Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

University of Padua

Department of Information Engineering

Master's Degree in
Computer Engineering

Increasing the Energy Efficiency of a Parallel Sparse Linear System Solver: Experiments with the COUNTDOWN Library

Supervisor:
Professor Carlo Fantozzi
Co-supervisor:
Professor Carlo Janna

Master Candidate:
Bortolin Simone
2038512

Academic Year: 2023/2024
Graduation Date: 07/03/2024

Abstract

This thesis investigates the environmental impact of solving linear systems with a parallel solver for sparse matrices. Possible optimization strategies are first explored through a literature review, with particular emphasis on software libraries. Then, the investigation concentrates on the COUNTDOWN library developed by Cineca, which promises a significant reduction in energy consumption (up to $25\,\%$) in MPI applications.

The thesis includes a review of basic mathematical concepts, such as linear solvers and preconditioners, before delving into the operation of parallel communication in MPI, which is essential for the chosen solver, named Chronos. An overview of available libraries is given, highlighting their strengths and limitations. Then, the COUNTDOWN library is analyzed in detail, its features are examined, and its impact on energy consumption is experimentally evaluated. The thesis concludes with a reflection on the limitations imposed by the operating system on the optimal management of CPU frequency, highlighting the challenges and opportunities for energy optimization in high-performance computing systems.

The work in this thesis has been partially supported by the Spoke 1 “Future HPC & Big Data” of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 - Next Generation EU (NGEU).

Contents

List of Figures

List of Tables

Thesis Outline

In the current context of growing environmental awareness and concern about climate change, the technology sector is at the center of a debate about its environmental impact. This debate is particularly relevant in the context of computing and cloud, where the rapid expansion of digital infrastructure has raised questions about the sustainability of current practices. To answer this question, data centers increasingly boast about their environmental choices, such as buying only green energy. Within this discussion, the present work aims to explore in detail the various dimensions of the environmental impact of computing and cloud, focusing on issues such as data center energy consumption, although there are many other challenges such as water consumption, e-waste management, and the impact of noise generated by technological infrastructures [1]. In this paper we aim to encourage critical reflection on current practices and promote initiatives aimed at reducing the environmental impact of High Performance Computing (HPC).

Until a decade ago, HPC was primarily used for scientific and engineering computations, but now HPC is used by computer engineers as well as scientists and engineers, and its use has increased dramatically as we have entered the age of artificial intelligence, machine learning, cryptocurrency mining, and the seemingly infinite data storage capacity made possible by cloud computing.This increase in HPC use requires greater attention to the efficiency of these processes.

Specifically, in this thesis we want to evaluate the energy impact, energy consumption and potential savings obtained by using a library developed by the CINECA consortium when applied to a sparse matrix linear equation solver developed by a spin-off of the University of Padua. The aim is to study the performance of the library, both in terms of performance and profiling, and to identify possible improvements or novel strategies. Using a combination of experimental simulations and analytical studies, this work aims to provide a comprehensive analysis of energy consumption patterns and computational efficiency.

The paper is organized as follows: chapter 1 gives a general introduction to the topic, listing the main topics covered; after this chapter, we begin with the mathematical premises, described in chapter 2. Chapter 3 discusses Chronos, a linear solver designed to run on clusters of computers and communicate via MPI, described in chapter 4. After that, the next chapters, 5 and 6, cover the state of the art relative to power management within computers and MPI, in particular Linux power management, and the current literature on algorithms for optimizing runs in MPI are described. Finally, chapter 7 provides an overview of the library we have chosen to see if it really brings about any energy savings through the experiments in chapter 8. In this chapter, an exploration is made between computational efficiency and energy consumption, where we try to find a delicate balance between the two to achieve optimal performance with the highest possible efficiency.

The analysis also assesses the practical implications of the results obtained, considering possible applications in various fields such as scientific research, engineering simulations, and high performance computing. Closing the analysis is chapter 9, which summarizes everything seen in the current work and presents ideas on how to pursue the path of energy efficiency in clusters, with some ideas for optimizing power consumption in big.LITTLE architectures, which are becoming increasingly popular.

1

Introduction

In this chapter we would see the main reasons why I decided to analyze this topic, namely the analysis of energy consumption and the potential that the use of energy saving algorithms and systems can bring. This chapter begins with an introduction to current parallel architectures, section 1.1, continues with sections from 1.2 to 1.4, which discusses the current environmental and ecological challenges that plague the world of supercomputers.

1.1 Parallelism in Computing Software and Architectures

Parallelism is a fundamental concept in computational science and engineering, offering a way to exploit the computational power of modern hardware architectures effectively. The method of sequential execution was great as long as the frequency of computers increased every year, but this stopped as the physical limits of silicon and manufacturing processes were reached in the early years of the new millennium, then the engineers involved in CPU development said that the best way to solve this problem was to create parallel systems, so that there were more processors running things in parallel. This marked a change from depending on faster clock speeds, which necessitated greater power usage, to a parallel approach that focused always on increasing performance but also on energy efficiency, since a system with two CPUs clocked at 2.0 GHz consumes less than a system with one CPU clocked at 4.0 GHz, as described in section 5.1.2. This approach led to the incorporation of various levels of parallelism in the hardware.

Insufficient use of parallel programming techniques is, again, a major limitation in most software frameworks. This problem is exacerbated by the common tendency not to exploit parallelism even in the slightest; especially on the part of programmers who are often so preoccupied with the main problems of parallelism, problems studied and solved, through excellent techniques for years, such as deadlocks, that is, that there are two processes that wait for each other, and starvation, that is, there are processes that never execute since others are stealing their resources, they have consistently overlooked the potential advantages of parallelism in their program, relegating it to specialized applications. However, awareness of the importance of parallel programming is growing, albeit belatedly. This change is partly attributed to the increasing adoption of asynchronous call abstractions along with event-oriented programming, as seen in languages such as JavaScript, C# and on the Android platform. The abstraction of asynchronous calls, offers a quick and easy

method of moving code from the main thread to background workers that do the computation, however, is advantageous especially in applications involving user interaction and web servers, but fails to even remotely exploit the capabilities of parallel execution.

In fact the concept of parallelism within processors is not new, within a CPU there is the arithmetic unit, ALU, which is already highly parallel in itself, if you think about it the various bit-to-bit addition and subtraction operations are already performed in parallel, there are dozens of transistors that do a bit-to-bit sum and carry the rest back to the next bit which is then summed. Same thing is done with the various sequential arithmetic instructions or for those more difficult than a simple sum. The level of thinking about this aspect is seldom considered by developers, and it is not particularly meaningful to do so. The task of organizing instructions in the most optimal sequence for the processor is handled by the compiler. Instead developers worry about higher-level parallel execution problems, such as deadlocks and starvation.

In addition to asynchronous programming, numerous alternative forms of parallel programming have been devised in recent years, such as massively parallel programming, introduced by GPUs, TPUs, and FPGAs. These methods of execution on dedicated accelerators competes with, but does not replace, classical multiprocessor execution. Based on the design of parallel systems and their execution environments, the following classifications of parallel applications are recognized, with certain cases of overlap [2]:

Data-parallelism This technique involves applying the same operation to multiple data elements in parallel. It is ideal for tasks that can be broken down into smaller components, allowing each component to be independently processed on separate data elements. For instance, image-processing tasks like convolution or matrix multiplication exhibit data-parallel characteristics, where the same operation is independently applied to each pixel or element of the matrix.

A data-parallel job that operates on an array of n elements can be evenly distributed between all processors. Data parallelism is widely used in matrix calculations, offering easy parallelization and delivering outstanding results. A prominent example is the straightforward multiplication of matrices. An instance of data parallelism, as demonstrated in fig. 1.1a, involves an image processing system designed to detect cats. This system can be run either in parallel or sequentially, with the clear advantage that parallel execution is faster and does not pose any issues. In this case, the image processing system looking for cats the smallest entity that can be parallelized, constituting an individual activity.

Task-parallelism This method entails the simultaneous execution of multiple independent tasks. As the name suggests, this approach assigns distinct tasks to different processors, recognizing each task as the smallest unit of parallel execution.

In task parallelism, each task operates autonomously and can be executed concurrently with other tasks. This method is particularly advantageous for scenarios where tasks can not be subdivided into independent units, such as parallel web crawling or parallel sorting.

Task parallelism is widely used in vector or matrix analysis, each task dealing with a distinct calculation, such as one the sum and the mean, one the variance, one the median, offering trivial parallelization of functions. An example of task parallelism

is the classic analysis of an image, illustrated in fig. 1.1b, one task is concerned with seeing if there are cats, one with seeing if there are dogs, one with possibly extracting text, and so on. In this case, the processing system looking for cats, or dogs or text is the smallest entity that can be parallelized, constituting an individual activity.

Massive parallelism or Vectorization This method is the ability to perform numerous calculations or operations simultaneously by leveraging a very high number of processing units with a reduced time for synchronization. It is a programming paradigm that involves breaking down a large computational problem into smaller parts that can be executed in parallel on multiple processing units without having to wait for other resources to provide faster execution and better performance.

This type of programming typically involves data parallelism at a low level and with a high degree of parallelism. Massive parallelism finds widespread application in scientific simulations, machine learning, and the realm of big data analytics. This capability can be harnessed through diverse architectures such as GPUs, TPUs, and FPGAs. The potency of massive parallelism enables the handling of vast datasets and the execution of intricate computations that would be unfeasible or beyond the scope of possibility with sequential processing.

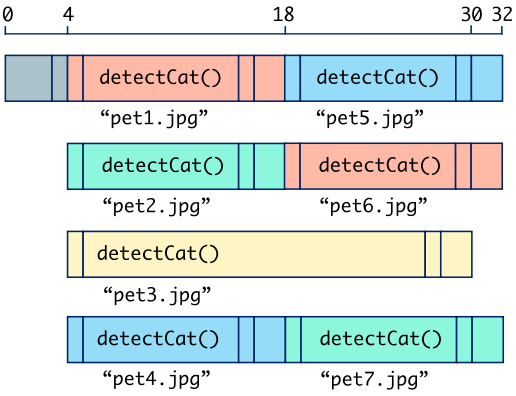
Vectorization is a specific form of massive parallelism, in which a computer program that operates on CPUs is transformed from a scalar implementation, which handles only one pair of operands at a time, to a vector implementation, which handles multiple pairs of operands simultaneously.

A practical example of massive parallelism is the classic multiplication of matrices, a task that demands access to numerous cells and can be highly parallelized, thereby avoiding synchronization delays through simultaneous execution by various threads.

Asynchronous event-based programming Asynchronous programming is a method that allows your program to initiate a task that may take a long time to complete and still remain responsive to other events while the task is ongoing, instead of being required to wait for the task to be completed. This form of programming typically involves task-oriented programming, where a user interface needs to react to user actions immediately. The event handler typically manages the user interface, initiating and stopping background tasks when there is any user interaction with the application or by running them on other threads.

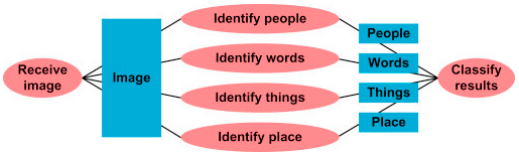
Typically, Data-parallelism and Task-parallelism are purely paradigms used at CPU level, while Massive parallelism is generally used in GPUs and hardware accelerators, while Vectorization is also used in CPUs through the Advanced Vector Extensions (AVX) and Streaming SIMD Extensions (SSE) instructions. It must be emphasized that an application may correspond to several types, for example it is generally easier to find task-parallelism applications that are also data-parallelism. Or Data-parallelism applications that make use of asynchronous event management.

Furthermore, according to the different hardware architectures. Four classes are identified, and the following classes are referred to as Flynn's taxonomy, illustrated in fig. 1.2 [5].



(a) Example of data parallelism applied to images: each image is analyzed in parallel.

Source: Kha [3].



(b) Example of task parallelism applied to an image: multiple war tasks are executed in parallel and each task is executed in parallel does something different.

Source: Oshana [4].

Figure 1.1: Example of date and task parallelism.

Single Instruction Single Data or SISD The SISD process executes instructions and handles data one at a time. It sequentially processes one instruction after another, without any parallelization.

Multiple Instruction Single Data or MISD In this method, we continue to manipulate a single data block while simultaneously executing multiple instructions.

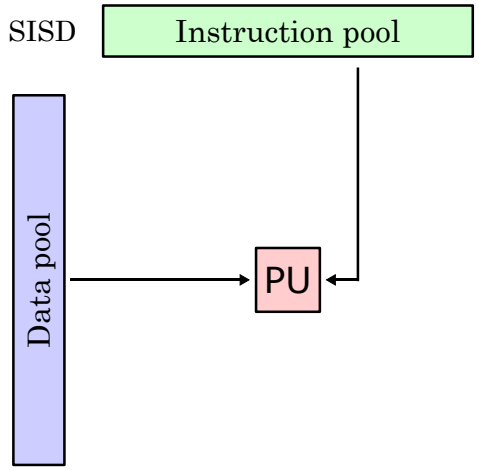
Single Instruction Multiple Data or SIMD The processing resources in question have control units that are shared among multiple cores. This particular design determines the features of these computing resources. One notable feature is the capability to execute a single instruction simultaneously on all available processing resources. As a result, the same instruction can be applied to a large set of data elements simultaneously, utilizing all available processing resources. However, it is important to note that not all processing resources in SIMD machines are universal. These machines typically have a limited set of instructions, which means that SIMD systems are often used for solving specific problems.

Multiple Instruction Multiple Data or MIMD In this case, every processing resource possesses its own control unit, allowing it to execute various instructions independently on a distinct set of data. As a result, this architecture can encompass multiple cores, CPUs, or even machines, enabling the simultaneous execution of different tasks on multiple devices.

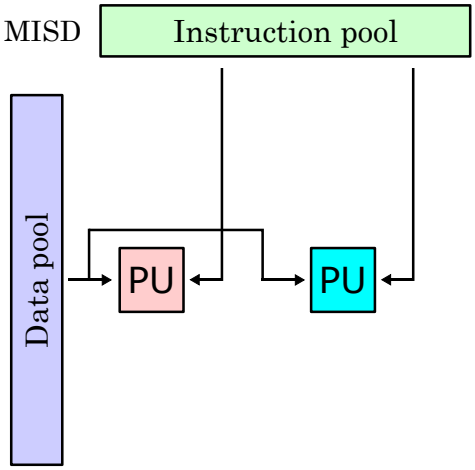
The MISD architecture may appear unconventional, but it is employed in scenarios where ensuring fault tolerance is crucial, such as the flight control computer of the Space Shuttle program.

1.2 Challenges of High Performance Computing

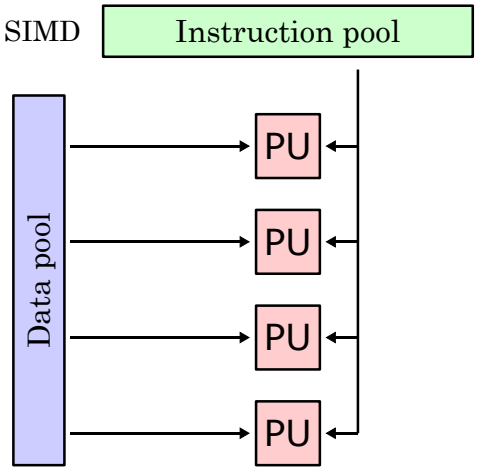
High performance computing (HPC) involves combining computing resources to achieve higher performance levels compared to a single workstation, server, or computer. An HPC



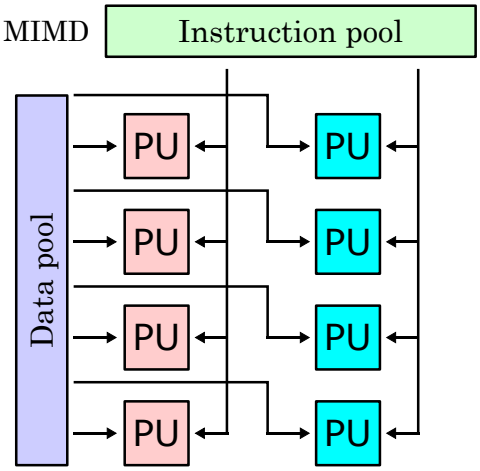
(a) Single Instruction Single Data.



(b) Multiple Instruction Single Data or MISD.



(c) Single Instruction Multiple Data or SIMD.



(d) Multiple Instruction Multiple Data or MIMD.

Figure 1.2: Flynn's taxonomy.

Source: Wikimedia Commons [6].

Figure 1.3: Architecture of a distributed computer.

Source: Kha [7].

system is a system consisting of many CPUs, GPUs, hard disks placed in parallel and interconnected at very high speed through a local area network. HPC systems can range from specialized supercomputers to clusters composed of multiple individual computers.

Typically in an HPC system a small group of two or four CPUs are part of a single node, each with several cores, thus forming a distributed architecture. Some of these nodes are intended for computation, others for storage, and they are all interconnected by high or very high-speed networks. A graphic illustration of a data-parralelism performed on a distributed system can be found in fig. 1.3.

High Performance Computing (HPC) presents a multitude of challenges, often stemming from the nature of executing complex computations on vast datasets. One prevalent issue is that of execution problems, that encompass a range of obstacles hindering the rapid execution of HPC applications. These problems may include bottlenecks in data transfer, resource contention, load balancing issues, and synchronization overheads, among others. Additionally, factors such as hardware limitations, software inefficiencies, and algorithmic complexities can exacerbate these challenges.

Despite this, a distributed system is only the basis of today's HPC, as there other problems. The main challenges are listed below.

Execution time and scalabilty In order to obtain results quickly and minimize execution queues, it is necessary for supercomputers to have a large number of nodes. This is since there is a growing demand for high performance and fast systems. In order to handle workload fluctuations, machines must possess scalability capabilities, i.e. to respond well to an increase or decrease in load, and optimisation capabilities, i.e. to manage nodes wisely by trying to assign to a load of 8 nodes, 8 nodes with a more optimised network path, furthermore, the load must be equidistributed among all the machines available and avoid that the machines with a lower name are subject to greater use than the others. This ensures that each machine is given appropriate periods of rest, enabling a balanced distribution of machine usage rather than some machines being more heavily utilized than others.

Fault Tolerance In HPC systems, the probability of hardware failures rises due to the presence of numerous components. It is essential to guarantee fault tolerance in order to uphold the reliability and availability of the system. To mitigate the consequences of failures and ensure uninterrupted operation, techniques like redundancy,

checkpointing, and error detection and correction mechanisms are utilized.

Power consumption and Energy Efficiency The power consumption of HPC systems is a significant concern, both from an environmental and economic perspective. Energy-efficient design and operation strategies are essential to reduce the environmental impact and operating costs of HPC facilities. This includes optimizing hardware components, implementing power management techniques, and exploring alternative cooling solutions. Optimizing energy consumption more can also be done with smart grid strategies, such as turning off some nodes when there is any sunshine and related photovoltaic production, or when energy is cheaper with hourly billing. Selecting efficient air conditioning systems that can effectively remove heat is crucial. In many cases, the amount of heat that needs to be dissipated makes it impractical to use traditional HVAC evaporators to release it into the external air. Instead, more advanced cooling systems like geothermal or other alternatives are required. By utilizing these more efficient systems, energy consumption can be significantly reduced. For instance, a geothermal system has the potential to decrease energy consumption by up to 40 percent.

Storage and Data Management The storage, retrieval, and analysis of large volumes of data produced and handled by HPC applications present substantial difficulties. To guarantee prompt data access and reduce storage expenses, it is essential to employ effective data management techniques such as data compression, caching, and distributed file systems.

1.3 Prioritize Power and Energy Efficiency

The emphasis on energy efficiency, even when faced with time constraints, is a novel concept in the current computing environment. While computational speed is typically prioritized, as we have seen an HPC system has many challenges, but the one we would analyze here is primarily a focus on Power consumption and Energy Efficiency, this topic perhaps being a recent concept, as computational speed is typically prioritized over energy efficiency. However, it is crucial to redirect our attention toward sustainable and environmentally-friendly computing solutions. This shift not only supports global initiatives to reduce carbon emissions, but also tackles the rising energy expenses linked to high performance computing.

For example, in 2007, Seager of Lawrence Livermore National Laboratory (LLNL) noted that the large consumption of electricity to power and cool his supercomputers leads to exorbitant energy bills, e.g., $\$14\,\mathrm{M/year}$ ($\$8\,\mathrm{M/year}$ to power and $\$6\,\mathrm{M/year}$ to cool) and in total spent a good $\$150\,\mathrm{M}$ of electricity for a system that cost $\$180\,\mathrm{M}$. At Los Alamos National Laboratory (LANL), the building for the ASC Q supercomputer cost nearly $\$100\,\mathrm{M}$ to construct [8]. The italian CINECA consortium in 2021 absorbed as much as $38\,\mathrm{GW\,h}$, or about $€2.2\,\mathrm{M}$. Therefore, it is essential to try to minimize the costs of powering and conditioning the data centers.

1.4 Top500 and Green500

The Top500 list is a comprehensive ranking of the 500 most powerful computer systems currently in use worldwide. First published in 1993, this list is updated twice a year to offer a detailed overview of the highest-performing systems. The ranking methodology is based on the Linpack benchmark, with the best performance achieved in floating point operations per second (FLOPs) serving as the primary yardstick. Unfortunately, the historical focus on speed has resulted in supercomputers known for their high power consumption and the requirement for advanced cooling systems. Currently, we have reached a level of computing performance that amounts to 10^{18} operations per second, utilizing the IEEE 754 Double Precision standard. This significant achievement has given rise to the popular term "Exascale computing".

Given the increasing environmental footprint and the increased energy consumption of high performance computing (HPC) facilities, researchers at Virginia Tech introduced the Green500 list. This list, initiated by Sharma et al. [8], Feng and Cameron [9], aims to reassess the Top500 list of supercomputers by placing greater importance on performance per Watt rather than focusing solely on processing power.

Although the Top500 persists in its role as a benchmark for computational prowess, the Green500 emerges as a crucial counterpart, directing attention to the ecological impact of HPC operations. The Green500 leverages the "FLOPs-per-Watt" metric. This metric underscores the efficiency of computational power utilization, providing a green perspective beyond raw processing capabilities.

Over time, the Green500 has been incorporated into the Top500 ranking in order to encourage the energy efficiency of supercomputers. This integration allows for community input of new and diverse perspectives on energy-efficient supercomputers. The Green500 endeavors to reshape the discourse around supercomputer evaluations, encouraging a balanced consideration of factors such as reliability, availability, and usability, ultimately contributing to a more environmentally responsible and economically feasible landscape in supercomputers.

Tables A.1 and A.2, in appendix, showcase the top 10 computers from the Top100 and Green500 lists. It is intriguing to note that among these, 3 supercomputers secure positions in both rankings, indicating a convergence between high performance and energy efficiency. At the same time, the remaining positions are filled by other systems, underscoring diversity in the landscape of supercomputer architectures and emphasizing the significance of considerations spanning both performance and energy efficiency in evaluating excellence in high performance computing.

The observation of a predominance of European supercomputers among the most energy-efficient is noteworthy and likely attributable to higher energy costs in the region [10]. The focus on energy efficiency in the Green500 list is in line with the wider global movement towards sustainable computing practices. European countries, often characterized by higher energy prices and a strong commitment to environmental sustainability, may be incentives to invest in supercomputer systems that prioritize efficiency to mitigate operational costs and minimize environmental impact. This regional trend underscores the influence of economic and environmental factors in shaping the landscape of energy-efficient supercomputers.

2

Linear Solvers

This chapter is dedicated to an overview of linear systems and their solving methods, focusing on large linear systems with a very large number of unknowns, solving a linear system is a very energy-intensive activity and widely used in simulations of physical and engineering processes. This necessity introduces the general problem of solving linear systems of equations, which can be found in both engineering and science. Solving linear systems of equations, as expressed in the form described in eq. (2.2), emerges as a critical computational task, typically accounting for a substantial portion, ranging between $70\,\%$ and $80\,\%$, of the total computational time in various domains of computational science and engineering. This prominence is particularly evident when problems in continuous solid mechanics are addressed by the implicit finite element method. In large-scale simulations, solving linear systems can be the most expensive task, accounting for up to $99\,\%$ of the total cost of the simulation. A linear system is represented as a set of equations, as shown in eq. (2.1):

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m \end{cases} \tag{2.1}$$

However, this representation does not fit well with the large linear equations found in physical and engineering models; so one prefers to express it in matrix form, where each individual coefficient $a_{ij}, i \in [1,n], j \in [1,n]$ can be seen as an element of the matrix $\mathbf{A} \in \mathbf{R}^{n\times n}$ that represents the system, every single known term b_i is part of the vector $\mathbf{b} \in \mathbb{R}^n$ and every single unknown term x_i is part of the vector $\mathbf{x} \in \mathbb{R}^n$. The number of equations is denoted by n. This results in the following matrix form, described in eq. (2.2).

$$\mathbf{Ax} = \mathbf{b} \tag{2.2}$$

Generally in linear systems found in engineering and science, one works with symmetric and positive definite matrices (SPD). An example of a symmetric and positive definite matrix (SPD) is shown in eq. (2.3), where the letters $a, b, ..., h$ represent the values of the diagonal and the colored dots represent the symmetric and positive values in the matrix.

$$\mathbf{A} = \left[\begin{array}{cccccccc} a & & & & & & \bullet & \bullet \\ \bullet & b & & \bullet & & \bullet & & \\ & & c & & & & & \bullet \\ & \bullet & & d & \bullet & & & \\ & & & & e & & & \\ & \bullet & & & & f & & \\ \bullet & & & & & & g & \bullet \\ \bullet & & \bullet & & & & \bullet & h \end{array}\right] \tag{2.3}$$

In contemporary industrial applications, the number of unknowns, expressed as n, can quickly escalate to several hundred million. In contrast, research experiments have successfully tackled systems with billions of unknowns. The crucial difference lies in the problems: Industrial challenges are characterized by complex geometries, irregular discretization, and matrix coefficients that exhibit heterogeneity. In contrast, large-scale research experiments often involve incremental refinement of regular or relatively regular grids. Despite their smaller scale in terms of unknowns, real-world industrial problems present formidable challenges due to their inherent complexity arising from complex geometries, irregular discretization, and matrix coefficient heterogeneities.

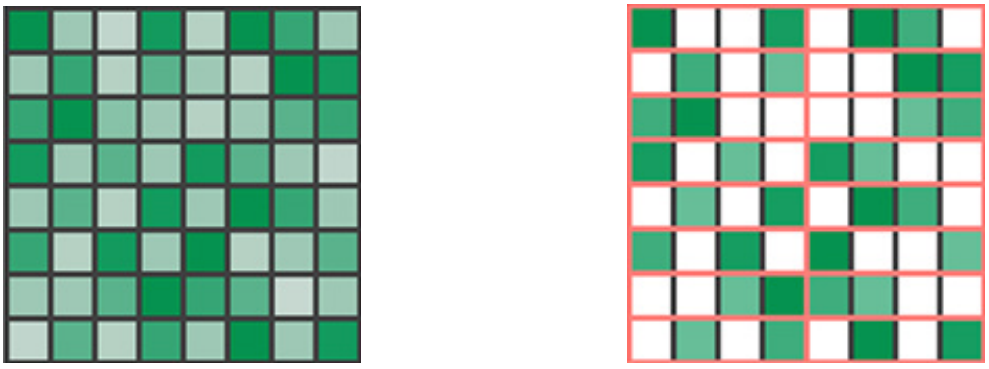
The nature of the matrices involved in these linear systems varies considerably. Sparse matrices, where the majority of elements are zero, are prevalent in simulations with irregular geometries or irregular discretizations. These matrices require special techniques for efficient solution methods. On the other hand, dense matrices, where most elements are nonzero, are common in simulations with regular or structured discretization. A graphical representation of dense and sparse matrices is given in fig. 2.1. Triangular matrices, whether upper or lower, also appear in certain computational scenarios and may require customized solution approaches. The size of these matrices ranges from the small scale encountered in research experiments to the large scale characteristic of industrial applications. The sections from 2.1 to 2.4 describe the functionality of a linear system solver through the approximation methods.

For nonlinear problems, a more complex approach is required. Within each static time step, the system of nonlinear equations is linearized and a solution is sought using the Newton-Raphson (NR) method [11, 12].

2.1 Algebraic Solver

In the context of finite element simulations, the matrix $\mathbf{A}$ associated with eq. (2.2) is characterized by sparsity, symmetry, and often positive definiteness. This is especially true for problems in solid mechanics. For linear problems, a singular solution to eq. (2.2) is adequate. Several methods exist to solve eq. (2.2), encompassing both direct and iterative methods that demonstrate excellent performance on parallel computers.

The most straightforward method to solve eq. (2.2) is by calculating the inverse of matrix $\mathbf{A}$. This process can be accomplished through various techniques, including Crout's



(a) Dense Matrix. (b) Sparse Matrix.

Figure 2.1: Graphical representation of a dense matrix, where there are no zero values (white) and many non-zero values (green), and a sparse one where there are many zero values and few non-zero values.

Source: Pool et al. [13].

factorization (**LU**) and Cholesky factorization. Crout's factorization (**LU**) decomposes matrix **A** into two components: a lower triangular matrix (**L**) and an upper triangular matrix (**U**). Once matrices **L** and **U** are obtained, the system of equations can be more readily solved. Cholesky factorization, specifically applicable to symmetric and positive definite matrices, further simplifies the resolution process.

Methods involving the calculation of the inverse, such as Crout's factorization (**LU**) or Cholesky factorization, are generally excellent for small and dense matrices. A matrix **L** obtained by Crout or Cholesky factorization from the matrix in eq. (2.3) is seen as that represented in eq. (2.4). Where the coloured dots assume the same position and the white dots with a black border are new. However, for large-sized matrices or sparse matrices, there are additional considerations to take into account.

$$\mathbf{L} = \begin{bmatrix} a & & & & & & & \\ b & & & & & & & \\ & c & & & & & & \\ & & d & & & & & \\ & & & e & & & & \\ & & & & f & & & \\ & & & & & g & & \\ & & & & & & h & \end{bmatrix}. \tag{2.4}$$

Large-sized matrices may demand significant computational resources for inverse calculation, and in certain cases, the outcome might be influenced by numerical issues. Moreover, in the case of sparse matrices, where the majority of elements are zero, calculating the inverse becomes even more computationally expensive, and the sparse characteristics may not be fully exploited.

To address these challenges, alternative approaches are often preferred, such as iterative methods (e.g., the conjugate gradient method) or direct solution algorithms specifically designed for sparse matrices (e.g., METIS or SuperLU). These methods are designed to maximize computational efficiency and minimize resource usage when dealing with large or sparse matrices.

2.2 Preconditioner

A preconditioner [12] is an entity that is used in the context of solving linear systems, particularly in iterative methods, to improve the convergence of the algorithm. Simply put, a preconditioner is used to render the linear system so that it becomes more amenable to resolve with iterative solvers. In iterative methods, the preconditioner allows the iterative algorithm to converge more rapidly to the desired solution.

The concept of preconditioning arises from the need to solve large and sparse linear systems, which can be computationally expensive and difficult to solve with direct methods such as LU factorization. Preconditioning is a highly active research field that plays a critical role in solving linear systems of equations containing millions or even billions of unknowns.

There are various types of preconditioner, each suitable for the specific characteristics of the problems to be solved. Some common examples include Jacobi preconditioners, Gauss-Seidel preconditioners, incomplete LU factorization (ILU) preconditioners, incomplete Cholesky factorization (IC) preconditioners, and QR factorization preconditioners, among others.

Consider the linear system of equations described in eq. (2.2), for solving we need to inverse the matrix $\mathbf{A}$ and calculate $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. The fundamental concept of preconditioning involves the multiplication of the system by a matrix $\mathbf{P}^{-1}$, which serves as an approximation to $\mathbf{A}^{-1}$, so that $\mathbf{P}^{-1} \approx \mathbf{A}^{-1}$ in a specific manner, although determining the precise criteria for this approximation can be challenging.

We can obtain the left-preconditioned system, as shown in eq. (2.5a), or the right-preconditioned system, as shown in eq. (2.5b). In the second scenario, we must solve the equation $\mathbf{Px} = \mathbf{y}$ as an additional step [12].

$$\mathbf{P}^{-1}\mathbf{Ax} = (\mathbf{P}^{-1}\mathbf{b}) \tag{2.5a}$$

$$\mathbf{AP}^{-1}\mathbf{y} = \mathbf{b}. \tag{2.5b}$$

Among the many preconditioners to be mentioned, we focus on the Preconditioned Conjugate Gradient (PCG), in particular the accelerated Newton-Chebyshev polynomial preconditioner [14].

This preconditioner leverages Chebyshev polynomials, which are a type of orthogonal polynomial, to approximate the inverse of the matrix. By using Chebyshev polynomials in combination with Newton's method, the preconditioner aims to accelerate the convergence of iterative solvers by improving the spectral properties of the system.

The Newton-Chebyshev polynomial preconditioner is particularly effective for symmetric positive definite (SPD) matrices, where it can efficiently reduce the condition number of the system, leading to faster convergence of iterative solvers such as the conjugate

gradient method. A comprehensive explanation of the PCG method can be found in Bergamaschi et al. [14], the final expression of the discrete function that needs be minimized:

$$\Psi(\mathbf{h},\mathbf{u},\mathbf{p}) = \begin{bmatrix} \mathbf{h} & \mathbf{u} \end{bmatrix}^T \begin{bmatrix} \mathbf{G}^h & -\alpha\mathbf{B} \\ -\alpha\mathbf{B}^T & \mathbf{G}^u \end{bmatrix} \begin{bmatrix} \mathbf{h} \\ \mathbf{u} \end{bmatrix} + \mathbf{p}^T(\mathbf{A}\mathbf{h} - \mathbf{C}\mathbf{u} - \mathbf{q}). \tag{2.6}$$

The algebraic problem can be obtained by applying the first order optimality conditions:

$$\begin{aligned} \mathbf{G}^h\mathbf{h} - \alpha\mathbf{B}\mathbf{u} + \mathbf{A}\mathbf{p} &= \mathbf{0}, \\ -\alpha\mathbf{B}^T\mathbf{h} + \mathbf{G}^u\mathbf{u} - \mathbf{C}^T\mathbf{p} &= \mathbf{0}, \\ A\mathbf{h} - \mathbf{C}\mathbf{u} &= \mathbf{q} \end{aligned} \tag{2.7}$$

where α is typically around 1, $\mathbf{h} \in \mathbb{R}^{n^h}$ represents the hydraulic head on fractures, $\mathbf{u} \in \mathbb{R}^{n^u}$ represents the flux on the traces, and $\mathbf{p} \in \mathbb{R}^{n^p}$ represents the Lagrange multipliers. The vector $\mathbf{q} \in \mathbb{R}^{n^h}$ includes the boundary conditions and forcing terms. Generally, $n^p = n^h$, although in this problem, n^u may be either larger or smaller than n^h. The matrices in eq. (2.7) are defined as follows:

- The matrices $\mathbf{G}^h$ and $\mathbf{G}^u$ are symmetric positive semi-definite (SPSD) matrices, but they are often rank-deficient. The matrix $\mathbf{G}^h$ has a fracture-local structure, with block-diagonal elements that vary in size depending on the dimensions of each fracture. On the other hand, $\mathbf{G}^u$ has a global nature and operates on degrees of freedom associated with different fractures.

- The rectangular coupling blocks $\mathbf{B}$ and $\mathbf{C}$ are both in the set of real numbers $\mathbb{R}^{n^h \times n^u}$. These blocks are defined by the inner products between the basis functions of $\mathcal{H}^h$ and $\mathcal{U}^h$. The matrix $\mathbf{C}$ is fracture-local, meaning it has rectangular blocks whose size depends on the dimension of each fracture and the related traces. On the other hand, matrix $\mathbf{B} = \mathbf{C} + \mathbf{E}$ is a combination of matrix $\mathbf{C}$ and matrix $\mathbf{E}$, where $\mathbf{E}$ contributes globally and has zero entries in the positions corresponding to the nonzero entries of the rectangular blocks of matrix $\mathbf{C}$.

- The matrix $\mathbf{A} \in \mathbb{R}^{n^h \times n^h}$ is both symmetric positive definite (SPD) and fracture-local, meaning it has a block diagonal structure. Each diagonal block is obtained by discretizing the $\nabla \cdot (\mathbf{K}\nabla)$ operator over a fracture, and therefore has the typical structure of a 2-D discrete Laplacian.

Equation (2.7) can be written in a compact form as:

$$\begin{bmatrix} \mathbf{G}^h & -\alpha\mathbf{B} & \mathbf{A} \\ -\alpha\mathbf{B}^T & \mathbf{G}^u & -\mathbf{C}^T \\ \mathbf{A} & -\mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{h} \\ \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{q} \end{bmatrix} \implies \mathcal{K}_0\mathbf{x} = \mathbf{f}_0, \tag{2.8}$$

where $\mathcal{K}_0$ is a symmetric matrix with a leading block that has a deficient rank.

2.3 Algebraic Multigrid (AMG)

This is where Algebraic Multigrid (AMG) comes into play. AMG is an advanced technique focused on improving the efficiency of solving linear systems, especially those involving large sparse matrices. AMG employs multigrid methods, an algebraic approach, and techniques like aggregation, smoothing, and interpolation. The use of AMG preconditioning ensures that convergence is achieved in a few iterations, regardless of the mesh size or with only a minor dependence on it.

One major limitation of AMG preconditioning is that it is not yet a fully automated method, and instead relies on the expertise of the user to properly set up and fine-tune the parameters. Incorrect configuration of AMG solvers can result in slow convergence, excessively costly preconditioners, and in extreme cases, failure to obtain a solution [11].

Typically, an AMG method relies on some primary components, the interaction of which determines the overall effectiveness of the method:

Multigrid Methods These are iterative approaches to solving systems of linear equations. They use approximate solutions at different scales (or grids) of the computational grid, gradually reducing the size of the system for efficient problem solving.

