CAP Bench: a benchmark suite for performance and energy evaluation of low-power many-core processors

Matheus A. Souza, Pedro Henrique Penna, Matheus M. Queiroz, Alyson D. Pereira, Luís Fabricio Wanderley Góes, Henrique Cota de Freitas, Márcio Castro, Philippe O.A. Navaux, Jean-François Méhaut

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-Power Many-Core Processors

Matheus A. Souza[1]*, Pedro H. Penna[1], Matheus M. Queiroz[1], Alyson D. Pereira[1], Luís F. W. Góes[1], Henrique C. Freitas[1], Márcio Castro[2], Philippe O. A. Navaux[3], Jean-François Méhaut[4]

[1]*Department of Computer Science, Pontifical Catholic University of Minas Gerais (PUC Minas), Brazil*
[2]*Department of Informatics and Statistics, Federal University of Santa Catarina (UFSC), Brazil*
[3]*Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Brazil*
[4]*CEA/DRT, University of Grenoble, France*

SUMMARY

The constant need for faster and more energy-efficient processors has been stimulating the development of new architectures, such as low-power many-core architectures. Researchers aiming to study these architectures are challenged by peculiar characteristics of some components such as Networks-on-Chip and lack of specific tools to evaluate their performance. In this context, the goal of this paper is to present a benchmark suite to evaluate state-of-the-art low-power many-core architectures such as the Kalray MPPA-256 low-power processor, which features 256 compute cores in a single chip. The benchmark was designed and used to highlight important aspects and details that need to be considered when developing parallel applications for emerging low-power many-core architectures. As a result, this paper demonstrates that the benchmark offers a diverse suite of programs with regard to parallel patterns, job types, communication intensity and task load strategies, suitable for a broad understanding of performance and energy consumption of MPPA-256 and upcoming many-core architectures. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: benchmark; low-power; many-cores; network-on-chip

1. INTRODUCTION

Computer Architecture is a research field that is largely based on a rapid evolution of processors in order to support the high demand for performance in software development. The ever-increasing amount of data to be processed demands large scale platforms that are able to surpass the petaflop barrier and deliver exaflop performance. However, the development of large scale platforms with exponentially scaling performances also led to an exponential growth in power consumption. This problem was also pointed out by the Defense Advanced Research Projects Agency (DARPA) in their report [1], which states that current trends are insufficient to achieve exascale systems due to power and energy consumption constraints. This concern is now enough to warrant the research on the use of low-power many-core processors [2, 3]. Indeed, several research efforts are looking for alternatives to place a large number of processing cores inside a single chip to increase the processor's efficiency [4, 5, 6].

In this context, the use of busses and crossbar switches as global interconnection mechanisms is no longer viable when scalability is necessary, due to physical wire-related constraints, such as low

*Correspondence to: matheus.alcantara@sga.pucminas.br

bandwidth, signal attenuation, delays due to the long wire sizes, and concurrency [7, 8]. To address this issue, Networks-on-Chip (NoCs) have emerged as an appealing approach for interconnecting a large number of cores [9, 10]. The Multi-Purpose Processor Array (MPPA-256) is a state-of-the-art low-power many-core processor designed by Kalray that embraces such structure, featuring 16 multi-core clusters of 16 cores each, totaling 256 cores in a single chip. In this architecture, clusters are interconnected by two NoCs. It is a next-generation many-core architecture built for low energy consumption that may be potentially used to attain exaflop performance with better energy efficiency [11, 12].

To make efficient use of such many-core processors, parallel applications developed for them must efficiently exploit both shared and distributed memory programming models [2]. In MPPA-256, for instance, applications must use a shared memory model inside multi-core clusters, whereas they must use a specific Application Programming Interface (API) that implements a NoC Inter-Process Communication (IPC) model to exchange data between clusters. Writing code for such hybrid models demands deep knowledge about the target architecture. Unfortunately, most programmers are used to develop code based on the shared memory paradigm, which is the usual programming model for multi-cores. These challenges increase the complexity in the effective use of NoC-based many-core processors.

Another important issue concerning these emerging architectures is performance evaluation. Even though several benchmark suites have been previously proposed to assess the performance of multi-core processors, they are not suitable for conducting analyses on low-power many-core architectures [13, 14]. There are three main reasons for that. First, they were not designed to accommodate the greatly increased number of cores. Work units are either too small or outnumbered, which leads to load imbalance and thus poor performance. Second, they do not deal effectively with low-power many-core memory constraints such as limited amounts of on-chip memory and the absence of cache coherence protocols. The use of smaller memory units might be a key aspect in many-core architectures in order to achieve low-power consumption [15]. Third, they are not usually designed to specifically exploit platform-dependent features, such as vector processing units or NoCs, which might demand a specific understanding about how to deal with them.

New research efforts are needed to efficiently deal with NoC-based many-core architectures, since they are the state-of-the-art for future exascale computing. However, both industry and academy are now facing challenges on software development and performance evaluation of the emerging many-core processors. For instance, when new architectures based on different programming models such as MPPA-256 come up, they have no accompanying software to address these issues upon their arrival. Fortunately, this stalemate may be resolved by open benchmarks that are able to encourage programmers and demystify the architecture.

In this paper we discuss the design of an open benchmark suite (CAP Bench) to provide a solution to evaluate energy consumption and performance of low-power many-core architectures. As a case study, we used CAP Bench to evaluate an emergent state-of-the-art low-power many-core processor called MPPA-256 and to assess whether MPPA-256 can be used as an alternative for energy-efficient High Performance Computing (HPC). To the best of our knowledge no benchmark is available to be used as baseline to design, program, evaluate and learn about MPPA-256. Besides being used and validated specifically on MPPA-256, CAP Bench also features an OpenMP implementation of all applications. This implementation can be used in shared-memory many-core architectures such as the Intel Xeon Phi [16] and the Mellanox TILE-Gx [17], the PULP platform [18], as well as in full-system simulators such as Gem5 [19].

This paper is organized as follows: Section 2 presents an overview of related work. Section 3 quickly introduces the MPPA-256 architecture. Section 4 presents the design method used throughout the development of the benchmark suite. Section 5 discusses the algorithms available in CAP Bench as well as their main characteristics. An evaluation of the benchmark applications is presented in Section 6. Finally, Section 7 concludes this paper and suggests future research paths.

DOI: 10.1002/cpe

2. RELATED WORK

The need for energy-efficient processors led to the development of low-power many-core architectures. Early studies such as the GigaNetIC [20], the CHNoC [21] and the MCNoC [22] focused on NoC aspects. Currently, cluster-based architectures are under development such as the SMYLEref many-core [23] and a cluster-based multi-core SoC [24]. The STHORM/P2012 project [25], which is developed by CEA and ST Microeletronics, and the Parallel Ultra-Low-Power Processing-Platform (PULP) [18], a joint project between groups at ETH Zurich and UNIBO, are also under development. Both were designed on cluster-based NoCs towards low-power many-core architectures with strong similarities to the already mentioned MPPA-256 platform.

With regard to benchmark suites, several ones have already been proposed to assess the performance of multi-core processors. In this section, we first underline the main aspects of the most popular ones, and then discuss why these benchmark suites are not suitable to evaluate emerging low-power many-core processors such as the MPPA-256.

The NAS Parallel Benchmark (NPB) [26] was developed at NASA Ames Research Center in 1991 with the goal of testing the performance of highly parallel supercomputers running scientific applications. NPB originally consisted of five parallel kernel benchmarks and three simulated applications that were developed to resemble Computational Fluid Dynamics (CFD) applications. The benchmark applications were implemented in C/Fortran77 and involved larger computations than other benchmarks that were available at the time, and therefore were suitable for the evaluation of parallel machines. Additionally, they were conceived to be simple enough to be implemented in new machines without much effort and time.

