Investigating Parallel Programming Paradigms in HeMPS MPSoC Platform

G. Lopes [1,3], A. Mello [1], E. Carvalho [2], C. Marcon [3]

[1] Ciência da Computação – Universidade Federal do Pampa (UNIPAMPA)
CEP: 97546-550 – Alegrete – RS – Brazil

[2] Centro de Ciências Computacionais – Universidade Federal do Rio Grande (FURG)
CEP: 96203-900 – Rio Grande – RS – Brazil

[3] Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
CEP: 90619-900 – Porto Alegre – RS – Brazil

{geaninne.mnl, alinevieiramello, ewerson.carvalho}@gmail.com,
cesar.marcon@pucrs.br

Abstract. *This work investigates the use of parallel programming paradigms in the development of applications targeting a Multiprocessor System-on-Chip (MPSoC). We implemented Matrix Multiplication, Image Manipulation and Advanced Encryption Standard (AES) applications in the Master-Slave, Pipeline and Divide-and-Conquer paradigms, and applied execution time and power dissipation as criteria for evaluating the performance of the applications executing according to the paradigms on an MPSoC architecture. The obtained results allowed us to conclude that there are optimal application-paradigm relations. Pipeline presents lower execution time and lower power dissipation for the Image Manipulation application; whereas, Master-Slave performs better for the Matrix Multiplication and AES applications. However, when the input size of the applications increases, the Divide-and-Conquer paradigm tends to minimize the execution time for Matrix Multiplication application. The main contributions of this work are the development of applications, considering different paradigms, and the impact evaluation of these paradigms on MPSoC architecture.*

1 Introduction

A Multiprocessor System-on-Chip (MPSoC) consists of an architecture composed of heterogeneous resources, including multiple embedded processors, dedicated hardware modules, memories, and an interconnection structure [Wolf, 2004]. The use of MPSoCs allows embedded systems to be cheaper and more efficient, since a single chip integrates multiple processors, reducing embedded system area, increasing the operation speed, and reducing power dissipation in the execution of applications. Therefore, the use of MPSoCs has become a trend in the embedded systems design [Tanurhan, 2006].

Some factors interfere with application performance in MPSoCs and should be considered to get an efficient application execution. Among these factors are the programming paradigm, which affects the behavior of the processors and the amount of information exchanged among application tasks. Despite the existence of several industrial MPSoCs, most of them have not yet reached the consumer. Carvalho (2009) cites as a possible cause for such effect the lack of suitable models of programming. Well-known programming paradigms applied to parallel applications, in the high-performance computing area, are Master-Slave, Pipeline, Phase-Parallel, and Divide-and-Conquer.

Since an MPSoC is a very complex architecture, at this first moment, the objective of this paper is to analyze the impact, in terms of execution time and power dissipation, of different programming paradigms in an MPSoC. With this purpose, three applications, with different behaviors and complexities, were implemented in different paradigms (Master-Slave, Pipeline, and Divide-and-Conquer), and simulated in an MPSoC built on the Hermes Multiprocessor System-on-Chip (HeMPS) platform. Evidently, the type of application also affects the choice of the best paradigm, and as it can be analyzed in the results, are optimal application-paradigm relations. However, the main focus of this paper is to analyze only the impact of the paradigm itself, and for that each application was implemented in the three paradigms.

Table 1 presents the related works, architecture, and paradigms used, as well as the purpose of each one. Some works, such as [Shee et al., 2006] and [Gorev and Ubar, 2014], evaluate different paradigms in multicore processors. These architectures differ structurally from an MPSoC, integrating from 2 to 8 cores, enabling the use of shared memory and bus interconnection. Due to the structural similarity between MPSoCs and computer clusters, used in [Raeder et al., 2011] and GPUs, it is believed that the same programming paradigms can be employed in MPSoCs. However, MPSoCs has limited power processing and storage capacity compared to the architectures cited previously. Table 1 shows that Souza et al. (2017), and Aguilar and Leupers (2015) propose a testbench for a Multi-Purpose Processor Array (MPPA), and a parallelism extraction tool, respectively. Although there are similar works, none of the above compares the effects resulting from different paradigms in an MPSoC. To our best knowledge, this is the first work that performs this evaluation targeting an MPSoC architecture.

Table 1. Summary of the related work.