Algebraic Approach Unlike multigrid geometric methods that operate directly on the geometric grid, AMG is based on an algebraic representation of the matrix of the linear system. This means that AMG can be applied even to matrices for which a natural geometric grid is not available.

Aggregation A key phase in AMG, where grid points are grouped together to form "aggregates". These aggregates are then used to build a more efficient representation of the system, accelerating convergence.

Smoothing This involves applying an inner preconditioner to dampen high-frequency error components.

Smooting also derives from the preconditioner. Similarly as before, the operator of $\mathbf{A}^{-1}$ is typically approximated as $\mathbf{M}^{-1}$, and is defined by the following equation:

$$\mathbf{S} = \mathbb{I} - \omega \mathbf{M}^{-1}\mathbf{A}$$

where $\mathbb{I}$ is the identity matrix and ω is relaxation factor to ensure $\omega\rho(\mathbf{M}^{-1}\mathbf{A}) < 2$.

$$\mathbf{M}^{-1} = \mathbf{G}^T\mathbf{G}$$

with $\mathbf{G}$ lower triangular. The second element of AMG is known as the coarse grid correction (CGC), which refers to the operation of $\mathbf{A}$-orthogonal projection. This operation is designed to handle the low-frequency components of the error. In classical AMG, the unknowns of a given level are partitioned into fine and coarse variables (F/C), with coarse variables becoming the unknowns of the next level. The system matrix is reordered according to this partitioning:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{ff} & \mathbf{A}_{fc} \\ \mathbf{A}_{fc}^T & \mathbf{A}_{cc} \end{bmatrix} \tag{2.9}$$

with $\mathbf{A}_{ff}$ and $\mathbf{A}_{cc}$ are square matrices of size, respectively, $n_f \times n_f$ and $n_c \times n_c$. The prolongation operator $\mathbf{P}$ is expressed using the F / C ordering eq. (2.9) as follows:

$$\mathbf{P} = \begin{bmatrix} \mathbf{W} \\ \mathbf{I} \end{bmatrix}$$

where $\mathbf{W}$ is an $n_f \times n_c$ matrix that stores the weights to interpolate variables from the coarse level to the fine level. Since the system matrix is semi-positive definite (SPD), the restriction operator $\mathbf{R}$ is defined using a Galerkin approach, which involves taking the transpose of $\mathbf{P}$. The coarse-level matrix $\mathbf{A}_c$ is obtained by performing a triple matrix product:

$$\mathbf{A}_c = \mathbf{P}^T\mathbf{A}\mathbf{P}$$

In real applications, it is always desirable to have fast convergence and rapid coarsening, which means having high F / C ratios. To achieve this, it is crucial to construct effective prolongation operators that can reconcile these conflicting requirements. Once all the components mentioned above have been defined, the setup phase of the two-level multigrid method is finished, and the iteration matrix is provided:

$$(\mathbf{S})^{\nu_2}\left(\mathbb{I} - \mathbf{P}\mathbf{A}_c^{-1}\mathbf{P}^T\mathbf{A}\right)(\mathbf{S})^{\nu_1}$$

where ν_1 and ν_2 denote the number of smoothing iterations executed prior to and following the coarse-grid correction, respectively. AMG employs smoothing techniques to improve iterative convergence, efficiently reducing approximate errors in solutions.

Coarsening The selection of coarse-level variables for the construction of the next level.

Interpolation defining the transfer operator between coarse and fine variables.

AMG preconditioning has a significant limitation in that it is not yet a fully automated method. It relies on the expertise of the user and often requires careful adjustment of the setup parameters. If the setup is incorrect, it can result in slow convergence or excessively costly preconditioners. In the worst scenario, it may even cause the solution to fail [11].

2.4 Factorized Sparse Approximate Inverse (FSAI)

Factorized Sparse Approximate Inverse is an alternative technique to calculate the preconditioning and Smoothing. The FSAI preconditioner $\mathbf{M}^{-1}$ for an SPD matrix $\mathbf{A}$ is defined in the classical manner:

$$\mathbf{M}^{-1} = \mathbf{G}^T\mathbf{G} \simeq \mathbf{A}^{-1} \tag{2.10}$$

where $\mathbf{G}$) is computed by minimizing the Frobenius norm of eq. (2.11).

$$\|\mathbb{I} - \mathbf{G}\mathbf{L}\|_F \tag{2.11}$$

over the collection $\mathcal{W}_{\mathcal{S}}$ of matrices that have a specified non-zero pattern $\mathcal{S}$ in the lower triangular part. The matrix $\mathbf{L}$ in eq. (2.11) is the lower triangular factor of $\mathbf{A}$ and is not

necessarily needed to obtain $\mathbf{G}$. In fact, by differentiating eq. (2.11) with respect to the entries g_{ij} of $\mathbf{G}$ and setting it to zero, we get:

$$[\mathbf{GA}]_{ij} = \left[L^T\right]_{ij} \quad \forall (i,j) \in \mathcal{S} \tag{2.12}$$

Given that the transpose of matrix $\mathbf{L}$, denoted as $\mathbf{L}^T$, is upper triangular and the pattern $\mathcal{S}$ is lower triangular, the expression for matrix eq. (2.12) can be restated as follows:

$$[\mathbf{GA}]_{ij} = \begin{cases} 0 & i \neq j, \\ l_{ii} & i = j \end{cases} \quad (i,j) \in \mathcal{S} \tag{2.13}$$

where $[\cdot]_{ij}$ represents the element in the i-th row and j-th column of the matrix enclosed in square brackets, and l_{ii} refers to the diagonal element at position i in matrix $\mathbf{L}$. Since $\mathbf{L}$ is not known, the value of l_{ii} in equation 2.13 is substituted with 1. The matrix $\widetilde{\mathbb{G}}$ is obtained by solving the following equation:

$$[\widetilde{\mathbf{G}}\mathbf{A}]_{ij} = \delta_{ij} \tag{2.14}$$

when multiplied by the Kronecker delta δ_{ij}, it is proportionally adjusted.

$$\mathbf{G} = \mathbf{D}\widetilde{\mathbf{G}}, \quad \mathbf{D} = [\text{diag}(\widetilde{G})]^{-1/2} \tag{2.15}$$

thus obtaining the matrix $\mathbf{G}$ used in the definition of FSAI (Equation (2.10)). The scaling (Equation (2.15)) ensures that the diagonal entries of the preconditioned matrix $\mathbf{GAG}^T$ are normalized to unity. Additionally, the Kaporin condition number of this matrix is minimized among all matrices $\mathbf{G} \in \mathcal{W}_{\mathcal{S}}$. The Kaporin number of a symmetric positive definite (SPD) matrix is defined as the ratio between the arithmetic and geometric mean of its eigenvalues, and it provides an indication of the number of iterations needed for the Preconditioned Conjugate Gradient (PCG) method to converge. The FSAI preconditioner is highly robust as it can be computed for any choice of the non-zero pattern $\mathcal{S}$, and the resulting preconditioned matrix is guaranteed to be SPD.

The primary computational expense in the FSAI configuration is solving a series of n small dense linear systems, where n represents the size of $\mathbf{A}$, as a result of the component-wise eq. (2.13). The cost is particularly influenced by the non-zero pattern $\mathcal{S}$, which can be chosen either statically, meaning it is predetermined, or dynamically, meaning it is generated during the computation of $\mathbf{G}$. In this study, we propose the utilization of supernodes in the static FSAI calculation [15].

3

Chronos

This chapter analyzes in detail Chronos [11], a library that solves large linear systems. Chronos is a solver developed by the University of Padua spin-off M3E (Mathematical Methods and Models for Engineering), this library is designed to run in parallel on HPC clusters as well as with GPU acceleration. This library is already very efficient in itself, and within this thesis the idea is to try to make it even more efficient by treating it as a black box, i.e. without modifying a single line of code.

To solve the linear system, Chronos uses Algebraic Multigrid (AMG, section 2.3) preconditioners or Adaptive Factorized Sparse Approximate Inverse (aFSAI) within high performance computing (HPC) clusters, as described in sections 3.1 and 3.3. It uses a hybrid approach that integrates Message Passing Interface (MPI) and OpenMP directives to enable multi-threaded communication and parallel execution. The MPI communication is described in chapter 4. This hybrid MPI-OpenMP implementation provides greater flexibility and efficiency compared to pure MPI by taking advantage of the fine-grained parallelism of modern computing resources.

Chronos uses object-oriented programming (OOP) to create a distributed matrix object that can be used for multiple purposes. These purposes include representing a linear system, a smoother, an AMG hierarchy, or a preconditioner. Regardless of which preconditioner is used, the same iterative methods can be used to solve linear systems or eigenproblems. The modular design allows seamless integration of CPU kernels with graphics processing unit (GPU) and field programmable gate array (FPGA) kernels, while maintaining the overall integrity of the library structure. Sections 3.3 and 3.4 describes all this.

Chronos is a fairly standard linear solver with AMG, but with minor differences, we see that in AMG, unlike most smoothers, Chronos implements the adaptive Factorized Sparse Approximate Inverse (aFSAI, section 2.4) for approximating the matrix $\mathbf{M}^{-1}$ within the AMG, where the matrix $\mathbf{M}^{-1}$ takes an explicit form. In addition, the cost of implementing aFSAI applications is typically much lower compared to Gauss-Seidel and Chebyshev methods. This is mainly because the number of non-zeros in the inverse matrix $\mathbf{M}^{-1}$ is generally only $20\,\%$ to $40\,\%$ of the number of non-zeros in the matrix $\mathbf{A}$. Chronos is distinguished from other options by its superior flexibility and efficiency when used in AMG.

Chronos includes a fourth component inspired by the principles of bootstrap and adaptive AMG. This component reveals hidden elements of the near-kernel of the linear operator when they are not previously available, introducing a valuable method for revealing latent structure within the computational process.

In particular, Chronos employs a Distributed Sparse Matrix (DSMat) storage scheme that optimizes memory usage and computational efficiency for large-scale simulations, as described in section 3.2. [11]. Finally, section 3.5 describes the distribution of the computational usage of the various components of Chronos.

3.1 Algebraic Solver and Preconditioner

The preconditioner used by Chronos are described in Bergamaschi et al. [14], this framework that takes advantage of the nice properties of matrix $\mathbf{A}$, that is, semi-positive definite (SPD), block diagonal, and such that its inverse can be applied exactly to a vector at relatively low cost and with polynomial acceleration.

From eq. (2.8) an appropriate permutation of $\mathcal{K}_0$ is used:

$$\mathcal{K} = \left[\begin{array}{cc|c} \mathbf{A} & \mathbf{0} & -\mathbf{C} \\ \mathbf{G}^h & \mathbf{A} & -\alpha\mathbf{B} \\ \hline -\alpha\mathbf{B}^T & -\mathbf{C}^T & \mathbf{G}^u \end{array}\right], \quad \mathbf{x} = \left[\begin{array}{c} \mathbf{h} \\ \mathbf{p} \\ \mathbf{u} \end{array}\right], \quad \mathbf{f} = \left[\begin{array}{c} \mathbf{q} \\ \mathbf{0} \\ \mathbf{0} \end{array}\right],$$

to avoid a singular leading block. Although the permuted matrix is no longer symmetric, the 2×2 principal submatrix has a block diagonal structure and, hence, is invertible. In a more compact form, the permuted system $\mathcal{K}\mathbf{x} = \mathbf{f}$ can be written as

$$\left[\begin{array}{cc} \mathbf{M} & -\mathbf{Z} \\ -\mathbf{W}^T & \mathbf{G}^u \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{u} \end{array}\right] = \left[\begin{array}{c} \mathbf{f}_1 \\ \mathbf{0} \end{array}\right].$$

3.2 Chronos DDMat and DSMat

The Distributed Dense Matrices (DDMat) and Distributed Sparse Matrices (DSMat) are managed by the DDMat and DSMat classes, respectively. The implemented classes offer a storage system for both mass and RAM memory. Both the DDMat and DSMat storage schemes necessitate partitioning the matrix into n_p horizontal stripes of consecutive rows, where n_p denotes the total number of active MPI processes. DDMat utilizes a storage scheme where each stripe is arranged in rows among the processes. This arrangement enhances memory access during multiplication operations, thus improving the efficiency of DDMat for linear systems with multiple right-hand sides and eigenproblems. Additionally, distributed vectors are stored as a single column DDMat. In DSMat, each stripe is subdivided into a matrix of Compact Sparse Row (CSR), as shown in fig. 3.1.

The CSR matrices use a local numbering system where the rows and columns of the block IJ are assigned numbers ranging from 0 to n_{I-1} and 0 to n_{J-1}, respectively. Here, n_I and n_J represent the number of lines assigned to processes (I) and (J), respectively. This approach allows the use of 4-byte integers, resulting in memory savings and improved efficiency [11, 14]. For a schematic representation of the DSMat matrix storage scheme implemented in Chronos, see fig. 3.2b.

Chronos uses the compressed sparse matrix (CSR) format to handle shared sparse matrices, with the CSRmat class responsible for their management. The DSMat storage scheme used in Chronos is particularly efficient for both the preconditioner computation

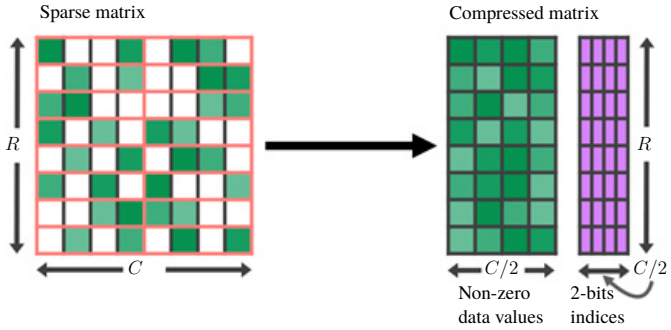
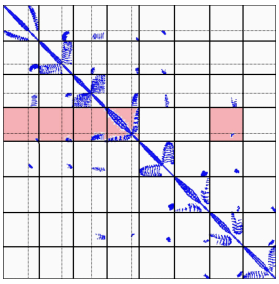


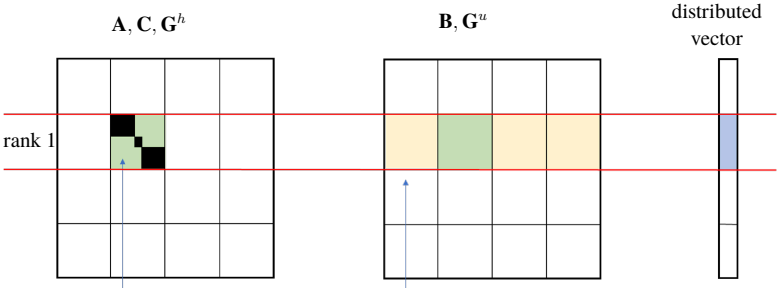
Figure 3.1: Graphical representation of the compression from a sparse matrix (left) to a CSR (right), the representation used by DSMat to store each individual subsection of the matrix.

Source: Pool et al. [13].



(a) Suddivision of the matrix using 8 MPI processes. The blue pixel represents a non-zero value. The red blocks are assigned to the process 3.

Source: Isotton et al. [11].



(b) Suddivision of matrix using 4 MPI process with the matrix $\mathbf{A}, \mathbf{C}, \mathbf{G}^h$ (left) and $\mathbf{B}, \mathbf{G}^u$ (center). The portions of the matrices and vectors stored by processor 1 are colored different colors.

Source: Bergamaschi et al. [14].

Figure 3.2: Schematic representation of the DSMat matrix storage scheme implemented in Chronos.

and the SpMV product. This efficiency is due to the significant overlap between communication and computation, which is illustrated in fig. 3.2a and further discussed in the following subsection.

3.3 Preconditioner and MatrixProd

The Preconditioner class operates at the highest level of abstraction, managing the approximation of the inverse of a Distributed Sparse Matrix (DSMat). It takes a DSMat-type object as input, along with an optional test space represented by a DDMat-type object. The derived classes from Preconditioner include Jac, aFSAI, and aAMG, each handling Jacobian-type, adaptive-FSAI-type, and aAMG-type preconditioners, respectively. Specifically, each of these classes has the capability to function as a smoother within the

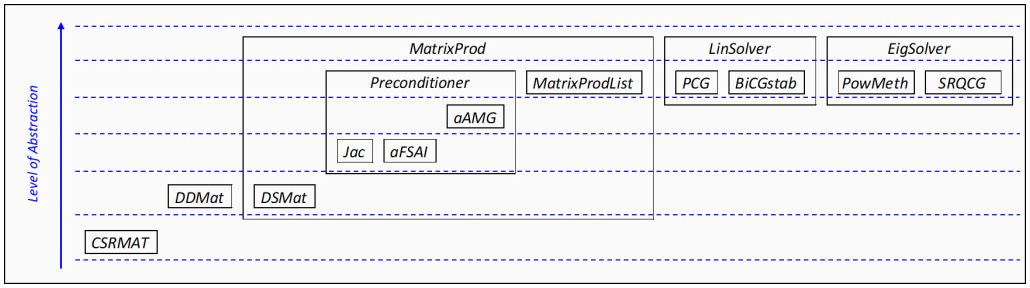


Figure 3.3: Chronos main classes and hierarchies.

Source: Isotton et al. [11].

AMG framework [11].

The DSMat and Preconditioner classes both derive from the MatrixProd class, which is responsible for managing the Sparse-Matrix-by-Vector product (SpMV) at the highest level of abstraction. The SpMV operation is known to be the most resource-intensive task in any iterative solver that utilizes preconditioning., has significantly influenced the design of the entire library. As depicted in fig. 3.3, the MatrixProd class plays a central role in the Chronos structure, working in conjunction with iterative solvers. Additionally, the MatrixProdList class facilitates the creation of more generalized MatrixProd elements. This class handles an implicit MatrixProd object, which is defined as the product of a sequence of MatrixProd objects arranged in a list [11].

3.4 LinSolver and EigSolver

At the highest level of the pyramid hierarchy, we encounter the solvers for linear systems and eigenproblems, specifically referred to as LinSolver and EigSolver, respectively. LinSolver is responsible for managing the Krylov methods used to solve linear systems. It requires a preconditioner and a linear system, both represented as MatrixProd-type objects, a right-hand side represented as a DDMat-type object, and an optional initial solution represented as a DDMat-type object. The LinSolver class currently has two derivatives: PCG and BiCGstab. PCG manages the Preconditioned Conjugate Gradient iterative method, while BiCGstab manages the Preconditioned Biconjugate Gradient Stabilized iterative method.

In contrast, EigSolver is responsible for handling the Krylov methods used to solve eigenproblems, requesting an optional preconditioner and a linear system as MatrixProd-type objects, together with a DDMat-type object representing the initial eigenspace. Currently, PowMeth and SRQCG are the two classes that inherit from EigSolver. PowMeth implements the Power Method, while SRQCG implements the Simultaneous Rayleigh Quotient Minimization iterative method [11].

3.5 Chronos MPI Call Distribution

The Chronos MPI call distribution, encapsulated within its hierarchy of classes, exhibits a balance with around 20 % of the computational workload allocated to setup and pre-

conditioning phases, and the remaining $80\,\%$ dedicated to the actual solution process, in particular in the Preconditioned Conjugate Gradient (PCG) class. This deliberate allocation strategy is deeply ingrained in the design of Chronos, where the lower-level classes, including DDMat, DSMat, and MatrixProd, collaboratively contribute to the setup and preconditioning stages. These foundational classes lay the groundwork, ensuring optimal management of distributed dense and sparse matrices, as well as handling the sparse matrix-by-vector product (SpMV) at the highest level of abstraction. As the hierarchy progresses, the computational responsibility is transferred, the Preconditioner class, with its derived subclasses such as Jac, aFSAI, and aAMG, further refines the preconditioning process, making it adaptive and efficient. Eventually, at the apex of the class hierarchy, the LinSolver and EigSolver classes take center stage, orchestrating the Krylov methods for linear system and eigenproblem solutions, respectively.

4

MPI (Message Passing Interface)

This chapter introduces MPI, the main communication system between nodes within parallel and distributed programming. The first section (4.1) analyses the main primitives that MPI offers and their various types of communication. The second section (4.2) analyses how communication between MPI processes is organized. The third and fourth sections (4.3 and 4.4) attempt to summarize the current literature on the efficiency of MPI communication and communication models.

MPI (Message Passing Interface) [16, 17] serves as a widely used standard for the programming of parallel and distributed computing systems, especially within high performance computing (HPC) environments. This framework provides a collection of functions and protocols designed to facilitate communication and coordination among multiple concurrently running processes.

MPI is one of the most popular programming models for developing parallel applications. MPI offers various point-to-point and collective communication primitives. Collective communication is an important subset of MPI operations that involve the coordination and exchange of data between multiple processes.

The MPI standard only defines communication between various processes and the various processes may be on different nodes, which means that the execution times of the processes on the various nodes may be different. This means that the waiting times for MPI primitives, which are always executed simultaneously on all nodes, can be different. MPI, in its default implementation, puts the process in a power state and performs a busy waiting to wait for all nodes to reach the MPI primitive [16, 17].

4.1 MPI Primitives

MPI communication operations are typically used when a group of processes needs to work together to perform a common task, such as distributing data, gathering results, or synchronizing their execution. These operations are designed to be efficient and are often optimized for specific hardware and network topologies. Primitives can be divided into point-to-point Communication and Collective Communication and can be identified into blocking and non-blocking.

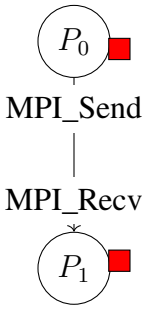
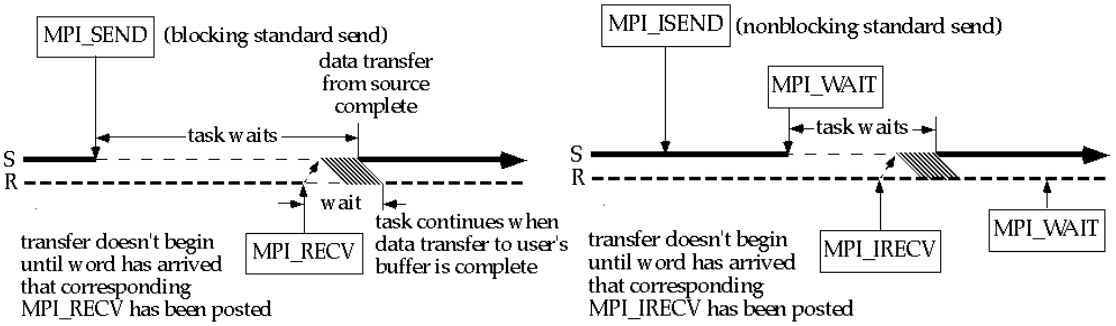


Figure 4.1: MPI Point-to-Point Communication.



(a) Flow example of a blocking and synchronous operation.

(b) Flow example of a non-blocking and asynchronous operation.

Figure 4.2: Flow example of Point-to-Point Communication.

4.1.1 Point-to-Point Communication

The main point-to-point instructions are `MPI_Send` and `MPI_Recv`, which are blocking and synchronous operations, and their non-blocking and asynchronous twins, `MPI_ISend` and `MPI_IRecv`. These instructions deal with sending and receiving data between two nodes; they are illustrated in fig. 4.1.

4.1.2 Blocking and non-Blocking Communication

MPI provides two primary communication modes: synchronous and blocking, and asynchronous and non-blocking. In synchronous and blocking mode, both threads involved in the communication must reach the same MPI call simultaneously to exchange data. The blocking wait mechanism for this mode is depicted in fig. 4.2a.

On the other hand, the asynchronous mode allows data to be stored in a sender buffer until the recipient process is ready to receive or send it. However, it is crucial to confirm that communication has taken place using the `MPI_Wait` function before reusing the buffer. This step is necessary and may result in a busy wait in one of the two threads, as illustrated in fig. 4.2b. Despite the increased efficiency of the non-blocking mode, its applicability is not universal, and there are situations where it may not be feasible or advisable to use it.

4.1.3 Collective Communications

Collective instructions are instructions for exchanging data and dividing data between all threads. MPI offers numerous intuitive techniques to divide and unite data, including `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather` and `MPI_Reduce`, they are illustrated in fig. 4.3, also called one-to-many or many-to-one instructions [16].

`MPI_Bcast` This operation allows one process (the root) to send data to all other processes in a communicator. It is useful when one process has data that need to be shared with all the others. The collective communication mechanism is shown in fig. 4.3a.

`MPI_Scatter` This operation allows one process (the root) to distribute data from one process (the root) to all processes in a communicator. Each process receives a distinct portion of the data. The collective communication mechanism is shown in fig. 4.3b.

`MPI_Gather` Gather is the reverse of Scatter. Collect data from all processes in a communicator and send it to one process (the root). The collective communication mechanism is shown in fig. 4.3c.

`MPI_Reduce` Reduction operations perform a specified operation (e.g., sum, maximum, minimum) on data from all processes in a communicator and return the result to one process (the root). The collective communication mechanism is shown in fig. 4.3d.

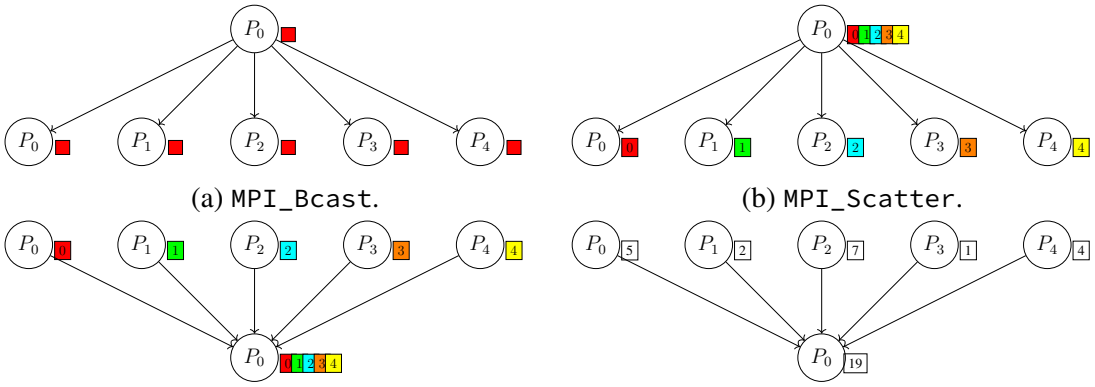
Finally, MPI also offers techniques for working on split data and exchanging portions of split data in many-to-many and not one-to-many or many-to-one mode; these instructions are: `MPI_Alltoall`, `MPI_Allgather`, `MPI_Allreduce` and `MPI_Reduce_Scatter`, there are illustrated in fig. 4.4 [16].

`MPI_Alltoall` This operation enables every process to transmit data to all other processes within a communicator and can be viewed as a collective transposition operation that operates on data chunks. It is useful when each process needs to share data with all others. `MPI_Alltoall` works as combined `MPI_Scatter` and `MPI_Gather`. The collective communication mechanism is shown in fig. 4.4b.

`MPI_Allgather` This operation allows all processes in a communicator to share their data with all other processes in the communicator. Each process sends its data and receives data from other processes. At the end of `MPI_Allgather`, each process has a copy of all the data gathered from all the other processes in the group. The collective communication mechanism is shown in fig. 4.4a.

`MPI_Allreduce` Similar to Reduce, but the result is returned to all processes in the communicator. This is often used for global reductions. The collective communication mechanism is shown in fig. 4.4c.

`MPI_Reduce_Scatter` This operation is a collective communication operation that performs both a reduction operation and a scatter operation in a single step. The collective communication mechanism is shown in fig. 4.4d.



(a) `MPI_Bcast`. (b) `MPI_Scatter`.

(c) `MPI_Gather`. (d) `MPI_Reduce`.

Figure 4.3: Graphic Representation of One-to-Many and Many-to-One MPI Collective Communication, illustrating the structured flow and interconnection of data exchange among multiple MPI processes.

Once again, instructions can fall into two categories: synchronous, signifying they are blocking, or asynchronous, indicating they are non-blocking. In the latter scenario, the non-blocking nature is denoted by adding the prefix `I` to the method name.

It is crucial to note that the `MPI_Barrier` instruction manages the synchronization of all threads, as exemplified in fig. 4.5. The implementation of the `MPI_Barrier` involves utilizing `MPI_Gather` with a 0 B message to the processor, followed by the `MPI_Bcast` with a 0 B message from the processor. This example makes us realize that considering a collective one-to-all or all-to-one instruction is not considered as functions that guarantee that all threads start up at the same time, as there are MPI implementations that work via prefix communication [18].

The main challenge within the realm of energy consumption is the energy wastage incurred by collective instructions that encompasses multiple processes. As is well known, the more processes there are, the more nodes to be expected, the more energy can be wasted. In figs. 4.3 and 4.4, the key collective barriers introduced by MPI are presented in the context of energy consumption. Furthermore, in fig. 4.5, one can observe how the execution of MPI on 4 nodes contributes to a significant delay in the context of energy consumption, as elucidated by Walker [16].

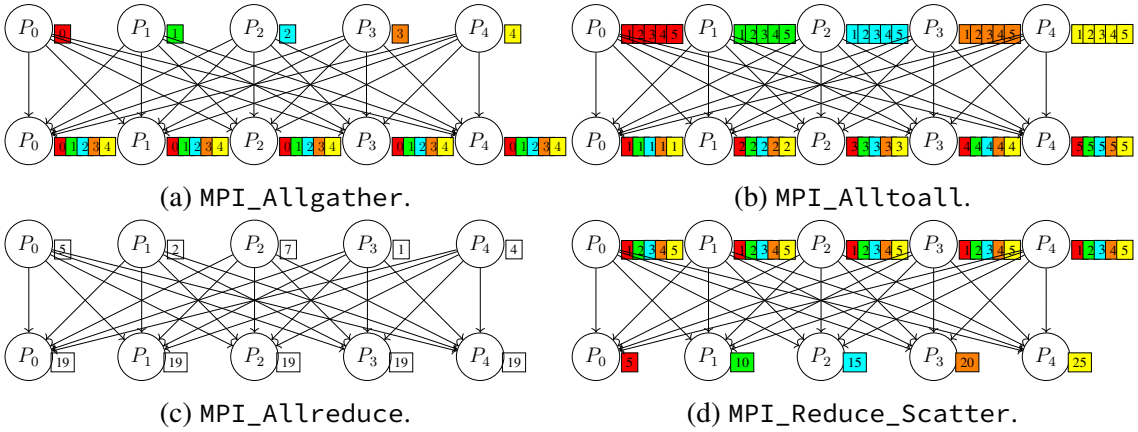
4.2 MPI Nodes, Tasks and OpenMP Threads

Given the extensive versatility of MPI in conjunction with OpenMP, allowing for use in both intra-node and inter-node communications, the following entities can be identified during program execution:

Nodes the number of distinct computers on which to run the program.

Tasks within a node, the number of threads dedicated to a specific task.

OpenMP Thread within a task, the number of threads used for parallel execution.



(a) `MPI_Allgather`. (b) `MPI_Alltoall`.

(c) `MPI_Allreduce`. (d) `MPI_Reduce_Scatter`.

Figure 4.4: Graphic Representation of Many-to-Many MPI Collective Communication, illustrating the structured flow and interconnection of data exchange among multiple MPI processes.

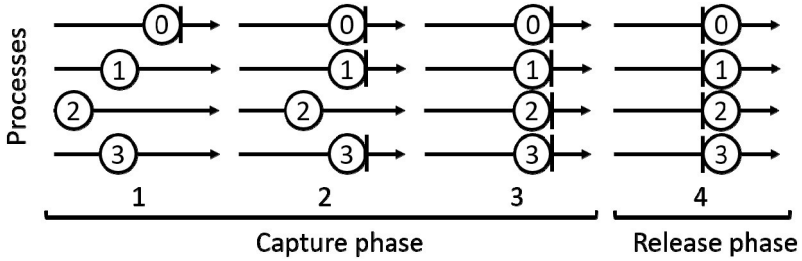


Figure 4.5: `MPI_Barrier`: Capture and Release Phases with Synchronization among Threads.

Source: Zharikov et al. [19].

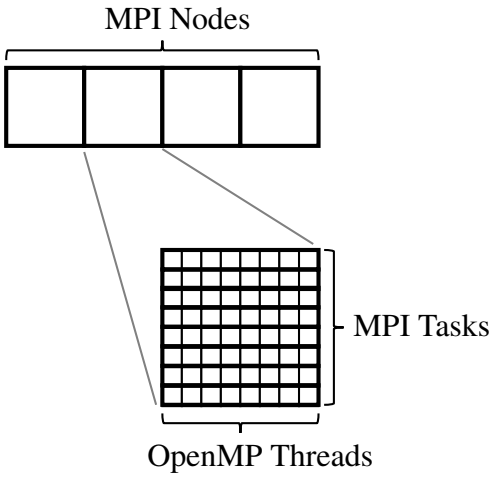


Figure 4.6: Graphic representation of a computational structure implemented with MPI Nodes, Tasks, and OpenMP Threads, showcasing the hierarchical organization of MPI processes within nodes, individual tasks distributed across processes, and the concurrent execution facilitated by OpenMP threads within each task.

The product of the number of Tasks and OpenMP Threads must equal the number of cores in the respective node. This alignment ensures optimal utilization of resources and efficient parallelization, enhancing the overall performance of the program.

Figure 4.6 is a graphical representation of a computational structure implemented with MPI nodes, tasks, and OpenMP threads, showing the hierarchical organization of MPI processes within nodes, individual tasks distributed across processes, and concurrent execution facilitated by OpenMP threads within each task.

In the combined world of MPI and OpenMP, communication unfolds in three stages. Threads communicate through via an entity called sockets or aggregators. Tasks share data using MPI and Inter-Process Communication (IPC), as shown in fig. 4.7. The nodes exchange information via MPI through the encapsulated Ethernet network in the UDP datagram. Each of the 3 systems has advantages and disadvantages, in particular, it is very important to consider the overhead that each package carries, which is why two different strategies are used.

In OpenMP, data is shared through a common pool among threads. In contrast, data exchange between Task and Nodes is based on the duo of multicast, particularly for broadcast operations, and the sophisticated Twisted Reflected Tree (TRT) system, which takes the lead in operations such as reduce, scatter and gather, as shown in fig. 4.8. This system not only facilitates data exchange but also optimizes communication pathways, since is used to obtain a logarithmic rather than exponential data exchange system, optimizing the time and number of data exchanged [20].

These differences are very important for analyzing certain phenomena that might not be easy to understand at first glance. It's crucial to balance the number of OpenMP threads to avoid issues. An imbalance can strain socket memory, leading to performance degradation due to the underuse of the advanced twisted-reflected tree (TRT) system. Correct task and thread management is essential to maximize the potential of the MPI-OpenMP hybrid framework.

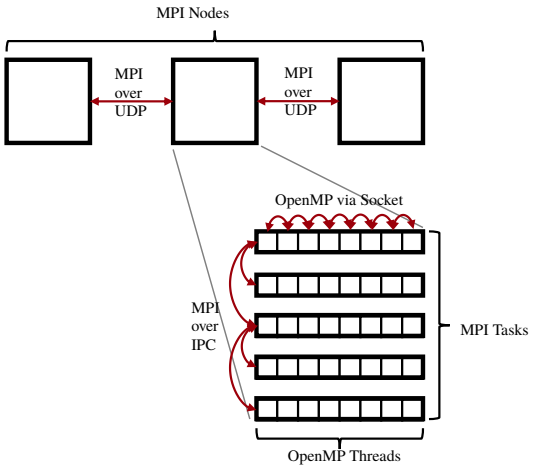


Figure 4.7: Graphical representation of the communication via UDP, IPC, and Socket of an MPI application.

4.3 Modelling the Execution Time and Energy of an MPI Program

The LogP model [21] is the main and first model that sums up the execution time of a classical parallel MPI process. It consists of 4 elements:

Latency An upper bound on the delay incurred in transmitting from the source task to its target task, often measured in time units required for communication, with a relatively small message.

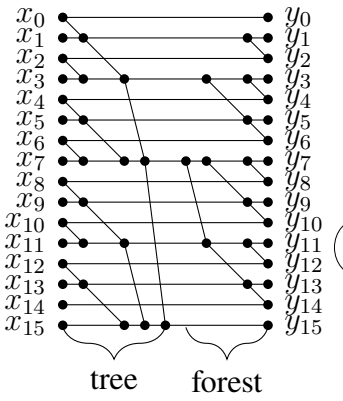
Overhead The time that a processor is actively involved in the transmission or reception of each message. During this period, the processor is dedicated to communication tasks, restricting its ability to perform other operations.

Gap The minimum time interval between consecutive message transmissions or receptions at a processor. The reciprocal of the Gap represents the available per-processor communication bandwidth, influencing the efficiency of communication.

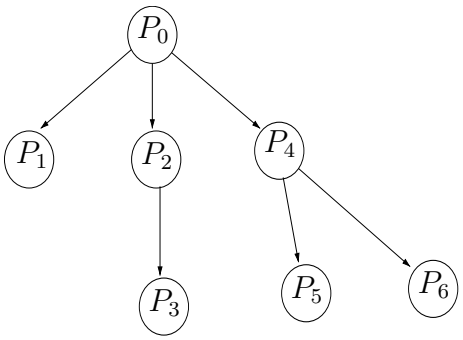
Processor The count of processor/memory modules. For analysis, we assume the unit time for local operations, called a cycle.

These metrics, *Latency* (L), *Overhead* (o), and *Gap* (g), are measured as multiples of the processor cycle and provide a comprehensive model for estimating the communication time. Figure 4.9 provides a visual representation of LogP parameters, specifically when transmitting a message from processor A to processor B. This visualization serves as a reference point for understanding the intricacies of communication patterns within the LogP framework.