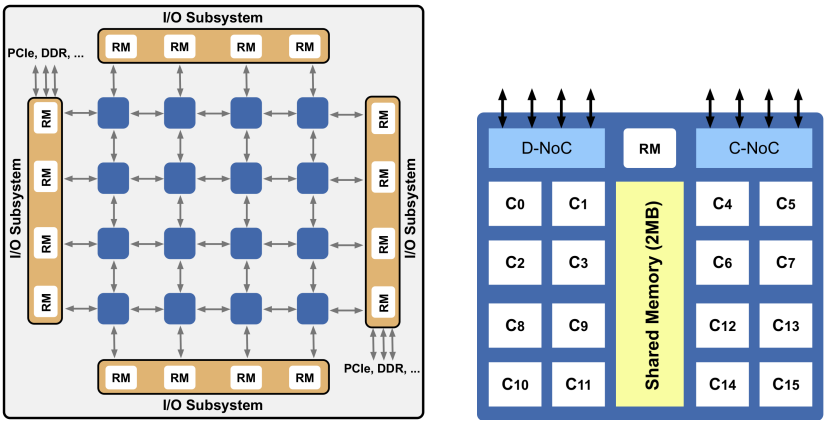
SPLASH-2 [14] is a parallel application suite proposed in 1995 to study shared-memory multiprocessors. It is composed of 8 complete applications and 4 kernels, representing a variety of scientific, engineering and graphic applications. The paper characterizes the applications with respect to aspects that interfere in their performance, namely: concurrency and load balance; working sets and temporal locality; communication-to-computation ratio and traffic; and spatial locality and false sharing. The authors remark that this characterization was made to help people to better understand the applications and thus to make a more efficient use of them.

In 2006, the Standard Performance Evaluation Corporation (SPEC) announced an update to its benchmark suite CPU2000, introducing CPU2006 [27]. CPU2006 contains applications implemented in C, C++ and Fortran, inspired by real life applications instead of using synthetic benchmarks. It has a total of 30 benchmarks, of which 12 are integer benchmarks and the remaining 18 are floating-point benchmarks. Each of the applications has a varied set of pertinent inputs and different outputs. One of the intended uses of the CPU2006 benchmark suite is early design analysis of new architectures, hence the focus on including varied, real-world applications.

PARSEC [28] is a parallel benchmark suite designed in 2008 with the goal of studying Chip Multiprocessor (CMP) architectures. The need for a new benchmark suite is due to old suites being biased towards HPC architectures, as well as using dated implementation techniques and not necessarily being research-friendly. PARSEC is composed of 9 applications and 3 kernels, selected to be as diverse as possible. Several different parallelism strategies are used in the applications. The paper also characterizes the applications with regard to metrics such as locality exploration, communication-to-computation ratio, off-chip traffic, parallelization and working sets.

The Rodinia benchmark was conceived to enable the study of heterogeneous computing architectures, which might use, for instance, Graphics Processing Units (GPUs) [29]. The authors developed 9 applications or kernels aiming to evaluate multi-core architectures and GPU platforms, characterizing them with respect to inherent architectural characteristics, parallelization, synchronization, communication overhead and power consumption. They concluded that each application might present different and specific behaviors.

Although these benchmark suites do provide researchers with a variety of applications to assess the performance of multi-core processors, they are not suitable for conducting analyses on low-power many-core architectures [13]. They neither consider severe local memory constraints, which may be just a handful of megabytes like in MPPA-256, nor the specific NoC topology, which



(a) MPPA-256 chip. (b) Compute cluster.

Figure 1. Kalray MPPA-256 overview. (a) The chip is composed of 16 compute clusters interconnected by NoCs. (b) Inside each cluster, there are 16 cores that share 2 MB of memory.

plays an important role in many-cores. Furthermore, applications with small work units and load imbalance may not be suitable to evaluate many-core architectures. For instance, the SPLASH-2 characterization reveals that some of the applications present poor performance when the number of cores were greatly increased [14]. The Rodinia benchmark concept is quite similar to ours, however, its characterization does not cover parallel patterns applied to hybrid programming models, which is an alternative in low-power many-core architectures that improves the explicit parallelism and the programming flexibility.

Strategies to address the aforementioned problems could be employed, such as load balance of loop iterations and vectorization. To accommodate these changes, however, these applications would need to be completely remodeled and redesigned to be suitable for use in the aforementioned architecture, which is an endeavor that can be time-consuming [2, 12, 30, 31].

Given the above considerations, it is important to evaluate and understand low-power many-core architectures, by setting a baseline, using proper benchmarks, in order to support innovations in the same direction. Our proposal, CAP Bench, also differs from the aforementioned benchmarks since it takes into account a design methodology, which is substantiated in parallel patterns, energy efficiency of many-core architectures and workload balance. This design method will be further detailed in Section 4.

It is worth noting that some of the reported benchmark suites have been proposed in the past decades, thus, they did not cover application design aspects that are essential to conduct evaluations and research on the state-of-the-art many-core architectures, which reinforces the need for new approaches .

3. THE MPPA-256 MANY-CORE PROCESSOR

MPPA-256 [11] is a many-core processor developed by Kalray which features 16 compute clusters in a single chip. An overview of MPPA-256 is presented in Figure 1(a), and a more detailed view of a compute cluster is shown in Figure 1(b). Each compute cluster is composed of 16 processing cores (named C_0–C_{15}) and a Resource Manager (RM), thus totaling 256 processing cores. Processing cores are dedicated to run user threads (one thread per core) in non-interruptible and non-preemptive mode, whereas RMs execute kernel routines and communication services. Each compute cluster has 2 MB of memory shared by the 16 processing cores of the cluster.

MPPA-256 also has 4 quad-core clusters, named I/O subsystems, that make it possible to perform I/O operations, such as getting data from the DDR memory (2 GB) and sending it to compute clusters. A compute cluster cannot directly access the DDR or another cluster's memory. Because

DOI: 10.1002/cpe

of that, the 16 compute clusters and the 4 I/O subsystems are connected by two parallel NoCs with bi-directional links, one for data (D-NoC), and the other for control (C-NoC). The D-NoC allows any of the compute clusters to write data to the internal memory of other compute clusters. Moreover, it allows the I/O subsystem to write data from the DDR to the compute clusters' internal memories and vice-versa. The way in which communication occurs is covered in Section 4.

4. DESIGN METHOD

To enable the design of a benchmark suite to evaluate low-power many-core architectures, MPPA-256 processor was chosen as target and a development method was defined and adopted. Our main goal was to keep the source code as clear as possible, while optimizing the applications to make a more efficient evaluation of the target architecture. This way, the benchmark itself can be easily extended and applications may help new programmers to learn about low-power many-core architectures, MPPA-256 and next generations.

Applications were developed using the C language with two parallel programming libraries: (i) OpenMP 3.0 and (ii) a proprietary Application Programming Interface (API) from Kalray. The former is based on a shared memory model and was used to parallelize the applications inside the clusters and the I/O subsystems. The latter, on the other hand, follows a distributed memory model and was used for inter-cluster communication and for communication between clusters and the I/O subsystem through the NoC. The API is based on the classic POSIX Inter-Process Communication (IPC) with synchronous and asynchronous operations adapted to the NoC and PCI features. It is worth noting that the proprietary API relies on explicit parallelism, in which the work units are totally independent in terms of data and computing. It means that the programmer must use functions and directives from the API in order to determine how the parallelism, with respect to inter-cluster communication, should happen. To actually build applications, the Kalray toolchain that specifically targets MPPA-256 was used.

The first step of the development cycle was the selection of the algorithms. We selected algorithms that solve general scientific problems, such as image processing and clustering. The selected algorithms (Section 5) can be used to develop relevant applications with diverse characteristics with respect to parallel patterns, job types, communication intensities and task loads. We believe that these characteristics are relevant to evaluate MPPA-256.

Once the algorithms were chosen, they were parallelized with OpenMP (shared memory model). This first parallelization allowed us to identify code snippets that would work in parallel units. Different parallel patterns were considered during this process. Several runs of the applications were conducted with the shared memory model. On that stage, after running an application, the perceived results were analyzed and code redesigns were done. This way, the parallel pattern might have been changed in this process, until we get a version that have performance improvements. Once the parallel versions of all applications using the shared memory model were finished, they were tested on MPPA-256 using a single cluster. This allowed us to evaluate the performance of our parallel solutions without any communication overhead. The next step was then to adapt these solutions to a hybrid model, which uses OpenMP inside the clusters and the Kalray API to perform cluster-to-cluster and cluster-I/O communications.

The execution flow on MPPA-256 is the following. The main process runs on an RM of the I/O subsystem and is responsible for spawning worker processes. These processes are then executed on compute clusters and may create up to 16 threads, one for each core (PE) inside them. Figure 2 shows an overview of how this process occurs in MPPA-256.