Work	Architectu re	Paradigm	Goal
[Shee et al., 2006]	Multicore	PP and MS	Explores the parallelization of a JPEG application, using two paradigms, with heterogeneous components
[Raeder et al., 2011]	Cluster of PCs	MS, DC and PP	Proposes an analytical method to evaluate the best parallel programming paradigm for matrix multiplication
[Gorev and Ubar, 2014]	Multicore	DLP and PP	Combines paradigms, where it is preferable to use pipeline when data must be transferred several times
[Aguilar and Leupers, 2015]	MPSoC	PP, DLP and TLP	Proposes a tool that identifies multiple forms of parallelism from sequential embedded applications in a unified manner
[Souza et al., 2017]	MPPA-256	DC, Map, MapReduce, Stencil and Workpool	Presents a benchmark suite to evaluate a low-power many-core architecture
This	MPSoC	PP, DC and MS	Analyzes the execution time and power dissipation of MPSoC applications based on programming paradigms

Legend - MS: Master-Slave; DC: Divide-and-Conquer; PP: Pipeline; DLP: Data Level Parallelism; TLP: Task Level Parallelism.

This paper is structured as follows. Section 2 presents the theoretical basis for this study, including the descriptions of the target MPSoC and the programming paradigms. Section 3 presents the selected applications, and Section 4 refers to the development of these applications, considering their implementation according to each paradigm. Section 5 presents the adopted evaluation scenario and discuss the obtained results. Finally, Section 6 concludes the paper and suggests issues for future work.

2 Theoretical Background

This section presents concepts and definitions, regarding the investigated programming paradigms, and Hermes Multiprocessing System (HeMPS), which is the target MPSoC.

2.1 HeMPS MPSoC Platform

The applications implemented in this work were executed on HeMPS, a homogeneous and distributed memory MPSoC, composed by Plasma processors that are interconnected by Network-on-Chip (NoC) Hermes [Moraes et al., 2004]. Figure 1 illustrates the main elements of HeMPS, where the Plasma-IPs are Processing Elements (PEs). There are two types of PEs: the Cluster Manager Processor (CM) and Slave Processors (SPs). CM manages the MPSoC resources, while SPs are responsible for running user applications. Each PE contains a memory (partitioned in pages), a router, and a Direct Memory Network Interface (DMNI) for direct access between network and memory. The application task repository is performed by a memory placed out of the MPSoC, which contains the object codes of all tasks running on the system.

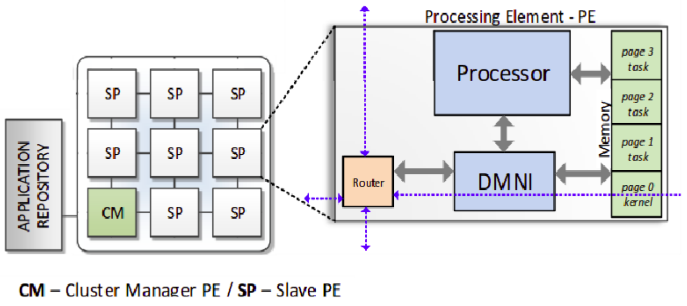

Figure 1. HeMPS MPSoC structure [Moraes et al., 2019].

Applications of HeMPS are represented through a task graph, where the nodes are tasks and edges are task communications; each task executes part of the application, cooperating and communicating with the others.

SPs execute the user applications employing an Operating System, which provides an Application Programming Interface (API) to implement application tasks. This API defines the structure for task communications and the following functions for performing system calls: (i) send and receive messages; (ii) capture the number of clock cycles performed by the processor; (iii) stop application execution and (iv) print log at the console. When the Send function is called, the DMNI copies the packet from memory and injects it into NoC. Then the packet stays in the pipe until it receives a message request from an application that called the Receive function. The HeMPS platform uses ".c" files for describing the computation and communication of the application task. Additionally, HeMPS requires two configuration files to simulate an application: (i) the Project file, which specifies the initial task of the application and the task dependencies; and (ii) the

Testcase file, which specifies hardware, software and application settings.

2.2 Parallel Programming Paradigms

Raeder et al. (2011) describes that the difficulty in defining the best parallel programming paradigm is one of the most significant problems in the high-performance computing area. Among the main paradigms are Master-Slave, Pipeline and Divide-and-Conquer. These paradigms imply the way that the application is implemented; thus, affecting the application performance. Following, we contextualize the paradigms already mentioned.