The model introduced by Alexandrov et al. [22], which constitutes the second paradigm of parallel computation, builds on LogP and extends its capabilities by incorporating support for long messages. In this augmentation, an additional metric, denoted as *Gap per*

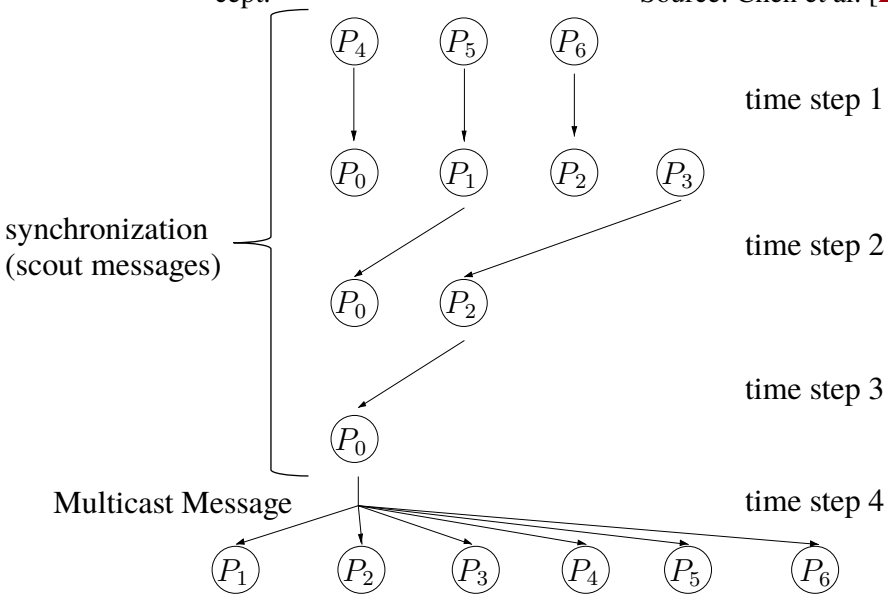


(a) Twisted Reflected Tree (TRT) Communication concept.



(b) Forest step of TRT on MPI.

Source: Chen et al. [20].



(c) Tree step of TRT and multicast on MPI.

Source: Chen et al. [20].

Figure 4.8: Illustrating the communication principles of Twisted Reflected Tree (TRT) and showcasing specific examples of Forest and Tree communication, along with multicast strategies, in a distributed MPI environment.

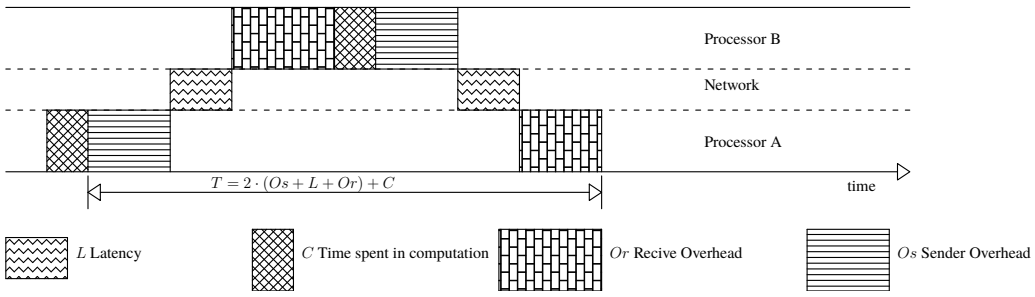


Figure 4.9: Sending a message from processor A to processor B with the `MPI_Send` and `MPI_Recv` primitive, doing some computation in B, and sending back to A in the context of LogP.

Source: Al-Tawil and Moritz [27].

Data Unit, is introduced. This metric is defined as the reciprocal of the available per-processor communication bandwidth specifically tailored for long messages that consist of more than one data unit. Similarly to LogP, the values for G are measured in multiples of the processor cycle.

The LogGPO model, introduced by Chen et al. [23], represents a significant advancement by extending the LogGP model. It incorporates the ability to capture the overhead resulting from high-level communication libraries. Moreover, it characterizes the potential overlap between computation and communication in MPI programs, offering a more comprehensive understanding of the dynamics involved.

The LoGPX model, as proposed by Lin et al. [24], stands out as a complete model. It possesses the versatility to degenerate into several popular models, including LogP, LogGP, LoGPC, and LogGPO. This flexibility makes LoGPX a powerful and adaptable tool for analyzing various parallel computing scenarios.

The work by Frank et al. [25] presents an abstraction of the LogPC model, the LoPC model that goes a step further by eliminating the need for the parameter g. This abstraction streamlines the LogPC model, potentially simplifying its application and analysis. The removal parameter g assumes that the gap is negligible and does not influence the communication pattern too much.

In a similar vein, the LoGPC model, introduced by Moritz and Frank [26] takes a distinctive approach by incorporating application-specific parameters. These parameters are introduced to account for network and resource contention effects.

4.4 Analysis of the Execution Time and Energy of an MPI Program

According to the findings presented in fig. 4.5, Hackenberg et al. [28] reports that latency times for MPI instructions exhibit a uniform distribution, ranging from a minimum of $21\,\mu s$ to a maximum of $524\,\mu s$ on Intel Haswell processor platforms.

Similar results are corroborated in earlier research papers. For one-to-one communication scenarios, Abandah and Davidson [29] indicates that message latencies do not exceed $500\,\mu s$ for messages up to $1\,kB$ in size. In the context of one-to-many communications, la-

Table 4.1: Timing Formulas for Collective Communications in MPI on the IBM SP2 Architecture, where n represents the number of tasks and m represents the message size in bytes to be exchanged.

Operation	**OpenMPI communications time**	**IBM MPI communications time**	**OpenMPI latency**	**IBM MPI latency**
`MPI_Send` and `MPI_Recv`	$46 + 0.035m$	$67 + 0.035m$	56	67
`MPI_Bcast`	$(40 \log n) + (0.037 \log n)m$	$(16 \log n) + (0.025 \log n)m$	$40 \log n + 20$	$16 \log n$
`MPI_Gather`	$(24n + 84) + (0.045n)m$	$(17 \log n + 15) + (0.025n - 0.02)m$	$24n + 84$	$17 \log n + 15$
`MPI_Scatter`	$(24n + 105) + (0.026n + 0.03)m$	$(17 \log n + 15) + (0.025n - 0.02)m$	$24n + 105$	$80 \log n$
`MPI_Alltoall`	$(125n - 22) + (0.06n^{1.29})\, m$	$(80 \log n) + (0.03n^{-1.29})m$	$105n - 22$	$60 \log n + 60$

tencies remain consistently below $1000\,\mu s$ for the same message size. Meanwhile, many to many communications exhibit slightly higher latencies, typically just over $1000\,\mu s$, again for messages up to $1\,kB$, as demonstrated by Abandah and Davidson [29]. In alignment with these findings, Xu and Hwang [30] underscores that, for messages up to $1\,kB$, latency times generally remain under $500\,\mu s$, with certain exceptions in many to many communication scenarios.

Furthermore, Heinrich et al. [31] reaffirms that latency times for these communication operations are consistently below $500\,\mu s$. In addition, Xu and Hwang [30] performed a regression analysis of communication times, considering the size of data m and the number of nodes n as variables, as shown in table 4.1.

5

Strategies for energy saving

Energy efficiency is a critical goal within data centers as these operations become increasingly energy-intensive. It is clear that there is a growing demand for more efficient cooling systems to address the increasing energy usage caused by frequent hardware upgrades. The current hardware no longer adheres to Moore's law, which was in effect until about a decade ago, wherein the number of transistors would double (and clock speeds), and energy consumption would decrease every 18-24 months. Presently, neither the number of transistors nor energy consumption experiences such doubling, but with each CPU generation, both consumption and performance see marginal increases.

In this scenario, the focus is shifting from optimizing hardware to streamlining software to minimize energy consumption. The focus is on making software as environmentally friendly as possible, even at the expense of slowing down code execution. This strategic shift underscores the industry's commitment to achieving greater ecological sustainability in computing practices.

In response to these challenges, we analyze strategies for reducing power consumption on the software side. Specifically, we investigated the use of a power-aware algorithm designed to automatically and seamlessly adjust voltage and frequency settings. This adaptive approach aims to achieve significant reductions in power consumption and energy savings, with minimal impact on system performance. Our exploration involves the incorporation of a well-established technology called "dynamic voltage and frequency scaling" into the runtime system of standard HPC systems described in section 5.1 [17]. The next section, 5.2, analyzes what are the granted options related to DVFS in Linux, and the last two (sections 5.3 and 5.4) analyze how to save energy in the case of parallel and HPC computations.

5.1 Power Management

Power management, within the realm of computing and HPC systems, involves a set of methodologies and approaches that aim to regulate and optimize electrical energy consumption. This objective is pursued while either maintaining or enhancing system performance and functionality. This critical discipline addresses the increasing demand for energy-efficient solutions that span various domains such as information technology, telecommunications, transportation, and consumer electronics.

The importance of power management has experienced a notable upswing in recent years, propelled by various compelling factors. The prevalence of portable devices such as smartphones, laptops, and wearables has accentuated the need for prolonged battery life, which requires innovative strategies to minimize power usage. Simultaneously, rising energy costs and environmental concerns have prompted organizations to embrace environmentally friendly practices, urging a reduction in power consumption in data centers, supercomputers, and industrial facilities. Additionally, the persistent pursuit of higher computational performance has driven the evolution of power-efficient processors and components.

The significance of power management has increased in recent years due to several compelling factors. First, the proliferation of portable devices, such as smartphones, laptops, and wearables, has underscored the need for extended battery life, which requires innovative approaches to minimize power usage. Second, increasing energy costs and environmental concerns have compelled organizations to adopt greener practices, requiring the reduction of power consumption in data centers, supercomputers, and industrial facilities. Finally, the relentless drive for improved computational performance has led to the development of power-efficient processors and components.

Efficient power management covers a diverse range of techniques, ranging from hardware-level mechanisms embedded within individual components such as processors and memory modules to software-driven strategies that govern the overall system behavior. Fundamental aspects of power management include Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Voltage Scaling (DVS), P-States, C-States and T-States.

5.1.1 P-States, C-States, and T-States

In the realm of power management for computer systems and processors, the P-States, C-States, and T-States, according to ACPI terminology, are fundamental concepts that have crucial roles in maximizing energy efficiency and thermal effectiveness. Since most contemporary processors have the capability to function at different clock frequencies and voltage settings these states are essential for balancing the trade-offs between power efficiency and computational performance in modern computing devices. Generally, higher clock frequencies and voltages allow the CPU to execute more instructions per unit of time, but they also consume more energy or power in the given P-States. Thus, there is a trade-off between CPU capacity (number of instructions executed over time) and power consumption [32, 33].

P-States (also known as Performance States or Power States) are hardware-controlled mechanism used in modern processors to manage power consumption and performance levels dynamically. Processors can operate at different P-States, each with varying clock frequencies and voltages. Higher P-States correspond to higher performance but also higher power consumption [34, 35].

C-States (also known as Idle States) are power-saving states in processors where specific components are turned off or slowed down to reduce power consumption during periods of inactivity. The deeper the C-States, the more components are powered down, resulting in greater energy savings but longer wake-up times [34, 35].

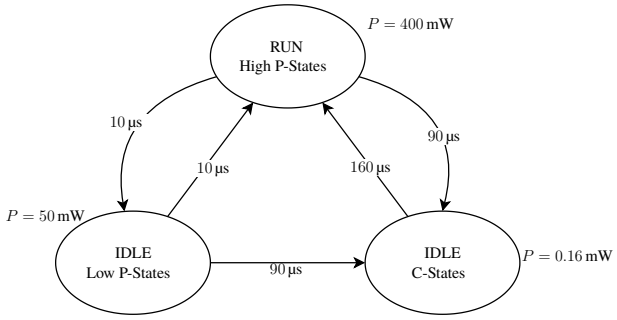


Figure 5.1: Power state machine for the StrongARM SA-1100 processor.

Source: Benini et al. [32].

T-States (also known as Throttling States) are related to thermal management in processors. When a processor reaches a certain temperature threshold, it may enter a T-States, which reduces its performance to lower temperatures and prevents overheating. This state helps protect the processor from damage [36].

In addition to P-States, C-States, and T-States, the "Sleep States" is a variant of the C-States. The Sleep States involves placing specific hardware components or the entire system in a very low power mode during long periods of inactivity, further enhancing energy efficiency and extending device lifespan [32], this state is illustrated in fig. 5.1.

5.1.2 Dynamic Voltage Scaling (DVS)

Dynamic Voltage Scaling (DVS) or Dynamic Voltage and Frequency Scaling (DVFS) represents a power management technique implemented in computer systems to optimize energy consumption without compromising performance. DVS dynamically adjusts the voltage and frequency of a processor or component to align with the workload's specific requirements. The primary objective of DVS is to decrease energy consumption during periods of low computational demand, while still allowing the system to operate at elevated performance levels when necessary.

Within the realm of DVS, the voltage supplied to the processor or component undergoes dynamic scaling, accompanied by adjustments to the clock frequency. During lighter or idle workloads, DVS reduces both the voltage and frequency, resulting in diminished power consumption and heat generation. On the contrary, when there is a heavy computational load, DVS can increase the voltage and frequency to ensure that tasks are completed more quickly.

DVS is particularly valuable in mobile devices, laptops, and data centers, where energy efficiency is a critical concern. By dynamically adjusting the power supply, DVS helps extend the battery life in portable devices and reduces electricity costs in data centers, while maintaining acceptable performance levels [37].

Dynamic Voltage and Frequency Scaling (DVFS) evolved from the concept of Dynamic Voltage Scaling (DVS), which introduces the capability for a processor or microcontroller to dynamically adjust its operating voltage (Vdd) and clock frequency (CPU frequency) in real time. This advancement offers precise control over both voltage and

frequency, allowing the processor to optimize power consumption and performance simultaneously.

Under DVFS, the processor has the flexibility to operate at lower voltages and frequencies during periods of light workload. This adaptive approach conserves energy and minimizes heat generation, contributing to overall efficiency. On the contrary, when facing demanding tasks that require higher performance, DVFS allows the processor to seamlessly ramp up both voltage and frequency to meet those requirements effectively. The relationship between CPU clock frequency, power, and energy is discussed in the Intel Corporation [38] reference manual. The study investigates this relationship through the use of the eqs. from (5.1a) to (5.1c). The supply voltage is denoted as V_{dd} and the CPU clock frequency is denoted as f.

$$\text{Power} \propto f V_{\text{dd}} \tag{5.1a}$$

$$\text{Delay} = \frac{1}{f} \propto \frac{1}{V_{\text{dd}}} \tag{5.1b}$$

$$\text{Energy} = \propto V_{\text{dd}}^2 \tag{5.1c}$$

Increasing the frequency of a system will result in a linear decrease in delay, which, in turn, will lead to a quadratic increase in both power and energy. The importante of DVFS is found in the opposite of this statement: when there is a linear increase in delay, there will be a quadratic decrease in power and energy consumption. We can express the practical impact in a simpler way: increasing the CPU clock frequency requires progressively more energy [39–41].

5.2 Power Management in Linux

The documentation in `kernel.org` [42, 43] is one of the main sources of information on power management in the Linux environment. In certain situations, it may be desirable or necessary to run a program as fast as possible, and using the highest P-states (highest-performance frequency/voltage configuration) is justified. However, there are cases where it might be unnecessary to execute instructions quickly, and maintaining the highest CPU capacity for an extended period without utilizing it entirely could be considered wasteful. Additionally, maintaining maximum CPU capacity for too long may not be feasible due to thermal or power supply constraints. To address these scenarios, hardware interfaces allow CPUs to switch between different frequency/voltage configurations or P-states.

These hardware interfaces are often used in conjunction with algorithms to estimate the required CPU capacity, determining which P-states to put the CPUs into. However, in specific situations, it may be necessary for the application to manage CPU usage independently to avoid kernel penalties or for greater efficiency.

Linux abstracts these concepts within the `CPUFreq` (CPU Frequency scaling) subsystem, comprising three layers of code: the core, scaling governors, and scaling drivers. The `CPUFreq` core offers a shared code infrastructure and user space interfaces for all platforms that enable CPU performance scaling. It establishes the fundamental framework for the other components.

The first component is the governor, which can be automatic or user-governed. Scaling governors implement algorithms to estimate the required CPU capacity, with each governor typically implementing a specific scaling algorithm.

Scaling drivers communicate with the hardware, providing scaling governors with information on available P-states and utilizing hardware interfaces specific to the platform to modify CPU P-states as directed by governors.

Although it is theoretically possible to use any scaling governor with any scaling driver, the `CPUFreq` provides a way for scaling drivers to bypass the governor layer and implement their own algorithms for performance scaling. During kernel initialization, the `CPUFreq` core creates a `sysfs` directory called `cpufreq` under `/sys/devices/system/cpu/`, containing subdirectories for each policy object. These directories hold policy-specific attributes to control `CPUFreq` behavior, with some being generic and others added by scaling drivers for driver-specific aspects:

`affected_cpus` List of online CPU currently using the (`policyX`) policy (i.e. sharing the hardware performance scaling interface represented by the `policyX` policy object).

`bios_limit` Reports the upper limit on CPU frequencies if directed by the platform firmware (BIOS), potentially influenced by BIOS settings, service processor restrictions, or other BIOS/Hardware-based mechanisms. Does not cover ACPI thermal limitations.

`cpuinfo_cur_freq` Current frequency of CPUs in this policy is obtained from hardware (in kHz), representing the actual running frequency. This attribute may not be present if the frequency can not be determined.

`cpuinfo_max_freq` The highest achievable operational frequency for the CPUs associated with this policy (in kHz).

`cpuinfo_min_freq` The lowest achievable operating frequency for the CPUs associated with this policy is expressed in kilohertz (kHz).

`cpuinfo_transition_latency` Time taken to switch CPUs in this policy from one P-states to another, measured in nanoseconds. Returns -1 (`CPUFREQ_ETERNAL`) if unknown or too high for the scaling driver to work with the on-demand governor.

`related_cpus` List of all CPUs (online and offline) associated with this policy.

`scaling_available_governors` List of `CPUFreq` scaling governors or scaling algorithms (for `intel_pstate` driver) available in the kernel for attachment to this policy.

`scaling_cur_freq` Current frequency of all CPUs in this policy (in kHz), usually reflecting the last P-states requested by the scaling driver, though not necessarily the actual CPU frequency.

`scaling_driver` Currently used scaling driver.

scaling_governor Currently attached scaling governor or scaling algorithm (for the `intel_pstate` driver).

The attribute mentioned is both readable and writable. When writing to it, a new scaling governor will be associated with this policy or a new scaling algorithm will be applied to it, depending on the string written to this attribute. In the case of `intel_pstate`, the string must be one of the names listed in the `scaling_available_governors` attribute mentioned earlier.

scaling_max_freq The highest achievable frequency for the CPUs associated with this policy (in kHz). Read-write attribute; writing an integer string sets a new limit (not lower than `scaling_min_freq` attribute).

scaling_min_freq The lowest achievable operating frequency for the CPUs associated with this policy is expressed in kilohertz (kHz). Read-write attribute; writing a non-negative integer string sets a new limit (not higher than `scaling_max_freq` attribute).

scaling_setspeed This function is effective only when the userspace scaling governor is connected. It retrieves the most recent frequency requested by the governor, measured in kilohertz. Alternatively, it can be used to set a new frequency for the policy.

In this paper, our attention is directed towards the following registers: `cpuinfo_max_freq`, `cpuinfo_min_freq`, `scaling_cur_freq`, `scaling_max_freq`, `scaling_min_freq`, `scaling_setspeed`, and `scaling_available_governors`. The first three registers are read-only, while the others are changed.

5.3 Energy Efficiency and MPI

The rise in energy consumption of contemporary supercomputing systems is a cause for the global HPC community. In particular the Message Passing Interface (MPI) has long been the predominant programming model for parallel applications, and MPI libraries have been crafted to deliver optimal communication performance on contemporary computing architectures. However, the balance between performance and energy efficiency in these designs has yet to be thoroughly explored. Consequently, it is imperative to gain insights into the energy consumption characteristics of MPI routines and to discern the trade-offs between performance and energy efficiency inherent in various protocols and library designs utilized within MPI.

By default, when MPI processes find themselves in a state of synchronization, MPI libraries employ a busy-waiting/polling mechanism to avoid entering C-states which can induce performance penalties. Nevertheless, it is worth noting that during MPI primitives, a significant portion of the workload consists of waiting times and IO/memory accesses. Running an application in a low-power mode during these periods may result in reduced CPU power consumption, often with little to no adverse impact on overall execution time [34–36].

While MPI libraries do incorporate idle-waiting mechanisms, they are seldom employed in practice due to concerns about the performance penalties associated with transitioning in and out of low-power states, as documented in previous studies [34–36, 44, 45].

5.3.1 Slack and Communication Time

In MPI communications can be divided into slack time and copy time, this subdivision was proposed by Rountree et al. [39]. The authors proposed a division of communication time, as described in the LogP model, into two distinct components: the slack time (*Tslack*) and the copy time (*Tcopy*). The authors defined a "task" as the section of code located between two MPI communication calls and formulated an optimization problem aimed at minimizing slack time.

Slack Time (*Tslack*) This component represents the time spent waiting for a critical task to enter an MPI primitive. Since in parallel computation, each process is independent, each process has its own execution time, and therefore some of these tasks may be ready to communicate before others, which results in a waiting period. This waiting time can be regarded as the time during which the processor is not fully engaged in active computation. The waiting time can be eliminated by adopting a branchless programming style, which is in use in GPUs, for example.

Copy Time (*Tcopy*) The communication or copy time accounts for the duration required for the actual data transfer during communication. When data is sent or received between processes, there's an inherent cost associated with copying this data between memory locations, and this is the copy time. This time can not be reduced, but at most accelerated [39–41].

Slack times are generally much higher in synchronous communications than in asynchronous ones, as a portion of the time in MPI collective functions (e.g., `MPI_Barrier` or `MPI_Allreduce`) is spent waiting for the last rank to reach the synchronous point, that are illustred in fig. 4.5.

By reducing the P-States of the processor core only during slack time, this will lead to a reduction in consumption without compromising performance, whereas reducing the frequency in both slack time and copy time may compromise performance. A common strategy widely used to identify and separate this slack time from the copy time involves inserting a call to `MPI_Barrier` before each MPI collective, as illustrated in fig. 5.2. This strategy is described are present in COUNTDOWN and MPInside libraries [35, 46]. This division into slack time (*Tslack*) and communication time (*Tcopy*) represents a pivotal concept in the optimization of MPI power consumption.

5.3.2 Optimising Parallel Execution to Reduce Slack Times

There are several strategies to optimize parallel execution and reduce slack time. One approach involves minimizing the number of conditional statements (ifs) within the code and utilizing temporal patterns.

For instance, when programming in CUDA with massively parallel processing, it is crucial to avoid branches as they can lead to warp divergence (in CUDA programming a

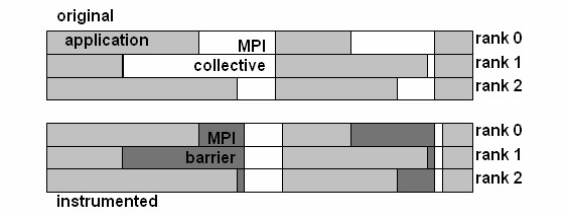


Figure 5.2: `MPI_Barrier` insertion for collective wait time.

Source: Thomas et al. [46].

warp is a group of threads that executes the same instruction simultaneously). This form of parallelism is adverse to branch conditions, such as ifs, as they result in inconsistent executions and increased synchronization times. Conditional statements in GPU programming, like branch conditions, may pose challenges to parallelism since, in a parallel execution model, all threads within a thread block must follow the same execution path. Deviations caused by a conditional statement in one thread can lead to divergent execution, causing significant performance penalties.

To mitigate this issue, it is advisable to minimize the use of conditional statements within a thread block. Alternative techniques, such as employing data-dependent operations or utilizing lookup tables, can be explored to achieve desired outcomes while preserving parallelism. If the use of conditional statements is unavoidable, structuring the code in a manner that minimizes divergence between threads becomes crucial.

A simple branch conditions, like:

```
if(dis[i][j] < dis[i][h] + dis[h][j])
    dis[i][j] = dis[i][h] + dis[h][j];
```

Can be removed using a simple boolean logic:

```
t = dis[i][h] + dis[h][j];
dis[i][j] = t * (t < dis[i][j]) + dis[i][j] * (t >= dis[i][j]);
```

This works since C++ treats logical true as 1 and logical false as 0. What happens if different threads in a warp need to do different things?

```
if (x < 0.0)
    z = x - 2.0;
else
    z = sqrt(x);
```

This is called warp divergence. CUDA will generate the correct code to handle this, but to understand the performance you need to understand what CUDA does with it. To avoid loss of synchronization all threads execute both conditional branches:

```
p = (x < 0.0);
a = x - 2.0;
b = sqrt(abs(x));
z = p * a + !p * b;
```

The execution cost becomes the sum of both branches, potentially resulting in a significant loss of performance, nevertheless, care must be taken, as the root of negative numbers does not exist and roots of different numbers may have uneven lengths. This issue extends to memory access, such as when accessing out-of-bounds elements.

When dealing with substantial branches, the nvcc compiler incorporates code to check if all threads in a warp take the same branch (warp voting) and subsequently branch accordingly. If each warp follows a uniform path, the process is highly efficient. While warp voting incurs a few additional instructions, the compiler opts for predication without voting for very simple branches.

It is important to note that each warp is treated independently, it does not matter what is happening with other warps. Warp divergence can consequently lead to a substantial loss of parallel efficiency. In the worst-case scenario, there is an effective performance loss of a factor of 32x if one thread requires an expensive branch while the rest remain inactive [47].

In non-massive parallel programming, it is generally impossible to avoid divergences in thread execution flows, furthermore, threads are managed within an operating system that takes precedence over memory management, so having an implicit synchronism, which is realized thanks to the absence of branch conditions, is impossible, despite this, it is possible to reduce waiting times through patterns as well and for example by reducing the processor speed in shorter threads [48]. The management of different threads by the Linux kernel is explained in section 5.2, while the strategies for analyzing and conserving energy in MPI are discussed in chapter 6.

5.4 Clusters

The power-scalable cluster [49, 50] are clusters where an attempt is made to save energy by dynamically adjusting processor performance to lower energy levels. The pivotal aspect of interest revolves around the dynamic adjustment of processor performance and its direct impact on energy consumption. The studies conducted by Freeh et al. [49] and Springer et al. [50] show the relationship between energy efficiency and execution time within the context of MPI programs.

Freeh et al. [49] specifically delve into the balance between energy consumption and the time required for execution in MPI programs. The concept of a power-scalable cluster allows for the exploration of trade-offs, presenting a unique opportunity to optimize both performance and energy efficiency concurrently. By dynamically adjusting the energy levels of individual nodes, the authors demonstrate the potential for achieving dual benefits: a reduction in energy consumption and faster execution times.

Springer et al. [50] extends this exploration by placing a particular emphasis on minimizing execution time while adhering to energy constraints. The study recognizes the challenges inherent in maintaining efficiency within the limitations of energy consumption. The authors focus on strategies for optimizing execution time within the defined power-scalable cluster, highlighting the delicate balance required to achieve optimal performance with the least amount of energy consumed.

In essence, both studies contribute significantly to our understanding of the energy-time tradeoff in power-scalable clusters. These studies demonstrate that for certain types

of programs, there exists a compelling opportunity to achieve a dual benefit: reduced energy consumption and faster execution times. This is accomplished by employing a larger number of nodes, with each node operating at a reduced energy level, thereby enhancing overall efficiency in terms of both energy usage and execution speed.

6

Performance Analysis and Energy Saving in MPI

This chapter is dedicated to an in-depth examination of tools devised for the analysis of both performance and energy-saving aspects within the context of MPI applications. The tools under scrutiny include Jitter, MPInside, COUNTDOWN, ATFaVSCP, and other minor ones in conjunction with energy efficiency implementations within Intel MPI and MVAPICH2. The focal point of our exploration revolves around unraveling the common and unique strategies employed by these tools to optimize energy consumption, particularly during the most energy-intensive phases, namely, the busy wait periods during MPI synchronization primitives. These tools, with the exception of MPInside perform a dynamic voltage scaling for the dynamic adjustment of CPU frequency (and voltage), to minimize these wait times and consequently reduce energy consumption, we refer to this process as frequency variation.

Throughout this chapter, we will meticulously dissect each tool and implementation, elucidating its methodologies and functionalities. Within this chapter, the attention is specifically directed towards the COUNTDOWN tool, distinguished not only as the sole open-source solution but also for its compatibility with OpenMPI, rendering it the most comprehensive tool among those outlined. COUNTDOWN emerges as a pivotal subject of exploration, presenting a unique and valuable perspective in the landscape of performance and energy-saving tools for MPI environments.

The organization of this chapter is as follows: sections from 6.1 to 6.10 talk about the most important energy saving techniques, libraries and algorithms, all of which are summarized in a final comparison in section 6.11.

6.1 Jitter Library

Jitter [37] is a library that uses a scheduled iteration methodology for energy-awareness. Jitter seamlessly inserts itself between an application and the MPI library, operating in a way that is typically transparent to both the application itself and the MPI library. Its primary function is to actively monitor the periods during which a program waits for external events.

The core idea of Jitter is to reduce speed on the computation region (*Tcomp*) in order to minimize the slack time (*Tslack*), for doing this the Jitter algorithm identifies non-critical nodes. A node that does not belong on the critical path has slack time, that can be optimized

```
1: constant N, F                                   ▷ Number of nodes and frequencies
2: constant B = 1.5                                ▷ Bias to stabilize Jitter, see Section 4.3.2
3: constant α = 0.25                               ▷ Upshift factor, see Section 3.4
4: global F                                        ▷ Current frequency of the microprocessor
5: global S                                        ▷ Slack Threshold, see Section 3.7
6: global downShift[F], upShift[F]                 ▷ Down and up shift factors

7: procedure INITIALIZE
8:     F ← fastest                                 ▷ F is current frequency; initially set to fastest
9:     upShift[i] ← 1, ∀i = 1 to F
10:    downShift[i] ← 1, ∀i = 1 to F
11:    S ← default_S                               ▷ Default is 0.05, see Section 4.3.2
12: end procedure

13: procedure JITTER(i, slack[N], iterationTime)
    ▷ Called every iteration for each node i
14:    local netSlack ← slack[i] − min_{i=1}^{N} slack[i]

15:    if netSlack > S ×downShift[F] and F ≠ slowest then
    ▷ Reduce clause, see Section 3.3
16:        downShift[F] ← downShift[F] ×B
17:        F ← decr(F)                             ▷ reduce frequency

18:    else if netSlack < α × S/upShift[F] and F ≠ fastest then
    ▷ Increase clause, see Section 3.4
19:        upShift[F] ← upShift[F] ×B
20:        F ← incr(F)                             ▷ increase frequency
21:    end if
22: end procedure
```

Figure 6.1: Jitter algorithm.

Source: Kappiah et al. [37].

through slower execution by lowering the less energetic frequency. Since this node at full speed finishes its tasks and remains idle until it receives a message from another node. The Jitter algoritmh sets the non-critical node to operate with decreased CPU speed to reduce energy consumption, aiming to finish its tasks right before receiving the message from the remote node, the pseudocode of the algorithm is shown in fig. 6.1. Energy savings can be achieved without extending the application completion time, as long as a node with reduced performance completes its computation before the bottleneck node. Figure 6.2 shows an example of DVFS management performed by Jitter.

The optimizations performed by Jitter are based on the assumption that two instances of the same task have the same *Tcomp*, *Tslack*, and number of instructions: programs follow an iterative structure and that the duration of each iteration remains relatively consistent. In other words, the variability in the time taken for each iteration is minimal, allowing for the use of past iterations to predict future ones. If this happens, Jitter can drastically improve energy efficiency each iteration, also called a step or timestep. It is important to note that the aforementioned conditions hold true for the vast majority of scientific programs.

Jitter executes critical nodes at maximum clock speed and non-critical nodes at a reduced speed to avoid affecting the execution time. When a node is operating at a lower frequency and lacks sufficient slack, Jitter is employed to enhance CPU speed. This situation can arise due to the uncertainty surrounding the impact of frequency reduction, which

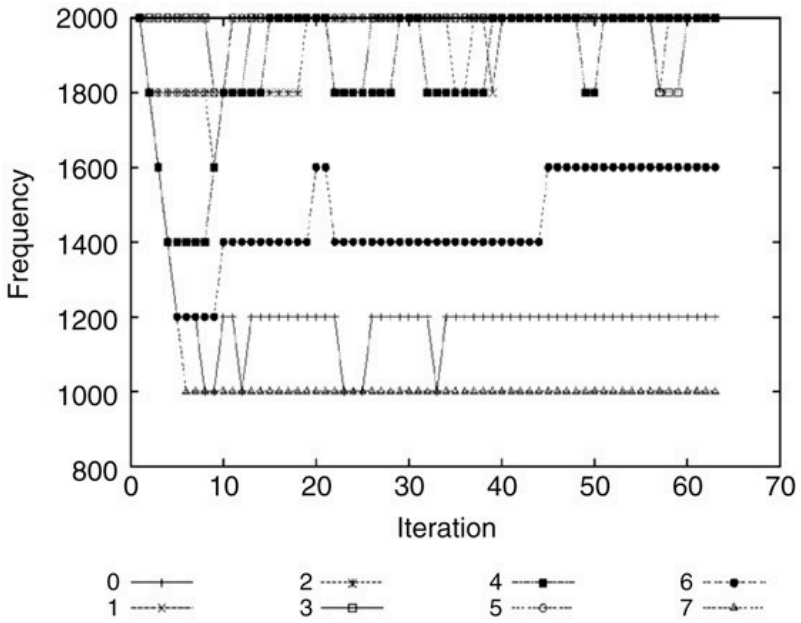


Figure 6.2: Example of DVFS performed by Jitter in Aztec. The graph shows that critical threads have a higher frequency compared to less critical threads, which have a lower frequency.

Source: Kappiah et al. [37].

varies depending on the program's CPU, memory, and I/O usage.

Initially, an optimistic approach is employed, reducing the frequency, and subsequently reverting back if needed. One more factor to consider is the potential variation in load distribution among nodes over time. Therefore, a lower frequency may be suitable at a certain moment but not ideal later on. Consequently, Jitter constantly observes the available time on each node and modifies the frequency accordingly.

The current implementation of Jitter requires that the code be recompiled to add the `MPI_Jitter` instruction at the beginning of the iteration loop and to change the function references when linking, since changing the power management of a thread is an onerous operation, Jitter shortens multiple iterations to a maximum of 2.5 seconds if they are too short. By doing this, Jitter determines various aspects such as the limits of iterations, the net slack of each node, the appropriate times to decrease or increase performance, when to reset algorithm parameters, and how to adjust the slack threshold.

When applied to unbalanced programs, the Jitter system achieves a notable $8\,\%$ reduction in energy consumption while incurring only a modest $2.6\,\%$ increase in execution time. It's important to note that these energy savings and performance improvements are realized without needing any modifications to either the application source code or the underlying communication library. Additionally, it's noteworthy that the system's performance is within a $5\,\%$ margin of what could be achieved through manual, hand-tuned optimization, often referred to as the "optimal" solution. Furthermore, Jitter demonstrates its adaptability by effectively responding to changes in workload, a feature that conventional hand-tuned solutions are typically unable to match [37].

6.2 MPInside Library

MPInside [46] is a tool for profiling MPI applications. It allows us to analyze the performance of an MPI application and provides valuable insights into MPI communications. MPInside examines the actions sent and received by a process, and produces data that indicates the level of synchronization between the two. By analyzing the data, you have the ability to identify specific areas within the application that you would like to concentrate your optimization efforts on [46].

MPInside is also a transparent wrapper, but unlike Jitter, which requires recompilation, it acts as a wrapper to the executed program, a bit like other various program analysis tools, for example valgrind. To execute MPInside, simply insert the command `MPInside` between mpirun and the executable, as described in listing 6.1.

```
mpirun -np 128 MPInside ./a.out args...
```

Listing 6.1: Shell execution of an application with the MPInside profiler.

The default behavior of MPInside is to provide a simple and efficient implementation of the MPI functions. The generated information includes the size of each data request, the number of data requests, the size of the communicator used for MPI collective functions, and the number of times each rank acted as the root of a collective function. MPInside reports the total sum of these statistics for the entire run. After the MPI application completes successfully, the measurement and the statistics data are reported in a text file called `mpinside_stats` [46, 51].

6.3 Offline Scheduling

The Offline DVFS Scheduling [39–41] is an algorithm computed over one or more training runs. This can be especially beneficial in embedded and real-time computing. In this context, rather than having program execution as a singular event, programs are designed to loop continuously and be replicated across all instances of a specific device (e.g., temperature sensors). The scheduling cost can be spread out over all executions on all devices, which justifies even minor energy savings. In the field of HPC, the traditional assumptions mentioned above are no longer applicable. Supercomputers are hardly ever idle, and the key measure of success is the time it takes to complete a task. To make use of dynamic voltage and frequency scaling (DVFS) in this scenarios, we leverage load imbalance, where certain processors have less workload compared to others, resulting in periods of idle time. In this context, DVFS can be employed to schedule the execution of tasks at a lower CPU clock frequency without impacting the overall time it takes to complete the computation.

6.4 Fermata Algorithm

The Fermata algorithm [39–41] implements a straightforward algorithm aimed at reducing the P-States of cores during communication regions (*Tcomm*). This scheduling is called

Scheduled Communication. Using a prediction algorithm, Fermata makes decisions on scaling down the P-States based on the time spent in communication during the preceding call. If this duration exceeds or equals twice the switching threshold, Fermata sets a timeout scheduled to expire at the threshold time. If the MPI call is completed before the timer runs out, the callback is canceled. The default threshold time for Fermata is 100 µs, although the literature in section 4.4 suggests other values. Identification of specific MPI primitives in the application code is done by hashing the pointer (Task ID) that forms the stack trace. Task ID is generated when an MPI primitive is encountered, ensuring a unique identification for each MPI primitive in the code. The details of the last call are stored in a lookup table, which helps to determine whether to set the timer in the next call [39–41].

