mapping to MPPA-256. Their findings show that the performance of these traditional processors is on par with the performance of the MPPA-256 embedded processor which provides 6.4 times better energy efficiency.

Unlike the previous works, we have focused on the passage from multicores to manycores from the perspective of an irregular problem (TSP). We pointed out some of the programming issues that must be considered when developing parallel applications to manycores. Moreover, we analyzed the performance and energy consumption of a set of state-of-the-art multicore and manycore platforms, ranging from low-power processors to general-purpose processors. The study was based on a parallel implementation of the TSP using Pthreads.

7. CONCLUSION AND FUTURE WORK

Manycore processors such as MPPA-256 seem to be the trend in the development of faster energy-efficient processors. The efficient use of an embedded manycore processor

demands adaptations to the application code so that it can efficiently use the whole chip. These modifications are not trivial. In a processor such as MPPA-256, the absence of a globally addressable memory space makes these modifications similar to those needed to port an application to the MPI paradigm. Additionally, the absence of a coherent cache considerably increases the implementation complexity, forcing the use of full memory barriers or proprietary system calls designed to completely bypass the cache.

As many other parallel applications, these modifications introduce redundant computations in order to improve the parallelism of the whole solution. Not every application is suitable to this kind of modification and, in the worst case scenario, an strictly serial application might be limited to the performance of a single core. For a fairly parallelizable application, such as the TSP problem, we showed that the MPPA-256 can be very competitive, even if the application is irregular in nature. Our results show that this manycore processor has a performance that is roughly equivalent to that of a traditional multi-purpose processor while providing better energy efficiency (∼13x). When compared to other embedded processors, MPPA-256 also shows better energy-to-solution (∼9.8x) and performance results (∼13.8x).

This work can be extended in two directions. On the one hand, our chosen application shows irregular communication patterns. However, there is not enough communication between the clusters to make it the bottleneck of the execution. As future work, we intend to assess how well the performance of this processor fares when compared to an application with heavier communication patterns, and not as embarrassingly parallel. On the other hand, we compared the energy efficiency of one state-of-the-art Intel processor (Xeon E5) to other low-power processors (MPPA-256 and Carma). This specific Intel processor is optimized for performance, not for energy consumption. As future work, we intend to compare the performance of these processors to other Intel energy optimized processors such as the Intel Atom and the mobile versions of the Sandy Bridge architecture. Additionally, we have used low-level tools, as for example POSIX threads and cache bypass functions to carry out our tests. A performance and energy consumption comparison of these processors with applications that employ higher-level tools such as MPI and OpenCL is planned.

Acknowledgments

The authors would like to thank CAPES for funding this research under project CAPES/Cofecub 660/10. Part of this work was done in the context of the European 'Mont-Blanc: European scalable and power efficient HPC platform based on low-power embedded technology' #288777 project of call FP7-ICT-2011-7.

8. REFERENCES

[1] P. Aubry, P.-E. Beaucamps, and F. Blanc *et. al.* Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *International Conference on Computational Science (ICCS)*, volume 18, pages 1624–1633, Barcelona, Spain, 2013. Elsevier.

[2] D. Brooks, P. Bose, and S.E. Schuster *et. al.* Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[3] B. D. de Dinechin, P. G. de Massasa, and G. Lagera *et. al.* A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In *Intl. Conference on Computational Science (ICCS)*, volume 18, pages 1654–1663, Barcelona, Spain, 2013. Elsevier.

[4] D. Göddeke and Dimitri Komatitsch *et al.* Energy Efficiency vs. Performance of the Numerical Solution of PDEs: An Application Study on a Low-power ARM-based Cluster. *J. Comput. Physics*, 237:132–150, 2013.

[5] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *ACM Sigmetrics Performance Evaluation Review*, 40(3):13–17, 2012.

[6] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar and X. Li, editors, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, chapter 13. John Wiley & Sons, Inc., Hoboken, NUSA, 2009.

[7] G. Laporte. The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms. *European Journal of Operational Research*, 59(2):231–247, June 1992.

[8] J. Larus. Spending Moore's Dividend. *Communications of the ACM*, 52:62–69, 2009.

[9] Li, Hui *et. al.* Locality and Loop Scheduling on NUMA Multiprocessors. In *International Conference on Parallel Processing (ICPP)*, volume 2, pages 140–147, Syracuse, USA, 1993. IEEE Computer Society.

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

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

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

[13] E. Rotem, A. Naveh, A. Ananthakrishnan, and E. Weissmann *et al.* Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[14] L. Stanisic, B. Videau, J. Cronsioe, and A. Degomme *et al.* Performance Analysis of HPC Applications on Low-Power Embedded Platforms. In *Design, Automation & Test in Europe (DATE)*, pages 475–480, Grenoble, France, 2013. IEEE Computer Society.

[15] Tilera Corporation. TILE-Gx Processor Family. `http://www.tilera.com/products/processors/TILE-Gx_Family`. Accessed: September 2013.

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