- **Master-Slave** consists of a master entity and multiple slaves. The master decomposes the problem into tasks, performs load balancing, sends the data to the slave tasks and stores the results. The slaves receive a message with the task, process this task, and return the result to the master;
- **Pipeline** requires dividing the application into sequential stages. Each stage runs part of the application and sends its results to the next stage, where another task is executed. Depending on the number of available PEs, the application can be executed through several parallel execution streams;
- **Divide-and-Conquer** decomposes an instance of a problem into smaller sub-instances that are resolved apart. The problem is decomposed until it is simple enough to have an immediate solution. At last, the partial solutions are combined until the entire solution is obtained.

3 Target Applications

This section presents the sequential versions of Matrix Multiplication, Image Manipulation and Encryption AES, which are the applications implemented in the present work. We choose these applications because they present different complexity degrees.

3.1 Matrix Multiplication

The most common method to perform matrix multiplication is to multiply the elements of the lines of matrix A by the elements of the columns of matrix B, followed by the sum of the resulting products [Press et al., 2007]. An algorithm for the multiplication of two square matrices of order n must contain three nested loops, which run through matrices from position 0 to $n - 1$ by performing the multiplications and applying the summation. This algorithm has complexity $O(n^3)$.

3.2 Image Manipulation

This application is introduced in Meyer (2016), and performs three transformations on an image represented in the RGB standard; each image transformation is explained next.

- **Gray Scale**: After averaging the primary values of each pixel, the values are replaced, resulting in a grayscale image;
- **Color Inversion**: Subtracts the maximum value (255) from the value of the pixel in question. Thus, the dark areas have become clear and vice versa;
- **Thresholding**: The image is converted to black and white, considering a threshold value. Each value is checked. If it is below the threshold, it is changed to black. Otherwise, it is replaced to white color.

Figure 2 shows an example of an output of the image manipulation algorithm, whose stages are sequentially executed. This sequentiality means that the image passes through the first processing stage, and its result serves as input to the second processing stage, and so on. This application is O(*log(n)*) complex.

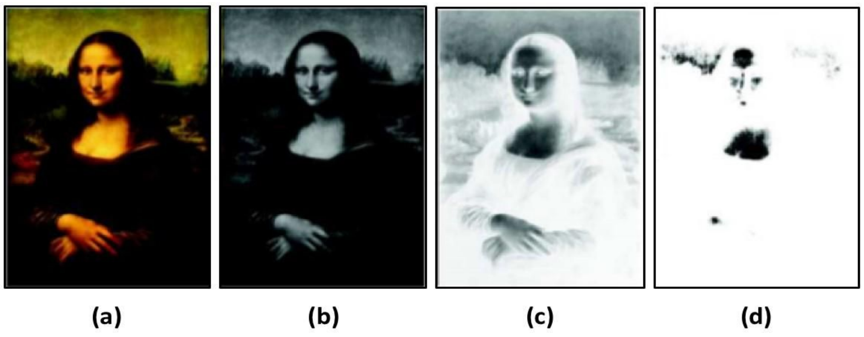

Figure 2. Example of Image Manipulation, starting from an (a) input image, and performs three sequential transformations: (b) Gray Scale, (c) Color Inversion and (d) Thresholding [Meyer, 2016].

3.3 Encryption AES

AES or Rijndael [Daemen and Rijmen, 2019] is a cryptographic algorithm that can be used to protect electronic data. This algorithm was designed to encrypt *n*-bit messages (*n* is multiple of 128), using ciphers of 128, 192, or 256-bit lengths. The encryption procedure is composed of different methods, which are executed in rounds, and the number of rounds depends on the key size.

Figure 3 shows the flow performed by the AES algorithm, which is O(*n*) complex. The application starts generating different subkeys for each round; subsequently, the application performs the four procedures explained below.

- **AddRoundKey**: For each round, a subkey is derived from the primary key. This operation adds the subkey to the current state using XOR logic operations;
- **SubBytes**: Transforms all bytes of the State array using the S-box table;
- **ShiftRows**: The rows of the matrix, which represents the block, are rotated to the left by 0, 1, 2 and 3 positions;
- **MixColumns**: Performs a linear transformation on each column of the array.