6.5 Adagio-Computation Algorithm

Adagio-Computation [39–41], also called Andante in Cesarini et al. [36], takes care of reducing power consumption when the program is not within the MPI communication blocks. Unlike Jitter, Adagio-Computation is a scheduled timeslice (a fraction of a timestep) algorithm based on the same assumption of Jitter. The difference between Jitter and Adagio-Computation is that the former requires the program to be iterative, and so the first few runs are used to train the algorithm and find for each node the ideal frequency. The latter searches previous instances of a task, and uses the most similar one if it does not find it, and uses this instance as an estimate for the current one, thus allowing optimisation of execution time and *Tcomp* energy.

The first step for each task, after finding the previous instance, is to calculate the *TComp*, *Tcomm*, *TSlack* and *Tcopy*, the first and second are calculated by summing the time intervals inside and outside the MPI calls, *Tcopy* is calculated using the models described in section 4.3 and *Tslack* is calculated as *Tcomm* minus *Tcopy*. After determining the execution time of *TSlack* and *TComp*, Adagio-Computation makes an estimate of the ideal working frequency by means of Instructions Per Second (IPS), keeping in mind that the sum of *TSlack* and *TComp* must remain unchanged.

Like Fermata, Adagio-Computation differentiates tasks by utilizing the stack trace located at the conclusion of each collective MPI primitive. A lookup table is employed to store details regarding the most recently executed task, including the IPS for each distinct P-States of the system, as well as the subsequent P-States to be assigned [39–41].

6.6 Adagio Algorithm

Adagio [39–41] is designed to integrate the capabilities of both Fermata and Adagio-Computation into a unified, energy-aware runtime. While Adagio-Computation focuses on slowing down computation regions, Fermata handles communication phases. This integrated approach aims to strike a balance between optimizing computation and communication aspects to achieve energy efficiency in the overall system runtime. Adagio utilizes a simple and robust algorithm that does not require any specific knowledge about the application. Adagio schedules tasks based on predicted computation time and makes slowdown decisions at runtime.

Adagio is task based and to identify the task from the individual MPI calls, Adagio uses a hashing technique on the stack trace, which is stored in a hash table. Adagio must not only accurately predict the computation, communication and slowdown associated with the upcoming call, but also predict the upcoming call itself. Initially, scheduling is based on worst-case slowdown, and then more aggressive scheduling is performed based on observed performance. After finding a task with the same hash in the hash table, Adagio uses this purely local data to determine the critical paths. Adagio's scheduler then tries to reduce the frequency in the *Tcomp* so that all the different threads reach the MPI call at the same time. It's important to note that this may introduce some overhead and delay.

In Adagio, the prediction of the properties of the next task relies on the identification of the task that will occur next. This identification is achieved by creating a signature for each task using a hash of the pointers in the stack trace. The hash is generated when the MPI call associated with the task is intercepted by the library. The completed task record includes the hash of the task that immediately follows it. Prior to the computation of a task, Adagio retrieves the frequency schedule for that task and sets the operating frequency to the first one in the schedule. It also initializes hardware performance monitors (HPMs) to monitor the code. Following the completion of a task, Adagio collects data and determines the frequency schedule for the next execution of that task.

The pseudocode for Adagio in the case of using a single frequency per task is presented in fig. 6.3. Since runtime algorithms do not have any prior information about program execution characteristics, Adagio initially schedules task execution at the fastest frequency denoted by $\hat{f}$. If a task recurs, Adagio assumes a worst-case slowdown, where the execution slowdown is proportional to the change in frequency. However, the computation will not slow down by more than the ratio of the change in frequencies.

6.7 COUNTDOWN Library

COUNTDOWN [34–36] is an algorithm for energy saving, a runtime analysis library, and a runtime library designed to be performance-neutral. It conserves energy exclusively during MPI synchronization without introducing any increase in time-to-solution for applications.

This library enhances the application's functionality by capturing blocking MPI primitives. It utilizes a timeout strategy to prevent altering the power state of the cores when there are rapid applications and MPI context switches, preventing performance overhead without significant reductions in energy and power. If the MPI blocking phase terminates within this time frame, COUNTDOWN does not enter low-power states, filtering out short MPI phases that incur costly overheads with negligible energy savings. This strategy is purely reactive and is triggered by the MPI primitives invoked by the application. COUNTDOWN implements the timeout strategy using standard Linux timer APIs.

When COUNTDOWN comes across an MPI phase where it can potentially conserve energy by transitioning to a low-power state, it enrolls a timer callback in the initial routine known as event start. Afterwards, the execution continues following the regular workflow of the MPI phase. Once the timer reaches its limit, a system signal is triggered, causing an interruption in the "normal" execution of the MPI code and the reduction of CPU speed. After the signal handler is triggered, it calls the COUNTDOWN callback, and once the callback finishes executing, execution of the MPI code resumes from the point where it

```
1  PreTask()
2
3     taskid = hash(stack pointer chain)
4     if isnew(taskid) then
          /* First instance of a task:  Choose fastest
             frequency.                                  */
5        ; f = f̂
6     else
          /* Look up correct frequency.                  */
7        ; f = Sched[taskid]
8     SetFreq(f)
9     InitPerformanceCounters()
10    RunTask(taskid)
11
12 PostTask()
13
      /* Generate schedule for next execution of this
         task.                                          */
14    ; Record I, t_comp, t_lib.
15    Rates[taskid][f] = I/t_comp
16    t = t_comp + t_lib
17    t_target = t − t_copy
18    if isnew(taskid) then
          /* First instance of a task:  Set slowdown
             rates to worst-case for each available
             frequency.                                 */
19       ; for f ∈ F do
20          Rates[taskid][f] =
21           Rates[taskid][f̂] ∗ f̂/f
22       end
23    /* Find slowest frequency that respects the
         critical path.  Default is fastest frequency.
         */
24    ; Sched[taskid] = f̂
25    for f from slowest (f̄) to fastest (f̂) do
26       if I/Rates[taskid][f] ≤ t_target then
27          Sched[taskid] = f
28          return;
29
30    end
```

Figure 6.3: Adagio algorithm.

Source: Rountree et al. [40], Rountree [41].

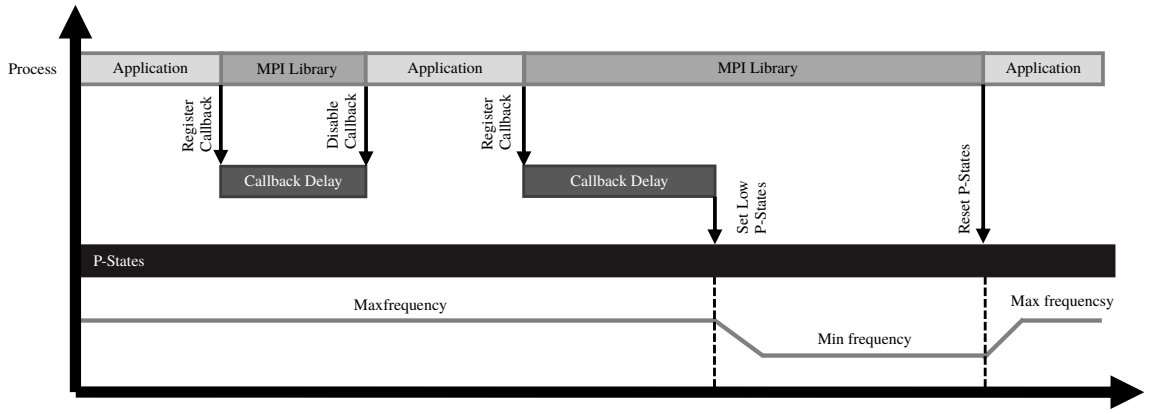


Figure 6.4: Approach used by COUNTDOWN library to save energy during communication times.

Source: Cesarini et al. [34, 36, 52].

was interrupted. If the execution of the "normal" phase returns to COUNTDOWN before the timer expires, COUNTDOWN will disable the timer in the epilogue routine, and the execution will proceed as if nothing had occurred.

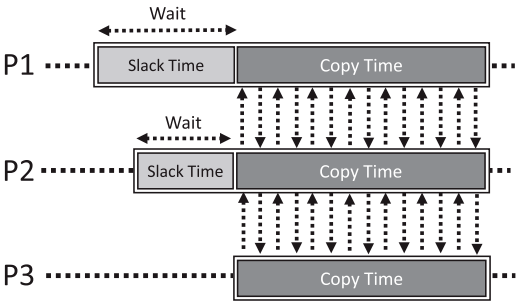
COUNTDOWN adopts a fixed-time approach, whereby it expects the MPI instruction to terminate within $500\,\mu s$, which according to literature is the ideal value section 4.4. If it does not terminate within this time frame, COUNTDOWN idles the node and waits. Regarding fig. 4.9, the operation of COUNTDOWN can be summarised as follows: if $T \geq 500\,\mu s$, the program transitions the processor to P-states until the MPI primitive is invoked. This approach proves to be highly efficient since as soon as the MPI primitive is invoked and any data exchange occurs, the program exits the P-states.

COUNTDOWN develops a strategy, described in fig. 4.9, to prevent the processor entering C-States even during MPI communications, but since MPI offers not only a communication system but also a data exchange system, if an MPI program enters a C-states, there is a high risk that without the possibility to communicate via hardware primitives, as we will see later for Intel MPI, the program gets stuck in a C-states even during the communication phase, which degrades the performance quite a bit. The division of time, communication, and synchronization, is described by section 5.3.1 and illustrated in fig. 6.5a, which in the case of big data can slow down performance: an `MPI_Barrier` is made before each collective instruction, this ensures that during copy time there is no performance degradation. This workaround is described in fig. 6.5b.

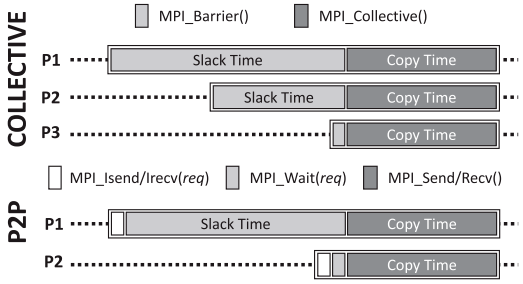
COUNTDOWN is described in three main papers:

1. Cesarini et al. [36] introduces the COUNTDOWN library designed to save energy in MPI applications without compromising performance. To prevent frequent switches between the application and MPI context, COUNTDOWN utilizes a timeout strategy instead of altering the power state of CPU cores. Moreover, COUNTDOWN can distinguish short and overhead-intensive MPI phases with negligible energy-saving impact. This reactive approach can be used with existing MPI applications without requiring code modifications.

2. Cesarini et al. [35] focuses on reducing the energy footprint in large-scale MPI ap-



(a) The execution model and classification. Each process is assigned a task. The sum of slack time (*Tslack*) and Copy Time (*Tcopy*) is equivalent to MPI Communication Time (*Tcomm*).



(b) Artificial barrier mechanism to split *Tslack* and *Tcopy* of an MPI primitive.

Figure 6.5: COUNTDOWN: mechanism to prevent the processor from going into the P-States while data is being copied.

Source: Cesarini et al. [35].

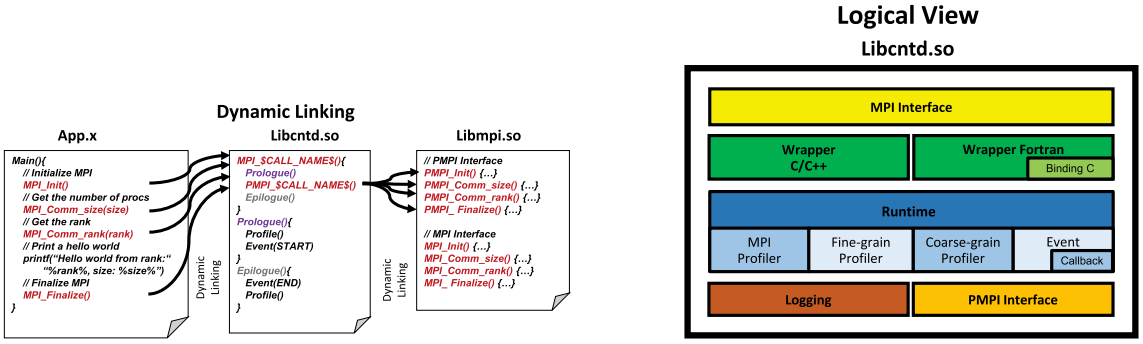
plications. It might address specific strategies for managing energy resource allocation in large MPI clusters, aiming to maximize energy savings without performance trade-offs.

3. Cesarini et al. [34] emphasises an "application-agnostic" approach to energy-saving management in MPI communication primitives. The COUNTDOWN library is designed to apply to a wide range of MPI applications without requiring significant customizations.

The Profiler is a crucial element of COUNTDOWN, which is used to examine the properties and actions of MPI primitives. Another notable module is the Event Module, which plays an important role in responding to events and monitoring power states. All MPI functions are wrapped by some transparent Wrapper Functions; this allows you to enclose MPI calls and enable COUNTDOWN to intercept and handle them. COUNTDOWN instruments an application by wrapping each MPI call in a wrapper function that contains prologue and epilogue routines. These routines are used for profiling and power management, respectively. The library interacts with the hardware power manager through an events module and can be triggered by system signals for timing purposes. Finally, COUNTDOWN's configuration options can be customized via environment variables, such as adjusting the logging verbosity and selecting hardware performance counters to monitor. More detailed explanations of these options can be found in chapter 7 [36].

COUNTDOWN provides a familiar interface that replicates a standard MPI library. It is written in C and intercepts all MPI calls from the application, and has separate wrappers for C/C++ and Fortran MPI libraries due to differences in assembly symbols. These wrappers provide a bridge between the different language libraries. COUNTDOWN supports dynamic linking to instrument applications without requiring source code modifications or toolchain changes [36].

Figure 6.6 illustrates the dynamic linking events that occur when COUNTDOWN is injected into the application during the loading process, providing a logical view of all the



(a) Dynamic Linking. (b) Logical View.

Figure 6.6: Dynamic linking events occured upon injecting COUNTDOWN into the application during the loading process and logical view of all the components

Source: Cesarini et al. [36].

components involved.

6.8 ATFaVSCP Tool

ATFaVSCP (Adaptive Transparent Frequency and Voltage Scaling for Communication Phases) Lim et al. [53] is a seamless system designed to enhance energy efficiency in MPI programs. It achieves this by intelligently reducing the processor's P-States during communication phases, effectively decreasing CPU performance to optimize energy consumption. ATFaVSCP identifies through patterns from CPU registers whether the processor is currently inside or outside an MPI call without creating a wrapper for MPI calls, thus with granularity not at the MPI call level but through an analysis of the operations performed by the CPU. Identification is done without the use of artificial intelligence systems, but through the recognition of predefined patterns. Simply due to this, it autonomously selects the appropriate CPU frequency to minimize the energy-delay product, contributing to both energy savings and improved performance.

Crucially, all the analysis and subsequent frequency and voltage scaling operations take place entirely within the MPI framework. This approach ensures complete transparency in the application. As a result, the vast array of existing MPI programs, as well as those under development, can seamlessly integrate and benefit from our system without requiring any modifications, as outlined in Lim et al. [53].

Unlike MPInside and COUNTDOWN, this approach does not act as a wrapper to MPI instructions, but merely transparently analyzes the execution flow and guesses where there is a busy wait for communication, and takes care of lowering the power consumption of the processor in this state. The detection mechanism is based on the number of instructions per clock (IPC), the fewer they are, the more plausible it is that it is a busy wait, but it could also be SIMD instructions such as AVXs or others, this creates false negatives, i.e., areas of misidentified busy wait. It is therefore necessary to train the algorithm in relation to the program to be optimized. This algorithm saves 10 % to 20 % energy in the NAS Parallel Benchmark Suite [53].

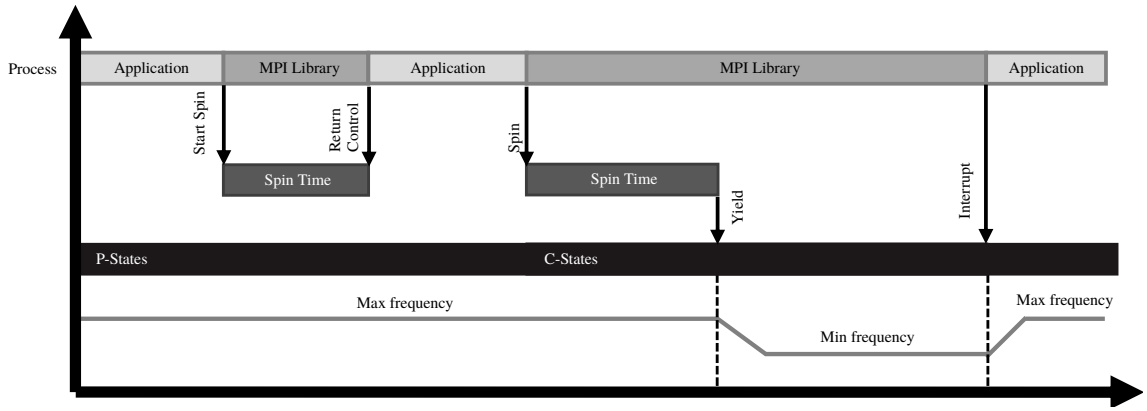


Figure 6.7: Approach used by the Intel MPI library to save energy during communication times.

Source: Cesarini et al. [36].

6.9 Intel MPI: Energy Efficient Approach

The Intel MPI library [35] follows a similar strategy to COUNTDOWN however instead of working with the P-States it uses the C-States (idle states) of the CPU cores. This strategy is not enabled by default since changing a state to a processor is a very onerous operation, much more so than working on the various P-states. The strategy is illustrated in fig. 6.7. By configuring environment variables like `I_MPI_WAIT_MODE` and `I_MPI_SPIN_COUNT`, it is feasible to define the duration of spin count, i.e., the time from the start of an MPI call, to the start of communication, a concept very similar to the COUNTDOWN timer. Once the spin count reaches zero, the Intel MPI library allows the CPU core to be allocated to its idle task, allowing it to enter a low-power state (C-States) to conserve power. Execution is resumed when a system interrupt wakes up the MPI library, indicating the end of the MPI call. This mode is referred to as MPI SPIN WAIT. It can be seen that the strategy is virtually identical to that of COUNTDOWN: both COUNTDOWN and the Intel MPI library aim to reduce power consumption during idle or waiting periods, enhancing energy efficiency without compromising performance [34, 36].

The main problems with Intel MPI Library are first of all that it uses C-States instead of frequency reduction, this causes more management overhead, and the second is that it is closed-sourced and only works on Intel processors and uses APIs that are not available on all x86 processors.

The use of a C-States instead of a low-power P-States means that there is an increase in execution time close to $25.85\,\%$ compared to the polling strategy [35]. Although the energy savings are negative ($-12.72\,\%$), there is a corresponding power reduction of $12.83\,\%$. This discrepancy is attributed to the entry into C-states during wait periods, which contributes to the overall power savings [35].

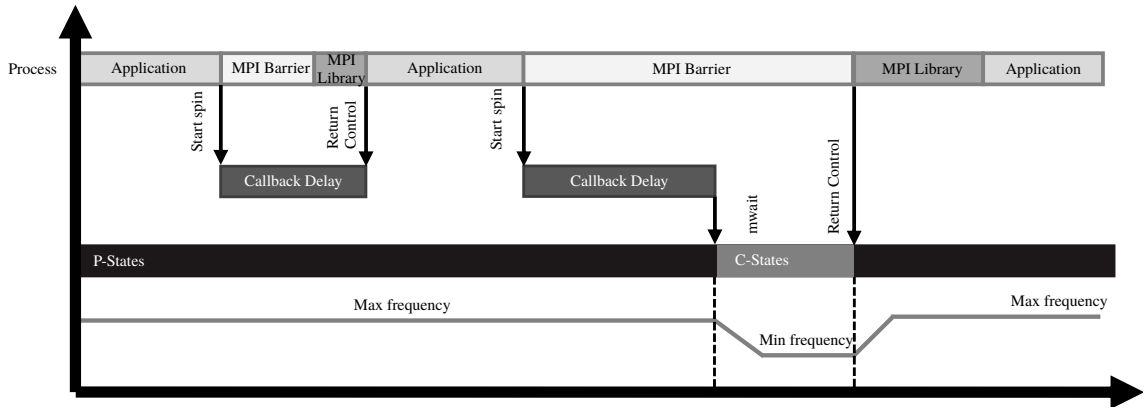


Figure 6.8: Approach used by MVAPICH2 library to save energy during communication times.

6.10 MVAPICH2: Energy Efficient Approach

The last approach is implemented in MVAPICH2 [54], an MPI implementation crafted by Ohio State University for compatibility with InfiniBand and Cuda GPUs which adopts a hardware-level strategy. This entails leveraging native hardware instructions, such as `mwait` for Intel processors, to temporarily pause processors during slack times. This approach not only reduces energy consumption but also alleviates the busy-wait problem.

In this strategy, Core-idling is executed by suspending processes, through a low P-States, during periods of inactivity in wait time. Processes are then resumed as needed, eliminating the necessity of entering the waiting mode.

This implementation is still in its early stages and currently only facilitates forcing the processor into a low or high state, always within a P-States. However, according to the authors Kim et al. [54], it will be possible to employ other energy-saving policies, given the numerous methods available for implementing process suspension and resumption. Various approaches can be employed, including assembly instructions like `mwait` on Intel processors, which sets a hardware busy wait, which follows the intel MPI liberia strategy through spin time for entering an energy-saving mode and await an event.

The authors Kim et al. [54] also screened the use of other tools to avoid using architecture-dependent instructions, such as timers, semaphores, and signals. However, timers are only suitable for controlling larger units of time, and semaphores have the risk of causing deadlocks. Therefore, in the initial implementation, the authors chose to use signaling. Consequently, the implementation is CPU-independent and easily extensible to support the inter-node communication channel.

The current implementation of this policy in MVAPICH2 v2.3.1 involves the framework activating a user-defined energy-saving policy when the process is about to enter the busy-waiting mode [54].

6.11 Comparison

This section provides a comprehensive look at various tools and libraries commonly used in MPI implementations. Table 6.1 outlines the main features of the MPInside Library,

Jitter, Offline Scheduler, Fermata, Adagio-Computation, Adagio COUNTDOWN Library, ATFaVSCP tool, Intel MPI and MVAPICH2. Specifically, the table details each tool's.

- **MPI implementation**: whether it works with all MPI implementations or requires a specific implementation
- Availability of a **profiler**: whether or not there is a low performance profiler profiling the program.
- **Mode of operation**: whether the run is online, whether it manages from previous estimates or data what the ideal frequency is, or whether it is offline, whether it uses data from a previous run to estimate the ideal frequency, or whether it requires training.
- **Granularity**: whether it works at per MPI call or at timestamp or timestep. Timestamp granularity indicates an analysis based on fixed time sampling, for example every 100 microseconds, timestep granularity indicates a granularity of detail for each iteration of a given scientific computing software; this dictates that there are a small number of MPI calls within that step. Finally, timeslice granularity indicates a granularity in which each individual step/iteration is divided into many slices.
- **Ideal frequency**: which allows energy saving capabilities during computation and busy waits.
- **Presence or absence of false negatives**: for example, if the tool exchanges a simple while that computes a sum with a busy wait.
- Whether it works with *blackbox code or requires recompilation*.
- **Availability of the source code** (Open Source).

The desired features are obviously an online scheduler that acts directly without training, with granularity at MPI call level or higher, the absence of false negatives, the achievement of the ideal frequency, and finally the ability to run with blackbox code and the presence of source code. Based on the information provided in table 6.1, it can be concluded that of all the algorithms listed, only two meet our requirements. Of these two, our main focus is on the Open Source one, COUNTDOWN.

Table 6.1: Comparison of tools and libraries. Bold entries show desired characteristics.

Tool/Library	**MPI Implementation**	**Profiler**	**Online or Offline**	**Granularity**	**Energy Saving on Computation**	**Energy Saving on Busy Wait**	**Ideal frequency**	**False Negatives**	**Work with blackbox code**	**Source code availability**	**Approach figure**
MPInside Library [46]	Any	**Yes**		**MPI call**	No	No			**Yes**	No	
Jitter [37]	Any	No	**Online**	Timestep	**Yes**	No	Only on computation	**No**	**Yes**	No	Figures 6.1 and 6.2
Offline Scheduler [39–41]	Any	**Yes**	**Online**	Timestamp	**Yes**	**Yes**	**Yes**	Yes	No		
Fermata [39–41]	Any	**Yes**	**Online**	**MPI call**	No	**Yes**	No	**No**	**Yes**	No	
Adagio-Computation [39–41]	Any	No	**Online**	Timeslice	**Yes**	No	No	**No**	**Yes**	No	
Adagio [39–41]	Any	No	**Online**	**MPI call and Timeslice**	**Yes**	No	**Yes**	**No**	**Yes**	No	Figure 6.3
COUNTDOWN Library [34–36]	Any	**Yes**	**Online**	**MPI call**	No	**Yes**	Only on slack time	**No**	**Yes** [55]	**Yes**	Figures 6.4 and 7.4
ATFaVSCP Tool [53]	Any	No	**Online**	Timestamp	No	**Yes**	Only on slack time	Yes	**Yes**	No	
Intel MPI [34–36]	Intel MPI	No	**Online**	**MPI call**	No	**Yes**	Only on slack time	**No**	**Yes**	No	Figure 6.7
MVAPICH2 [54]	MVAPICH2	No	**Online**	**MPI call**	No	**Yes**		**No**	**Yes**	No	Figure 6.8

7

The COUNTDOWN Library

This section provides an in-depth exploration of the COUNTDOWN library to illustrate its functionality, organisational structure, integration process with applications and the various source code components. The insights presented here are derived, in part, from the authors' articles, in particular Cesarini et al. [34, 35, 36, 52], Cesarini [56], Bartolini et al. [57]. Nevertheless, to ensure a complete and accurate representation of COUNTDOWN, some crucial details have been extracted directly from the source code accessible on GitHub at [55]. This decision is motivated by the realisation that the documentation is not exhaustive to the levels required for this study.

The organization of the chapter is as follows: section 7.1 introduces how to follow the COUNTDOWN library, followed by section 7.2 which tells what the configuration options of the said library are. After that, section 7.3 gives a detail about the general report that provides COUNTDOWN, followed by section 7.4 whitch talks about the low-level library architecture, and the next sections describe it at a high level (sections from 7.5 to 7.10). Finally, sections 7.11 and 7.12 talks about the overhead and performance evaluation of COUNTDOWN by the authors.

7.1 Shell Execution of an Application with the COUNTDOWN Library

COUNTDOWN employs a technique where the COUNTDOWN library is loaded prior to any other system libraries. This is achieved by utilizing the environment variable `LD_PRELOAD`, which specifies the paths that the shared library will load before any other library, including the C runtime library `libc.so`. This method is commonly used to wrap system function like the malloc, thanks to this to profile an application via COUNTDOWN you only need to run add an environment variable as shown in listing 7.1.

```
export LD_PRELOAD=/path/to/libcntd.so
mpirun -np 128 ./a.out args...
```

Listing 7.1: Shell execution of an application with the COUNTDOWN profiler.

Since COUNTDOWN is loaded as an environment library that overrides the MPI library by overwriting the environment variable `LD_PRELOAD` it can not access the Linux model-specific registers (MSRs) including those pertaining to power management described in section 5.2 since SELinux (Security-Enhanced Linux) policies prohibit access to these registers from source code external to the running application and/or kernel modules, so for example all external libraries.

The strategy so far used to overcome this problem is the one used by MPInside described in section 6.2, that is, to handle the execution of the code herself, but this could be a problem if the code itself requires the use of MRS registers, but this was contrary to the principles of COUNTDOWN, i.e. being a wrapper between the application and MPI callable via an environment variable, in order to maintain this transparency and avoid the need for recompilation the authors of COUNTDOWN decided to use MSR-SAFE (Model-Specific Register - Software Access to Flags and Events) to modify the architectural register independently for each core, allowing it to change the current P-States (processor frequency and voltage) per core. MSR-SAFE is a driver for the Linux kernel, which must be compiled for the specific platform, that acts as a wrapper between the system files that govern the CPU and the rest, ensuring that all applications have the aforementioned permissions [58, 59]. This driver is a kernel module that implements access-control lists for model-specific registers, allowing controlled userspace access to these registers. By using MSR-SAFE, system administrators can grant trusted users read access to registers at the register level and write access at the bit level. This is particularly useful in production environments where kernel drivers may not support new processor features or where batch access to multiple registers is needed to meet performance requirements.

7.2 COUNTDOWN Options

As seen in section 7.1, using the COUNTDOWN profiler is a very simple operation, similarly so is enabling the COUNTDOWN and COUNTDOWN Slack algorithms, as seen in listing 7.2 and listing 7.3. An exhaustive list of all the options are collected in the next two pragraphs, the first is devoted to compile-time settings, the second to run-time environment settings.

```
export LD_PRELOAD=/path/to/libcntd.so
export CNTD_ENABLE=1
mpirun -np 128 ./a.out args...
```

Listing 7.2: Shell execution of an application with the COUNTDOWN algorithm.

```
export LD_PRELOAD=/path/to/libcntd.so
export CNTD_ENABLE=1
export CNTD_SLACK_ENABLE=1
mpirun -np 128 ./a.out args...
```

Listing 7.3: Shell execution of an application with the COUNTDOWN algorithm and COUNTDOWN Slack algorithm.

Build options COUNTDOWN offers five compilation options that modify what COUNTDOWN does. In the experiments carried out in chapter 8 we use all the default build options, as we do not want to make any changes that would externally affect the results. To compile COUNTDOWN, we can use the commands shown in listing 7.4 to run in a Linux environment containing `build-essential`, `openmpin-bin`, `libopenmpi-dev`.

- `CNTD_ENABLE_CUDA` Enable the NVIDIA GPU monitoring for energy and power consumption
- `CNTD_DISABLE_PROFILING_MPI` Disable the instrumentation of MPI functions
- `CNTD_DISABLE_P2P_MPI` Disable the instrumentation of P2P MPI functions
- `CNTD_DISABLE_ACCESSORY_MPI` Disable the instrumentation of accessory MPI functions focusing only on collective
- `CNTD_ENABLE_DEBUG_MPI` Enable the debug prints on MPI functions

```
mkdir build
cd build
cmake ..
cd ..
make
```

Listing 7.4: Compilation of COUNTDOWN.

Environment variables The experiments described in chapter 8 utilized various settable COUNTDOWN Environment variables. These variables included CNTD and CNTD slack, max and min pstate, timeout, and enable timeseries report.

- `CNTD_ENABLE` Enable the COUNTDOWN algorithm (when the value is `enable`, `on`, `yes`, `true` or `1`) or enable only the analysis of energy-aware MPI (when value is `analysis`). This parameter is required.
- `CNTD_SLACK_ENABLE` Enable the COUNTDOWN Slack algorithm (when the value is `enable`, `on`, `yes`, `true` or `1`) or enable only the analysis of energy-aware MPI (when the value is `analysis`). This parameter is required.

- `CNTD_MAX_PSTATE` Force an upper bound frequency to use E.x. `24` is 2.4 GHz frequency.
- `CNTD_MIN_PSTATE` Force a lower bound frequency to use E.x. `12` is 1.2 GHz frequency.
- `CNTD_TIMEOUT` Timeout of energy-aware MPI policies in microseconds, default 500 µs.
- `CNTD_FORCE_MSR` Force the use of MSR instead (when the value is `enable`, `on`, `yes`, `true` or `1`) of MSR-SAFE driver, the application must run as root.
- `CNTD_SAMPLING_TIME` Timeout of system sampling, default 1 s, max 600 s.
- `CNTD_OUTPUT_DIR` Output directory of report files.
- `CNTD_TMP_DIR` Temporary directory of report files.
- `CNTD_PERF_ENABLE` Enable Linux perf monitoring when the value is `enable`, `on`, `yes`, `true` or `1`.
- `CNTD_PERF_EVENT_X` Configure the perf event X, where X is between 0 and the maximum available PMUs of the CPU architecture, while the value must be in hex format. COUNTDOWN support at most custom 8 performance events. By default, it already analyzes 12 perf events.
- `CNTD_DISABLE_POWER_MONITOR` Disable the energy/power monitoring when the value is `enable`, `on`, `yes`, `true` or `1`.
- `CNTD_ENABLE_REPORT` Save the summary report as a file.
- `CNTD_ENABLE_TIMESERIES_REPORT` Enable time-series reports, default sampling time 1 s.

Performance event The perf events are implementation-defined; see The Intel CPU manual Volume 3B documentation [60] or the AMD BIOS and Kernel Developer Guide [61]. After this, we can use `libpfm4` [62] library. This library is designed to facilitate the translation process from the name used in architectural manuals to the corresponding raw hex value required in the `CNTD_PERF_EVENT_X` field of COUNTDOWN. One possible use of this command is to be able to see what state the processor is in, from the Intel manual you can see that the P-states information is contained in the `IA32_MPERF` and `IA32_APERF` instructions. The `IA32_MPERF` instruction provides the actual performance state of the processor, representing the count of elapsed core cycles in the maximum performance state. On the other hand, the `IA32_APERF` instruction provides the accumulated performance state, indicating the count of elapsed core cycles in any performance state.

The default 12 metrics for COUNTDOWN monitoring are:

- `PERF_INST_RET`: This metric represents the number of retired instructions, indicating the total count of instructions that have been executed and completed.

- `PERF_CYCLES`: Measures the total number of CPU cycles executed for a given operation or code segment, providing insight into the overall computational workload.
- `PERF_CYCLES_REF`: This metric specifically counts the reference cycles, offering a refined perspective on the cycles that directly contribute to the computation.
- `PERF_SCALAR_DOUBLE`: This represents the count of scalar double-precision floating point operations, offering insights into the performance of operations involving double-precision floating point numbers.
- `PERF_SCALAR_SINGLE`: Similarly to the previous metric, this one counts scalar single-precision floating point operations, providing details on the performance of operations involving single-precision floating point numbers.
- `PERF_128_PACKED_DOUBLE`: Measure the count of 128-bit packed double-precision floating point operations, giving visibility into the performance of operations involving vectors of double-precision numbers.
- `PERF_128_PACKED_SINGLE`: This metric counts 128-bit packed single-precision floating point operations, offering insights into the performance of operations involving vectors of single-precision numbers.
- `PERF_256_PACKED_DOUBLE`: Similarly to the 128-bit version, this metric counts 256-bit packed double-precision floating point operations.
- `PERF_256_PACKED_SINGLE`: This metric counts 256-bit packed single-precision floating point operations.
- `PERF_512_PACKED_DOUBLE`: Similarly to the previous two, this metric counts 512-bit packed double-precision floating point operations.
- `PERF_512_PACKED_SINGLE`: This metric counts 512-bit packed single-precision floating point operations.
- `PERF_CAS_COUNT_ALL`: It represents the total count of Compare-And-Swap (CAS) operations, providing information on the number of such atomic operations performed.

Hardware power monitor The hardware monitor requires that can be read the access of CPU (`/sys/devices/system/cpu/*`) in the Intel architecture via the MSR-SAFE driver, the On-Chip Controller (`/sys/firmware/opal/exports/occ_inband_sensors`) on the IBM Power 9 architecture, and the `tx2mon` driver on Marvell ThunderX2 on `/systems/devices/platform/tx2mon/`. In cases where the MSR-SAVE driver is not accessible, performing these operations may necessitate root or other privileged permissions. If obtaining such permissions is not feasible, an alternative option is to deactivate hardware monitors.

7.3 COUNTDOWN Report

The COUNTDOWN report contains the following information, an example of a report is given in listing 7.5.

- **EXE time**: Time of execution of the program.
- **GENERAL INFO**: Contains general information on MPI Ranks, Nodes, Sockets, CPUs, and GPUs.
  - **Number of Nodes**: Number of nodes in the system.
  - **Number of MPI Ranks**: Total number of MPI ranks (nodes $\times$ task).
  - **Number of CPUs**: Total number of CPUs (nodes $\times$ tasks $\times$ threads).
  - **Number of Sockets**: Number of sockets for OpenMPI communication.

  A graphical representation is shown in fig. 4.6.
- **PKG**: Information about the package (a single CPU) with consumed Energy (Joule) and AVG Power (Watt).
- **DRAM**: Information about RAM (across all nodes) with consumed Energy (Joule) and AVG Power (Watt), available only on intel platform.
- **GPU**: Information about the GPU (across all nodes) with consumed Energy (Joule) and AVG Power (Watt).
- **MPI network** (TOT, SENT, and RECV): Data shared between nodes and threads (total, send, and receive data).
- **MPI file** (Read, Write): Data read and written by `MPI_file` instructions.
- **MAX Memory usage**: Maximum memory (RAM) used.
- **AVG IPC (Instructions per clock), AVG CPU frequency, Cycles, Instructions retired**: Information about CPU usage, given by the first 4 metrics of section 7.2.
- **SIMD information** (Single Instruction, Multiple Data): performance in handling floating point operations at different vector sizes.
  - **DP UOPs** (total, 64, 128, 256, 512): Total number of double-precision floating point micro-operations, determined by the 6th, 8th, and 10th metrics in section 7.2.
  - **DP FLOPs** (total, 64, 128, 256, 512): Total number of floating point operations equivalent to double precision, equivalent to the DP UOPs but considering the appropriate factor for each vector size ($\times 1, 2, 4, 8$).
  - **SP UOPs** (total, 64, 128, 256, 512): Total number of single-precision floating point micro-operations, given by the 7th, 9th, and 11th metrics in section 7.2.

  - **SP FLOPs** (total, 64, 128, 256, 512): Total number of floating point operations equivalent to single precision, equivalent to the DP UOPs but considering the appropriate factor for each vector size ($\times 1, 2, 4, 8$).