The I/O subsystem creates one process per cluster using the `mppa_spawn()` function. Communication channels are allocated using `mppa_open()` on the I/O subsystem and on all clusters involved in the communication. These communication channels are associated with preallocated buffers on the receiver's memory. By using these communication channels, the I/O subsystem can perform write operations (`mppa_write()` function) on the memory allocated in each cluster (2 MB) and clusters can perform write operations on the DDR connected to the I/O subsystem (2 GB). Each `mppa_write()` operation executed by the transmitter must be combined

DOI: 10.1002/cpe

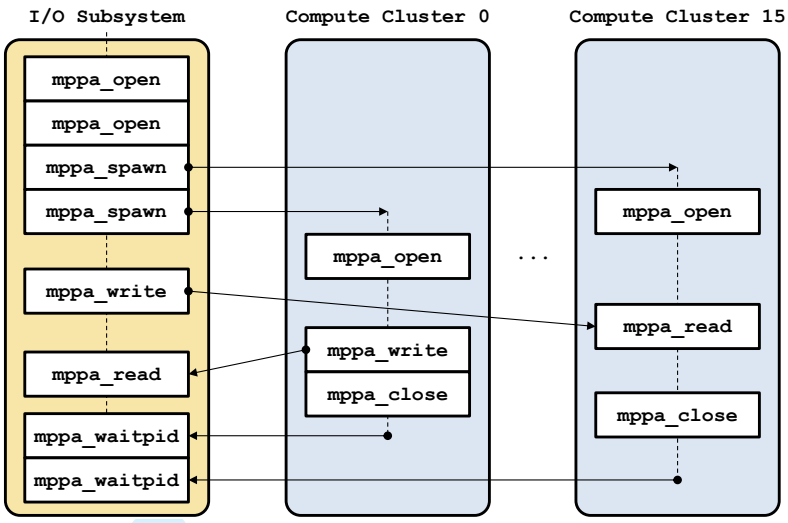


Figure 2. Kalray MPPA-256 execution flow.

with an `mppa_read()` on the receiver. The API allows both synchronous and asynchronous communication. Threads in compute clusters are then created and managed by either OpenMP 3.0 or Pthreads, allowing thread parallelism through the 2 MB shared memory. Once all computations are finished, each compute cluster should call `mppa_close()`, which will be responsible for performing a synchronization with the I/O subsystem (`mppa_waitpid()` function).

Each application was adapted to the hybrid model described before. After that, their performance was analyzed in order to found opportunities for possible improvements. For instance, an empirical study of the impact of the task size on the performance of each application was conducted. We concluded that the tradeoff between communication time and load imbalance is key to achieve high performance on MPPA-256. This study led us to fine-tune the task sizes and communications done through the NoC, which improved the performance of all applications due to a better use of the MPPA-256 resources.

All results presented in Section 6 represent the average values for the metrics considered in this study (*i.e.*, time and energy). Averages were calculated based on the values obtained from at least 10 runs, presenting statistical confidence of 95% by Student's t-distribution and less than 0.7% of relative error. Confidence intervals were omitted from the results due to very low relative error. The rationale behind such a low relative error is due to the fact that there is almost no overhead from the operating system on MPPA-256. For instance, threads within the clusters run without being interrupted by the operating system that runs on the Resource Manage RM.

The MPPA-256 was evaluated with respect to performance, scalability and energy consumption. To measure performance, the applications were executed with *small*, *default* and *huge* input sizes, using 2, 4 and 16 compute clusters, respectively. Then, all applications were executed with all input sizes, varying the number of used compute clusters in a base-2 logarithmic scale (1, 2, 4, 8 and 16 compute clusters) to evaluate both scalability and energy consumption on MPPA-256. Additionally, all applications were executed with the *default* input size comprising any possible quantity of compute clusters (1 to 16). The energy consumption was collected through the use of specific tools for Kalray MPPA-256, while the executions were conducted.

5. THE CAP BENCH SUITE

To compose the benchmark, seven applications are proposed, covering a wide range of characteristics. Applications follow five different *parallel patterns* based on [32, 33]:

DOI: 10.1002/cpe

1. **Divide and Conquer**, which starts with a division of the problem into small subproblems solvable in parallel, merging partial solutions into a result;
2. **Map**, where operations are mostly uniform and applied individually over elements of a data structure;
3. **MapReduce**, which combines Map with a consolidation of results in a Reduce procedure;
4. **Stencil**, in which a function can access an element in a collection and its neighbors, given by relative offsets; and
5. **Workpool**, where the algorithm can be divided into independent tasks and distributed among the execution units.

Job types pertain to what resource is critical to the application: the CPU, memory or NoC. When using the MPPA-256, if the time to complete a specific application task is determined mainly by the compute clusters' speed, the application is said to be CPU-bound. If the application time to solution is decided primarily by the amount of memory required, it is said do be memory-bound. In the case of MPPA-256, with limited memory inside compute clusters, it was an effort to adjust the applications to be as little memory-bound as possible. Finally, the application is said to be NoC-bound if the time to solution is mainly determined by the period spent transferring data through the D-NoC. This behavior occurs, for instance, when the task sizes are small, causing high communication intensity.

Communication intensity concerns how often the applications require the use of the NoC. Three levels of communication intensity were defined: low, for applications that do not require the NoC often; average, for applications that make moderate use of the NoC; and high, for applications that rely heavily on the NoC in their operations. It is worth noting that the data sent to compute clusters must not be larger than 2 MB, since the memory space in these clusters is limited. Thus, communication intensities are mainly dependent on the number of times the NoC is required.

Finally, *task loads* can be regular, if they have the same size, or irregular, if they have diverse sizes due to irregularities in task generation and/or computation.

In the following sections we describe the strategies used to implement these applications for the MPPA-256 processor. These sections are meant to introduce the applications and briefly outline how they were implemented. CAP Bench is open source software and it is available online[†].

5.1. Features from Accelerated Segment Test - FAST

Features from Accelerated Segment Test (FAST) [34, 35] is a corner detection method that follows the Stencil parallel pattern. It is usually used to extract feature points and to track and map objects in computer vision tasks. It uses a circle of 16 pixels to test whether a candidate point p is actually a corner. Each pixel in the circle is labeled from integer number 1 to 16 clockwise. If all N contiguous pixels in the circle are brighter than the intensity of the candidate pixel p plus a threshold value t or all darker than the intensity of $p - t$, then p is classified as a corner.

In the MPPA-256 implementation, we use randomly generated images as inputs and a mask containing the positions relative to p that must be analyzed. The N value is set to 12 and threshold t is set to 20. Due to very large input images and the compute cluster memory restriction, the I/O subsystem partitions the input image into 256 KB chunks and sends them to compute clusters for corner detection. Such a fine granularity causes intense communications through the NoC. After that, output chunks are sent back to the I/O subsystem, which in turn puts them together to build the output image that indicates all corners present in it. Moreover, the I/O subsystem receives the amount of corners detected by each compute cluster and summarizes them to indicate the overall number of corners detected.

[†] github.com/cart-pucminas/CAPBenchmarks

5.2. Friendly Numbers - FN

In number theory, two natural numbers are friendly if they share the same abundancy. The abundancy A of a given number n is defined as $A(n) = \frac{\sigma(n)}{n}$, where $\sigma(n)$ denotes the sum of divisors of n, i.e., $\sigma(n) = \sum_{d|n} d$. FN computes and compares the abundancy of all numbers in a given interval $[m, n]$ to determine which pairs of numbers are friendly. The parallel pattern used in FN implementation is MapReduce.

The MPPA-256 implementation uses a master/slave approach. Since every processing task for FN can be executed independently, we split the input interval into equal sized tasks in the master process and distribute them among the compute clusters to be simultaneously processed. These tasks are balanced, causing a regular task load in the slave processes. The abundancy results are sent back to the master process, which then performs abundancy comparisons using the 4 I/O clusters in parallel. This computation is not influenced by the NoC use or memory access, being exclusively CPU-bound.

5.3. Gaussian Filter - GF

The Gaussian blur (also known as Gaussian smoothing) filter is an image smoothing filter that seeks to reduce noise and achieve an overall smoothing of the image. It consists in applying a specially computed two-dimensional Gaussian mask (m) to an image (i), using a matrix convolution operation. It uses the Stencil parallel pattern.