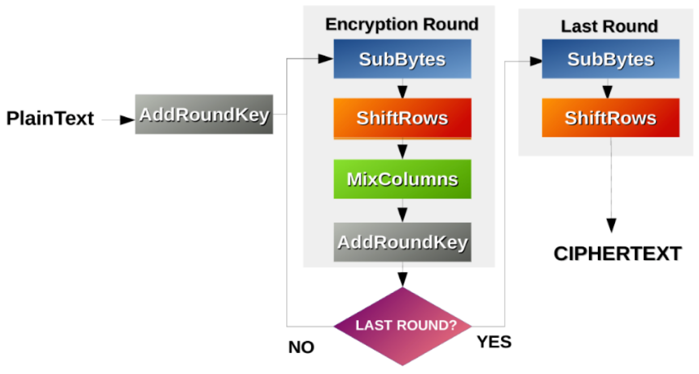

Figure 3. Example of the AES algorithm processing.

The applications were classified in Job Type, Complexity, and Number of Processing Phases (NPT). As it can be seen in Table 2, AES and MM applications have the same Job Type and only one processing task, so similar results are expected in the paradigm evaluation. The NPT is equal to the number of phases of the sequential algorithms that demands processing power. Then, Matrix Multiplication and AES were classified with just one phase, "Multiplication" and "Cipher", respectively. Whereas, Image Manipulation has three phases which demands the same processing. Job Types pertain to what resource is critical to the application [Souza et al., 2017]. This work classifies the application in CPU-Bound or Memory-Bound. A description of each job type is given below [Null and Lobur, 2014].

- **CPU-bound**: A system performance condition where a process or set of processes spend most of their execution time in the CPU or waiting for CPU resources;
- **Memory-bound**: A system performance condition where a process or set of processes spend most of their execution time in the main system memory or waiting for memory resources.

Table 2. Attributes for application characterization.

Application	Job type	Complexity	NPT
MM	CPU-bound	$O(n^3)$	1
MI	Memory-bound	$O(log(n))$	3
AES	CPU-bound	$O(n)$	1

4 Application Development according to the Parallel Paradigms

This section describes how Matrix Multiplication, Image Manipulation, and AES applications were implemented in each of the paradigms evaluated in this work. In all developed applications, the input data is stored in a one-dimensional vector. The communication between tasks is done by exchanging messages using the primitives Send and Receive of HeMPS.

The slave tasks (Master-Slave), leaf tasks (Divide-and-Conquer), and execution flows (Pipeline) represent the threads used. The data distribution is performed through a function that determines the amount of data each thread will process circularly. If the number of parallel elements is 10, and the amount of available threads is 3, threads 2 and 3 receive three elements and thread 1 receives four elements. The complete code of implemented applications in the different paradigms is available at https://bit.ly/2kRm0BI.

In the master-slave paradigm, the master task did the load distribution and sent the data to the slave tasks that performed the computation. In the Pipeline paradigm, applications were divided into different stages, and each stage was executed by different processors. And finally, in the division and conquest paradigm the root task performed the load distribution, and this division process was performed until the data arrived at the sheets tasks, which performed the computation. The subsections below discuss the particularities of each application.

4.1 Matrix Multiplication

Matrix Multiplication is implemented with two input square matrices (*A* and *B*). The load distribution is performed on the array lines *A* since all elements of a given line must be sent to the same task. The Pipeline stages of this application are: (i) Fill the matrices; (ii) Multiplication; and (iii) Integrate the results.

4.2 Image Manipulation

Image Manipulation is implemented using an input vector, where each element is a structure of pixel information. This vector represents RGB values, which are generated randomly. The workload distribution is performed on the pixels of the image since the RGB values of each pixel must be sent to the same task. The Pipeline stages of this application are: (i) Gray-Scale; (ii) Color Inversion; and (iii) Thresholding.

4.3 Encryption AES

Encryption AES was developed based on the work of [Rezende and Caimi, 2016], which implemented the application in the Master-Slave paradigm. The application has as input a message that is stored in a vector of size *n*. The input vector contains a sequence of letters, which are repeated 16 times. The encryption key has 256 bits. The workload distribution is performed in the same way as the other applications. In this case, the data are divided into blocks of 16 bytes, and the workload distribution is performed according to the number of blocks. The Pipeline stages of this application are: (i) Filling of the message to be encrypted; (ii) Generation of the subkeys and *AddRoundKey*; (iii) Encryption rounds; (iv) Last Round; and (v) Building the encrypted block.