  The dimensions (32, 128, 256, 512) denote the widths of SIMD (Single Instruction, Multiple Data) registers employed to concurrently execute multiple operations. In practice, floating point operations are performed on data organized in vectors (or "packed" vectors) to harness parallelization.

- **MEM UOPs and MEM GLOBAL DATA**:
  - **MEM UOPs** (Memory Micro-Operations): This metric provides insights into the memory-related micro-operations, reflecting the efficiency and usage patterns of memory operations within the system.
  - **MEM GLOBAL DATA**: This metric focuses on global memory data with the overall utilization of the system's global memory.
- **GPU reporting**:
  - **GPU util**: Indicates GPU utilization, representing the proportion of time the GPU spends actively processing tasks.
  - **GPU mem util**: Reflects GPU memory utilization, showcasing the percentage of GPU memory actively in use.
  - **GPU temp**: Providing information on the thermal conditions (temperature) of the GPU.
  - **GPU freq**: Indicates the GPU frequency, representing the clock speed at which the GPU is operating.
- **MPI timing**: time for APP and MPI
  - **EXE time**: process execution time measured from the first Rank started to the last Rank to finish
  - **TOT time**: sum of all execution times of the Ranks
  - **APP time**: sum of all code execution times excluding the MPI calls of the Ranks
  - **MPI time**: sum of all MPI timing of the Ranks

  A graphic representation is in fig. 7.1.
- **MPI detail exchange data**: This feature provides a detailed list of all MPI primitives intercepted by COUNTDOWN, along with the corresponding count of calls made to each of them. It offers a granular view of the MPI operations executed during the program's runtime.
- **MPI slack report**: The MPI slack report presents a list of MPI primitives for which the artificial barrier has been enabled.

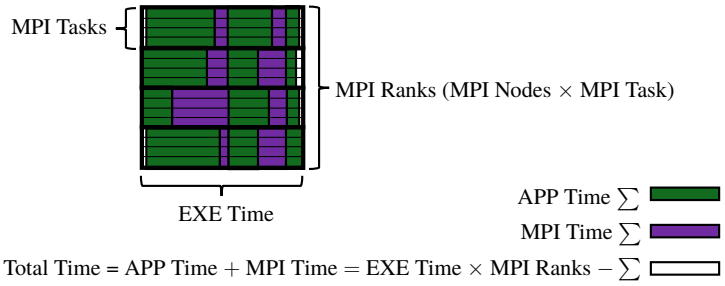


Figure 7.1: Graphical representation of MPI timing.

```
#######################################################
#################### COUNTDOWN ########################
#######################################################
EXE time: 10.637 sec
#################### GENERAL INFO #####################
Number of MPI Ranks:        96
Number of Nodes:             16
Number of Sockets:           32
Number of CPUs:             768
#################### ENERGY ###########################
PKG:                    38417 J
DRAM:                   9353 J
#################### AVG POWER ########################
PKG:                    3611.54 W
DRAM:                   879.28 W
################## PERFORMANCE INFO ###################
MPI network - SENT:     2.21 GByte
MPI network - RECV:     2.21 GByte
MPI network - TOT:      4.41 GByte
MPI file    - WRITE:    0 Byte
MPI file    - READ:     0 Byte
MPI file    - TOT:      0 Byte
MAX Memory usage:           33.86 GByte
AVG IPC:                                    2.19
AVG CPU frequency:                       3094 MHz
Cycles:                                  2956364089090
Instructions retired:                    6498795275628
DP FLOPs:                                                                184216886431
DP FLOPs 64:                                                             180042585091
DP FLOPs 128:                                                              4163398812
DP FLOPs 256:                                                                       0
DP FLOPs 512:                                                                10902528
DP UOPs (TIME_EN/TIME_RUN):         182125647313 (   3916346134636/   3916346134636)
DP UOPs 64 (TIME_EN/TIME_RUN):      180042585091 (    979085882214/    979085882214)
DP UOPs 128 (TIME_EN/TIME_RUN):       2081699406 (    979086322487/    979086322487)
DP UOPs 256 (TIME_EN/TIME_RUN):               0 (    979086745428/    979086745428)
DP UOPs 512 (TIME_EN/TIME_RUN):         1362816 (    979087184507/    979087184507)
SP FLOPs:                                                                       5376
SP FLOPs 32:                                                                    5376
SP FLOPs 128:                                                                      0
SP FLOPs 256:                                                                      0
SP FLOPs 512:                                                                      0
SP UOPs (TIME_EN/TIME_RUN):                5376 (   3916346989999/   3916346989999)
SP UOPs 32 (TIME_EN/TIME_RUN):             5376 (    979086099830/    979086099830)
SP UOPs 128 (TIME_EN/TIME_RUN):               0 (    979086533160/    979086533160)
SP UOPs 256 (TIME_EN/TIME_RUN):               0 (    979086961881/    979086961881)
SP UOPs 512 (TIME_EN/TIME_RUN):               0 (    979087395128/    979087395128)
MEM UOPs (TIME_EN/TIME_RUN):          7055202636 (    127652337360/    127652337360)
MEM GLOBAL DATA:                                                          451532968704
#################### MPI TIMING #######################
APP time: 746.249 sec (72.82%)
MPI time: 278.545 sec (27.18%)
TOT time: 1024.794 sec (100.00%)
#################### MPI REPORTING ####################
MPI_INIT_THREAD: 96 - 0.000 Sec (0.00%)
MPI_ALLGATHER: 2688 - 1.092 Sec (0.39%) - SEND 2.43 MByte - RECV 2.43 MByte
MPI_ALLGATHERV: 96 - 0.046 Sec (0.02%) - SEND 47.11 MByte - RECV 47.11 MByte
MPI_ALLREDUCE: 481632 - 119.801 Sec (43.01%) - SEND 2.15 GByte - RECV 2.15 GByte
MPI_ALLTOALL: 1344 - 13.364 Sec (4.80%) - SEND 1.97 MByte - RECV 1.97 MByte
MPI_BARRIER: 5376 - 70.101 Sec (25.17%)
MPI_BCAST: 5184 - 45.254 Sec (16.25%) - SEND 5.80 MByte - RECV 5.74 MByte
MPI_COMM_SPLIT: 96 - 0.024 Sec (0.01%)
MPI_REDUCE: 1728 - 0.010 Sec (0.00%) - SEND 17.81 KByte - RECV 18.00 KByte
MPI_SCATTER: 64512 - 4.067 Sec (1.46%) - SEND 252.00 KByte - RECV 249.38 KByte
MPI_WAITALL: 68160 - 20.020 Sec (7.19%)
MPI_WAIT: 900516 - 4.763 Sec (1.71%)
MPI_FINALIZE: 96 - 0.001 Sec (0.00%)
#######################################################
```

Listing 7.5: Aggregate data report provided by COUNTDOWN.

7.4 Components of COUNTDOWN

The following description outlines key source code files, each serving a specific role in the COUNTDOWN application's implementation.

init This section is responsible for initializing the COUNTDOWN library. It manages the loading of COUNTDOWN settings from environment variables or default values and ensures that these settings are applied to the running program. The set of environment variables is in section 7.2.

wrapper_pmpi_c_cpp This subsection intercepting MPI calls within C/C++ code. It applies COUNTDOWN and/or slack mechanisms and then calls the MPI primitive. This code supports 344 MPI functions specified in the MPI 4.0 Specification.

wrapper_pmpi_fortran Similar to the C/C++ counterpart, this subsection focuses on intercepting MPI calls, but specifically within Fortran code. It applies COUNTDOWN and/or slack and calls the corresponding MPI primitive. This code supports 344 MPI functions specified in the MPI 4.0 Specification.

timer This is responsible for managing time-related aspects. It generates an alarm in the CPU after a 500 500 µs and ensures that the processor enters a low P-states if the MPI instruction is not completed. Importantly, it employs an asynchronous, event-driven alarm mechanism without resorting to busy waits or other CPU-intensive modes. More details are in section 7.7.

eam Short for "event alarm manager", this subsection handles the alarms generated by the timer. More details are in section 7.6.

eam-slack This subsection handles the alarms generated by the timer if the slack is enabled. More details are in section 7.10.

sampling This section is dedicated to real-time sampling of various hardware parameters. It collects data and saves it to reports, providing crucial insights into the system's performance. More details on this subsection are in section 7.5.

report Building on the data gathered through real-time sampling, the report section generates summary information. It provide a report with as a comprehensive overview of the system's status and performance. If enabled it also generates the report of custom performance events as well as all MPI calls, all data transfers, and the various general counters. More details on this subsection are in section 7.3.

pm Abbreviated for "power management", this subsection manages the DVFS of the processor, specifically dealing with frequencies of P-states. It plays a role in optimizing power consumption based on the system's needs. More details on this subsection are in section 7.8.

hwp Standing for "hardware performance", this part of the codebase deals with data obtained from the hardware performance monitor. It likely involves processing and utilizing information related to the system's hardware components.

cpufreq This subsection is responsible for limiting the CPU frequency if requested via the environment variables `CNTD_MAX_PSTATE` and `CNTD_MIN_PSTATE`.

7.5 COUNTDOWN Profiler

The COUNTDOWN Profiler is a tool comprising two essential components: the Event Profiler/MPI Profiler and the Time-based Profiler, each serving distinct yet interconnected purposes.

Event Profiler/MPI Profiler The Event Profiler/MPI Profiler takes a detailed approach to monitor the hardware (HW) performance of Intel processors. Utilizing the RDPMC instruction, it tracks micro-architectural events such as clock cycles, instructions retired, and configurable HW performance counters. This profiler is adept at extracting MPI information from parameters passed to MPI primitives. It creates a comprehensive MPI profile, capturing crucial details like MPI communicators, MPI groups, core ID, and meticulously profiling each MPI call. This includes recording the call type, entrance and exit times, and the data exchange with other MPI processes.

Time-based Profiler The Time-based Profiler adopts a broader strategy, collecting a diverse set of HW performance counters at regular intervals. It works by sampling every few seconds through timers, each dedicated to each MPI rank that collects all the data from the various OpenMP threads, ensuring an equitable distribution of the profiling overhead among the MPI ranks. Leveraging the MSR-SAFE kernel driver and Intel RAPL registers, it monitors CPU and DRAM energy/power consumption.

- **Fine-grain Micro-architectural Profiler** At a more granular level, the Fine-grain Micro-architectural Profiler captures micro-architectural insights with precision. It accesses performance counters using the RDPMC instruction, monitoring average frequency (in some architecture), time stamp counter (TSC), and instructions retired during each MPI call.
- **Coarse-grain Profiler** The Coarse-grain Profiler extends its focus to a broader set of HW performance counters available in Intel architectures. Overcoming access limitations, it employs the MSR-SAFE driver and a scheduler plugin to grant users access. At its fundamental level, it keeps track of TSC, the number of executed instructions, the average frequency, the durations of C-states, and the temperature. Furthermore, it monitors the energy consumption of the CPU package, the residency of C-states, and the temperature of the package by utilizing RAPL. Due to the relatively higher overhead of accessing monitored counters, it employs a time-based sample rate. Activation is triggered by the fine-grain profiler based on periodic timestamp checks.

The profiler generates a summary report, described in section 7.3, which not only summarizes events and time-based traces but also improves readability and simplifies long-term compression, and then produces a CSV output containing a time-series, if activated with `CNTD_ENABLE_TIMESERIES_REPORT`, and this output can be found in fig. 7.2. Despite the negligible overhead of the hierarchical report, the storage performance remains

Figure 7.2: COUNTDOWN time-series report.

a key factor. The memory footprint is consistent per MPI process, typically within the range of a few megabytes. The flexibility of configuring and deactivating different profiler modalities adds versatility to its usage [35, 36].

7.6 COUNTDOWN Event Alarm Manager

COUNTDOWN interacts with the hardware power controller of each CPU core to analyze and reduce power consumption. Due to limitations in SELinux, the interactions occur through the MSR-SAFE driver, and the Events module determines the appropriate performance level to execute a specific phase [36].

COUNTDOWN employs a timeout strategy using standard Linux timer APIs, which include system calls like `setitimer()` and `getitimer()` for manipulating user-space timers and registering callback functions. The methodology is illustrated in fig. 6.4. When COUNTDOWN reaches an MPI phase where it has the opportunity to conserve energy by transitioning to a C-States, it sets up a timer callback in the prologue function named “event start“. Afterwards, the execution continues with the usual workflow of the MPI phase. When the timer runs out, a system signal is produced, causing the ”normal” execution of the MPI code to be interrupted. The signal handler then activates the COUNTDOWN callback, and once the callback finishes, the execution of the MPI code resumes from the point where it was interrupted. If the execution of the MPI phase returns to COUNTDOWN (indicating its termination) before the timer expires, COUNTDOWN will disable the timer in the epilogue function, and the execution will continue as if nothing had happened. The callback can be configured to enter a lower C-States (12.5 % of the load), referred to as COUNTDOWN THROTTLING, or a lower P-States (1.2 GHz), referred to as COUNTDOWN DVFS [36], the second is the default strategy and the THROTTLING strategy is not recommended by the authors of the COUNTDOWN.

The `eam_callback()` function plays a crucial role in the Event Alarm Manager (EAM) by responding to triggered events. It sets a flag (`flag_eam`) to indicate an EAM

event and adjusts the system's minimum processor state (P-states) through the invocation of `set_min_pstate()` of `pm.c`. This function is essential for coordinating actions in response to EAM events and optimizing processor states for improved efficiency.

In the case of `eam_start_mpi()`, it acts as the initiator for Message Passing Interface (MPI) processes within COUNTDOWN. It ensures the reset of `flag_eam` to FALSE, denoting the absence of an EAM event. Depending on the EAM timeout setting, the function either initiates a timer using `start_timer()` of `timer.c` or directly calls `eam_callback()`. This orchestration is crucial for efficiently managing MPI-related events within the COUNTDOWN application.

For `eam_end_mpi()`, its primary responsibility lies in concluding MPI-related processes. The function checks the EAM timeout and resets the timer if applicable. If the `flag_eam` is set (indicating timer expiration), it sets the maximum processor state (P-states) through `set_max_pstate()` of `pm.c` and returns TRUE. Otherwise, it returns FALSE. This function is pivotal for handling MPI events and adjusting processor states accordingly.

During initialization, `eam_init()` sets up the Event Alarm Manager (EAM) by initializing the timer. If the EAM timeout is greater than zero, it calls `init_timer()` with `eam_callback` as the callback function. This function is a crucial part of the EAM setup during the COUNTDOWN application's initialization.

Lastly, `eam_finalize()` is responsible for concluding the operations of the Event Alarm Manager (EAM). If the EAM timeout is greater than zero, it calls `finalize_timer()` of `timer.c` to reset the timer and set the maximum system P-States. This function is executed during the COUNTDOWN application's finalization process.

7.7 COUNTDOWN Timer

As per the title this section of code deals with time management within the COUNTDOWN library. The `start_timer()` function is responsible for initializing and starting a timer. It utilizes the interval structure to define the timer's value, particularly setting the time interval based on the EAM timeout. By invoking `setitimer()` with the `ITIMER_REAL` parameter, the timer is initiated, triggering an alarm after the specified interval. This functionality is crucial for managing time-sensitive operations within COUNTDOWN.

To ensure precise control over timers, the `reset_timer()` function is employed to reset the timer, effectively canceling any ongoing alarm. This capability enhances the adaptability of COUNTDOWN, allowing for dynamic adjustments to timing behavior during runtime. The `finalize_timer()` function encapsulates the process of resetting the timer, contributing to the orderly conclusion of timer-related operations.

The code also introduces advanced timer configuration functions, such as `make_timer()` and `delete_timer()`. The former enables the creation of timers with user-defined intervals and expiration times, offering flexibility in adapting to varying time requirements. Conversely, the latter function facilitates the deletion of a specified timer, providing a mechanism to efficiently manage and release timer resources when they are no longer needed. These advanced timer functionalities extend COUNTDOWN's temporal control capabilities, enabling sophisticated handling of time-sensitive events and operations within the application.

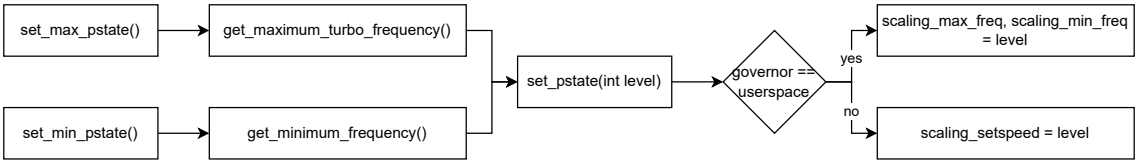


Figure 7.3: Graphical representation of power management logic of COUNTDOWN.

7.8 COUNTDOWN Power Management and DVFS

COUNTDOWN interacts with the hardware power controller of each core to reduce power consumption using the DVFS states (P-states). This integration is achieved through the Power Management Strategies of the Linux kernel, described in section 5.2, specifically the files `/sys/devices/system/cpu/cpu*/cpufreq/*` exposed by the Linux kernel. When COUNTDOWN needs to modify the CPU's power management, it handles changes in the values present in the `scaling_max_freq`, `scaling_min_freq`, and `scaling_setspeed` registers, depending on the type of governor. The first two are modified for some governors, while the third is modified for others [35, 43].

For a high-power state, both `scaling_max_freq` and `scaling_min_freq`, or `scaling_setspeed` if the governators are in userspace, are set to the value obtained either through an environment variable or, if not present, from `cpuinfo_max_freq`.

For a low-power state, both `scaling_max_freq` and `scaling_min_freq`, or `scaling_setspeed` if the governators are in userspace, are set to the value obtained either through an environment variable or, if not present, from `cpuinfo_min_freq`.

It is crucial to verify the successful writing of values to `scaling_max_freq`, `scaling_min_freq`, and `scaling_setspeed` as these parameters may be subject to rejection by the system. This verification ensures that the desired power management settings are applied effectively, allowing the CPU to operate at the specified clock frequency and bypass any other energy-saving strategy present in the system.

The response time of the hardware (HW) power controller is estimated to be $500\,\mu s$ ([28, 35]). This implies that any new setting for the core frequency applied within a time frame faster than $500\,\mu s$ may be either successfully implemented or completely ignored. The outcome depends on when the register was sampled the previous time.

The DVFS management in COUNTDOWN is encapsulated within `pm.c`, utilizes the functions `get_minimum_frequency()` and `get_maximum_turbo_frequency()` in `pm.c`. Upon examining these functions, it is observed that the former exclusively reads the `cpuinfo_min_freq` register, converts it to an integer, and returns the result divided by 10^{-5}. The second function, based on the presence or absence of the `SCALING_SETSPEED` value (which can either be the user-defined value through environment variables or not), reads the value of `cpuinfo_max_freq` or the user-defined value. Subsequently, it reads the file and returns the integer result divided by 10^{-5}. The algorithm follows the logic illustrated in fig. 7.3.

7.9 COUNTDOWN Slack

The COUNTDOWN Slack is a straightforward mechanism to distinguish between slack time (*Tslack*) and copy time (*Tcopy*) in collective primitives, as illustrated in the upper part of fig. 6.5b. To maintain performance during the *Tcopy* phase, a barrier was introduced using the `MPI_Barrier` function in the MPI collective operation, and a similar artificial barrier was also created for P2P operations. This mechanism is called Collective COUNTDOWN Slack barrier. This barrier effectively prevents any decrease in performance and can be used with different MPI implementations that are based on standard MPI primitives. Its flexibility allows it to be easily integrated with various MPI libraries, making it suitable for a wide range of situations. This mechanism was primarily used with blocking MPI primitives while still maintaining the functionality of non-blocking, one-sided, file, and support MPI primitives.

As already mentioned, the approach also considers both collective and Point-to-Point (P2P) primitives since the time spent in other types of primitives is relatively negligible for the specific benchmarks under consideration. Two distinct mechanisms were implemented to isolate the slack time designed for collective primitives and the other for P2P primitives. These mechanisms are instrumental in optimizing energy efficiency during the MPI communication phases. The artificial barrier for collective communication is described in section 5.3.1.

COUNTDOWN Slack enforces an artificial `MPI_Barrier` on the communicator whenever the application initiates a collective primitive. If the `MPI_Barrier` extends beyond a specified time threshold, COUNTDOWN Slack can reduce the P-States to the minimum level available in the system. As soon as all processes involved in the collective primitive reach this synchronization point, the barrier concludes, and COUNTDOWN Slack restores the highest frequency when profiling the duration of slack time. Following this, the execution flow returns to the application, allowing it to invoke the actual collective primitive. The mechanism for collective barriers functions as expected since it involves all processes in the communicator, ensuring synchronization. However, when it comes to P2P primitives, they are exclusively invoked by the processes engaged in P2P communication.

Inserting `MPI_Barrier` in P2P primitives is not possible since it would lead to a deadlock for subsequent `MPI_Wait`. To address this challenge, COUNTDOWN has implemented a waiting mechanism based on non-blocking primitives, as depicted in the lower part of fig. 6.5b. Before an `MPI_Send` primitive, COUNTDOWN Slack appends an artificial `MPI_Isend` with 0 B content, followed by an `MPI_Wait`. Similarly, before an `MPI_Recv` primitive, COUNTDOWN Slack includes an artificial `MPI_Irecv` with 0 B content, followed by an `MPI_Wait`. However, in the case of `MPI_Isend`, COUNTDOWN Slack only introduces an artificial `MPI_Isend`, and for `MPI_Irecv`, it adds solely an artificial `MPI_Irecv`.

The non-blocking P2P primitive returns a request object, which COUNTDOWN Slack utilizes in the subsequent `MPI_Wait` primitive. `MPI_Wait` is a blocking operation used to await completion of the request object. During artificially introduced `MPI_Wait`, COUNTDOWN Slack reduces the processor's frequency if the `MPI_Wait` duration exceeds a certain threshold. This mechanism enables COUNTDOWN Slack to establish a P2P barrier exclusively among the processes involved in P2P communication, effectively segregating slack from copy time. We refer to this approach as the P2P COUNTDOWN Slack barrier.

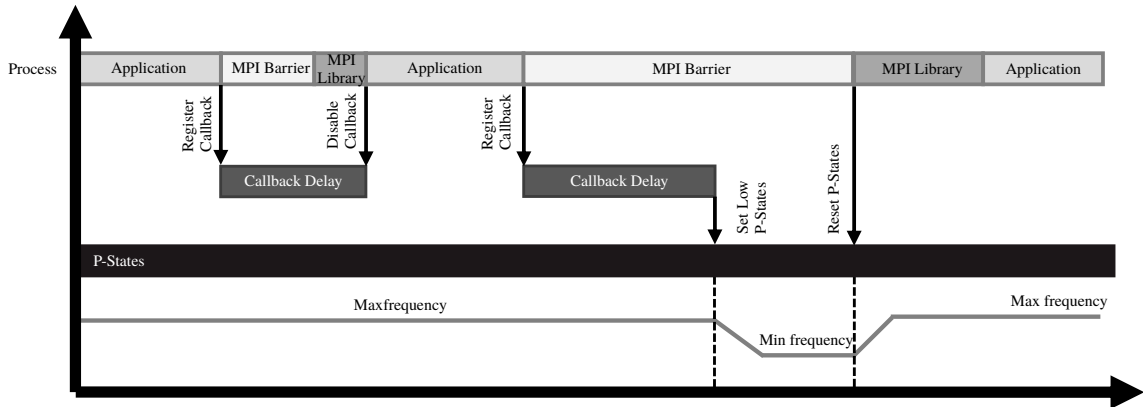


Figure 7.4: Approach used by COUNTDOWN Slack library to save energy during communication times.

Source: Cesarini et al. [35].

When all processes participating in the P2P primitive reach this synchronization point, the artificial barrier is completed and the execution flow reverts to the COUNTDOWN Slack. Subsequently, the library restores the maximum frequency and invokes the application's original P2P primitive.

While COUNTDOWN Slack augments blocking P2P primitives with their non-blocking counterparts, it accommodates applications that utilize a mix of blocking and non-blocking P2P primitives. To maintain consistency and avoid discrepancies in P2P COUNTDOWN Slack barriers, we have included non-blocking P2P primitives before every non-blocking P2P primitive called by the application, ensuring a balanced number of non-blocking P2P primitives.

To assess the overhead introduced by both the collective and P2P COUNTDOWN Slack barriers, we conducted experiments by running our benchmarks both with and without the artificial barrier mechanism, subsequently comparing their execution times. The experimental results demonstrate (see chapter 8) that the overhead is negligible in all our benchmarks [35].

Figure 7.4 describes the approach of COUNTDOWN Slack to split the time spent in communication (*Tcomm*) into slack time (*Tslack*) and copy time (*Tcopy*). COUNTDOWN Slack uses the same logic regarding the management of the separation between *Tslack* and *Tcomm* of Adagio (section 6.6), but only the part of Fermata (section 6.4) is implemented in COUNTDOWN, with the empirical switching threshold set to 500 µs. This was only applied to the slack regions isolated by the collective and P2P COUNTDOWN Slack barrier logic.

7.10 COUNTDOWN Slack Event Alarm Manager

This code segment introduces functionalities to manage slack time within the COUNTDOWN application, expanding on the time-related capabilities discussed above. The code classifies MPI instructions into three main types: waiting operations, collective barriers, and point-to-point communications.

The `eam_slack_callback()` callback function is associated with slack time events. When triggered, it sets a flag (`flag_eam_slack`) and adjusts the minimum processor state (`set_min_pstate()`). The code also includes functions to intelligently detect whether a given MPI instruction is a wait operation (`is_wait_mpi`) or a P2P communication (`is_p2p`). For the latter flag, the mapping function `is_collective_barrier()` is capable of identifying collective barrier types within MPI instructions. Otherwise, this module is similar to the event alarm manager described in section 7.6.

7.11 Overhead of COUNTDOWN

For COUNTDOWN overhead analysis, the COUNTDOWN authors used the profiler module to evaluate the performance impact of running MPI applications without modifying the CPU core frequency. To assess this, tests are performed on a single node using the QE-CP-EU application, which presents a worst-case scenario for COUNTDOWN due to the high number of MPI calls and high granularity to profile. In this setup, the overhead caused by network-related factors in MPI calls is minimal, and communication and synchronization within the chip are much faster compared to communication between chips or nodes. Consequently, the waiting times in MPI calls that can be used for power management are typically shorter [36].

In this run, each process in the diagonalization task utilizes over 1.1 million MPI primitives, resulting in COUNTDOWN profiling an average MPI call every 200 µs for each process. To evaluate the overhead, the authors of COUNTDOWN compare the execution time with and without COUNTDOWN instrumentation. The test is repeated five times and the median case is reported. The results show that, in most scenarios, the overhead introduced by the profiler in the execution is less than 1 % [36].

The same test is repeated with a change in CPU core frequency to assess the overhead of fine-grain Dynamic Voltage and Frequency Scaling (DVFS) control [36]. The experimental results report an overhead of 1.04 % for accessing the DVFS control register and the profiling routines, with COUNTDOWN configured to force the highest P-States in the DVFS to isolate the overhead caused by interaction with power management in Linux, which causes an overhead [36].

7.12 Evaluation of COUNTDOWN

In the evaluation of the COUNTDOWN made by COUNTDOWNS authors begins with a single compute node and then extends to a real HPC system, specifically an IBM NeXtScale cluster, which is recognized in the Top500 supercomputer list. The HPC system is equipped with two Intel Broadwell E5-2697 v4 CPUs, each having 18 cores at a 2.3 GHz nominal clock speed and a 145 W TDP. Nodes are interconnected using an Intel QDR (40 Gbit s^{-1}) InfiniBand high performance network [36].

Three sets of applications are used for benchmarking in the target HPC system:

NAS Parallel Benchmark Suite This suite comprises seven benchmarks with various mathematical workloads, including FFT, differential equations, and ordering. It runs on 29 compute nodes with a total of 1024 cores [45].

OMEN OMEN is a computational tool that employs atomistic quantum transport simulation to calculate the ab initio I-V characteristics of nano-devices. It has been specifically optimized for supercomputers and was awarded the ACM Gordon Bell Prize in 2019.

QuantumESPRESSO (QE) QE is configured for complex large-scale simulations and includes iterative steps involving linear algebra and FFT. This benchmark runs on 96 compute nodes with 3456 cores.

The COUNTDOWN authors compare the performance with and without the COUNTDOWN methodology on the same nodes. The results show that COUNTDOWN significantly reduces energy consumption while incurring only a small time-to-solution overhead, typically below $5\,\%$. Specifically, COUNTDOWN Slack reduce energy consumption up to $20\,\%$ [36].

8

Experiments

This chapter presents a detailed exploration of ten different experiments carried out on the Cineca supercomputers. Each experiment is designed to provide information on the functionality, efficiency and impact of the COUNTDOWN tool in conjunction with the Chronos algebraic solver described in chapter 3.

Before going into the detailed description of the experiments, it is essential to familiarise with the Galileo100 architecture. Understanding the architecture of the Cineca supercomputer provides a fundamental background to the experiments that follow. After exploring the Galileo100, we turn our attention to the matrices used in these experiments. An in-depth study of these matrices is crucial as they form the basis of the computational scenarios in which the COUNTDOWN tool will be evaluated. Finally, we will illustrate the graphical representations used throughout this chapter. Particular emphasis will be placed on the interpretation of box plots and event plots. These graphical tools serve as indispensable aids in visually conveying the results and nuances of each experiment.

The organization of the chapter is as follows: section 8.1 introduces the architecture of the machine I ran COUNTDOWN and Chronos on, followed by section 8.2 which tells what matrices were used for these tests. After that, section 8.3 talks about the experiments that will be performed, which are described in the next sections (sections from 8.4 to 8.13). Finally, section 8.15 draws preliminary conclusions about the experiments.

8.1 Galileo100 Architecture

The experiments were carried out on Cineca's Galileo100 supercomputer, a powerful computing cluster equipped with 2nd Generation Intel® Xeon® Platinum 8260 Processors, codenamed Cascade Lake. Each node has 2 CPUs with 24 cores and 48 threads, for a total of 48 cores and 192 threads, and a maximum turbo frequency of $3.90\,\text{GHz}$, with $768\,\text{GB}$ of RAM. The architecture is based on Linux Infiniband cluster technology and has 636 nodes, including 10 login nodes. Each CPU has a TDP of $165\,\text{W}$ and each node has a TDP of $330\,\text{W}$.

These servers are interconnected through a high-speed $100\,\text{Gbit}\,\text{s}^{-1}$ Ethernet network. The storage infrastructure is extensive, with $20\,\text{PB}$ accessible from both the cloud and the nodes. In particular, dedicated $1\,\text{PB}$ fast storage with full NVMe/SSD support meets cloud storage requirements, while $720\,\text{PB}$ fast storage using the IME DDN solution ensures

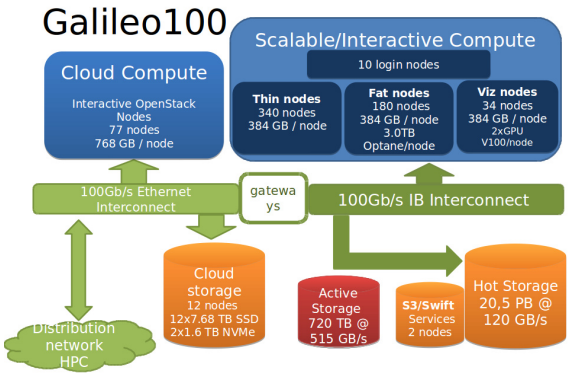


Figure 8.1: System Architecture of Galileo100 supercomputer.

Source: Besker and Bari [64].

fast data access. This network architecture enables a maximum bandwidth of $100\,\text{Gbit}\,\text{s}^{-1}$ between each pair of nodes, ensuring seamless communication.

Some Galileo100 nodes are equipped with NVIDIA V100 PCIe3 GPUs, with 32 GB RAM, integrated into 36 visualization nodes (viz). This GPU configuration enhances the computing capabilities and IT complements the significant memory resources. Standard nodes, known as "thin nodes", are equipped with 480 GB SSD storage, while data processing nodes, known as "fat nodes", have a significant 2 TB SSD, augmented by 3 TB Intel Optane memory. This versatile memory architecture enables efficient handling of a wide range of computing tasks, such as CPU-intensive and mixed calculations, to be handled efficiently. The Galileo100 architecture is also described in fig. 8.1, and this supercomputer provides peak performance of around $2\,\text{PFlop}\,\text{s}^{-1}$ [63, 64].

8.2 Test Matrices

We use a collection of sparse matrices [65, 66] to run Chronos and analyze the performance evaluation.

The first matrix, *agg14m.bin*, is derived from the 3D mechanical equilibrium of a loosely constrained symmetrical machine cutter. The domain has dimensions of $50 \times 50 \times 50\text{mm}^3$ and contains 2644 spherical polystyrene inclusions. The cement matrix has properties $(E1, \nu_1) = (25\,000\,\text{MPa}, 0.30)$, while the polystyrene inclusions have properties $(E2, \nu_2) = (5\,\text{MPa}, 0.30)$. As a result, there is a significant contrast in Young's modulus between these two linear elastic materials. The discretization is performed using tetrahedral elements in the finite element method.

The matrix *Cubo_1772481.Ext_bin* represents a regular discretization of an elastic cube with tetrahedral finite elements and is used for linear elasticity testing. It is commonly used to perform linear elasticity tests and it provides researchers and engineers with a versatile platform for analyzing structural behavior under mechanical loading and deformation [67].

The matrices *Wing_4538k.csr.Ext_bin* and *Wing_BIG_BC.csr.Ext_bin* are derived from an airplane wing problem, modeling the structural properties of the wing in an aircraft [66].

Table 8.1: Collection of sparse matrix in analysis.

Matrix	Size [GB]	Row Number	Non-zero Terms	Non-zeros/Row
agg14m.bin	15.0	14 106 408	633 142 730	44.88
Cubo_1772481.Ext_bin	5.0	5 317 443	222 615 369	41.86
Wing_4538k.csr.Ext_bin	4.2	4 538 205	187 714 431	41.36
Wing_BIG_BC.csr.Ext_bin	62.0	33 654 357	2 758 580 899	81.97
M20.bin	71.0	20 056 050	1 634 926 088	81.52

Finally, the matrix *M20.bin* is derived from the 3D mechanical equilibrium of a loosely constrained symmetrical machine cutter. The unstructured mesh is composed of 4 577 974 second-order tetrahedra and 6 713 144 vertices, resulting in 20 056 050 DOFs. The material is linear elastic with $(E, \nu) = (108MPa, 0.33)$ [65]. Table 8.1 provides a summary of key details for each matrix, including size, number of rows, number of non-zero terms, and non-zero terms per row.

Initially, all the matrices were used in the different experiments, the matrices *M20.bin* and *Wing_BIG_BC.csr.Ext_bin* caused memory management problems, requiring an amount of GB that could not be allocated in the Cineca machines, and very high execution times.

The matrix *agg14m.bin* was used until experiment 2, when it showed scalability problems with a number of cores, so that the execution could not be finished in 24 hours, to avoid problems we preferred to use the other two matrices, which have similar exit times.

8.3 Overview of the Experiments

The first experiment, labeled Experiment 0 (section 8.4), serves as a basic verification process. Its primary objectives are to confirm the accuracy of the data generated by the COUNTDOWN tool. It also aims to validate the information obtained by reverse engineering as described in chapter 7. Experiment 1 (section 8.5) shifts the focus to verifying the seamless operation of COUNTDOWN alongside the algebraic solver Chronos. This experiment evaluates the performance of COUNTDOWN when used with Chronos, as described in chapter 7.

Experiment 2 (section 8.6) involves the search for optimal points for more in-depth COUNTDOWN tests. Experiment 2 uses a configuration with a lower number of iterations to reduce the influence of the iterative part, which computes the matrix solution, with respect to the setup and preconditioning part. In this way, about half of the runtime is used by the setup and preconditioning and the remaining time is utilized by the iterative computation, as opposed to the normal unbalanced distribution, which is described in section 3.5. This choice also allows for more samples at the same cost.

Experiments 3 (section 8.7), 4 (section 8.8), and 5 (section 8.9) analyze the performance of COUNTDOWN applied to the optimal points, which have been identified in Experiment 2. Specifically, Experiment 3 analyzes the point with the best energy-time ratio, Experiment 4 analyzes the fastest point, and finally Experiment 5 analyzes whether the value of the COUNTDOWN callback delay, namely 500 µs, is the ideal one.

Experiment 6 (section 8.10), on the other hand, selects a few runs with a full iteration configuration, i.e. with a sufficient number of iterations for the problem to converge.

Experiments 7 and 8 (sections 8.11.1 and 8.11.2), investigate the effects of lowering

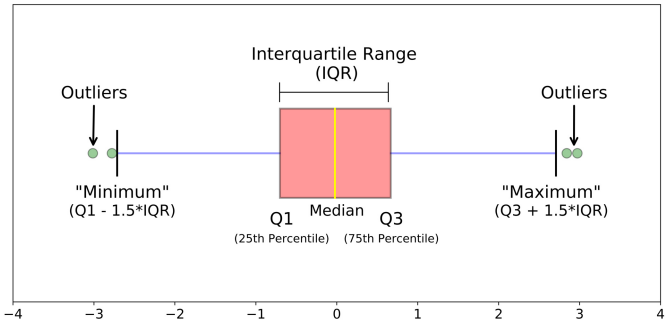


Figure 8.2: Different parts of a boxplot.

Source: Galarnyk [68].

the processor frequency in the configuration with the best energy/time ratio. In particular, Experiment 7 tests the configuration with a reduced number of iterations, and Experiment 8 attempts to converge the problem with the correct number of iterations.

Experiments 9, 10, and 11 (sections from 8.12 to 8.14) investigate the actual duration of MPI calls and analyzing their absolute relative frequency, as well as whether the Linux kernel actually accepts processor downlock requests, and propose a small modification to COUNTDOWN to verify some of the considerations made in the previous experiments.