In the MPPA-256 implementation, we use randomly generated masks and images as inputs. Since some input images are very large and the compute clusters have a 2 MB memory restriction, the I/O subsystem partitions the image into 1 MB chunks and sends them to compute clusters to be filtered. This is an average chunk size, causing a moderate use of the NoC that in general does not overwhelm the general computation. After the individual chunks are filtered, they are be sent back to the I/O subsystem, which puts them together to build the output image.

5.4. Integer Sort - IS

The integer sort problem consists in sorting a very large amount of integer numbers. We implemented a variation of the integer sort problem called *bucket sort*, which divides the elements to be sorted into *buckets*. A bucket is a structure that stores numbers in a certain range. The integer numbers used as input are randomly generated and range from 0 to $2^{20} - 1$. IS uses the Divide and Conquer parallel pattern.

In the MPPA-256 implementation, the buckets are further subdivided into *minibuckets* of 1 MB. As input elements are mapped to the appropriate buckets, they are placed in a minibucket. When the minibucket becomes full, a new minibucket is allocated inside its parent bucket and starts receiving elements. This takes place in the I/O subsystem, which will also send minibuckets for compute clusters to work on. Each compute cluster receives only one minibucket at a time, due to memory restrictions. Inside a compute cluster, minibuckets are sorted using a parallel mergesort algorithm, and as the starting order are random, the task load is irregular. Sorted minibuckets are then sent back to the I/O subsystem to be merged in parallel by its 4 cores. Because of this flow of minibuckets, IS is a high-intensity algorithm in terms of communication, therefore being NoC-bound.

5.5. K-Means - KM

K-Means clustering is a data clustering solution employed in clustering analysis. We opted to use Lloyd's algorithm [36] in our work. Given a set of n points in a real d-dimensional space, the problem is to partition these n points into k partitions. The data points are evenly and randomly distributed among the k partitions, and the initial centroids are computed. Then, the data points are re-clustered into partitions taking into account the minimum Euclidean distance between them and the centroids. Next, the centroid of each partition is recalculated taking the mean of all points in the partition. The whole procedure is repeated until no centroid is changed and every point is farther than the minimum accepted distance. During the execution, the number of points within

each partition may differ, implying different recalculation times for each partitions centroid. The parallel pattern of KM is Map.

The MPPA-256 version of the K-Means algorithm (KM) takes additional parameters p, specifying the number of compute clusters to be used, and t, which specifies the total number of execution flows. Each cluster spawns t working threads, so the total number of threads equals $p \times t$. This strategy causes intense communication between the I/O subsystem and the compute clusters. We first distribute data points and replicate data centroids among clusters, and then loop over a two-phase iteration. First, partitions are populated. Then, data centroids are recalculated, which is a memory-intensive process. For this recalculation, each cluster uses its local data points to compute partial centroids, i.e., a partial sum of data points and population within a partition. Next, clusters exchange partial centroids so each cluster ends up with the partial centroids of the same partitions. Tasks in KM are irregular, since the amount of work for each thread may vary during each iteration. Finally, clusters compute their local centroids and send them to the master process.

5.6. LU Factorization - LU

LU is a matrix decomposition algorithm which factors a matrix A as a product of two triangular matrices: lower (L) and upper (U). We opted to implement Gaussian elimination to compute L and U, which requires $n-1$ iterations, where n is the order of A (always square). In each iteration, a sub-matrix of a smaller order (1 unit smaller) is analyzed, starting from order n. First, the biggest element in the submatrix (pivot) is found. Then, this element is moved to the $(1,1)$ position, shifting rows and columns appropriately. The line containing the pivot is subsequently divided by itself, thus the pivot element becomes 1. The last step (reduction) aims to nullify elements below the main diagonal. For every line l below the pivot, we multiply the pivot line p by the opposite of the first element e of l and replace l with $l+p$. We store $-e$ in a separate matrix. After the iterations are finished, every line will have undergone reduction, and the resulting matrix is the U matrix. The L matrix is formed by the $-e$ factors that were stored in the separate matrix. Therefore, both matrices are computed simultaneously. LU uses the Workpool parallel pattern.

Our MPPA-256 solution assigns rows to compute clusters so the largest element can be found. Each compute cluster receives no more than 1 MB of data, in this case, leading to intensive communication. On the other hand, the distributed task loads are regular. The same restriction of 1 MB applies when distributing lines among the compute clusters to apply reduction. Row swapping is done in the master process (I/O subsystem), so the pivot becomes the first element in the matrix. The I/O subsystem is also used to rebuild the L and U matrices from chunks processed in the compute clusters.

5.7. Traveling-Salesman Problem - TSP

The Traveling-Salesman Problem consists in solving the routing problem of a hypothetical traveling salesman. Such a route must pass through n towns, only once per town, return to the town of origin and have the shortest possible length. Our solution (TSP) is based on the branch-and-bound method using brute force [12]. It takes as input the number of towns and a cost matrix, and outputs the minimum path length. The algorithm does a depth-first search looking for the shortest path, pruning paths that have a bigger cost than the current minimum cost. This pruning introduces irregularities in the algorithm, since the depth-first search needs to discard branches depending on the order in which the branches are searched. TSP uses the Workpool parallel pattern.

The MPPA-256 implementation uses a task queue in which tasks are branches of the search tree. Compute clusters take jobs from the queue and run them. The number of clusters and the number of threads define the total number of lines of execution. For each cluster, n threads will be spawned, totaling $n\_threads \times n\_clusters$ threads. When the minimum path is updated, the new value is broadcast to every cluster so they can also use it to optimize their execution. At the end of the execution, one of the clusters (typically the 0-th) prints the solution. The final solution might be discovered by any one of the clusters, however all of them are aware of it due to the broadcasts of each path update.

DOI: 10.1002/cpe

Table I. Characteristics of each application available in CAP Bench.

App.	Parallel Patterns	Job Types	Communication Intensities	Task Loads
FAST	Stencil	CPU-bound; NoC-bound	High	Irregular
FN	MapReduce	CPU-bound	Low	Regular
GF	Stencil	CPU-bound	Average	Regular
IS	Divide and Conquer	NoC-bound	High	Irregular
KM	Map	CPU-bound; Memory-bound	High	Irregular
LU	Workpool	NoC-bound	High	Regular
TSP	Workpool	CPU-bound	Low	Irregular

5.8. Design Decisions

We had to make several decisions in order to develop suitable applications for the MPPA-256. Diverse parallel patterns were considered for each application running on the processor with regard to communication intensities and strategies, job types, and task loads. Table I summarizes the general characteristics of CAP Bench applications that allow for a better use of the MPPA-256 resources.

The applications that deal with image processing, such as FAST and GF, commonly have to take an image data array as input to operate on. The Stencil pattern, used in both FAST and GF, enables the use of low-level operations over highly parallel data that can be optimized by using the numerous MPPA-256 cores. The communication intensities when running FAST are higher than GF due to the chunk sizes, which are bigger in GF. Smaller chunks create more communication between the I/O subsystem and the compute clusters and therefore a heavier use of the NoC. There is a trade-off in FAST, which pertains to the fact that if we increase the chunk size, the compute cluster would become very overloaded by the irregular tasks.

The Divide and Conquer pattern is almost inherent to the bucket sort used in the IS application. This sort of strategy is effective when an optimal number of buckets are used, which is easy to achieve when using MPPA-256. We conclude that a big number of buckets with small sizes generates high communication intensities, by the constant use of the NoC. On the other hand, bigger buckets would not fit on the limited memory size of a compute cluster. Generally speaking, the division of numbers in many independent parallel units has no impact in the sorting algorithm's performance, even though the task load is irregular.

FN and KM use the Map parallel pattern. The former also uses the Reduce pattern. Applications using these patterns scale well in MPPA-256 due to the explicit parallelism of the architecture and patterns. As FN and KM are algorithms with highly independent tasks, the other patterns' results were worse than Map or MapReduce. FN tasks consist of operations over sets of positive integers. Each cluster receives one set of integers, regardless of the input size (i.e., there is only one send and one receive operation per cluster). Thus, FN does not use the NoC very intensely. In the case of KM, the opposite occurs. There is high-intensity communication between the I/O subsystem and the compute clusters to distribute the points.