Each paradigm presents a specific organization. Therefore, depending on the paradigm used, the number of tasks required to parallelize the application may differ. For example, the Master-Slave paradigm always employs the number of tasks equal to the number of slaves plus one. Whereas in the Divide-and-Conquer paradigm, the task number depends on the maximum number of children that each task can have. Additionally, in the Pipeline paradigm, the number of tasks depends on the number of the application stages. As the tasks can be allocated in different PEs, the time for communication, the power dissipated and the area used are larger. Thus, the number of tasks used must be justified by the performance obtained in parallel processing.

5 Obtained Results

This section presents (i) the setup and MPSoC configuration used in the experiments; (ii) the results obtained through the simulation of Matrix Multiplication, Image Manipulation and Encryption AES that were modeled in the Master-Slave, Pipeline and Divide-and-Conquer paradigms, (iii) the criteria adopted in the evaluation of the results, as well as (iv) the analysis and comparison of these results.

5.1 Experimental Setup and MPSoC Configuration

Figure 4 depicts the methodology employed for the experiments. For each application, we explored three paradigms, four or five input size and the number of threads; these variations result in 420 simulations. Through the log files generated in each simulation, we calculated the power dissipation and the execution time of each experiment. Finally, we generated graphs and analyzed the results.

All simulations were performed in the HeMPS MPSoC with the following characteristics: (i) 6×6 PEs, being 1 master PE and 35 slave PE; (ii) 256 KB of page size for each PE; (iii) One task running per PE; (iv) NoC buffers with 8-word depth; (v) Task mapping performed statically; and (vi) XY routing algorithm.

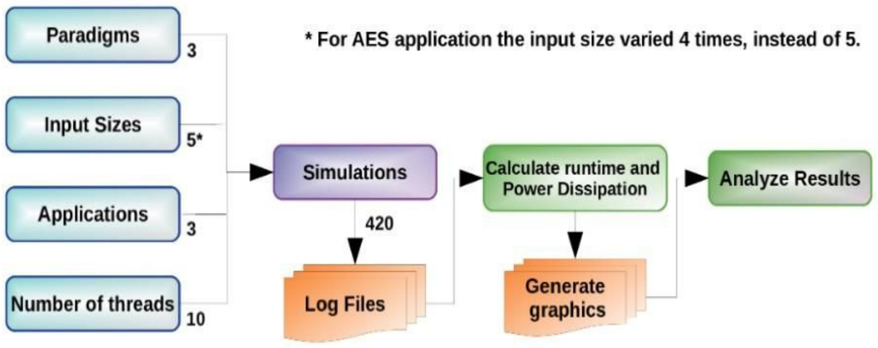

Figure 4. Methodology applied in the experimental results.

5.2 Evaluation Criteria

We used the execution time and power dissipated by the Plasma processors as evaluation criteria. The execution time, expressed in milliseconds (ms), is the time required to complete a test case. We inserted the function GetTick into the application to capture the execution time; this function returns the value of the current clock at the beginning and the end of the application. Thus, the execution time was obtained by subtracting the final time from the initial time.

We calculated the power dissipated in each simulation taking into account that instructions of a given class have similar power dissipation [Filho et al., 2012], and using the table containing the power dissipated by each class of instructions presented in [Guindani, 2014], which targets the HeMPS MPSoC implemented in 65 nm TSMC technology operating in 100 MHz. During the simulation, we summed the power dissipation of all the instructions that were executed, using a power dissipation table of each instruction class. The power dissipation model of the processor does not include memory accesses, but only the power dissipated due to the execution of instructions.

5.3 Result Analysis

This section presents and discusses the simulation results. We varied the number of threads from 2 to 10 and the input size for each simulation. Table 3 presents the shortest execution time, considering all paradigms, for each input size of each application, and its number of threads and power dissipation.

Table 3. Shorter execution time according to input sizes.