With the exception of Experiments 1 and 2, all the other experiments change a single parameter, so we can be sure that we are framing the phenomenon well. The results are presented in various forms, the most common of which are Box Plots, a graph showing the median (50$^{\text{th}}$ percentile), first and third quantiles ($Q1$ or 25$^{\text{th}}$ percentile and $Q3$ or 75$^{\text{th}}$ percentile). The interquartile range (IQR) is defined as the range of values between the $Q1$ and $Q3$ percentiles, this value also defines the minimum and maximum, through the formula $Q1 - 1.5 \cdot \text{IQR}$ and $Q3 + 1.5 \cdot \text{IQR}$. Finally, the outliers are the values outside the maximum and minimum. The boxplot box represents the first and third quantiles and thus the IQR, and the median line represents the median. The whiskers represent the maximum and minimum values, and the scattered dots represent outliers. Figure 8.2 displays the components of the boxplot just described.

Another graph used is the Event Plot, which is useful for analysing the variance of runs, a value that the Box Plot only provides through quantile data. Bar, column, castesian plot and scatter plots are used to display the remaining data. In column and cartesian plots, pink is the maximum value, blue is the average value and orange is the minimum value.

8.4 Experiment 0

The initial experiment serves as a crucial validation step, with the aim of confirming the reliability, applicability, and precision of the data provided by COUNTDOWN.

To accomplish this, we devised a stress test suite, which comprises six distinct programs, with their source code referenced in [69]. Here's a brief overview of each program:

`timer` This iterative method keeps the processor under stress by running a timer until a specified duration elapses, primarily stressing the CPU.

Table 8.2: COUNTDOWN consistency stress test results.

Program	EXE Time [s]	Energy PKG [J]	Power PKG [W]	IPC
Intel-defined, high-complexity workload (1 node)			330	
Intel-defined, high-complexity workload (2 node)			660	
`timer` (1 node)	60.200	8859	147.16	1.25
`timer` (2 node)	60.733	18 480	304.29	1.25
`eratostene` (1 node)	60.572	9455	156.09	0.78
`eratostene` (2 node)	60.377	19 444	322.04	0.77
`pi` (1 node)	60.475	9065	149.90	1.42
`pi` (2 node)	60.252	18 625	309.13	1.42
`matrix` (1 node, int)	72.807	11 124	152.78	1.45
`matrix` (2 node, int)	73.595	23 248	315.89	1.49
`matrix` (1 node, double)	73.884	22 141	299.68	1.01
`matrix` (2 node, double)	73.178	41 051	560.98	0.99
`sse` (1 node)	60.706	18 574	305.97	1.49
`sse` (2 node)	60.098	37 105	617.41	0.68
`avx` (1 node)	60.014	18 297	304.88	1.49
`avx` (2 node)	60.315	38 481	638.02	1.31

`eratostene` Similar to `timer`, this program involves multiplication operations and numerous memory accesses, thereby also stressing the RAM.

`pi` Utilizes the Gregory-Leibniz series to compute the decimal digits of pi, making use of the FPU within the CPU, which is known to consume significant resources.

`matrix` Performs various operations such as sums and products between matrices, supporting different data types like double, float, integer, and long. For simplicity, we utilized the double data type in this scenario.

`sse` A modified version of [70], this program executes calculations using SSE primitives.

`avx` Another modified version of [70], this program performs calculations using AVX2 primitives.

The "Intel-defined, high-complexity workload" program represents the workload defined by Intel for conducting its own power consumption and TDP calculations [63]. For Intel's processor architecture, we anticipate that maximum power consumption occurs exclusively with SSE and AVX instructions, with consumption decreasing as instruction complexity decreases. In other words, simpler integer instructions should consume the same amount or less power compared to complex double instructions, as depicted in table 8.2.

8.5 Experiment 1

The first experiment devised involved a comprehensive exploration of all four potential scenarios offered by the COUNTDOWN feature. These scenarios included:

COUNTDOWN with Analysis only (Baseline) COUNTDOWN executed in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario served as a baseline to examine the basic functionality of COUNTDOWN.

COUNTDOWN with slack Analysis (Analysis) COUNTDOWN was run with simultaneous slack analysis and without implementing any power saving algorithms. The purpose of this scenario was to evaluate the impact of COUNTDOWN when coupled with slack analysis, but without power optimization.

COUNTDOWN without slack (CNTD) Execution of the energy saving algorithm without the inclusion of the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization.

COUNTDOWN with slack (CNTD SLACK) Execution of the energy saving algorithm along with optimization of slack times. This scenario investigated the combined impact of energy optimization and slack time optimization through the COUNTDOWN mechanism.

By systematically exploring each of these scenarios, the experiment aimed to provide a nuanced understanding of how COUNTDOWN operates under various configurations, shedding light on its potential benefits and limitations in different operational contexts. The configurations in this experiment involve a variable number of nodes, 2 MPI Tasks, and 24 for OpenMP Threads, to obtain the number 48, as recommended in Besker and Bari [64].

In the following paragraphs, we present a summary of the main results noted in Experiment 1 and expressed numerically in the tables from 8.3 to 8.5. A description of the matrices is presented in section 8.2. Tables reporting the numerical results of Experiment 1 contain the following data:

Node and run config Number of nodes, task and openMP threads (section 4.2) and configuration used, the acronyms are as listed above.

Solver normRES Accuracy of the Chronos result.

Solver ITER Number of iterations to reach the solution with the required accuracy.

EXE time Execution time, calculated by COUNTDOWN (section 7.3).

MAX Memory usage Maximum memory (RAM) usage, calculated by COUNTDOWN profiler (section 7.3).

AVG IPC Average number of instructions per clock executed by the program, calculated by COUNTDOWN (section 7.3).

AVG CPU frequency Average number of CPU clocks during the runtime of the program, calculated by COUNTDOWN (section 7.3).

MPI Network - TOT Total MB exchanged between nodes, task from MPI, calculated by COUNTDOWN (section 7.3).

Energy Total energy consumed by program execution, calculated by COUNTDOWN (section 7.3).

AVG Power Average power consumed by the program, calculated by COUNTDOWN (section 7.3).

The tables from 8.3 to 8.5 also shows a percentage comparison to the base case for all runs with COUNTDOWN. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

agg14m.bin The first matrix examined is *agg14m.bin*, boasting a sizable dimension of 15 GB and containing 14 106 408 rows. Key data points are succinctly presented in table 8.3. Upon examination of the three explored configurations of this matrix, it becomes evident that energy savings are realized exclusively in the $8 \times 2 \times 24$ configuration, resulting in a total energy savings of 6 %. However, it should be noted that despite the nearly 11 % reduction in execution time, the average power has increased by 5 %. Despite this increase in power, there remains a notable efficiency gain as a result of the substantial reduction in both time and energy consumption.

Furthermore, it is intriguing to observe that the COUNTDOWN algorithm plays a pivotal role in indirectly enhancing synchronization among the various nodes. This improvement in synchronization is believed to be a contributing factor to the observed time savings. It should be noted that while the $8 \times 2 \times 24$ configuration demonstrates commendable efficiency gains, the intricate interaction between time, energy, and power factors requires a nuanced consideration to optimize overall system performance.

Cubo_1772481.Ext_bin The second matrix subjected to analysis is the *Cubo_1772481.Ext_bin* matrix, presenting a size of 4.2 GB and containing 5 317 443 rows. The key data points are succinctly presented in table 8.4. In this particular case, notable observations emerge, especially in the $1 \times 2 \times 24$ configuration, where energy savings reach impressive 8 %. This is attributed to the COUNTDOWN mechanism, which not only reduces execution times but also contributes to a 4 % savings in power. Once again, the improved synchronization among nodes facilitated by COUNTDOWN is identified as a key factor. However, it is intriguing to note that in the other two configurations, energy savings drop to 6 % and 2 %, respectively. This variance could be linked to the nuanced interplay between thread synchronization and overall system dynamics.

The disappearance of the time reduction phenomenon in the $16 \times 2 \times 24$ configuration is particular interesting. This occurrence is hypothesized to be a result of the increased number of threads that need to be synchronized, leading to a more complex synchronization process. This observation underscores the intricate relationship between thread count, synchronization, and the effectiveness of the COUNTDOWN algorithm.

Wing_4538k.csr.Ext_bin The third matrix subjected to analysis is *Wing_4538k.csr.Ext_bin* matrix, featuring a considerable size of 5.0 GB and containing 4 538 205 rows. The essential data points are summarized concisely in table 8.5.

In contrast to the previous matrices, this matrix does not exhibit improvements in execution times attributable to enhanced synchronization via COUNTDOWN. However, a discernible range of energy savings, ranging from 6 % to 2 %, is observed in the two configurations. In particular, power consumption experiences a 7 % increase in one configuration and a 2 % decrease in the other.

This distinctive behavior underscores the matrix-specific impact of COUNTDOWN on the measured metrics. Although the synchronization-related time gains may not be evident, the energy and power dynamics show variability.

Table 8.3: Average results of Experiment 1 for the matrix *agg14m.bin*. Three blocks of runs were conducted, each comprising three different configurations. For each configuration, all four COUNTDOWN operating modes were explored.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Node and run config	**Solver norm-RES**	**Solver ITER**	**EXE time [s]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**MPI network - TOT [MB]**	**Energy [J]**	**AVG Power [W]**
$4 \times 2 \times 24$, baseline	9.66×10^{-10}	904	1591.1	52.4	1.9	3100	1.6	930629	584.9
$4 \times 2 \times 24$, analysis	9.66×10^{-10}	904	1591.3 0 %	52.4	1.9	3100	1.6	930722 0 %	584.9 0 %
$4 \times 2 \times 24$, CNTD	9.64×10^{-10}	902	1569.7 −1 %	52.4	1.8	3099	1.6	934281 0 %	595.2 −2 %
$4 \times 2 \times 24$, CNTD SLACK	9.94×10^{-10}	901	1585.5 0 %	52.4	1.9	3096	1.6	925031 1 %	583.4 0 %
$8 \times 2 \times 24$, baseline	9.71×10^{-9}	831.5	96.6	52	1.8	3099	7.3	105703	1094.3
$8 \times 2 \times 24$, analysis	9.93×10^{-9}	831	98.6 2 %	52	1.7	3099	7.2	99359 6 %	1008.0 8 %
$8 \times 2 \times 24$, CNTD	9.71×10^{-9}	831.5	86.2 −11 %	52	1.7	3099	7.3	99411 6 %	1153.2 −5 %
$8 \times 2 \times 24$, CNTD SLACK	9.73×10^{-9}	831.5	87.1 −11 %	52	1.7	3099	7.3	100054 6 %	1148.9 −5 %
$2 \times 2 \times 24$, baseline	9.80×10^{-10}	901.2	121.7	52.4	1.6	2985	20.3	80152	659.0
$2 \times 2 \times 24$, analysis	9.80×10^{-10}	901.3	121.7 0 %	52.4	1.6	3002	20.3	81755 −2 %	671.6 −2 %
$2 \times 2 \times 24$, CNTD	9.82×10^{-10}	902.8	122.0 0 %	52.4	1.6	3001	20.3	80232 0 %	658.0 0 %
$2 \times 2 \times 24$, CNTD SLACK	9.80×10^{-10}	901.0	121.7 0 %	52.4	1.7	2994	20.3	78299 2 %	643.3 2 %

Table 8.4: Average results of Experiment 1 for the matrix *Cubo_1772481.Ext_bin*. Three blocks of runs were conducted, each comprising three different configurations. For each configuration, all four COUNTDOWN operating modes were explored.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Node config and Run config	**Solver norm-RES**	**Solver ITER**	**EXE time [s]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**MPI network - TOT [MB]**	**Energy [J]**	**AVG Power [W]**
$1 \times 2 \times 24$, baseline	0.2	1151	131	18	1	3100	410	38151	292
$1 \times 2 \times 24$, analysis	0.2	1151	129 2 %	18	1	3100	410	35297 −7 %	274 −6 %
$1 \times 2 \times 24$, CNTD	0.2	1151	126 4 %	18	1	3099	410	35169 −8 %	279 −4 %
$1 \times 2 \times 24$, CNTD SLACK	0.2	1151	126 −3 %	18	1	3100	410	35227 8 %	279 4 %
$8 \times 2 \times 24$, baseline	9.71×10^{-9}	6273	109	19	2	3097	29835	139602	1269
$8 \times 2 \times 24$, analysis	9.77×10^{-9}	6273	109 0 %	19	1	3098	29580	130507 −7 %	1162 −8 %
$8 \times 2 \times 24$, CNTD	9.93×10^{-9}	6275	108 1 %	19	1	3096	29590	129624 −7 %	1168 −8 %
$8 \times 2 \times 24$, CNTD SLACK	9.61×10^{-9}	6397	110 0 %	19	1	3096	30065	131840 6 %	1168 9 %
$16 \times 2 \times 24$, baseline	6.92×10^{-9}	2203	16	28	2	3093	320350	68731	4256
$8 \times 2 \times 24$, analysis	6.94×10^{-9}	2203	15 6 %	28	2	3095	320370	65752 −4 %	4404 3 %
$8 \times 2 \times 24$, CNTD	6.71×10^{-9}	2204	15 6 %	28	2	3095	320410	65987 −4 %	4411 4 %
$8 \times 2 \times 24$,CNTD SLACK	6.86×10^{-9}	2203	15 −6 %	28	2	3012	320350	67046 2 %	4396 −3 %

Table 8.5: Average results of Experiment 1 for the matrix *Wing_4538k.csr.Ext_bin*. Two blocks of runs were conducted, each comprising three different configurations. For each configuration, all four COUNTDOWN operating modes were explored.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Node and Run config	**Solver norm-RES**	**Solver ITER**	**EXE time [s]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**MPI network - TOT [MB]**	**Energy [J]**	**AVG Power [W]**
$4 \times 2 \times 24$, baseline	9.31×10^{-9}	833	81.7	51.9	1.6	3097	10.3	97939	1033.0
$4 \times 2 \times 24$, analysis	9.50×10^{-9}	832	89.9 10 %	51.9	1.8	3099	8.0	96393 2 %	1071.8 -4 %
$4 \times 2 \times 24$, CNTD	9.54×10^{-9}	832	83.5 2 %	51.9	1.7	3099	8.0	92453 6 %	1107.5 -7 %
$4 \times 2 \times 24$, CNTD SLACK	9.90×10^{-9}	831	83.1 +2 %	51.9	1.7	3099	7.2	92248 6 %	1109.9 -7 %
$8 \times 2 \times 24$, baseline	9.91×10^{-9}	10486	174.2	17.3	1.2	3098	46.8	218320.5	1253.4
$8 \times 2 \times 24$, analysis	9.82×10^{-9}	10623	175.7 1 %	17.3	1.2	3097	47.3	220555.5 -1 %	1255.5 0 %
$8 \times 2 \times 24$, CNTD	9.92×10^{-9}	10525	173.8 0 %	17.3	1.2	3098	46.9	218267 0 %	1255.8 0 %
$8 \times 2 \times 24$, CNTD SLACK	9.93×10^{-9}	10369	175.2 +1 %	18.0	1.2	3098	46.3	214337 2 %	1223.5 2 %

8.6 Experiment 2

The second experiment involves an exhaustive analysis of node configurations ranging from 1 to 32, which is the maximum number of cores that can be used on Cineca G100 with a type C project. The main goal is to find the most environmentally sustainable node configuration, referred to as the **greenest point**. We identify the greenest point as the point where the ratio of $\frac{\text{N. Core} \times \text{EXE Time}}{\text{Energy}}$ is minimized. This concept represents an optimal state where the balance between energy efficiency and time effectiveness is achieved, with the goal of reducing environmental impact and resource consumption.

In addition, the experiment aims to determine the optimal performance point, called the **fastest point**. This refers to the configuration in which the execution time is equal to or less than other configurations with a different number of nodes. It is well known that programs have a scalability limit, where they stop scaling after a certain number of nodes, an example of which can be seen in the graph in fig. 8.3.

In this experiment, we only change the core value and it was decided not to allocate threads exclusively to OpenMP, as realized in the previous experiment section 8.5, in order to achieve a more balanced distribution. In this and the next Experiment, the threads were divided between OpenMP and MPI node threads, with a split of 8 and 6 threads respectively, to get the number 48, as recommended in Besker and Bari [64]. All runs are performed with this configuration:

COUNTDOWN with Analysis only (Baseline) COUNTDOWN executed in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario served as a baseline to examine the basic functionality of COUNTDOWN.

In the following paragraphs, we present a summary of the main results noted in Experiment 2 and expressed numerically in the figs. from 8.4 to 8.6. A description of the matrices is in section 8.2. Tables containing the numerical results of Experiment 2 contain the following data:

Node and run config Number of nodes, task and openMP threads (section 4.2) and configuration used, the acronyms are as listed above.

EXE time Execution time, calculated by COUNTDOWN (section 7.3).

APP time Total computation time of the application, understood as the sum of the various *Tcomp* of each node and rank, calculated by COUNTDOWN.

MPI time Total communication time of the application, understood as the sum of the different *Tcomm* of each single node and rank, calculated by COUNTDOWN.

TOT time Total time of the application, understood as the sum of the various *Tcomp* and *Tcomm* of each single node and rank, calculated by COUNTDOWN.

Energy PKG Total energy consumed by the CPU, calculated by COUNTDOWN.

Energy DRAM Total energy consumed by the RAM, calculated by COUNTDOWN.

AVG Power PKG Average power consumed by the CPU, calculated by COUNTDOWN.

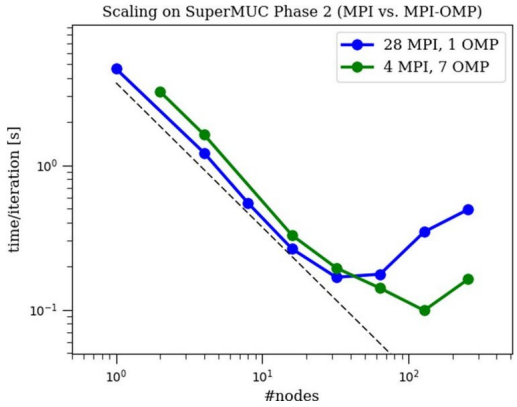


Figure 8.3: Scalability plot for the ECHO-3DHPC software in two OpenMP configurations.

Source: Bugli et al. [71].

AVG Power DRAM Average power consumed by the RAM, calculated by COUNTDOWN.

AVG CPU frequency Average number of CPU clocks during the runtime of the program, calculated by COUNTDOWN.

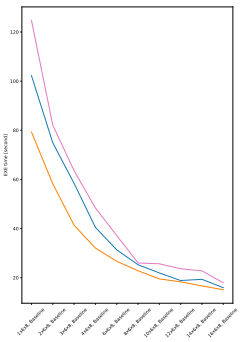
Energy Total energy consumed by program execution, calculated by COUNTDOWN.

AVG Power Average power consumed by the program, calculated by COUNTDOWN.

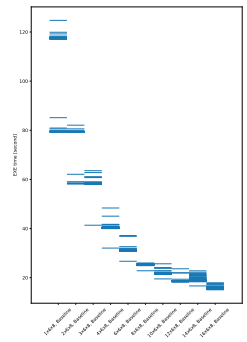
The figs. from 8.4 to 8.6 also shows a percentage comparison to the base case for all runs with COUNTDOWN. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

agg14m.bin The results of the experiments carried out with the matrix *agg14m.bin* are visually presented in fig. 8.4. From figs. 8.4a and 8.4e the **greenest point** is identified, which is located at $4 \times 6 \times 8$. Also, the **fastest point** is located at $16 \times 6 \times 8$. It should be noted that the execution of the $32 \times 6 \times 8$ configuration failed to complete within acceptable time limits, underscoring the intricacies and challenges associated with this particular setup. A configuration with more than 16 nodes increases the execution time exponentially.

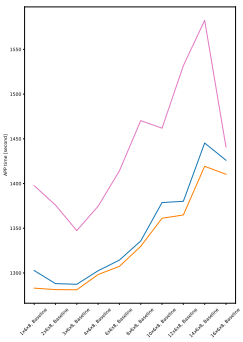
Cubo_1772481.Ext_bin The results of the experiments performed with the matrix *Cubo_1772481.Ext_bin* are visually presented in fig. 8.5. The most noteworthy data is graphically presented in fig. 8.5l, while additional insights can be derived from figs. 8.5a and 8.5e. In this context, the table 8.7 indicates that the **greenest point** is located at $8 \times 6 \times 8$, as shown in the graphs. At the same time, the **fastest point** is located at $32 \times 6 \times 8$. With 32 usable nodes as the maximum limit, it is not possible to examine how it behaves with more than 32 nodes, but it is plausible that configurations beyond 32 nodes may yield improved execution times, albeit at the expense of increased energy consumption.



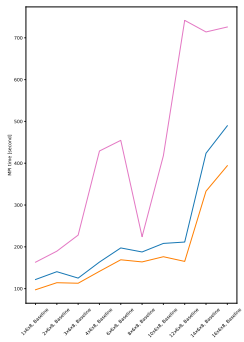
(a) EXE time: variations as threads increase.



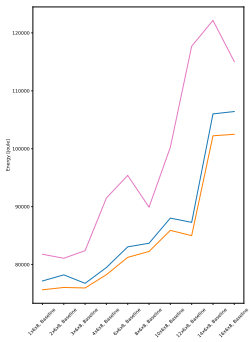
(b) EXE time: distribution of execution times for the 11 runs per instance.



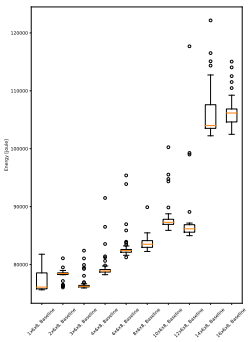
(c) APP time: distribution of execution times for the 11 runs per instance.



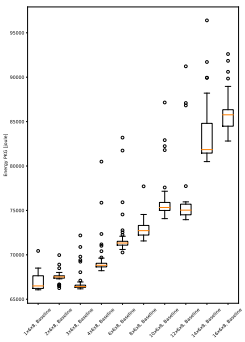
(d) MPI time: distribution of execution times for the 11 runs per instance.



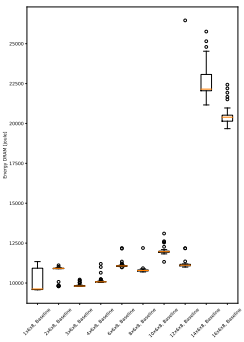
(e) Energy: variations as threads increase.



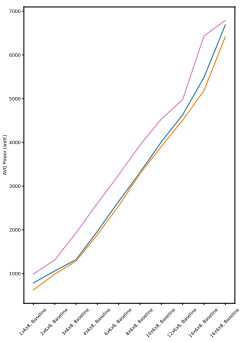
(f) Energy: boxplot of distribution as threads increase.



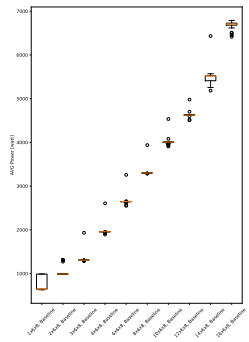
(g) Energy: boxplot of distribution of the energy consumed by CPU.



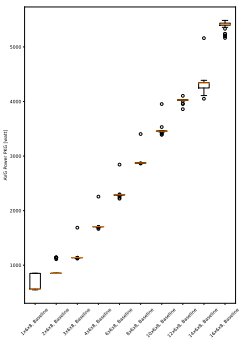
(h) Energy: boxplot of distribution of the energy consumed by RAM.



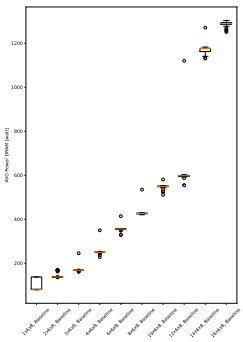
(i) Power: variations as threads increase.



(j) Power: boxplot of distribution as threads increase.



(k) Power: boxplot of distribution of the power consumed by CPU.



(l) Power: boxplot of distribution of the power consumed by RAM.

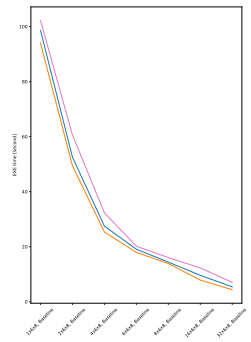
Figure 8.4: Experimental Results for Experiment 2 on the *agg14m.bin* matrix.

Run config	EXE time [s]	Energy [J]	AVG Power [W]	Ratio [(core/W)$^{-1}$]
$1 \times 6 \times 8$	117.3	76 104	645.95	0.001 54
$2 \times 6 \times 8$	79.3	78 445	989	0.002 02
$3 \times 6 \times 8$	58.3	76 242	1307	0.002 29
$4 \times 6 \times 8$	40.3	78 890	1956	0.002 04
$6 \times 6 \times 8$	31.2	82 402	2645	0.002 27
$8 \times 6 \times 8$	25.3	83 509	3300	0.002 42
$10 \times 6 \times 8$	21.8	87 295	4007	0.002 49
$12 \times 6 \times 8$	18.6	86 165	4623	0.002 59
$14 \times 6 \times 8$	18.9	103 990	5516	0.002 54
$16 \times 6 \times 8$	15.8	106 164	6705	0.002 39

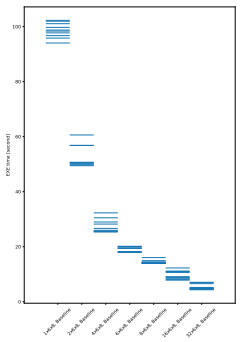
Table 8.6: Median values for Experiment 2 on the *agg14m.bin* matrix.

Run config	EXE time [s]	Energy [J]	AVG Power [W]	Ratio [(core/W)$^{-1}$]
$1 \times 6 \times 8$	98.5	31 241	318	0.003 16
$3 \times 6 \times 8$	50.6	34 550	650	0.003 04
$4 \times 6 \times 8$	26.3	34 953	1330	0.003 01
$6 \times 6 \times 8$	19.4	39 009	2009	0.002 99
$8 \times 6 \times 8$	14.2	38 031	2679	0.002 99
$16 \times 6 \times 8$	9.0	44 398	4941	0.003 24
$32 \times 6 \times 8$	4.9	48 022	9824	0.003 26

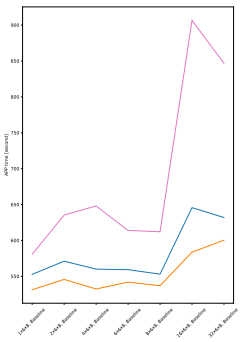
Table 8.7: Median values for Experiment 2 on the *Cubo_1772481.Ext_bin* matrix.



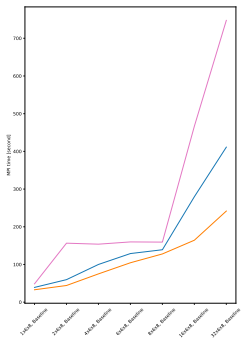
(a) EXE time: variations as threads increase.



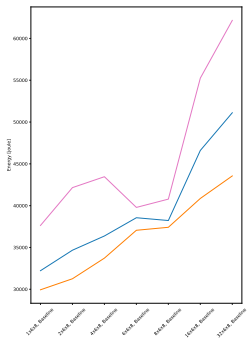
(b) EXE time: distribution of execution times for the 11 runs per instance.



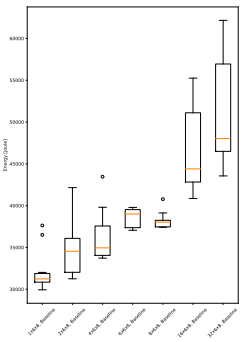
(c) APP time: distribution of execution times for the 11 runs per instance.



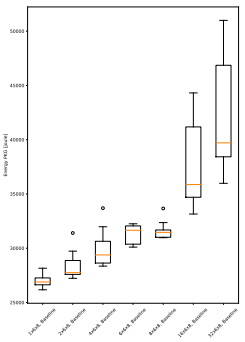
(d) MPI time: distribution of execution times for the 11 runs per instance.



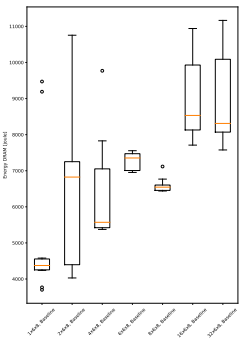
(e) Energy: variations as threads increase.



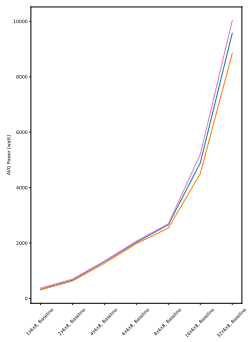
(f) Energy: boxplot of distribution as threads increase.



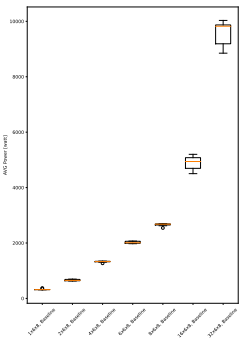
(g) Energy: boxplot of distribution of the energy consumed by CPU.



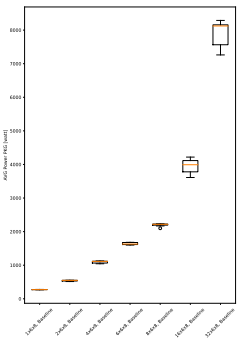
(h) Energy: boxplot of distribution of the energy consumed by RAM.



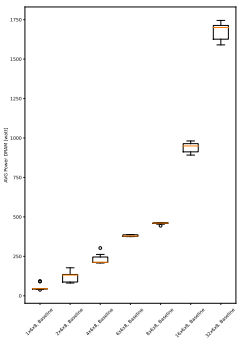
(i) Power: variations as threads increase.



(j) Power: boxplot of distribution as threads increase.



(k) Power: boxplot of distribution of the power consumed by CPU.



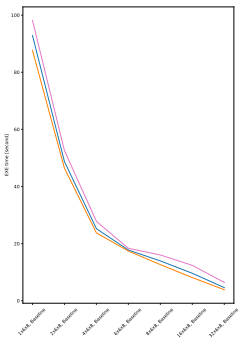
(l) Power: boxplot of distribution of the power consumed by RAM.

Figure 8.5: Experimental Results for Experiment 2 on the *Cubo_1772481.Ext_bin* matrix.

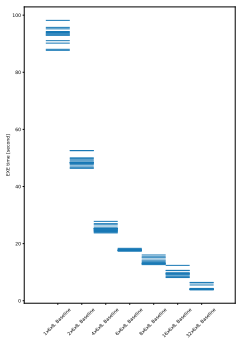
Run config	EXE time [s]	Energy [J]	AVG Power [W]	Ratio [(core/W)$^{-1}$]
$1 \times 6 \times 8$	93.4	29 643	326	0.003 15
$2 \times 6 \times 8$	48.3	32 544	668	0.002 97
$4 \times 6 \times 8$	25.1	33 844	1336	0.002 96
$6 \times 6 \times 8$	17.6	35 207	1997	0.003 01
$8 \times 6 \times 8$	13.6	35 056	2581	0.002 89
$16 \times 6 \times 8$	9.4	43 525	4636	0.003 46
$32 \times 6 \times 8$	4.1	40 530	9900	0.003 23

Table 8.8: Median values for Experiment 2 on the *Wing_4538k.csr.Ext_bin* matrix.

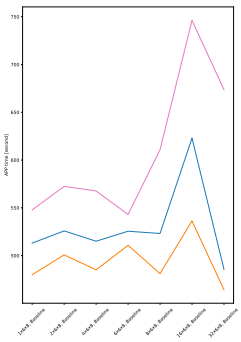
Wing_4538k.csr.Ext_bin The results of the experiments performed with the matrix *Wing_4538k.csr.Ext_bin* are visually presented in fig. 8.5. The most remarkable data are graphically presented in fig. 8.6, while additional insights can be derived from figs. 8.6a and 8.6e. In this context, the table 8.8 indicates that the **greenest point** is located at $8 \times 6 \times 8$, as shown in the graphs. At the same time, the **fastest point** is located at $32 \times 6 \times 8$.



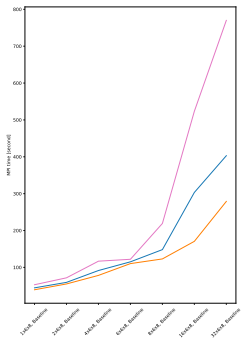
(a) EXE time: variations as threads increase.



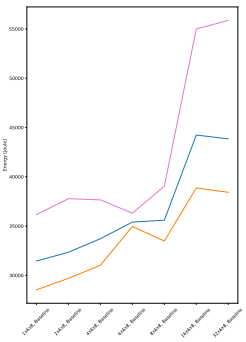
(b) EXE time: distribution of execution times for the 11 runs per instance.



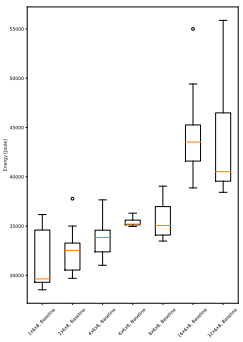
(c) APP time: distribution of execution times for the 11 runs per instance.



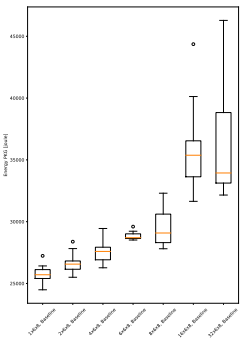
(d) MPI time: distribution of execution times for the 11 runs per instance.



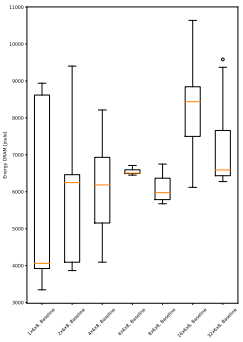
(e) Energy: variations as threads increase.



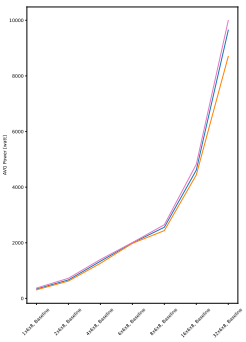
(f) Energy: boxplot of distribution as threads increase.



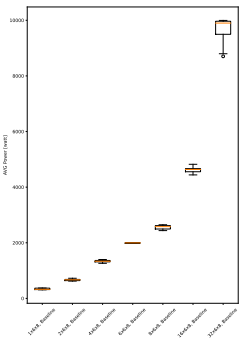
(g) Energy: boxplot of distribution of the energy consumed by CPU.



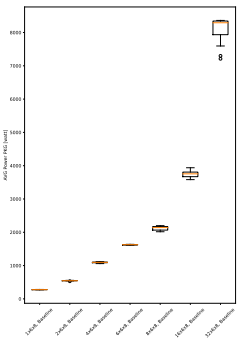
(h) Energy: boxplot of distribution of the energy consumed by RAM.



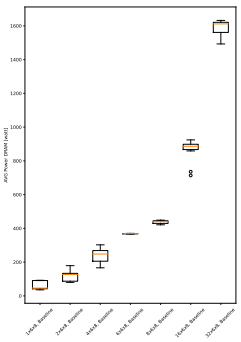
(i) Power: variations as threads increase.



(j) Power: boxplot of distribution as threads increase.



(k) Power: boxplot of distribution of the power consumed by CPU.



(l) Power: boxplot of distribution of the power consumed by RAM.

Figure 8.6: Experimental Results for Experiment 2 on the *Wing_4538k.csr.Ext_bin* matrix.

8.7 Experiment 3

The third experiment delves into the analysis of energy savings with the COUNTDOWN mechanism at the **greenest point** identified in the previous section 8.6. This inquiry is tailored to measure the magnitude of energy efficiency gains facilitated by COUNTDOWN when it harmonized with the most ecologically advantageous configuration as described in the previous section. Central to this effort is the quantification of energy consumption patterns at this specific point in time. By carefully isolating and examining the energy dynamics at play, our goal is to provide a focused assessment of the effectiveness of COUNTDOWN in promoting the highest level of environmental sustainability within the system.

In this experiment, we run our linear solver with 4 different configurations:

COUNTDOWN with Analysis only (Baseline) COUNTDOWN executed in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario served as a baseline to examine the basic functionality of COUNTDOWN.

COUNTDOWN with low power configuration (1.0 GHz) COUNTDOWN is set to keep the CPU frequency at 1.0 GHz, despite which the power saving algorithm is not enabled, since the Linux kernel as we will see, will not follow our instructions for long. We can also call this execution mode as COUNTDOWN enabled without timer, i.e. as soon as there is an MPI call, the CPU sets to low power.

COUNTDOWN without slack (CNTD) Execution of the energy saving algorithm without the inclusion of the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization.

COUNTDOWN with slack (CNTD SLACK) Execution of the energy saving algorithm along with optimization of slack times. This scenario sought to investigate the combined impact of energy optimization and slack time optimization through the COUNTDOWN mechanism.

Tables containing the numerical results of Experiment 3 contain the following data:

Node and run config Number of nodes, task and openMP threads (section 4.2) and configuration used, the acronyms are as listed above.

EXE time Execution time, calculated by COUNTDOWN (section 7.3).

APP time Total computation time of the application, understood as the sum of the various *Tcomp* of each node and rank, calculated by COUNTDOWN (section 7.3).

MPI time Total communication time of the application, understood as the sum of the different *Tcomm* of each single node and rank, calculated by COUNTDOWN (section 7.3).

TOT time Total time of the application, understood as the sum of the various *Tcomp* and *Tcomm* of each single node and rank, calculated by COUNTDOWN (section 7.3).

MAX Memory usage Maximum memory (RAM) usage, calculated by COUNTDOWN profiler (section 7.3)

AVG IPC Average number of instructions per clock executed by the program, calculated by COUNTDOWN (section 7.3).

Energy PKG Total energy consumed by the CPU, calculated by COUNTDOWN (section 7.3).

Energy DRAM Total energy consumed by the RAM, calculated by COUNTDOWN (section 7.3).

AVG Power PKG Average power consumed by the CPU, calculated by COUNTDOWN (section 7.3).