The Workpool pattern could be easily used in any MPPA-256 application. Given a set of independent tasks, they are distributed by the master process between the slave processes. LU and TSP follow this approach, in which the tasks are coordinated by the I/O Subsystem (master process) (for instance in a task queue) and then delivered to any available compute cluster (slave process). This was the best pattern we found to develop both LU and TSP. The LU application is mainly favored by this pattern because of task regularity. The size of rows sent to compute clusters is the same, leading to balanced computations, improved by the Workpool strategy. Just like in other applications, this brings up the NoC-bound behavior. In the case of TSP application, the data set is smaller, and time to solution is mainly determined by the compute cluster speed (CPU-bound).

In order to enable testing with different workload sizes, we defined 5 input set sizes for applications in CAP Bench:

- **Tiny**: very small input sizes, meant to test the behavior of the applications;
- **Small**: small input sizes, meant for fast execution of the applications;

Table II. Input parameters.

App.	Tiny	Small	Default	Large	Huge
FAST	2048×2048	4096×4096	8192×8192	16384×16384	24576×24576
FN	$8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{12}$	$8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{13}$	$8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{14}$	$8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{15}$	$8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{16}$
GF	2048×2048 (i) 7×7 (m)	4096×4096 (i) 7×7 (m)	8192×8192 (i) 11×11 (m)	16384×16384 (i) 11×11 (m)	32768×32768 (i) 15×15 (m)
IS	2^{23} integer	2^{24} integer	2^{25} integer	2^{26} integer	2^{27} integer
KM	$2^{12}\mathbb{R}^{16}$ points, 256 centroids	$2^{13}\mathbb{R}^{16}$ points, 512 centroids	$2^{14}\mathbb{R}^{16}$ points, 512 centroids	$2^{15}\mathbb{R}^{16}$ points, 1024 centroids	$2^{16}\mathbb{R}^{16}$ points, 1024 centroids
LU	512×512 matrix	1024×1024 matrix	1536×1536 matrix	2048×2048 matrix	2560×2560 matrix
TSP	14 towns	15 towns	17 towns	19 towns	20 towns

- **Default**: medium-sized inputs, meant to resemble input sizes of typical application workloads;
- **Large**: larger-scale inputs, meant to be above-average in resource needs, but not much;
- **Huge**: very large input sizes, sometimes greater than 1 GB, meant to more thoroughly evaluate the processing and NoC performance.

Each application was designed to fully support all workload sizes. Applications related to image processing, such as FAST and GF, have different symmetric image sizes. FN and IS, which deal with integer numbers, have input parameters that are directly related to these numbers. Thus, the higher the number or the range of numbers, the higher the computation cost. LU is an application that deal with symmetric matrix operations, thus the workload is associated to the size of the matrix. KM is a clustering application that computes a number of centroids based on a set of points. If these parameters are increased, the workload increases proportionally. Finally, in the case of TSP, the number of towns is required in order to determine the computation cost. The addition of a single town in the problem set can substantially increase the computation cost. Table II summarizes the parameters for each application.

6. RESULTS

In this section we first present a workload analysis of all applications in CAP Bench. Hence, we show the performance scalability and energy consumption results.

6.1. Workload Analysis

To point out the main characteristics and potential bottlenecks on all applications, we measured master time, slave time and communication time. Master time corresponds to processing time in the I/O subsystem, and excludes data transfer time or blocking time. Slave time is the average processing time of all compute clusters. Communication time corresponds to the *total* data transfer time between master and slave processes, and excludes the time spent by the master process waiting for slaves to be ready to send or receive data. We considered a total of ten executions for each application and input size combination, and we observed a relative standard deviation below 1.00%.

The applications were executed with small, default and huge input sizes, using 2, 4 and 16 compute clusters respectively. The results are presented in Table III. Total time in the table corresponds approximately to the sum of master and communication times, since slave time happens in parallel relative to the other two. Frequently, communication time can be close to slave time, which happens because the referred applications use blocking communication in the I/O subsystem.

The results for FN show a predominance of slave and communication time over master time for *small* and *default* input sizes, but the opposite occurs for *huge* input size and 16 compute clusters (a master time of 69.02% of total time). This happens because, as input sizes grow, more comparisons to identify which numbers are friendly have to be made in the master process during the Reduce

DOI: 10.1002/cpe

Table III. Breakdown of execution times (in seconds) with different number of clusters and input sizes: small with 2 clusters (left), default with 4 clusters (middle) and huge with 16 clusters (right).

App.	Small – 2 Clusters				Default – 4 Clusters				Huge – 16 Clusters			
	Master	*Slave*	*Comm.*	*Total*	*Master*	*Slave*	*Comm.*	*Total*	*Master*	*Slave*	*Comm.*	*Total*
FAST	0.00	0.79	1.02	1.03	0.00	1.58	2.50	2.52	0.0	3.55	13.32	13.40
FN	7.47	213.94	213.94	221.43	29.90	214.04	214.04	243.95	478.31	214.60	214.61	692.97
GF	0.00	0.87	1.04	1.05	0.00	3.46	4.11	4.12	0.00	24.09	33.89	33.95
IS	47.21	18.94	19.69	66.95	93.86	9.66	10.97	104.90	523.89	6.06	11.06	535.09
KM	0.07	6.89	8.19	8.28	0.38	22.72	25.54	25.97	2.89	42.68	56.17	59.17
LU	1.92	1.42	36.17	38.63	4.66	2.42	121.06	127.41	14.73	2.93	566.72	589.15
TSP	0.00	1.88	0.24	2.18	0.01	35.46	0.05	38.66	0.02	254.30	0.73	325.94

procedure. Communication times remained constant for the tested input sizes because the amount of numbers to be sent to each cluster during the Map procedure remained almost constant as input sizes and available resources were increased. FN also has regular task loads, which makes the slave time equal for all clusters.

For FAST and GF, results indicate that the majority of the time is spent sending data chunks to compute clusters (an average communication time of 99.54% and 99.21% of total time, respectively). It is worth noting that the communication part refers to blocking communication. These applications, characterized by high or average communication intensities, would benefit from a faster NoC with smaller delays, as well as faster and more numerous compute clusters.

FAST slightly has better communication times than GF due to its smaller chunks, which flow faster through the NoC. The I/O subsystem is not a bottleneck for these applications. A similar behavior can be observed for KM, with the difference that it spends more time in the slave processes relative to communication than FAST and GF do (KM presented a slave time average of 80.94% of total time). This occurs because slave processes in KM run more computationally expensive code than FAST and GF slave processes, therefore requiring proportionally more time to perform their tasks.

A result that immediately draws attention to IS is that the time spent in the slave processes decreases as input sizes increase, with a slave time of 29.29% of total time with 2 clusters and the *small* input size, 9.21% for 4 clusters with *default* input size and 1.13% with 16 clusters and *huge* input size. This means that even though there are more numbers to be sorted they are distributed among more compute clusters, improving the slave time. This behavior corroborates the use of the Divide and Conquer pattern, mainly on the divide phase. Time spent in the master process tends to increase with input, and this is due to the fact that more merge operations need to be carried out in the master process running in the I/O subsystem when inputs are larger.

Communication would be a bottleneck for IS if the minibuckets sent to the clusters were bigger, as IS is really a NoC-bound application. In other words, as the size of the data sent each time is relatively small, communication happens more often, and the NoC is required many times. But the communication time was not excessively large, as shown in Table III, due to the small data packets flowing through the NoC.

For LU, communication is the key performance-halting factor. Even though both the I/O subsystem and compute clusters are used in this application, the vast majority of the total execution time can be attributed to the use of the NoC to send matrix chunks back and forth (an average communication time of 94.95% of total time). This behavior explains the increase in communication time, since more chunks have to be sent for bigger matrices and when more compute clusters are employed.

Communication time is not a bottleneck for TSP, which does not use the NoC heavily in spite of using the same Workpool parallel pattern as LU, a NoC-bound application. Both applications have a pool of tasks to be executed, but LU generates more tasks to be sent to the compute clusters. TSP also spends most of its execution time in compute clusters. However, unlike the GF and KM, little time is spent in communication. This behavior, as mentioned before, can be explained by the fact that TSP performs few data transfers.