AP	Input size	Lower ET (ms)	NT	PD (mW)
MM	10×10	0.27 (DC)	2	0.0032
	10×10	0.27 (MS)	2	0.0032
	20×20	1.01 (MS)	4	0.0045
	30×30	2.49 (MS)	5	0.0071
	40×40	4.84 (DC)	8	0.0165
	50×50	8.31 (DC)	8	0.0270
IM	128	0.28 (PP)	3	0.0034

	256	0.51 (PP)	3	0.0037
	512	0.98 (PP)	3	0.0044
	1024	1.85 (PP)	3	0.0056
	2048	3.71 (PP)	3	0.0081
AES	256	0.69 (MS)	8	0.0051
	512	1.30 (MS)	8	0.0063
	1024	2.44 (MS)	10	0.0090
	2048	4.64 (MS)	10	0.0136

Legend: AP = Application; ET = Execution Time; NT = Number of Threads; PD = Power Dissipation; MS = Master-Slave; DC = Divide-and-Conquer; PP = Pipeline; MM = Matrix Multiplication; IM = Image Manipulation.

As it can be seen, in Matrix Multiplication application, the Master-Slave paradigm presents better results for smaller input sizes. However, for larger sizes, the Divide-and-Conquer paradigm has lower execution time, using a higher number of threads and dissipating more power. The Pipeline paradigm presents better results, using three threads, for Image Manipulation application for all input sizes simulated. AES application presents better results with Master-Slave paradigm. The lower execution time was obtained using 8 threads for smaller input sizes, and 10 threads for larger input sizes.

We generated two types of graphics for the larger input size of each application, one containing the execution time and another one comprising the power dissipation. The green, blue, red, and yellow lines represent the sequential version, Master-Slave paradigm, Divide-and-Conquer paradigm and Pipeline paradigm, respectively.

Case 1 of Figure 5 shows that the Pipeline paradigm presents the larger execution time for the Matrix Multiplication executing input matrices of order 50. Master-Slave results in shorter execution time for 2 to 7 threads, while using 8 to 10 threads, Divide-and-Conquer presents the shortest execution time. For example, considering matrices of order 50 and 10 threads, the Divide-and-Conquer implementation reduces 1.6%, 64% and 68% the execution time when compared to the Master-Slave, sequential version, and Pipeline paradigm, respectively. The best results for the Divide-and-Conquer paradigm by increasing the number of threads can be explained by a possible bottleneck in the communication between slave tasks and the master task in the Master-Slave paradigm. Then, the Divide-and-Conquer paradigm tends to reach shorter execution time than the other paradigms for matrices of order higher than 50. Moreover, in this scenario, it can be observed that the Divide-and Conquer paradigm, when configuring a maximum of 2 children per task, gets better results when the number of threads equals to a power of 2 (4 or 8 threads). This is an expected result, as this way the load distribution is balanced.

Case 2 of Figure 5 illustrates the power dissipated by the paradigms in Matrix Multiplication application. As we can observe, the Pipeline paradigm dissipated more power than the others, which is expected since it presents the larger execution time. Whereas Master-Slave and Divide-and-Conquer paradigms had similar execution times, the dissipated power was considerably higher in Divide-and-Conquer. It can be explained by the fewer number of tasks, and the less intense flow of communication in Master-Slave paradigm. Master-Slave paradigm has dissipated even less power than the sequential version, allowing a good trade-off between execution time and power dissipation. Considering matrices of order 50 and 10 threads, the Master-Slave implementation dissipates 52%, 29% and 2.8% less power than the Pipeline and Divide-and-Conquer paradigms, and sequential version, respectively.