AVG Power DRAM Average power consumed by the RAM, calculated by COUNTDOWN (section 7.3).

AVG CPU frequency Average number of CPU clocks during the runtime of the program, calculated by COUNTDOWN (section 7.3).

MPI Network - TOT Total MB exchanged between nodes, task from MPI, calculated by COUNTDOWN (section 7.3).

Energy Total energy consumed by program execution, calculated by COUNTDOWN (section 7.3).

AVG Power Average power consumed by the program, calculated by COUNTDOWN (section 7.3).

The tables also shows a percentage comparison to the base case for all runs with COUNTDOWN. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Cubo_1772481.Ext_bin As observed in section 8.6, the most environmentally friendly configuration for *Cubo_1772481.Ext_bin* is determined to be $8 \times 6 \times 8$, with key data points graphically depicted in appendix fig. B.1 and in table 8.9.

It is observed that in this scenario the utilization of the COUNTDOWN Slack mechanism results in an increase in energy consumption from fig. B.1e in the baseline from 37 006.0 to 38 840.0 J, marking a rise of 5 %. Similarly, the average power decrease from 2752.8 to 2695.1 W, representing a 2 % decrement. This increase in energy consumption and a reduction in power is caused by a 7 % increase in execution times, from 13.4 to 14.4 s. Notably, the MPI time experiences a substantial increase of 35 %, escalating from 124.3 to 167.4 s. Additionally, the energy consumption of the CPU (PKG) have increase by 5 % and the power decrease of 2 %, and the same changes also result in the DRAM. These findings suggest that in this specific scenario, the utilization of COUNTDOWN Slack leads to increased energy consumption and environmental entropy, indicating that it may not be advisable to utilize this mechanism.

The low power (1.0 GHz) configuration demonstrates relatively stable energy consumption and execution times compared to the COUNTDOWN-enabled scenarios. The energy consumption remains consistent, with a marginal reduction of only 6 %, and the

power consumption experiences a slight decrease of 4 %. As in experiment 3 relative to this matrix, this suggests that all MPI calls are generally take more than than 500 µs.

Enabling COUNTDOWN without the SLACK results in a similar effect to low-power execution, with a 10 % and 5 % reduction in energy in power.

Wing_4538k.csr.Ext_bin As evident from section 8.6, the most environmentally friendly configuration for the *Wing_4538k.csr.Ext_bin* matrix is also determined to be $8 \times 6 \times 8$, with key data points visually represented in fig. B.2 and table 8.10.

In this instance, the COUNTDOWN Slack mechanism yields to energy savings of 4 % (from 31 341 to 30 088 J) and a power reduction of 6 % (from 2622.4 to 2463.2 W), as illustrated in figs. B.2e and B.2i. However there is a simultaneous 2 % increase in execution times, from 12.0 to 12.2 s, as detailed in fig. B.2a. It is crucial to note that the MPI time (fig. B.2d) has experienced a substantial increase of 12 % escalating from 116.9 s to 131.0 s, while the energy and average power consumption of the RAM has decreased by 16 % and 18 %, respectively (figs. B.2h and B.2l), from 4285 to 3595 J and from 358.3 to 294.6 W, also the energy of the CPU have a little reduction from 27 056 to 26 484 J (2 %), and the average power has a reduction of 4 %, from 2264.2 to 3596 W. In this particular scenario, the usage or non-usage of COUNTDOWN leads to modest energy savings, quantifiable in 4 %.

The low power (1.0 GHz) configuration demonstrates relatively stable energy consumption and execution times compared to the COUNTDOWN-enabled scenarios. The energy usage remains consistent, with a marginal deviation of only 1 %, while the power consumption experiences a slight decrease of 8 %. Despite this, the execution times exhibit minimal fluctuations.

Enabling COUNTDOWN without the SLACK leads to a drastic increase in both energy consumption and execution times. The energy consumption surges by 145 %, accompanied by a substantial 195 % increment in execution times. This sharp rise underscores the significant impact of aggressive COUNTDOWN settings, resulting in heightened resource utilization and extended computation durations.

Table 8.9: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 3 on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8 \times 6 \times 8$, Baseline	13.4	522.0	124.3	646.8	30597	6400	2276.0	476.4	32.5	1.6	3089	37006	2752.8
$8 \times 6 \times 8$, 1.0 GHz	13.2 $-2\,\%$	516.1 $-1\,\%$	116.9 $-6\,\%$	633.2 $-2\,\%$	29747 $3\,\%$	5085 $21\,\%$	2259.6 $1\,\%$	386.3 $19\,\%$	32.0 $1\,\%$	1.6 $0\,\%$	3084 $0\,\%$	34831 $6\,\%$	2645.9 $4\,\%$
$8 \times 6 \times 8$, CNTD	13.5 $1\,\%$	521.5 $0\,\%$	125.8 $1\,\%$	650.4 $1\,\%$	30228 $1\,\%$	7704 $-20\,\%$	2241.2 $2\,\%$	570.2 $-20\,\%$	33.0 $-2\,\%$	1.6 $-1\,\%$	3097 $0\,\%$	37909 $-2\,\%$	2811.6 $-2\,\%$
$8 \times 6 \times 8$, CNTD SLACK	14.4 $7\,\%$	524.9 $1\,\%$	167.4 $35\,\%$	693.6 $7\,\%$	32112 $-5\,\%$	6728 $-5\,\%$	2228.2 $2\,\%$	466.9 $2\,\%$	32.5 $0\,\%$	1.7 $-2\,\%$	2976 $4\,\%$	38840 $-5\,\%$	2695.1 $2\,\%$

Table 8.10: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 3 on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8 \times 6 \times 8$, Baseline	12.0	457.9	116.9	575.2	27056	4285	2264.2	358.3	27.0	1.6	3084	31341	2622.4
$8 \times 6 \times 8$, 1.0 GHz	11.5 $-3\,\%$	452.8 $-1\,\%$	103.1 $-12\,\%$	556.0 $-3\,\%$	26217.5 $3\,\%$	3776.5 $12\,\%$	2267.6 $0\,\%$	327.1 $9\,\%$	26.4 $1\,\%$	1.6 $1\,\%$	3090.5 $0\,\%$	29997 $4\,\%$	2594.6 $1\,\%$
$8 \times 6 \times 8$, CNTD	13.0 $9\,\%$	462.1 $1\,\%$	163.4 $40\,\%$	625.3 $9\,\%$	28441 $-5\,\%$	4525 $-6\,\%$	2192 $3\,\%$	348.6 $3\,\%$	27.0 $0\,\%$	1.6 $-3\,\%$	2965 $4\,\%$	32966 $-5\,\%$	2540.0 $3\,\%$
$8 \times 6 \times 8$, CNTD SLACK	12.2 $2\,\%$	456.9 $0\,\%$	131.0 $12\,\%$	587.5 $2\,\%$	26484 $2\,\%$	3595 $16\,\%$	2168.0 $4\,\%$	294.6 $18\,\%$	26.4 $1\,\%$	1.6 $1\,\%$	2973 $4\,\%$	30088 $4\,\%$	2463.2 $6\,\%$

8.8 Experiment 4

The fourth experiment focuses on the analysis of energy savings with the COUNTDOWN mechanism at the **fastest point**, identified in the previous section 8.6. This point will be the base point, that is, the starting point, for the purposes of the final conclusion. The purpose of this study is to quantify the extent of the energy efficiency improvements facilitated by COUNTDOWN when integrated with the **fastest point** configuration described in section 8.6. In addition, this analysis aims to quantify the energy consumption behavior for this particular configuration. Central to this effort is the quantification of energy consumption patterns at this maximum performance point in time. By carefully isolating and examining the energy dynamics at play. Our goal is to provide a focused assessment of the effectiveness of COUNTDOWN in promoting the highest level of environmental sustainability within the system.

In this experiment, we run our linear solver with 4 different configurations, which are the same as in Experiment 3 (section 8.7). The tables with the numerical results of Experiment 4 contain the same data of Experiment 3 (section 8.7).

The table also shows a percentage comparison to the base case for all runs with COUNTDOWN. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Cubo_1772481.Ext_bin As shown in section 8.6, the faster configuration for the *Cubo _1772481.Ext_bin* matrix is $32 \times 6 \times 8$, with key data points visually represented in appendix (fig. B.3) and all data displayed in table 8.11.

In this case, the COUNTDOWN mechanism results in a higher energy consumption of $11\,\%$ and a power reduction of $6\,\%$, as clearly illustrated in figs. B.3e and B.3i. However, despite these gains, there is a simultaneous $18\,\%$ increase in execution times, as detailed in fig. B.3a, which effectively nullifies the average power savings of $6\,\%$. It is crucial to note that the APP time (fig. B.3d) has experienced a significant increase of $63\,\%$, escalating from $333.0\,\mathrm{s}$ to $495.6\,\mathrm{s}$, and the MPI time has also drastically increased by $2\,\%$. In addition, there is a higher variance in the data, as shown in fig. B.3b. Meanwhile, the average RAM energy consumption has increased by $15\,\%$ and the power consumption has decreased by $3\,\%$ (figs. B.3h and B.3l). On the other hand, the CPU consumption of the nodes has increased by $15\,\%$ in terms of energy and decreased by $7\,\%$ in terms of average power.

The low power ($1.0\,\mathrm{GHz}$) configuration has relatively stable energy consumption and execution times compared to the COUNTDOWN enabled scenarios. The energy consumption remains constant, with a decrease of $12\,\%$, while the power consumption experiences a decrease of $3\,\%$. Despite this, execution times show minimal variation, indicating stable performance under the baseline configuration.

Enabling COUNTDOWN without SLACK leads to a drastic increase in both power consumption and execution times. Energy consumption jumps by $7\,\%$, accompanied by a $2\,\%$ decrease in execution times. This sharp increase underscores the significant impact of aggressive COUNTDOWN settings, resulting in increased resource utilization and longer execution times.

In this particular scenario, using or not using COUNTDOWN does not result in any discernible environmental benefits or savings. Despite the reduction in RAM energy and power consumption, the significant increase in MPI execution times and the offsetting of

the average power savings highlight the complex tradeoffs involved.

Wing_4538k.csr.Ext_bin As shown in section 8.6, the faster configuration for the *Wing_4538k.csr.Ext_bin* matrix is also determined to be $32 \times 6 \times 8$, with key data points visually represented in appendix (fig. B.4) and all data displayed in table 8.12.

In this case, the COUNTDOWN mechanism results in a higher energy consumption of $6\,\%$ and a power reduction of $4\,\%$, as clearly illustrated in figs. B.4e and B.4i. However, despite these gains, there is a simultaneous $11\,\%$ increase in execution times, as detailed in fig. B.4a, which offsets the average power savings of $6\,\%$. It is crucial to note that the MPI time (fig. B.4d) has experienced a significant increase of $19\,\%$, escalating from $558.3\,\mathrm{s}$ to $701.0\,\mathrm{s}$, and the APP time has also increased by $1\,\%$. In addition, there is a higher variance in the data, as shown in fig. B.4b. Meanwhile, the average RAM energy consumption has increased by $10\,\%$ and the power consumption has decreased by $1\,\%$ (figs. B.4h and B.4l). On the other hand, the CPU consumption of the nodes has increased by $4\,\%$ in terms of energy and decreased by $5\,\%$ in terms of average power.

In this particular scenario, using or not using COUNTDOWN does not result in any discernible environmental benefits or savings. Despite the reduction in RAM energy and power consumption, the significant increase in MPI execution times and the offsetting of the average power savings highlight the complex tradeoffs involved.

The low power ($1.0\,\mathrm{GHz}$) configuration has relatively stable energy consumption and execution times compared to the COUNTDOWN enabled scenarios. The energy consumption remains constant, with a reduction of $33\,\%$, while the power consumption experiences an increase of $1\,\%$, due to an incongruity, i.e., the reduction in execution time. Similarly, COUNTDOWN without slack also performs very well, which is evident by the fact that the transmitted data is not affected by the clock reduction. Enabling COUNTDOWN without SLACK leads to a drastic reduction in energy of $29\,\%$, in time of $28\,\%$, and a small increase in power of $1\,\%$.

Table 8.11: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 4 on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	4.9	594.8	333.0	949.4	39904	9445	8144.8	1931.7	60.9	2.0	3091	49380	10078.1
$32 \times 6 \times 8$, 1.0 GHz	4.4 $-9\,\%$	580.9 $-2\,\%$	276.1 $-17\,\%$	857.7 $-10\,\%$	37686 $6\,\%$	5924 $37\,\%$	8480.8 $-4\,\%$	1336.2 $31\,\%$	56.3 $8\,\%$	1.9 $4\,\%$	3089 $0\,\%$	43615 $12\,\%$	9823.1 $3\,\%$
$32 \times 6 \times 8$, CNTD	4.6 $-5\,\%$	581.2 $-2\,\%$	313.3 $-6\,\%$	900.6 $-5\,\%$	39381 $1\,\%$	5309 $44\,\%$	8487.6 $-4\,\%$	1142.0 $41\,\%$	54.1 $11\,\%$	2.0 $3\,\%$	3088 $0\,\%$	44672 $10\,\%$	9622.6 $5\,\%$
$32 \times 6 \times 8$, CNTD SLACK	5.8 $18\,\%$	604.3 $2\,\%$	495.5 $49\,\%$	1119.0 $18\,\%$	44146 $-11\,\%$	10874 $-15\,\%$	7604.4 $7\,\%$	1879.1 $35\,\%$	60.9 $0\,\%$	2.1 $-2\,\%$	2818 $9\,\%$	54985 $-11\,\%$	9466.1 $6\,\%$

Table 8.12: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 4 on the *Wing_4538k.csr.Ext_bin* matrix. matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	5.6	483.1	575.2	1069.1	39903	13310.5	7287.5	2465.6	59.5	2.3	3093	53196	9735.4
$32 \times 6 \times 8$, 1.0 GHz	3.6 $-34\,\%$	459.5 $-5\,\%$	245.4 $-58\,\%$	705.9 $-34\,\%$	31141 $23\,\%$	4733 $65\,\%$	8518.0 $-18\,\%$	1297.8 $47\,\%$	46.2 $22\,\%$	1.9 $16\,\%$	3088 $0\,\%$	35883 $33\,\%$	9819.0 $-1\,\%$
$32 \times 6 \times 8$, CNTD	4.0 $-28\,\%$	461.5 $-5\,\%$	307.8 $-48\,\%$	770.1 $-28\,\%$	33522 $17\,\%$	4764 $65\,\%$	8428.6 $-16\,\%$	1197.8 $51\,\%$	45.3 $24\,\%$	2.0 $14\,\%$	3089 $0\,\%$	38286 $29\,\%$	9624.2 $1\,\%$
$32 \times 6 \times 8$, CNTD SLACK	4.6 $11\,\%$	469.8 $1\,\%$	402.1 $19\,\%$	885.3 $10\,\%$	35401 $-4\,\%$	7374 $-10\,\%$	7734.7 $5\,\%$	1614.0 $1\,\%$	49.6 $0\,\%$	2.0 $5\,\%$	2802 $17\,\%$	42775 $-6\,\%$	9336.7 $4\,\%$

8.9 Experiment 5

In this experiment we investigate if the default callback delay value of $500\,\mu s$ is the ideal value, since we know that this value is due to the fact that Linux updates the CPU speed every $500\,\mu s$ and that in general an MPI call does not take longer than $500\,\mu s$, as described in more detail in section 4.4.

For this experiment, we run our linear solver with the following callback delays $100\,\mu s$, $250\,\mu s$, $750\,\mu s$ and $1000\,\mu s$ along with the configurations already seen:

COUNTDOWN with Analysis Only (Baseline) COUNTDOWN run in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario was used as a baseline to examine the basic functionality of COUNTDOWN.

COUNTDOWN with low power configuration ($1.0\,GHz$) COUNTDOWN is set to keep the CPU frequency at $1.0\,GHz$, although the power saving algorithm is not enabled, since the Linux kernel will not follow our instructions for long. We can also call this execution mode as COUNTDOWN enabled without timer, i.e. as soon as there is an MPI call, the CPU sets to low power.

COUNTDOWN without slack (CNTD) Execution of the power saving algorithm without including the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization.

COUNTDOWN with slack and a callback delay of $100\,\mu s$ **(CNTD 100)** Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of energy optimization and slack time with a slack callback delay set to $100\,\mu s$.

COUNTDOWN with slack and a callback delay of $250\,\mu s$ **(CNTD 250)** Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of energy optimization and slack time with a slack callback delay set to $100\,\mu s$.

COUNTDOWN with slack and a callback delay of $750\,\mu s$ **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to $500\,\mu s$.

COUNTDOWN with slack and a callback delay of $750\,\mu s$ **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to $750\,\mu s$.

COUNTDOWN with slack and a callback delay of $1000\,\mu s$ **(CNTD 1000)** Running the power saving algorithm along with slack time optimization. This scenario seeks to investigate the combined impact of power optimization and slack with a slack callback delay set to $100\,\mu s$.

8.9.1 Experiment 5a

In this sub-experiment we investigate if the default value of callback delay of 500 µs is the optimal value in the **greenest point**.

Cubo_1772481.Ext_bin The key data of the result are visually represented in appendix (fig. B.5) and all data displayed in table 8.13, Varying the COUNTDOWN time, we observe that execution times remain quite table, with an increase of 12 % count 100 µs and 5 % in 750 and 1000 µs. In other cases, 250 µs and 500 µs, of an 8 and 7 %. Notably, the predominant increase is observed in MPI Time, with an escalation of 53 % at 100 µs, 37 % at 250 µs, and reaching 20 % at 750 µs. This highlights the significant impact of the COUNTDOWN algorithm on MPI communications, despite the presence of the MPI barrier, which is intended to mitigate such effects. This phenomenon also extends to APP time, with maximum increases of 1 % in all configurations, which is negligible and acceptable. The average energy generally increase by 3 %–5 % for all configurations, while average power decrease by 2 %–3 %.

Wing_4538k.csr.Ext_bin As evident from section 8.6, the most environmentally friendly configuration for the *Wing_4538k.csr.Ext_bin* matrix is also determined to be $8 \times 6 \times 8$, with key data points visually represented in appendix (fig. B.6) and all data displayed in table 8.14.

The situation is similar for the *Wing_4538k.csr.Ext_bin* matrix regarding execution times. Here, we observe minimal increases in the 750 and 1000 µs configurations, with a rise of 1 %–2 %, while other configurations exhibit increases ranging from 6 to 12 %. This results in a modest power saving of around 2 % in the 750 µs configuration but we have a 4 % on 500 µs. These values might seem marginal compared to expectations. This is evident in the significantly increased MPI times, up to 54 % in the 100 µs configuration, while the other configurations show only a 6 % increase at 750 µs, suggesting that perhaps this is the more suitable value for this specific case. The APP time does not vary significantly in this instance, with variations of at most 2 %.

Table 8.13: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5a on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8 \times 6 \times 8$, Baseline	13.4	522.0	124.3	646.8	30597	6400	2276.0	476.4	32.5	1.6	3089	37006	2752.8
$8 \times 6 \times 8$, 1.0 GHz	13.2 $-2\,\%$	516.1 $-1\,\%$	116.9 $-6\,\%$	633.2 $-2\,\%$	29747 $3\,\%$	5085 $21\,\%$	2259.6 $1\,\%$	386.3 $19\,\%$	32.0 $1\,\%$	1.6 $0\,\%$	3084 $0\,\%$	34831 $6\,\%$	2645.9 $4\,\%$
$8 \times 6 \times 8$, CNTD	13.5 $1\,\%$	521.5 $0\,\%$	125.8 $1\,\%$	650.4 $1\,\%$	30228 $1\,\%$	7704 $-20\,\%$	2241.2 $2\,\%$	570.2 $-20\,\%$	33.0 $-2\,\%$	1.6 $-1\,\%$	3097 $0\,\%$	37909 $-2\,\%$	2811.6 $-2\,\%$
$8 \times 6 \times 8$, CNTD 100	15.0 $12\,\%$	529.4 $1\,\%$	190.8 $53\,\%$	722.8 $12\,\%$	33031 $-8\,\%$	6929 $-8\,\%$	2199.0 $3\,\%$	461.0 $3\,\%$	32.5 $0\,\%$	1.7 $-2\,\%$	2932 $5\,\%$	39966 $-8\,\%$	2659.2 $3\,\%$
$8 \times 6 \times 8$, CNTD 250	14.6 $8\,\%$	525.1 $1\,\%$	170.4 $37\,\%$	700.9 $8\,\%$	32097 $-5\,\%$	6685 $-4\,\%$	2204.8 $3\,\%$	459.0 $4\,\%$	32.5 $0\,\%$	1.7 $-2\,\%$	2961 $4\,\%$	38782 $-5\,\%$	2663.9 $3\,\%$
$8 \times 6 \times 8$, CNTD 500	14.4 $7\,\%$	524.9 $1\,\%$	167.4 $35\,\%$	693.6 $7\,\%$	32112 $-5\,\%$	6728 $-5\,\%$	2228.2 $2\,\%$	466.9 $2\,\%$	32.5 $0\,\%$	1.7 $-2\,\%$	2976 $4\,\%$	38840 $-5\,\%$	2695.1 $2\,\%$
$8 \times 6 \times 8$, CNTD 750	14.1 $5\,\%$	522.3 $0\,\%$	149.6 $20\,\%$	676.4 $5\,\%$	31322 $-2\,\%$	6516 $-2\,\%$	2231.6 $2\,\%$	463.8 $3\,\%$	32.5 $0\,\%$	1.6 $-1\,\%$	2993 $3\,\%$	37877 $-2\,\%$	2695.2 $2\,\%$
$8 \times 6 \times 8$, CNTD 1000	14.1 $5\,\%$	523.5 $0\,\%$	152.0 $22\,\%$	676.5 $5\,\%$	31445 $-3\,\%$	6615 $-3\,\%$	2234.2 $2\,\%$	470.5 $1\,\%$	32.5 $0\,\%$	1.6 $-1\,\%$	2993 $3\,\%$	38052 $-3\,\%$	2704.6 $2\,\%$

Table 8.14: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5a on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8\times6\times8$, Baseline	12.0	457.9	116.9	575.2	27056	4285	2264.2	358.3	27.0	1.6	3084	31341	2622.4
$8\times6\times8$, 1.0 GHz	11.5	452.8	103.1	556.0	26217.5	3776.5	2267.6	327.1	26.4	1.6	3090.5	29997	2594.6
	$-3\,\%$	$-1\,\%$	$-12\,\%$	$-3\,\%$	$3\,\%$	$12\,\%$	$0\,\%$	$9\,\%$	$1\,\%$	$1\,\%$	$0\,\%$	$4\,\%$	$1\,\%$
$8\times6\times8$, CNTD	13.0	462.1	163.4	625.3	28441	4525	2192	348.6	27.0	1.6	2965	32966	2540.0
	$9\,\%$	$1\,\%$	$40\,\%$	$9\,\%$	$-5\,\%$	$-6\,\%$	$3\,\%$	$3\,\%$	$0\,\%$	$-3\,\%$	$4\,\%$	$-5\,\%$	$3\,\%$
$8\times6\times8$, CNTD 100	13.4	465.4	180.0	645.1	29104	4616	2168.9	344.3	26.6	1.6	2916	33736	2511.6
	$12\,\%$	$2\,\%$	$54\,\%$	$12\,\%$	$-8\,\%$	$-8\,\%$	$4\,\%$	$4\,\%$	$0\,\%$	$-3\,\%$	$5\,\%$	$-8\,\%$	$4\,\%$
$8\times6\times8$, CNTD 250	12.7	460.3	141.2	609.2	27616	4083	2178.9	322.6	26.4	1.6	2959	31687	2501.8
	$6\,\%$	$1\,\%$	$21\,\%$	$6\,\%$	$-2\,\%$	$5\,\%$	$4\,\%$	$10\,\%$	$1\,\%$	$0\,\%$	$4\,\%$	$-1\,\%$	$5\,\%$
$8\times6\times8$, CNTD 500	12.2	456.9	131.0	587.5	26484	3595	2168.0	294.6	26.4	1.6	2973	30088	2463.2
	$2\,\%$	$0\,\%$	$12\,\%$	$2\,\%$	$2\,\%$	$16\,\%$	$4\,\%$	$18\,\%$	$1\,\%$	$1\,\%$	$4\,\%$	$4\,\%$	$6\,\%$
$8\times6\times8$, CNTD 750	12.1	457.4	124.3	582.9	26696	3984	2204.6	328.8	26.4	1.6	2970	30682	2533.7
	$1\,\%$	$0\,\%$	$6\,\%$	$1\,\%$	$1\,\%$	$7\,\%$	$3\,\%$	$8\,\%$	$1\,\%$	$1\,\%$	$4\,\%$	$2\,\%$	$3\,\%$
$8\times6\times8$, CNTD 1000	12.7	460.3	150.3	610.5	28008	4454	2202.7	351.3	27.0	1.6	2981	32476	2554.2
	$6\,\%$	$1\,\%$	$29\,\%$	$6\,\%$	$-4\,\%$	$-4\,\%$	$3\,\%$	$2\,\%$	$0\,\%$	$-2\,\%$	$3\,\%$	$-4\,\%$	$3\,\%$

8.9.2 Experiment 5b

In this sub-experiment we investigate if the default value of callback delay of 500 µs is the optimal value in the **fastest point**.

Cubo_1772481.Ext_bin As evident from section 8.6, the faster configuration for the *Cubo_1772481.Ext_bin* matrix is determined to be $32 \times 6 \times 8$, with key data points visually presented in appendix (fig. B.7) and all data displayed in table 8.15.

In this scenario, the increase in execution times ranges from at 14 % to 51 %, with the configuration of 100 µs exhibiting the highest increase. The optimal choice, once again, is the 750 µs setup, with a 7 % power saving and a 6 % increase in energy. This increase in energy consumption is primarily attributed to the rise in MPI time, which increases by 35 %, resulting in an overall 14 % increase when considering no change in APP time. Energy, in all cases, increases from 6 % (750 µs) to 37 % (100 µs), while power ries from 6 % (500 µs) to 9 % (100 µs and 250 µs).

Wing_4538k.csr.Ext_bin As evident from section 8.6, the faster configuration for the *Wing_4538k.csr.Ext_bin* matrix is also determined to be $32 \times 6 \times 8$, with key data points visually represented in appendix (fig. B.8) and all data displayed in table 8.16.

In this configuration, we observe that, unlike the more environmentally friendly setup, COUNTDOWN can make a significant difference when well-configured. A remarkable 19 % energy savings is achieved when the barrier triggers after 750 µs, a value close to the 25 % mentioned in the paper Cesarini et al. [35]. However, it's worth noting a still substantial 17 % reduction in median time, probably due to improved synchronization, a side effect of COUNTDOWN observed in other scenarios, along with the increased variance in the distribution of APP times.

Examining individual data points, it is notable that the 250 µs configuration exhibits a substantial 32 % reduction in memory consumption, followed closely by the 750 µs setup with a 30 % reduction. In this scenario, when COUNTDOWN applies the consumption reduction algorithm, especially when the frequency is lower than the default, it significantly influences the average, resulting in genuine overall energy savings rather than just instantaneous reductions.

Nevertheless, not all execution times are decreased. For instance, with 100 and 1000 µs, there is a relative increase in execution times of 38 % and 13 %, primarily driven by MPI time, which sees an increase of 66 % and 24 %, respectively.

Table 8.15: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5b on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	4.9	594.8	333.0	949.4	39904	9445	8144.8	1931.7	60.9	2.0	3091	49380	10078.1
$32 \times 6 \times 8$, 1.0 GHz	4.4	580.9	276.1	857.7	37686	5924	8480.8	1336.2	56.3	1.9	3089	43615	9823.1
	$-9\,\%$	$-2\,\%$	$-17\,\%$	$-10\,\%$	$6\,\%$	$37\,\%$	$-4\,\%$	$31\,\%$	$8\,\%$	$4\,\%$	$0\,\%$	$12\,\%$	$3\,\%$
$32 \times 6 \times 8$, CNTD	4.6	581.2	313.3	900.6	39381	5309	8487.6	1142.0	54.1	2.0	3088	44672	9622.6
	$-5\,\%$	$-2\,\%$	$-6\,\%$	$-5\,\%$	$1\,\%$	$44\,\%$	$-4\,\%$	$41\,\%$	$11\,\%$	$3\,\%$	$0\,\%$	$10\,\%$	$5\,\%$
$32 \times 6 \times 8$, CNTD 100	7.4	621.9	766.4	1428.3	54274	13428	7353.5	1817.4	61.1	2.2	2785	67713	9164.6
	$51\,\%$	$5\,\%$	$130\,\%$	$50\,\%$	$-36\,\%$	$-42\,\%$	$10\,\%$	$6\,\%$	$0\,\%$	$-9\,\%$	$10\,\%$	$-37\,\%$	$9\,\%$
$32 \times 6 \times 8$, CNTD 250	6.5	605.3	576.6	1255.7	48978	10495	7581.5	1613.6	58.1	2.1	2811	59508	9196.8
	$33\,\%$	$2\,\%$	$73\,\%$	$32\,\%$	$-23\,\%$	$-11\,\%$	$7\,\%$	$16\,\%$	$5\,\%$	$-6\,\%$	$9\,\%$	$-21\,\%$	$9\,\%$
$32 \times 6 \times 8$, CNTD 500	5.8	604.3	495.5	1119.0	44146	10874	7604.4	1879.1	60.9	2.1	2818	54985	9466.1
	$18\,\%$	$2\,\%$	$49\,\%$	$18\,\%$	$-11\,\%$	$-15\,\%$	$7\,\%$	$3\,\%$	$0\,\%$	$-2\,\%$	$9\,\%$	$-11\,\%$	$6\,\%$
$32 \times 6 \times 8$, CNTD 750	5.5	596.3	448.4	1074.4	42982	9207	7766.3	1659.8	58.1	2.0	2846	52145	9419.1
	$14\,\%$	$0\,\%$	$35\,\%$	$13\,\%$	$-8\,\%$	$3\,\%$	$5\,\%$	$14\,\%$	$5\,\%$	$0\,\%$	$8\,\%$	$-6\,\%$	$7\,\%$
$32 \times 6 \times 8$, CNTD 1000	5.9	601.8	463.1	1135.2	44760	10995	7635.7	1877.0	60.8	2.1	2844	55755	9515.8
	$20\,\%$	$1\,\%$	$39\,\%$	$20\,\%$	$-12\,\%$	$-16\,\%$	$6\,\%$	$3\,\%$	$0\,\%$	$-2\,\%$	$8\,\%$	$-13\,\%$	$6\,\%$

Table 8.16: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5b on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	5.6	483.1	575.2	1069.1	39903	13310.5	7287.5	2465.6	59.5	2.3	3093	53196	9735.4
$32 \times 6 \times 8$, 1.0 GHz	3.6	459.5	245.4	705.9	31141	4733	8518.0	1297.8	46.2	1.9	3088	35883	9819.0
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD	4.0	461.5	307.8	770.1	33522	4764	8428.6	1197.8	45.3	2.0	3089	38286	9624.2
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD 100	7.5	505.7	953.4	1454.5	51403.5	18180	6856.7	2416.8	59.6	2.3	2611.5	69332	9277.9
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD 250	5.5	477.4	563.3	1059.2	41952	9191	7514.1	1676.1	50.7	2.2	2763	51264	9190.3
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD 500	4.6	469.8	402.1	885.3	35401	7374	7734.7	1614.0	49.6	2.0	2802	42775	9336.7
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD 750	4.6	472.1	397.4	895.0	35904	7946	7770.2	1719.0	50.9	2.0	2822	43850	9489.1
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %
$32 \times 6 \times 8$, CNTD 1000	6.3	489.6	727.7	1216.4	42714	15361	6799.3	2446.9	59.5	2.2	2587	58295	9219.8
	20 %	1 %	39 %	20 %	−12 %	−16 %	6 %	3 %	0 %	−2 %	8 %	6 %	6 %

8.9.3 Experiment 5c

In this sub-experiment we investigate if the default value of the callback delay of $500\,\mu s$ is the optimal value in the intermediate case between the greenest and the fastest case, i.e. the point $16 \times 6 \times 8$.

Cubo_1772481.Ext_bin In the *Cubo_1772481.Ext_bin* matrix case, the graphical results of which are in appendix (fig. B.9) and all data displayed in table 8.17, we note that the behavior is almost better than the previous situations. With an overhead of $9\,\%$, it consumes more than $5\,\%$ and it has a power reduction of $4\,\%$.

Analyzing the table, it's evident that the execution time (EXE time) varies significantly across different configurations. The baseline setup takes $7.2\,s$, while the $1.0\,GHz$ configuration drastically increases this time to $58.2\,s$, representing a $708\,\%$ increase. When COUNTDOWN (CNTD) is applied, there's a slight increase in execution time across all configurations, with the highest increase observed at the $100\,\mu s$ setup ($21\,\%$). However, the impact on execution time remains relatively low, with most configurations showing single-digit percentage increases.

Looking at the application time (APP time), similar trends are observed, with the baseline configuration taking $550.9\,s$ and the $1.0\,GHz$ setup taking $2199.7\,s$, indicating a $299\,\%$ increase. Although COUNTDOWN slightly increases the application time, the changes are minimal, with most configurations showing negligible percentage changes.

The message changes slightly when examining the MPI time. Here the effect of COUNTDOWN is more pronounced, especially at lower callback delays. For example, the $100\,\mu s$ setup sees a significant increase in MPI time ($92\,\%$), which has a noticeable impact on the total execution time. However, with a higher callback delay, the increase in MPI time is less significant.

In terms of overall energy consumption, both Energy PKG and Energy DRAM show an increase over over the baseline for most configurations. The largest increase is observed in the $1.0\,GHz$ configuration, with a $568\,\%$ increase in PKG energy consumption and a $423\,\%$ increase in DRAM energy consumption. Despite the slight reductions observed with COUNTDOWN, the overall trend remains an increase in power consumption compared to the baseline.

Wing_4538k.csr.Ext_bin The results for the *Wing_4538k.csr.Ext_bin* matrix case follow a similar pattern, the graphical results of which can be found in the appendix (fig. B.10) and all data displayed in table 8.18. Here we observe an overhead of $16\,\%$, which coincides with a consumption increase of $13\,\%$.

Analyzing the table, we notice that the baseline configuration has an execution time (EXE time) of $6.5\,s$, while the $1.0\,GHz$ setup slightly reduces this time to $6.3\,s$, marking a modest $3\,\%$ decrease. However, upon applying COUNTDOWN (CNTD), there's a notable increase in execution time across all configurations, with the highest spike observed at $8.4\,s$ ($29\,\%$ increase).

Examining the application time (APP time), there's a negligible decrease of $1\,\%$ in the $1.0\,GHz$ setup compared to the baseline. With COUNTDOWN, the application time increases slightly across all configurations, indicating minimal impact on application performance.

The most significant changes are observed in MPI time, where the baseline configuration consumes $169.1\,\mathrm{s}$. The $1.0\,\mathrm{GHz}$ setup reduces this time to $154.9\,\mathrm{s}$, indicating an $8\,\%$ decrease. However, with COUNTDOWN, there's a substantial increase in time across all configurations, with the highest one observed at $334.2\,\mathrm{s}$ ($98\,\%$ increase).

In terms of energy consumption, both Energy PKG and Energy DRAM show an increases across most configurations compared to the baseline. The highest increase is noted in the COUNTDOWN configuration, with a $16\,\%$ rise in PKG energy consumption and a $13\,\%$ increase in DRAM energy consumption.