Another particularity that is observed in TSP is the great difference between the total and slave time – for the *huge* input size, for instance, the slave time corresponds to 78.02% of total time. However, this behavior is expected due to the irregularity issue discussed in Section 5.7, related to

DOI: 10.1002/cpe

the branch-and-bound method. This also incurs in a reduction in average slave time, because some threads finish earlier than others, which brings the mean down.

6.2. *Performance and Energy Consumption*

To analyze the overall performance and energy consumption, we executed all the applications using every input size with 1, 2, 4, 8 and 16 compute clusters (Figure 3). We collected the average time-to-solution (TTS) and energy-to-solution (ETS) for the applications to evaluate how they scale when larger inputs and more resources are used. Additionally, a more detailed scaling test was conducted for all applications using the *default* input size, gradually increasing the number of compute clusters from 1 to 16 (Figure 4). The performance gains mentioned below use executions with 1 compute cluster as base for comparison. For each application and input size we ran the applications ten times, and observed a relative standard deviation below 1.00%.

Energy measurements were obtained from a proprietary tool called K1-POWER, which allows us to collect energy measurements of the whole chip, including all clusters, on-chip memory, I/O subsystems and NoCs when running the applications. According to the Kalray's reference manuals, energy measurements are accurate to within ± 0.25 W of the current power draw. In order to clarify how good this accuracy is, Table IV presents a sample analysis of the power consumption perceived on all applications when executed with default input set size. The minimum, maximum and average values of power consumption are presented, with the respective accuracy taking into account the ± 0.25 W margin. The worst accuracy value was 92.88% in FN, when running it with one cluster.

Table IV. Power consumption sample analysis (in Watts) of all applications with default input set size.

App.	Minimum	Accuracy	Maximum	Accuracy	Average	Accuracy
FAST	4.14	93.96%	4.34	94.24%	4.24	94.10%
FN	3.51	92.88%	9.22	97.29%	6.34	96.06%
GF	4.16	93.99%	4.35	94.25%	4.25	94.12%
IS	4.06	93.84%	4.73	94.72%	4.37	94.28%
KM	4.51	94.46%	7.70	96.75%	6.31	96.04%
LU	4.31	94.20%	6.88	96.37%	5.49	95.45%
TSP	4.34	94.24%	7.96	96.86%	6.28	96.02%

FAST, FN, GF, KM and TSP have similar well-scaling behaviors, which means that for a determined input size, TTS tends to decrease as more compute clusters are employed. These applications, as we can see in Table I, are all CPU-bound. IS and LU have their TTS mainly determined by the data transfers through the NoC, resulting in worse scalability results than the others.

Other exceptions are KM and GF for *tiny* input size. The inputs are so small that synchronization operations become a bottleneck as the number of compute clusters increases. Besides that, KM is a Memory-bound application which could not be executed with *large* input size using 1 compute cluster and with *huge* input size using 1 and 2 compute clusters, due to the 2 MB memory restriction in them. Concerning the applications that follow the Stencil pattern, the chunks in GF are bigger than in FAST and therefore the compute clusters are less often used by the former.

Regarding energy consumption, the same decreasing behavior is observed in some applications: since consumed power is constant and applications take less time to finish when more resources are used, energy consumption is smaller. FAST, GF, KM and TSP do not rely heavily on the I/O subsystem for their operations, causing TTS and energy consumption to be closely related to the time spent on compute clusters.

On the other hand, the Reduce procedure in FN heavily relies on the I/O subsystem for comparing abundancies. Besides that, it is a CPU-bound algorithm composed of a high number of arithmetic operations. These characteristics led the average power consumption to increase linearly. With that in mind, when changing from 4 to 5 clusters, the average power consumed by FN was more determinant for the energy consumption than the TTS. Changing from 5 to 6 clusters, the power consumption increased slightly, thus, in this case, the TTS was more determinant. Although FN and KM have similar parallel patterns, the latter does not use the Reduce procedure on the I/O subsystem. Nevertheless, the overall time spent in compute clusters in KM substantially surpasses the time spent

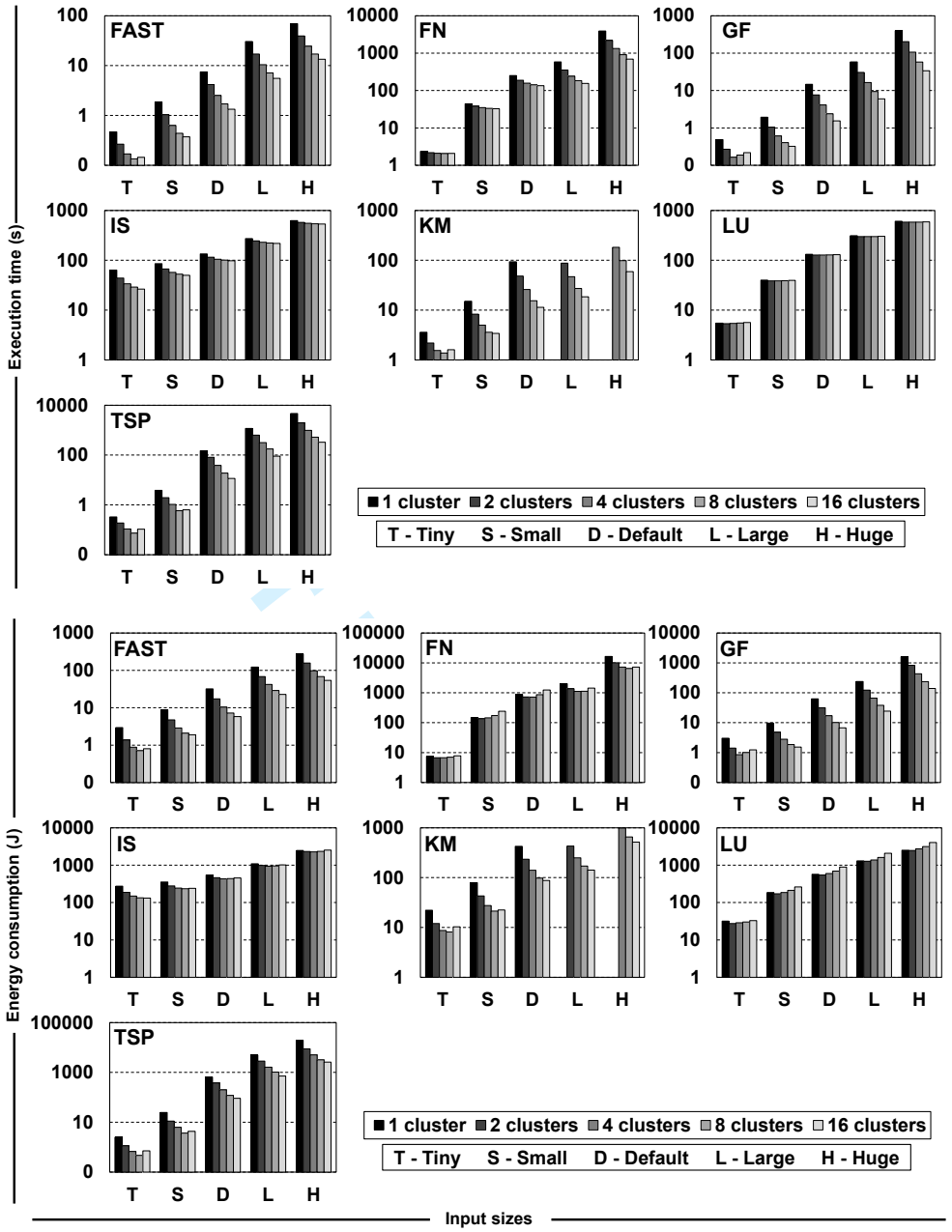


Figure 3. Execution time and energy consumption of all applications and input sizes.

on the I/O subsystem, causing TTS and energy consumption to decrease as more compute clusters are used.

The best execution time results for these applications were obtained with 16 clusters. In tests with 16 compute clusters, FAST obtained a decrease in execution time up to 82.24% and FN a decrease up to 92.91%. In GF, a reduction of about 89.80% was achieved. In KM and TSP, the achieved reduction in execution times was 87.84% and 93.01%, respectively.