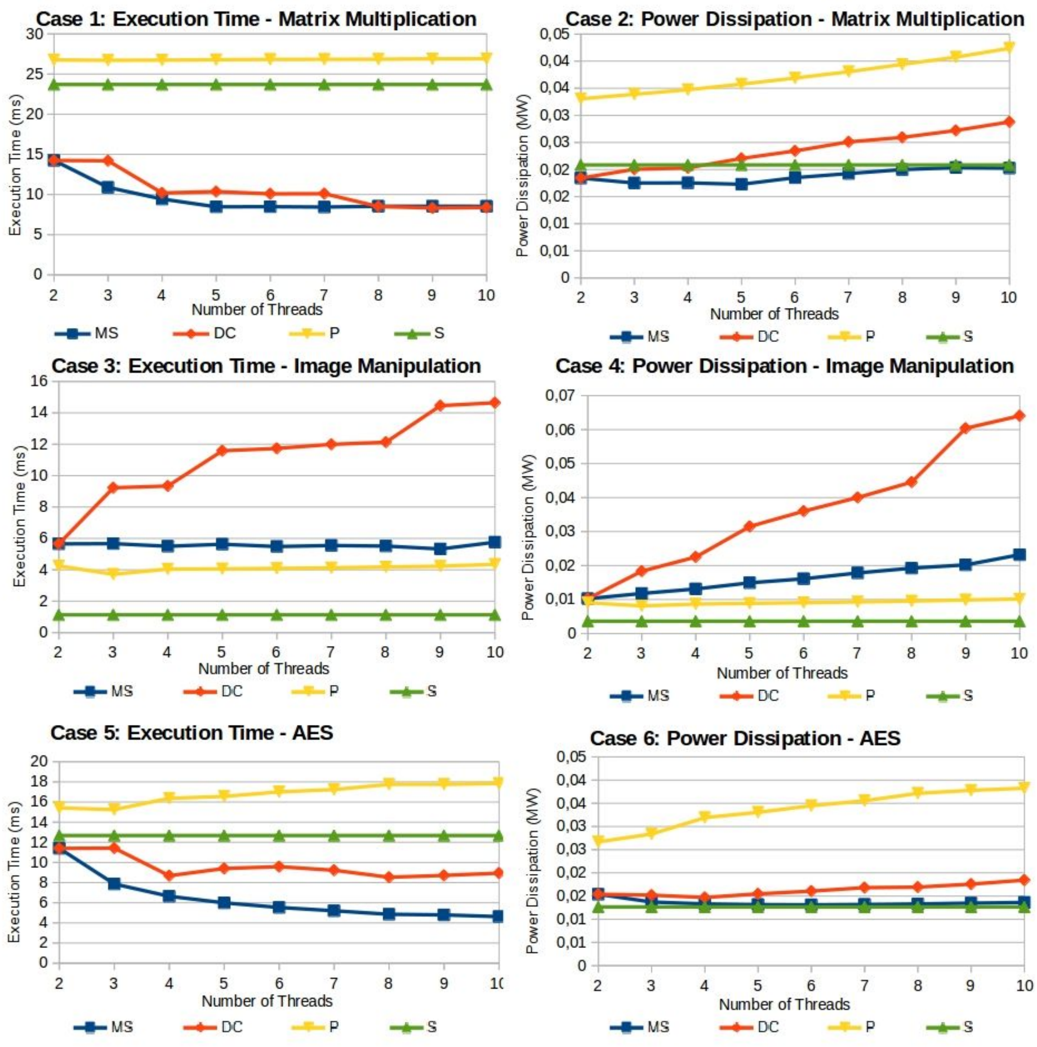

Figure 5. Execution time and power dissipation of six test cases.

Cases 3 and 4 of Figure 5 demonstrate that implementing the Image Manipulation application according to the Pipeline paradigm results is the lesser execution time and power dissipation when compared to other approaches. However, its results were worse than those obtained with the sequential version. For example, considering input images with 2048 pixels and 10 threads, the Pipeline implementation allows 24% and 70% shorter execution time than the Master-Slave and Divide-and-Conquer paradigms, respectively; however, the execution time of the Pipeline version is 26% longer than the sequential version. Additionally, the Pipeline paradigm for this same scenario dissipates 56% and 84% less power than the Master-Slave and Divide-and-Conquer implementations, but 36% more power than the sequential version. Thus, a parallelization of this application is not justified by the size of the input variables, which was limited by the platform.

Cases 5 and 6 of Figure 5 illustrate that the AES implementation obtained results of execution time and power dissipation similar to those obtained for the Matrix Multiplication application. The Master-Slave paradigm presents better results concerning both execution time and power dissipation compared to other paradigms. The Pipeline version dissipates higher power and executes for a longer time, even compared to the sequential version. The Divide-and-Conquer implementation dissipates approximately the same power of Master-Slave paradigm, but with longer execution time. For example, considering input messages with 2048 bytes and 10 threads, the Master-Slave approach executes 48% faster than the Divide-and-Conquer alternative, 63% faster than the sequential version and 74% faster than the Pipeline paradigm. Finally, for the same scenario, the Master-Slave implementation dissipates 26% less power than the Divide-and-Conquer and 64% less power than the Pipeline paradigms, but 7% more power than the sequential version. Consequently, we recommend using the Master-Slave paradigm for implementing an application with features similar to AES to allow a sound tradeoff between execution time and power dissipation.