In spite of achieving better results with 16 clusters, IS presented a decreasing scalability behavior, which means that performance gains tended to decrease as input set sizes increased. More precisely, the referred gain with 16 clusters was 58.42% with *tiny* input size, and 14.57% with *huge* input size. This happened because, as input sizes grow, IS relies more on the I/O subsystem to merge sorted

DOI: 10.1002/cpe

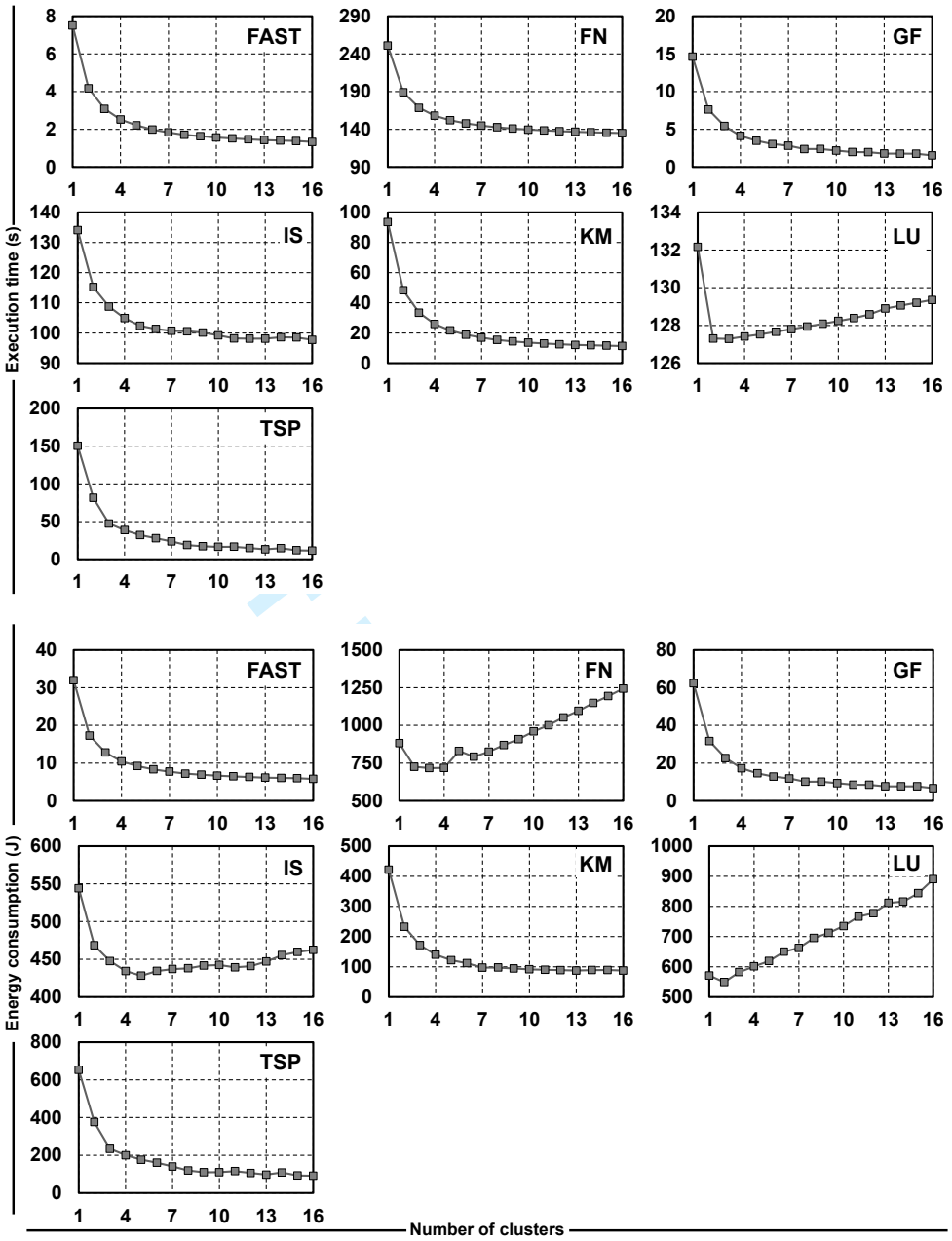


Figure 4. Detailed execution time and energy consumption results of all applications with default input set size.

minibuckets in the Conquer phase than sorting the numbers in the compute clusters. Larger input sets mean a larger number of minibuckets to be merged by the I/O subsystem, which in turn causes performance gains to diminish because the I/O subsystem has a constant number of cores and does not perform computations as well as the compute clusters.

Regarding energy consumption, Figure 4 shows that IS presented the most irregularities among all the applications in CAP Bench: it decreased proportionally when up to five clusters were used and then it increased almost linearly. This is because when more clusters are used, more communication takes place, causing the average power to increase linearly. The performance behavior is reflected

in energy consumption results, with smaller inputs presenting energy consumption gains which dwindle as input sizes augment.

The performance scalability of LU was negligible. Generally, we observed a more significant reduction in execution time when varying the number of compute clusters from 1 to 2. For more than 4 clusters, however, the overall execution time of LU usually increased. This poor scalability is justified by the fact that LU is heavily NoC-bound, and communication was its main performance bottleneck. As we increase the number of compute clusters we also increase the contention on the NoC. As a result, the amount of time spent in communications becomes higher than the amount of processing time. Overall, the highest reduction in execution time observed for this application was 4.21%, which was obtained with 4 clusters and *large* input size.

Regarding energy consumption, we can see in Figure 4 that it increases in a quasi-linear fashion for LU, except when increasing the number of compute clusters from 1 to 2, when a slight decrease is present. As in FN, the average power consumption in LU increases linearly when more compute clusters are employed. However, LU presented worse energy consumption results due to its poor scalability. The perceptible increase in energy consumption can also be attributed to the more intense use of the NoC, since bigger inputs will require more data chunks to be sent and more compute clusters also require the NoC to work more intensely.

To summarize, CAP Bench scales well enough to study the MPPA-256 architecture and its potential bottlenecks. We believe that CAP Bench applications cover important aspects of low-power many-core architectures, being useful to study emerging and upcoming low-power many-core architectures in terms of performance, energy consumption and communication bottlenecks. It is worth noting that MPPA-256 has all of the previously mentioned constraints, and the parallel patterns and design decisions related on Section 5.8 proved to be appropriated to address them. For instance, the tradeoffs between the NoC use and the data sizes (chunks) is something that some applications could expose, also pointing out that the NoC speed might be improved in spite of increasing the clusters' memory size. The results corroborate that CAP Bench enables a solid evaluation concerning energy efficiency of current low-power many-core architectures, such as the MPPA-256, and can be further used to evaluate and compare the upcoming ones.

7. CONCLUSIONS

In this paper we presented CAP Bench, an open source benchmark suite that includes parallel applications suitable to evaluate emerging low-power many-core processors such as MPPA-256. The benchmark contains a diverse set of applications that evaluated key aspects of MPPA-256, namely the use of its compute clusters, I/O subsystem, NoC and energy consumption. We expose development difficulties and potential bottlenecks that can stem from the shift in development paradigm when programming for low-power many-core architectures. The results showed us that different applications can have different performance bottlenecks, which is why a solid knowledge about the low-power many-core architecture is necessary for the development of efficient programs.

Our analysis shows that CAP Bench is prepared for the analysis of low-power many-core processors such as the MPPA-256, being scalable and concerned with new trends on this type of architectures. To achieve good performance and scalability, we developed applications considering aspects such as parallel patterns, load balance and architecture limitations. This allowed us to evaluate several aspects of the MPPA-256.

Our benchmark explores the hybrid programming model, which is a trend in low-power many-core processors, following parallel patterns. This enables us to verify that, in the case of MPPA-256, communication time may surpass computation time, which would ideally never occur. This behavior was highlighted by the LU application available in CAP Bench, which may indicate that the NoC should be improved to achieve better performance on NoC-bound applications. In this manner, CAP Bench comes up with the proposal to identify such bottlenecks, revealing potential improvements that might be done in future many-core architectures.