The results of the 420 simulations, allow us to conclude: (i) Master-Slave may be limited by a communication bottleneck when the number of threads is increased; (ii) Pipeline is indicated only for applications that have sequential stages and with a balanced processing load, as the Image Manipulation application; (iii) For low-size data input, the Master-Slave paradigm implements Matrix Multiplication and AES applications more efficiently; (iv) There are optimal application-paradigm relations; (v) For large-size data input, the Divide-and-Conquer paradigm tends to obtain shorter execution times for the Matrix Multiplication application; however, the power dissipation tends to continue to be high due to the number of tasks used; (vi) The Divide-and-Conquer paradigm should have 2^n leaf tasks to balance the tree to obtain the best results; and (vii) The number of tasks affects the power dissipation, regardless of the programming paradigm used.

6 Conclusions

This work aims to analyze the execution of different programming paradigms in a HeMPS MPSoC. The execution time and power dissipation of three applications in Master-Slave, Divide-and-Conquer and Pipeline paradigms were analyzed. The analysis of this work indicates that the parallel programming paradigms, used in the high performance area, can also be used in an MPSoC architecture, considering some aspects as type of application, amount of data to be processed, number of processors, etc. In future work, paradigms can be reevaluated using mappings that consider the task communication graph, other MPSoC configurations, and NoC power dissipation can be explored. In addition, the application-paradigm relation should be further explored.

References

Aguilar, M. A., & Leupers, R. (2015, October). Unified identification of multiple forms of parallelism in embedded applications. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (pp. 482-483). IEEE.

Carara, E. A., De Oliveira, R. P., Calazans, N. L., & Moraes, F. G. (2009, May). HeMPS-a framework for NoC-based MPSoC generation. In *2009 IEEE International Symposium on Circuits and Systems* (pp. 1345-1348). IEEE.

Carvalho, E. (2009). Mapeamento dinâmico de tarefas em mpsocs heterogêneos baseados em noc. Pontifícia Universidade Católica do Rio Grande do Sul.

Daemen, J. and Rijmen, V. (1999). *Aes proposal: Rinjdael. aes algorithm submission,* “http://www. nist. gov/CryptoToolKit”, January, 2019.

Gorev, M. and Ubar, R. (2014, October). Pipelined execution of data-parallel algorithms. In *2014 14th Biennial Baltic Electronic Conference (BEC)* (pp. 109-112). IEEE.

Guindani, G. M. (2014). Mecanismo de controle de QoS através de DFS em MPSOCS. Pontifícia Universidade Católica do Rio Grande do Sul.

Johann Filho, S. (2012). Suporte para aplicações dinâmicas em sistemas multiprocessados intra-chip homogêneos. PUCRS.

Meyer, V. (2016). *PIPEL: Modelo de Gerência da Elasticidade para Aplicações Organizadas em Pipeline*. Universidade do Vale do Rio dos Sinos.

Moraes, F. et al., (2017) "Hemps Platform v7.3". http://www.inf.pucrs.br/hemps/docs/HeMPS_presentation.pdf , January, 2019.

Moraes, F., Calazans, N., Mello, A., Möller, L., & Ost, L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, *38*(1), 69-93.

Null, L., & Lobur, J. (2014). *The essentials of computer organization and architecture*. Jones & Bartlett Publishers.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

Raeder, M., Griebler, D., Baldo, L., and Fernandes, L. G. (2011). Performance prediction of parallel applications with parallel patterns using stochastic methods. *Sistemas Computacionais (WSCAD-SSC), XII Simpósio em Sistemas Computacionais de Alto Desempenho*, 1-13.

Rezende, L. and Caimi, L. (2016) "Desenvolvimento da Aplicação AES para HeMPS. http://www.github.com/GaphGroup/hemps/blob/master/hemps8.5/applications/aes , January, 2019.

Shee, S. L., Erdos, A., and Parameswaran, S. (2006, October). Heterogeneous multiprocessor implementations for jpeg:: a case study. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis* (pp. 217-222). ACM.

Souza, M. et al., (2017). CAP Bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience*, *29*(4), e3892.

Tanurhan, Y. (2006). Processors and FPGAs quo vadis?. *Computer*, *39*(11), 108-110.

Wolf, W. (2004, July). The future of multiprocessor systems-on-chips. In *Proceedings. 41st Design Automation Conference, 2004.* (pp. 681-685). IEEE.