Application development challenges are still out there, and have to be solved to enable the evaluation of next generation many-core processors. As future work, we intend to incorporate other

applications to the benchmark, to make it more diverse and allow for a better characterization of the architecture and its aspects. We also intend to extend the benchmark use to other many-core architectures, to achieve a broader understanding of them and the differences between many-core processors. Another future plan is to collect full-system simulation results to strengthen the hypothesis that CAP Bench can be used to evaluate emerging many-core architectures with specific characteristics. These characteristics could be adjusted in a simulator to reveal strengths and potential bottlenecks of the simulated architecture. Thus, the use of a simulator along with CAP Bench opens up a wide range of possible evaluations of NoC and memory characteristics of many-core processors.

ACKNOWLEDGMENTS

CAP Bench was born from a joint initiative involving four research groups: *Computer Architecture and Parallel Processing Team (CArT)* at Pontifical Catholic University of Minas Gerais (PUC Minas), *Compiler Optimization and Run-time Systems (CORSE)* at Laboratoire d'Informatique de Grenoble (LIG), *Distributed Systems Research Laboratory (LAPESD)* at Federal University of Santa Catarina (UFSC) and *Parallel and Distributed Processing Group (GPPD)* at Federal University of Rio Grande do Sul (UFRGS).

This work was supported by FAPEMIG, FAPERGS, FAPESC, CNPq, CAPES, INRIA and STIC-AmSud and was developed in the context of EnergySFE and ExaSE cooperation projects.

REFERENCES

1. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, *et al.*. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* 2008; **15**.
2. Francesquini E, Castro M, Penna PH, Dupros F, Freitas HC, Navaux PO, Mhaut JF. On the energy efficiency and performance of irregular application executions on multicore, {NUMA} and manycore platforms. *Journal of Parallel and Distributed Computing* 2015; **76**(0):32 – 48, doi:http://dx.doi.org/10.1016/j.jpdc.2014.11.002. URL http://www.sciencedirect.com/science/article/pii/S0743731514002093, special Issue on Architecture and Algorithms for Irregular Applications.
3. Padoin EL, Pilla LL, Castro M, Boito FZ, Navaux POA, Mhaut JF. Performance/energy trade-off in scientific computing: the case of arm big.little and intel sandy bridge. *IET Computers Digital Techniques* 2015; **9**(1):27–35, doi:10.1049/iet-cdt.2014.0074.
4. Shalf J, Dosanjh S, Morrison J. Exascale computing technology challenges. *High Performance Computing for Computational Science (VECPAR)*. Springer: Berkeley, USA, 2010; 1–25.
5. Simon H. Barriers to exascale computing. *High Performance Computing for Computational Science (VECPAR)*. Springer: Kope, Japan, 2012; 1–3.
6. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, *et al.*. The landscape of parallel computing research: A view from berkeley. *Technical Report*, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley 2006.
7. Ho R, Mai KW, Horowitz MA. The future of wires. *Proceedings of the IEEE* 2001; **89**(4):490–504.
8. Dally WJ, Towles B. Route packets, not wires: on-chip interconnection networks. *Design Automation Conference, 2001. Proceedings*, 2001; 684–689, doi:10.1109/DAC.2001.156225.
9. Guerre A, Ventroux N, David R, Merigot A. Hierarchical network-on-chip for embedded many-core architectures. *ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, IEEE: Grenoble, France, 2010; 189–196.
10. Bjerregaard T, Mahadevan S. A survey of research and practices of network-on-chip. *ACM Comput. Surv.* 2006; **38**(1).
11. de Dinechin B, Ayrignac R, Beaucamps PE, Couvert P, Ganne B, de Massas P, Jacquet F, Jones S, Chaisemartin N, Riss F, *et al.*. A clustered manycore processor architecture for embedded and accelerated applications. *IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE: Waltham, USA, 2013; 1–6.
12. Castro M, Francesquini E, Nguélé TM, Méhaut JF. Analysis of computing and energy performance of multicore, numa, and manycore platforms for an irregular application. *Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, ACM, 2013; 5.
13. Ramachandran A, Vienne J, Van Der Wijngaart R, Koesterke L, Sharapov I. Performance evaluation of nas parallel benchmarks on intel xeon phi. *Parallel Processing (ICPP), 2013 42nd International Conference on*, 2013; 736–743, doi:10.1109/ICPP.2013.87.
14. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News* May 1995; **23**(2):24–36, doi:10.1145/225830.223990. URL http://doi.acm.org/10.1145/225830.223990.

15. Shiue WT, Chakrabarti C. Memory exploration for low power, embedded systems. *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, ACM: New York, NY, USA, 1999; 140–145, doi:10.1145/309847.309902. URL http://doi.acm.org.ez93.periodicos.capes.gov.br/10.1145/309847.309902.
16. Jeffers J, Reinders J. *Intel Xeon Phi Coprocessor High Performance Programming*. 1st edn., Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2013.
17. Fleig T, Mattes O, Karl W. Evaluation of adaptive memory management techniques on the tilera tile-gx platform. *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, VDE, 2014; 1–8.
18. Rossi D, Loi I, Conti F, Tagliavini G, Pullini A, Marongiu A. Energy efficient parallel computing on the pulp platform with support for openmp. *Electrical Electronics Engineers in Israel (IEEEI), 2014 IEEE 28th Convention of*, 2014; 1–5, doi:10.1109/EEEI.2014.7005803.
19. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, *et al.*. The gem5 simulator. *SIGARCH Comput. Archit. News* Aug 2011; **39**(2):1–7, doi:10.1145/2024716.2024718.
20. Niemann JG, Porrmann M, Ruckert U. A scalable parallel soc architecture for network processors. *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, 2005; 311–313, doi:10.1109/ISVLSI.2005.13.
21. Leng X, Xu N, Dong F, Zhou Z. Implementation and simulation of a cluster-based hierarchical noc architecture for multi-processor soc. *IEEE International Symposium on Communications and Information Technology, 2005. ISCIT 2005.*, vol. 2, 2005; 1203–1206, doi:10.1109/ISCIT.2005.1567085.
22. Freitas HC, Navaux POA, Santos TGS. Noc architecture design for multi-cluster chips. *2008 International Conference on Field Programmable Logic and Applications*, 2008; 53–58, doi:10.1109/FPL.2008.4629907.
23. Kondo M, Nguyen ST, Hirao T, Soga T, Sasaki H, Inoue K. Smyleref: A reference architecture for manycore-processor socs. *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, 2013; 561–564, doi:10.1109/ASPDAC.2013.6509656.
24. Tang Q, Mehrez H, Tuna M. Design for prototyping of a parameterizable cluster-based multi-core system-on-chip on a multi-fpga board. *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, 2012; 71–77, doi:10.1109/RSP.2012.6380693.
25. Mottin J, Cartron M, Urlini G. *Smart Multicore Embedded Systems*, chap. The STHORM Platform. Springer New York: New York, NY, 2014; 35–43, doi:10.1007/978-1-4614-8800-2_3. URL http://dx.doi.org/10.1007/978-1-4614-8800-2_3.
26. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, *et al.*. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 1991; **5**(3):63–73.
27. Henning JL. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 2006; **34**(4):1–17.
28. Bienia C, Kumar S, Singh JP, Li K. The PARSEC benchmark suite: Characterization and architectural implications. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, ACM: Toronto, Canada, 2008; 72–81.
29. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, 2009; 44–54.
30. Francesquini E, Goldman A, Méhaut JF. Improving the performance of actor model runtime environments on multicore and manycore platforms. *Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE)*, ACM: Indianapolis, USA, 2013; 109–114.
31. Kofman E, Millo JV, De Simone R. Application architecture adequacy through an fft case study. *Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, Sophia Antipolis, France, 2013; 25–28.
32. Wilkinson B, Villalobos J, Ferner C. Pattern programming approach for teaching parallel and distributed computing. *ACM Technical Symposium on Computer Science Education (SIGCSE)*, ACM: Denver, USA, 2013; 409–414.
33. McCool MD. Structured parallel programming with deterministic patterns. *USENIX Conference on Hot Topics in Parallelism (HotPar)*, USENIX Association: Berkeley, USA, 2010; 5–5.
34. Rosten E, Drummond T. Fusing points and lines for high performance tracking. *IEEE International Conference on Computer Vision (ICCV)*, vol. 2, IEEE: Beijing, China, 2005; 1508–1515.
35. Rosten E, Drummond T. Machine learning for high-speed corner detection. *IEEE International Conference on Computer Vision (ICCV)*, Springer: Graz, Austria, 2006; 430–443.
36. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2002; **24**(7):881–892.