Table 8.17: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5c on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$16 \times 6 \times 8$, Baseline	7.2	550.9	143.1	693.7	32445	4565	4503.5	633.9	35.9	1.7	3081	37006	5139.2
$16 \times 6 \times 8$, 1.0 GHz	58.2	2199.7	3389.0	5591.3	216868.2	33877	3728.1	410.4	35.8	2.6	3099	240800	4139.0
	708 %	299 %	2269 %	706 %	−568 %	−423 %	17 %	35 %	0 %	−51 %	−1 %	−551 %	19 %
16×6×8, CNTD	7.8	556.9	196.6	755.5	33830	7221	4287.0	813.3	39.1	1.8	3093	40881	5193.7
	9 %	1 %	37 %	9 %	−4 %	−58 %	5 %	−28 %	−9 %	−5 %	0 %	−10 %	−1 %
$16 \times 6 \times 8$, CNTD 100	8.7	559.7	274.8	836.9	36773	5130	4234.0	590.7	35.9	1.8	2898	41908	4824.3
	21 %	2 %	92 %	21 %	−13 %	−12 %	6 %	7 %	0 %	−5 %	6 %	−13 %	6 %
$16 \times 6 \times 8$, CNTD 250	7.7	553.8	193.2	748.1	33735	4772	4354.1	616.2	35.8	1.7	2945	38501	4969.1
	8 %	1 %	35 %	8 %	−4 %	−5 %	3 %	3 %	0 %	−1 %	4 %	−4 %	3 %
$16 \times 6 \times 8$, CNTD 500	7.8	553.9	203.2	757.0	34111	4813	4343.9	612.0	35.8	1.8	2938	38921	4955.9
	9 %	1 %	42 %	9 %	−5 %	−5 %	4 %	3 %	0 %	−2 %	5 %	−5 %	4 %
$16 \times 6 \times 8$, CNTD 750	7.7	554.3	191.3	746.1	33747	4770	4357.0	616.6	35.8	1.7	2937	38523	4972.1
	7 %	1 %	34 %	8 %	−4 %	−4 %	3 %	3 %	0 %	−1 %	5 %	−4 %	3 %
$16 \times 6 \times 8$, CNTD 1000	7.7	554.0	189.7	742.9	33649	4759	4353.3	616.7	35.8	1.7	2942	38417	4968.9
	7 %	1 %	33 %	7 %	−4 %	−4 %	3 %	3 %	0 %	−1 %	5 %	−4 %	3 %

Table 8.18: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 5c on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$16 \times 6 \times 8$, Baseline	9.6	612.1	317.8	923.2	35376	8571.5	3737.5	892.9	33.9	2.1	3093.5	43947.5	4636.3
$16 \times 6 \times 8$, 1.0 GHz	6.3	452.7	154.9	608.0	27903	3798	4429.1	602.2	30.3	1.7	3075	31696	5030.1
	$-34\,\%$	$-26\,\%$	$-51\,\%$	$-34\,\%$	$21\,\%$	$56\,\%$	$-19\,\%$	$33\,\%$	$11\,\%$	$18\,\%$	$1\,\%$	$28\,\%$	$-8\,\%$
$16 \times 6 \times 8$, CNTD	8.4	464.8	334.2	807.0	33104	8096	3960.0	967.2	34.3	2.0	3089	41163	4933.1
	$-13\,\%$	$-24\,\%$	$5\,\%$	$-13\,\%$	$6\,\%$	$6\,\%$	$-6\,\%$	$-8\,\%$	$-1\,\%$	$5\,\%$	$0\,\%$	$6\,\%$	$-6\,\%$
$16 \times 6 \times 8$, CNTD 100	7.5	463.4	262.0	726.5	31175	5947	4125.6	790.4	32.6	1.8	2870	37112	4916.3
	$-21\,\%$	$-24\,\%$	$-18\,\%$	$-21\,\%$	$12\,\%$	$31\,\%$	$-10\,\%$	$11\,\%$	$4\,\%$	$14\,\%$	$7\,\%$	$16\,\%$	$-6\,\%$
$16 \times 6 \times 8$, CNTD 250	7.6	463.9	268.2	730.6	31275	5994	4122.7	789.8	32.6	1.8	2863	37255	4913.6
	$-21\,\%$	$-24\,\%$	$-16\,\%$	$-21\,\%$	$12\,\%$	$30\,\%$	$-10\,\%$	$12\,\%$	$4\,\%$	$14\,\%$	$7\,\%$	$15\,\%$	$-6\,\%$
$16 \times 6 \times 8$, CNTD 500	7.6	462.5	263.9	728.7	31222	5978	4130.2	790.9	32.6	1.8	2864	37201	4921.9
	$-21\,\%$	$-24\,\%$	$-17\,\%$	$-21\,\%$	$12\,\%$	$30\,\%$	$-11\,\%$	$11\,\%$	$4\,\%$	$15\,\%$	$7\,\%$	$15\,\%$	$-6\,\%$
$16 \times 6 \times 8$, CNTD 750	7.1	459.9	222.8	685.3	29401	5696	4166.3	802.4	32.6	1.8	2868	35084	4971.6
	$-26\,\%$	$-25\,\%$	$-30\,\%$	$-26\,\%$	$17\,\%$	$34\,\%$	$-11\,\%$	$10\,\%$	$4\,\%$	$17\,\%$	$7\,\%$	$20\,\%$	$-7\,\%$
$16 \times 6 \times 8$, CNTD 1000	7.6	462.8	269.6	733.7	31345	5997	4119.9	788.8	32.6	1.8	2865	37355	4905.6
	$-21\,\%$	$-24\,\%$	$-15\,\%$	$-21\,\%$	$11\,\%$	$30\,\%$	$-10\,\%$	$12\,\%$	$4\,\%$	$14\,\%$	$7\,\%$	$15\,\%$	$-6\,\%$

8.10 Experiment 6

Experiments 3–5 (sections from 8.7 to 8.9) were performed with an insufficient number of iterations for the convergence of the problem, just over 1000 iterations as opposed to the more than 6000 required for *Cubo_1772481.Ext_bin* and 10000 for *Wing_4538k.csr.Ext_bin*. However, this was done in order to ensure a more even distribution between the parts and to avoid too much dwelling on the solving part of the problem, as described in section 3.5. Experiment 6 seeks to analyze real-life performance and to confirm or disprove the results obtained in the previous experiments. In this experiment, we see only the most promising configurations from Experiment 5, along with those also presented in Cesarini et al. [36].

COUNTDOWN with Analysis Only (Baseline) COUNTDOWN run in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario was used as a baseline to examine the basic functionality of COUNTDOWN.

COUNTDOWN with low power configuration (1.0 GHz) COUNTDOWN is set to keep the CPU frequency at 1.0 GHz, although the power saving algorithm is not enabled, since the Linux kernel will not follow our instructions for long. We can also call this execution mode as COUNTDOWN enabled without timer, i.e. as soon as there is an MPI call, the CPU sets to low power.

COUNTDOWN without slack (CNTD) Execution of the power saving algorithm without including the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization.

COUNTDOWN with slack and a callback delay of 750 µs **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to 500 µs.

COUNTDOWN with slack and a callback delay of 750 µs **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to 750 µs.

8.10.1 Experiment 6a

Sub-experiment 6a analyzes the performance of the linear solver Chronos in the **greenest point**, i.e., the configuration of $8 \times 6 \times 8$ nodes.

Cubo_1772481.Ext_bin The comparison between profiles with and without COUNTDOWN for Experiment 6a on the *Cubo_1772481.Ext_bin* matrix reveals notable insights, particularly when considering COUNTDOWN 500 and COUNTDOWN 750 relative to the baseline configuration.

In terms of performance, the addition of COUNTDOWN 500 results in an 8 % increase in total execution time compared to the baseline, while COUNTDOWN 750 shows a 3 % increase. Regarding energy consumption, there is a 7 % increase in package energy (CPU)

with COUNTDOWN 500 and a 2 % increase with COUNTDOWN 750 relative to the baseline.

Looking specifically at the application performance, COUNTDOWN 500 exhibits a 2 % increase in application time compared to the baseline, while COUNTDOWN 750 shows a marginal 1 % increase. Meanwhile, for memory usage, COUNTDOWN 500 shows a modest 6 % increase, while COUNTDOWN 750 does not see any significant change compared to the baseline.

These results are supported by the graphical results of which are in appendix (fig. B.11) and all data displayed in table 8.19.

Wing_4538k.csr.Ext_bin The comparison between profiles with and without COUNTDOWN for Experiment 6a on the *Wing_4538k.csr.Ext_bin* matrix unveils significant insights, particularly concerning COUNTDOWN 500 and COUNTDOWN 750 relative to the baseline configuration.

In terms of execution time, there is an 8 % increase with COUNTDOWN 500 and 8 % with COUNTDOWN 750 compared to the baseline. Meanwhile, the application time experiences a marginal 1 % increase with both COUNTDOWN 500 and COUNTDOWN 750, indicating relatively stable performance. However, MPI time exhibits a considerable 51 % increase with COUNTDOWN 500 and a 47 % increase with COUNTDOWN 750, showcasing the impact on parallel computing efficiency.

Regarding energy consumption, package energy (CPU) sees an 8 % increase with COUNTDOWN 500 and a 7 % increase with COUNTDOWN 750 relative to the baseline. Notably, DRAM energy consumption fluctuates, with a 19 % decrease observed with COUNTDOWN 500 and a 17 % increase with COUNTDOWN 750, suggesting varying memory utilization patterns.

Analyzing the IPC, both COUNTDOWN 500 and COUNTDOWN 750 show improvements of 6 % and 8 %, respectively, compared to the baseline. This increase signifies enhanced instruction throughout with the implementation of COUNTDOWN.

These results are supported by the graphical results of which are in appendix (fig. B.12) and all data displayed in table 8.20.

Table 8.19: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6a on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8\times6\times8$, Baseline	19.8	780.8	167.7	952.2	45641	11227	2307.6	559.9	32.8	1.6	3080	56744	2835.9
$8\times6\times8$, 1.0 GHz	106.2	2136.3	2967.1	5099.8	203099	22619	1913.3	212.9	31.7	2.6	3099	225686	2126.9
	436 %	174 %	1670 %	436 %	−345 %	−101 %	17 %	62 %	3 %	−66 %	−1 %	−298 %	25 %
$8\times6\times8$, CNTD	19.8	782.4	170.5	951.4	45853	10522	2319.0	532.0	32.5	1.6	3070	56380	2850.1
	0 %	0 %	2 %	0 %	0 %	6 %	0 %	5 %	1 %	1 %	0 %	1 %	−1 %
$8\times6\times8$, CNTD 500	21.5	786.7	246.0	1032.0	48656	11890	2266.8	546.1	32.8	1.6	2988	60481	2800.0
	8 %	1 %	47 %	8 %	−7 %	−6 %	2 %	2 %	0 %	−2 %	3 %	−7 %	1 %
$8\times6\times8$, CNTD 750	20.5	781.8	204.1	985.7	46776	11474	2283.1	554.7	32.8	1.6	3011	58124	2819.8
	3 %	0 %	22 %	4 %	−2 %	−2 %	1 %	1 %	0 %	−1 %	2 %	−2 %	1 %

Table 8.20: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6a on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8\times6\times8$, Baseline	67.5	2585.1	693.8	3244.5	158754	38261.5	2347.6	585.2	27.3	1.4	3068	200657.2	2933.7
$8\times6\times8$, 1.0 GHz	426.6	5890.7	14564.1	20476.6	784874	175395	1840	411	27.4	2.6	3099	960269	2251.2
	531 %	128 %	1999 %	531 %	−394 %	−358 %	22 %	30 %	0 %	−82 %	−1 %	−379 %	23 %
$8\times6\times8$, CNTD	72.4	2645.3	923.4	3477.4	168999	45754	2305.7	630.1	28.0	1.5	3043	214753	2937.9
	7 %	2 %	33 %	7 %	−6 %	−20 %	2 %	−8 %	−1 %	−4 %	1 %	−7 %	0 %
$8\times6\times8$, CNTD 500	66.3	2567.4	609.7	3184.3	156568	46040.5	2358.6	692.4	28.0	1.3	3072	202627	3047.8
	−2 %	−1 %	−12 %	−2 %	1 %	−20 %	0 %	−18 %	−2 %	4 %	0 %	−1 %	−4 %
$8\times6\times8$, CNTD 750	71.1	2549.4	854.4	3414.9	167474	45328	2353.3	634.6	28.0	1.4	3038	212824	2987.9
	5 %	−1 %	23 %	5 %	−5 %	−18 %	0 %	−8 %	−1 %	−3 %	1 %	−6 %	−2 %

8.10.2 Experiment 6b

Sub-experiment 6a analyzes the performance of the linear solver Chronos in the **fastest point**, i.e., the configuration of $32 \times 6 \times 8$ nodes.

Cubo_1772481.Ext_bin The comparison between profiles with and without COUNTDOWN for Experiment 6b on the *Cubo_1772481.Ext_bin* matrix reveals notable insights, particularly when considering COUNTDOWN 500 and COUNTDOWN 750 relative to the baseline configuration.

In terms of execution time, both COUNTDOWN 500 and COUNTDOWN 750 show an increases over the baseline, with COUNTDOWN 500 showing a $30\,\%$ increase and COUNTDOWN 750 showing a $11\,\%$ increase.

When analyzing application time, COUNTDOWN 500 shows a $2\,\%$ increase, while COUNTDOWN 750 shows a marginal $1\,\%$ increase over the baseline. This suggests a relatively stable application performance with the implementation of COUNTDOWN.

For MPI time, there is a significant $86\,\%$ increase with COUNTDOWN 500 and a $31\,\%$ increase with COUNTDOWN 750 compared to the baseline. This indicates a significant impact on parallel computing efficiency for both configurations.

In terms of power consumption, package power (CPU) shows a significant $21\,\%$ increase with COUNTDOWN 500 and a $6\,\%$ increase with COUNTDOWN 750 compared to the baseline. Conversely, DRAM power consumption shows a $20\,\%$ increase with COUNTDOWN 500 and a $7\,\%$ increase with COUNTDOWN 750.

When analyzing IPC, both COUNTDOWN 500 and COUNTDOWN 750 show improvements over the baseline, with increases of $7\,\%$ and $1\,\%$, respectively. These improvements suggest an improvement in instruction throughout with the implementation of COUNTDOWN.

These results are supported by the graphical results of which are in appendix (fig. B.13) and all data displayed in table 8.21.

Wing_4538k.csr.Ext_bin Comparing the profiles with and without COUNTDOWN for Experiment 6b on the *Wing_4538k.csr.Ext_bin* matrix reveals significant insights, particularly when considering COUNTDOWN 500 and COUNTDOWN 750 relative to the baseline configuration.

In terms of execution time, both COUNTDOWN 500 and COUNTDOWN 750 show increases over the baseline, with COUNTDOWN 500 showing a $27\,\%$ increase and COUNTDOWN 750 showing a $12\,\%$ increase. However, compared to the baseline configuration, the increase in execution time of the low-power configuration is $2421\,\%$ and the increase in energy is $1643\,\%$.

When analyzing application time, COUNTDOWN 500 shows a marginal $1\,\%$ increase, while COUNTDOWN 750 shows any significant change from the baseline. This suggests relatively stable application performance with the implementation of COUNTDOWN.

For MPI time, there is a significant $69\,\%$ increase with COUNTDOWN 500 and a $33\,\%$ increase with COUNTDOWN 750 compared to the baseline. This indicates a significant impact on parallel computing efficiency with both configurations.

In terms of power consumption, package power (CPU) shows a significant $21\,\%$ increase with COUNTDOWN 500 and a $8\,\%$ increase with COUNTDOWN 750 compared to

the baseline. Conversely, DRAM power consumption shows a 19 % increase with COUNTDOWN 500 and a 9 % increase with COUNTDOWN 750.

When analyzing IPC, both COUNTDOWN 500 and COUNTDOWN 750 show improvements over the baseline, with increases of 6 % and 1 %, respectively. These improvements suggest improved instruction throughout with the implementation of COUNTDOWN.

In summary, the introduction of COUNTDOWN 500 and COUNTDOWN 750 results in increases in execution time, MPI time, and package power (CPU) compared to the baseline. However, the improvements in IPC indicate improved processor efficiency despite fluctuations in energy consumption and MPI time.

These results are supported by the graphical results of which are in appendix (fig. B.14) and all data displayed in table 8.22.

Table 8.21: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6b on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	6.6	860.3	420.7	1282.9	59426	10340	8890.0	1530.6	57.0	1.9	3086	69956	10473.2
$32 \times 6 \times 8$, 1.0 GHz	106.1	6228.9	14140.2	20376.4	757681	101500	7138.3	956.6	56.4	2.7	3099	859141	8095.2
	1496 %	624 %	3261 %	1488 %	−1175 %	−882 %	20 %	38 %	1 %	−43 %	0 %	−1128 %	23 %
$32 \times 6 \times 8$, CNTD	6.9	860.1	471.1	1331.5	60968	10434	8848.0	1511.6	56.9	1.9	3089	71405	10358.3
	4 %	0 %	12 %	4 %	−3 %	−1 %	0 %	1 %	0 %	−1 %	0 %	−2 %	1 %
$32 \times 6 \times 8$, CNTD 500	8.6	876.7	784.4	1663.4	71645	12446	8318.8	1410.2	57.0	2.0	2895	84570	9794.9
	30 %	2 %	86 %	30 %	−21 %	−20 %	6 %	8 %	0 %	−7 %	6 %	−21 %	6 %
$32 \times 6 \times 8$, CNTD 750	7.4	866.7	550.8	1420.8	63238	11062	8552.0	1469.3	57	1.9	2925	73980	10048.6
	11 %	1 %	31 %	11 %	−6 %	−7 %	4 %	4 %	0 %	−1 %	5 %	−6 %	4 %

Table 8.22: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6b on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$32 \times 6 \times 8$, Baseline	19.5	2301.9	1448.2	3754.2	182891	38903	9386.7	1990.7	50.5	1.9	3077	221726	11379.8
$32 \times 6 \times 8$, 1.0 GHz	492.2	22742.4	71759.6	94501.3	3433389	451101.6	6981.4	875.9	45.0	2.7	3099	3865384	7857.3
	2421 %	888 %	4855 %	2417 %	−1778 %	−1008 %	26 %	56 %	11 %	−42 %	−1 %	−1643 %	31 %
$32 \times 6 \times 8$, CNTD	20.7	2287.9	1690.4	3975.6	194304	32853	9427.1	1590.9	46.9	1.9	3082	227157	11022.1
	6 %	−1 %	17 %	6 %	−6 %	16 %	0 %	20 %	7 %	−1 %	0 %	−2 %	3 %
$32 \times 6 \times 8$, CNTD 500	24.8	2313.6	2451.3	4760.5	221034	46312	8939.5	1872.3	50.6	2.0	2940	267210	10810.9
	27 %	1 %	69 %	27 %	−21 %	−19 %	5 %	6 %	0 %	−6 %	4 %	−21 %	5 %
$32 \times 6 \times 8$, CNTD 750	21.9	2301.8	1931.7	4221.7	198060	42306	9034.0	1928.3	50.6	1.9	2947	240737	10938.9
	12 %	0 %	33 %	12 %	−8 %	−9 %	4 %	3 %	0 %	−1 %	4 %	−9 %	4 %

8.10.3 Experiment 6c

The last sub-experiment analyzes the intermediate case between the greenest and the fastest case, i.e., the $16 \times 6 \times 8$ configuration, since we want to analyze how COUNTDOWN behaves in this situation.

Cubo_1772481.Ext_bin The comparison, as shown in table 8.23, illustrates notable trends. First, in terms of execution time, both COUNTDOWN 500 and COUNTDOWN 750 show an increases over the baseline, with COUNTDOWN 500 showing a $16\,\%$ increase and COUNTDOWN 750 showing a $8\,\%$ increase.

In terms of application time, COUNTDOWN 500 shows a marginal $1\,\%$ increase, while COUNTDOWN 750 shows no significant change from the baseline. This suggests a relatively stable application performance with the implementation of COUNTDOWN.

However, in terms of MPI time, both COUNTDOWN 500 and COUNTDOWN 750 show significant increases compared to the baseline, indicating a significant impact on communication efficiency.

In addition, the energy consumption metrics show different trends. Package (CPU) power shows significant increases with both COUNTDOWN 500 and COUNTDOWN 750, while DRAM power shows mixed results.

In addition, both COUNTDOWN 500 and COUNTDOWN 750 show improvements in IPC compared to the baseline.

These results are supported by the graphical results of which are in appendix (fig. B.15) and all data displayed in table 8.23.

Wing_4538k.csr.Ext_bin In Experiment 6c, a comparison was made between different configurations of the *Wing_4538k.csr.Ext_bin* matrix, with and without the COUNTDOWN feature enabled.

COUNTDOWN 500 shows a slight decrease in execution time ($-3\,\%$) and application time ($-2\,\%$), while COUNTDOWN 750 shows similar trends. However, there are significant decreases in MPI time for both COUNTDOWN 500 ($-4\,\%$) and COUNTDOWN 750 ($-3\,\%$).

In terms of energy consumption, configurations with COUNTDOWN generally show mixed results. While there are decreases in some metrics, such as package power and average power, there are increases in others, such as DRAM power. These fluctuations indicate a complex interaction between COUNTDOWN and power utilization. While COUNTDOWN 500 shows improvements in execution time and application metrics, COUNTDOWN 750 stands out for its significant energy savings, as evidenced by the decrease in package energy and average power. Specifically, for COUNTDOWN 750, there is a $9\,\%$ decrease in package energy and a $6\,\%$ decrease in average power compared to the baseline.

In addition, improvements in IPC are observed with COUNTDOWN configurations compared to the baseline, indicating improved processor efficiency. However, there is a slight decrease in average CPU frequency for COUNTDOWN configurations compared to the baseline.

These results are supported by the graphical results of which are in appendix (fig. B.16) and all data displayed in table 8.24.

Table 8.23: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6c on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$16 \times 6 \times 8$, Baseline	11.5	850.4	256.8	1109.9	52574.5	13054	4510.3	948.0	39.2	1.7	3089	64141.5	5488.3
$16 \times 6 \times 8$, 1.0 GHz	103.0	3547.4	6300.8	9892.9	381396	40724	3708.3	395.9	35.9	2.7	3099	422079.5	5104.3
	794 %	317 %	2353 %	791 %	−625 %	−212 %	18 %	58 %	8 %	−53 %	0 %	−558 %	25 %
$16 \times 6 \times 8$, CNTD	11.6	841.5	274.0	1120.0	53023	12981	4517.6	1116.2	40.0	1.7	3086	65979	5635.3
	1 %	−1 %	7 %	1 %	−1 %	1 %	0 %	−18 %	−1 %	1 %	0 %	−3 %	−3 %
$16 \times 6 \times 8$, CNTD 500	13.4	858.6	428.9	1289.6	58356.5	14582.5	4319.1	951.3	39.1	1.8	2944	71373	5260.1
	16 %	1 %	67 %	16 %	−11 %	−12 %	4 %	0 %	0 %	−5 %	5 %	−11 %	4 %
$16 \times 6 \times 8$, CNTD 750	12.4	854.0	337.0	1192.0	54137	13397.5	4393.3	944.6	39.2	1.7	2956	66334	5352.3
	8 %	0 %	31 %	7 %	−3 %	−3 %	3 %	0 %	0 %	−1 %	4 %	−3 %	2 %

Table 8.24: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 6c on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$16 \times 6 \times 8$, Baseline	35.9	2455.2	965.5	3448.5	165623	46970	4621.2	1300.0	35.9	1.7	3074	212333	5889.0
$16 \times 6 \times 8$, 1.0 GHz	420.4	8983.9	31380.8	40361.5	153386	21491	3647.0	392.3	30.3	2.7	3099	169877	4039.0
	1072 %	266 %	3150 %	1070 %	−826 %	−251 %	21 %	70 %	16 %	−60 %	−1 %	−700 %	31 %
16×6×8, CNTD	34.9	2406.5	929.7	3356.8	165342	29773	4735.7	855.2	31.2	1.6	3053	195172	5588.8
	−3 %	−2 %	−4 %	−3 %	0 %	37 %	−2 %	34 %	13 %	2 %	1 %	8 %	5 %
16×6×8, CNTD 500	41.4	2462.5	1494.7	3976.3	187180	50873.5	4528.7	1212.4	35.9	1.7	2971.5	236321	5586.4
	15 %	0 %	55 %	15 %	−13 %	−8 %	2 %	7 %	0 %	−5 %	3 %	−11 %	5 %
16×6×8, CNTD 750	34.8	2404.1	934.4	3344.1	163803	29674.5	4707.9	852.6	31.2	1.6	2990	193583	5557.0
	−3 %	−2 %	−3 %	−3 %	1 %	37 %	−2 %	34 %	13 %	4 %	3 %	9 %	6 %

8.11 Experiments 7 and 8

In Experiments 7 and 8, comprehensive power analyses were performed using COUNTDOWN, analyzing the frequencies not analyzed in previous experiments, i.e., 3.1 GHz (baseline) to 1.0 GHz (called low power in previous experiments), which we nevertheless report in order to have a complete comparison. Multiple runs were then performed with CPU frequencies between 1.2 GHz and 2.8 GHz, with a step size of 0.2 GHz.

The purpose of this experiments is to evaluate the impact of frequency reduction on energy efficiency and performance by allowing evaluation of energy saving or non-saving trends across different frequency settings. By systematically varying the maximum frequency within this range. In addition, this experiments sheds light on the extent to which the Linux scheduler accepts downclocking requests, which we have already mentioned are not well respected.

COUNTDOWN with Analysis Only (Baseline) COUNTDOWN run in parallel with the program, without any analysis of slack or power-saving algorithms. This scenario was used as a baseline to examine the basic functionality of COUNTDOWN.

COUNTDOWN with low power configuration (1.0 GHz) COUNTDOWN is set to keep the CPU frequency at 1.0 GHz, although the power saving algorithm is not enabled, since the Linux kernel will not follow our instructions for long. We can also call this execution mode as COUNTDOWN enabled without timer, i.e. as soon as there is an MPI call, the CPU sets to low power.

COUNTDOWN without slack (CNTD) Execution of the power saving algorithm without including the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization.

COUNTDOWN with slack and a callback delay of $750\,\mu$s **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to $500\,\mu$s.

COUNTDOWN with slack and a callback delay of $750\,\mu$s **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to $750\,\mu$s.

8.11.1 Experiment 7

Experiment 7 performs the analyses with a low configuration of iterations, a little more than 1000, so that both the effect generated by the setup and preconditioning and that generated by the iterative computation step can be analyzed in a more balanced manner, as opposed to the normal unbalanced distribution as described in section 3.5.

Cubo_1772481.Ext_bin In the case of the *Cubo_1772481.Ext_bin* matrix, the results showed a notable trend: as the CPU frequency decreased, there was a pronounced increase in execution times. For instance, the transition from the baseline frequency of 3.1 GHz to 1.0 GHz led to a 350 % increase in execution time. This shift was particularly noticeable in both application execution time and MPI time, which surged by 134 % and 1143 %, respectively. These findings underscore the substantial impact that CPU frequency adjustments can have on computational performance. Furthermore, a closer examination of the data reveals compelling insights into the performance at specific frequencies. At 2.2 GHz, the execution time exhibited a significant 217 % increase compared to the baseline configuration, highlighting the diminishing returns associated with lower frequencies. Similarly, at 2.8 GHz, the execution time experienced a notable 334 % increase, emphasizing the trade-off between power consumption and computational speed. Conversely, the 1.6 GHz configuration showcased a 328 % increase in execution time, indicative of the profound impact of lower frequencies on computational efficiency.

These results are supported by the graphical results of which are in appendix (fig. B.17) and all data displayed in table 8.25.

Wing_4538k.csr.Ext_bin In the case of the matrix *Wing_4538k.csr.Ext_bin*, similar trends were observed regarding the impact of varying CPU frequency on execution times. Transitioning from the baseline frequency of 3.1 GHz to 1.0 GHz resulted in a notable decrease in execution time across various metrics. For example, the execution time decreased by 4 %, 1 %, and 16 % for the application, MPI, and total time, respectively. However, the energy consumption saw a slight increase of 3 %, primarily attributed to the reduction in frequency. At 2.8 GHz, while there was a 6 % reduction in execution time, the energy consumption increased by 3 %, demonstrating a trade-off between performance and energy efficiency. Similarly, at 1.6 GHz, there was a 60 % increase in execution time, accompanied by a 50 % reduction in energy consumption, showcasing the significant impact of frequency adjustments on power usage. Moreover, the application time relative to the baseline configuration fluctuated, ranging from a 4 % decrease at 2.8 GHz to a 69 % increase at 1.6 GHz, highlighting the sensitivity of application performance to CPU frequency variations.

These results are supported by the graphical results of which are in appendix (fig. B.18) and all data displayed in table 8.26.

Table 8.25: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 7 on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	EXE time [s]	APP time [s]	MPI time [s]	TOT time [s]	Energy PKG [J]	Energy DRAM [J]	AVG Power PKG [W]	AVG Power DRAM [W]	MAX Memory usage [GB]	AVG IPC	AVG CPU frequency [MHz]	Energy [J]	AVG Power [W]
$8 \times 6 \times 8$, Baseline	13.4	522.1	124.9	647.2	30618	6410	2276.0	476.3	32.5	1.6	3089	37031	2752.8
$8 \times 6 \times 8$, 1.0 GHz	13.2	516.1	116.9	633.2	29747	5085	2259.6	386.3	32.0	1.6	3084	34831	2645.9
	−2 %	−1 %	−6 %	−2 %	3 %	21 %	1 %	19 %	1 %	0 %	0 %	6 %	4 %
$8 \times 6 \times 8$, CNTD	13.5	521.5	125.7	650.4	30228	7704	2241.8	570.5	33.0	1.6	3097	37909	2812.8
	1 %	0 %	1 %	1 %	1 %	−20 %	2 %	−20 %	−2 %	−1 %	0 %	−2 %	−2 %
$8 \times 6 \times 8$, CNTD 500	14.4	524.8	167.4	693.6	32112	6721	2227.1	466.0	32.5	1.7	2984	38840	2694.1
	7 %	1 %	34 %	7 %	−5 %	−5 %	2 %	2 %	0 %	−2 %	3 %	−5 %	2 %
$8 \times 6 \times 8$, CNTD 750	14.1	522.4	150.2	676.5	31339	6527	2231.3	463.9	32.5	1.6	2993	37877	2695.2
	5 %	0 %	20 %	5 %	−2 %	−2 %	2 %	3 %	0 %	−2 %	3 %	−2 %	2 %
$8 \times 6 \times 8$ 1.6 GHz	57.5	1219.2	1545.3	2762.3	108357.5	12472	1885.3	216.8	31.7	2.5	3099	120805	2102.1
	328 %	133 %	1137 %	327 %	−254 %	−95 %	17 %	54 %	2 %	−54 %	0 %	−226 %	24 %
$8 \times 6 \times 8$ 1.8 GHz	47.6	999.7	1287.5	2287.0	94672.5	13039.5	1987.8	273.8	32.1	2.5	3097	107709.5	2261.9
	254 %	91 %	930 %	253 %	−209 %	−103 %	13 %	43 %	1 %	−52 %	0 %	−191 %	18 %
$8 \times 6 \times 8$ 2.0 GHz	53.5	1108.2	1455.5	2570.0	104571.5	17010.5	1956.2	318.0	32.2	2.5	3098	121582	2273.7
	298 %	112 %	1065 %	297 %	−242 %	−165 %	14 %	33 %	1 %	−54 %	0 %	−228 %	17 %
$8 \times 6 \times 8$ 2.2 GHz	42.6	838.0	1212.4	2046.6	84122	14482	1973.0	339.9	32.3	2.4	3098	98592.5	2313.3
	217 %	60 %	870 %	216 %	−175 %	−126 %	13 %	29 %	1 %	−52 %	0 %	−166 %	16 %
$8 \times 6 \times 8$ 2.4 GHz	51.7	1074.1	1406.2	2484.3	101142	16359.5	1954.6	316.3	32.2	2.5	3098	117501.5	2271.1
	285 %	106 %	1025 %	284 %	−230 %	−155 %	14 %	34 %	1 %	−54 %	0 %	−217 %	18 %
$8 \times 6 \times 8$ 2.6 GHz	50.6	1064.4	1365.7	2431.8	97702	11368.5	1925.6	224.4	31.7	2.4	3098	109070.5	2150.5
	277 %	104 %	993 %	276 %	−219 %	−77 %	15 %	53 %	2 %	−52 %	0 %	−195 %	22 %
$8 \times 6 \times 8$ 2.8 GHz	58.4	1283.1	1532.3	2805.6	109808	34594.5	1874.3	592.3	33.9	2.6	3098	144402.5	2467.3
	334 %	146 %	1126 %	334 %	−259 %	−440 %	18 %	−24 %	−5 %	−60 %	0 %	−290 %	10 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 500	77.7	1363.4	2367.6	3731.3	131005.5	16099	1685.8	207.1	31.7	2.6	2957	147104.5	1892.6
	478 %	161 %	1795 %	477 %	−328 %	−151 %	26 %	57 %	2 %	−58 %	4 %	−297 %	31 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 500	64.5	1140.0	1945.6	3099.1	115485.5	16923	1789.3	262.2	32.0	2.6	2879.5	132440	2051.0
	380 %	118 %	1457 %	379 %	−277 %	−164 %	21 %	45 %	1 %	−57 %	7 %	−258 %	25 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 500	72.4	1213.8	2266.9	3479.5	126614	22217	1747.3	306.9	32.2	2.6	2945.5	148831	2054.0
	439 %	132 %	1714 %	438 %	−314 %	−247 %	23 %	36 %	1 %	−59 %	5 %	−302 %	25 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 500	49.7	883.8	1497.2	2388.2	90413	16566.5	1818.0	333.5	32.3	2.5	2837.5	107001	2151.7
	270 %	69 %	1098 %	269 %	−195 %	−158 %	20 %	30 %	1 %	−54 %	8 %	−189 %	22 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 500	68.2	1164.9	2118.2	3274.1	119678	20898.5	1759.0	306.5	32.2	2.6	2950	140576.5	2064.8
	407 %	123 %	1595 %	406 %	−291 %	−226 %	23 %	36 %	1 %	−58 %	4 %	−280 %	25 %
$8 \times 6 \times 8$ 2.6 GHz, CNTD 500	67.7	1176.4	2087.4	3253.2	118564	14450.5	1740.4	213.2	31.7	2.6	2981.5	133014.5	1953.6
	404 %	125 %	1571 %	403 %	−287 %	−125 %	24 %	55 %	2 %	−57 %	3 %	−259 %	29 %
$8 \times 6 \times 8$ 2.8 GHz, CNTD 500	75.0	1314.8	2283.4	3601.4	127373.5	43782.5	1700.6	583.9	34.0	2.7	2948.5	171156	2285.6
	458 %	152 %	1728 %	456 %	−316 %	−583 %	25 %	−23 %	−5 %	−65 %	5 %	−362 %	17 %
$8 \times 6 \times 8$ 1.2 GHz, CNTD 750	61.5	1152.1	1798.6	2953.9	110029	22479.5	1781.8	365.5	32.5	2.6	2828.5	132589	2149.3
	357 %	121 %	1340 %	356 %	−259 %	−251 %	22 %	23 %	0 %	−57 %	8 %	−258 %	22 %
$8 \times 6 \times 8$ 1.4 GHz, CNTD 750	66.1	1190.4	1969.6	3175.1	116680	17269.5	1768.0	261.2	32	2.6	2855	133901	2029.3
	392 %	128 %	1476 %	391 %	−281 %	−169 %	22 %	45 %	1 %	−56 %	8 %	−262 %	26 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 750	59.5	1089.1	1757.0	2858.9	103750	13034	1739.2	218.9	31.7	2.5	2863	116784	1959.0
	343 %	109 %	1306 %	342 %	−239 %	−103 %	24 %	54 %	2 %	−54 %	7 %	−215 %	29 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 750	58.5	1080.3	1733.5	2809.7	106233.5	15801.5	1805.7	265.4	32	2.5	2886	122800	2069.6
	335 %	107 %	1287 %	334 %	−247 %	−147 %	21 %	44 %	1 %	−55 %	7 %	−232 %	25 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 750	69.5	1189.2	2125.7	3336.7	121540	21413	1763	308.1	32.2	2.6	2933	142953	2071.1
	417 %	128 %	1601 %	416 %	−297 %	−234 %	23 %	35 %	1 %	−58 %	5 %	−286 %	25 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 750	46.6	851.8	1361.9	2237.4	85106	15592	1836.2	336.4	32.3	2.5	2817	100776	2175.0
	246 %	63 %	990 %	246 %	−178 %	−143 %	19 %	29 %	1 %	−52 %	9 %	−172 %	21 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 750	67.7	1261.2	2018.2	3250.3	117372	20928	1729.7	308.4	32.2	2.6	2804	138501	2039.3
	403 %	142 %	1515 %	402 %	−283 %	−226 %	24 %	35 %	1 %	−57 %	9 %	−274 %	26 %
$8 \times 6 \times 8$ 2.6 GHz, CNTD 750	62.0	1201.6	1875.4	2977.1	109736	14672	1762.7	216.3	31.8	2.6	2996	131938	1980.4
	361 %	130 %	1401 %	360 %	−258 %	−129 %	23 %	55 %	2 %	−57 %	3 %	−256 %	28 %
$8 \times 6 \times 8$ 2.8 GHz, CNTD 750	13.4	522.1	122.3	644.3	30185.5	6372.5	2250.8	475.8	32.5	1.6	3092	36558	2727.0
	0 %	0 %	−2 %	0 %	1 %	1 %	1 %	0 %	0 %	0 %	0 %	1 %	1 %

Table 8.26: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 7 on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	EXE time [s]	APP time [s]	MPI time [s]	TOT time [s]	Energy PKG [J]	Energy DRAM [J]	AVG Power PKG [W]	AVG Power DRAM [W]	MAX Memory usage [GB]	AVG IPC	AVG CPU frequency [MHz]	Energy [J]	AVG Power [W]
$8 \times 6 \times 8$, Baseline	12.0	457.9	117.2	575.7	27064	4285	2264.2	358.3	27.0	1.6	3084	31351	2622.4
$8 \times 6 \times 8$, 1.0 GHz	11.5	452.8	103.1	556.0	26217.5	3776.5	2267.6	327.1	26.4	1.6	3090.5	29997	2594.6
	−3 %	−1 %	−12 %	−3 %	3 %	12 %	0 %	9 %	1 %	1 %	0 %	4 %	1 %
$8 \times 6 \times 8$, CNTD	13.0	462.1	163.4	625.3	28441	4525	2192	348.6	27.0	1.6	2966	32966	2540.0
	9 %	1 %	39 %	9 %	−5 %	−6 %	3 %	3 %	0 %	−3 %	4 %	−5 %	3 %
$8 \times 6 \times 8$, CNTD 500	12.2	456.9	131.0	587.5	26484	3595	2168.0	294.6	26.4	1.6	2973	30088	2463.2
	2 %	0 %	12 %	2 %	2 %	16 %	4 %	18 %	1 %	1 %	4 %	4 %	6 %
$8 \times 6 \times 8$, CNTD 750	12.1	457.4	124.3	582.9	26696	3984	2204.6	328.8	26.4	1.6	2970	30682	2533.7
	1 %	0 %	6 %	1 %	1 %	7 %	3 %	8 %	1 %	1 %	4 %	2 %	3 %
$8 \times 6 \times 8$ 1.4 GHz	24.1	503.0	656.8	1160.0	49931	6221.5	2073.9	257.8	26.4	2.2	3095.5	56152.5	2330.2
	102 %	10 %	460 %	101 %	−84 %	−45 %	8 %	28 %	1 %	−39 %	0 %	−79 %	11 %
$8 \times 6 \times 8$ 1.6 GHz	19.1	477.2	437.1	918.5	40522.5	5185.5	2136.6	271.5	26.4	2.0	3084.5	45713.5	2408.0
	60 %	4 %	273 %	60 %	−50 %	−21 %	6 %	24 %	1 %	−29 %	0 %	−46 %	8 %
$8 \times 6 \times 8$ 1.8 GHz	51.6	935.2	1545.5	2477.9	99818	11326.5	1933.2	219.3	26.4	2.4	3099	111124	2152.6
	332 %	104 %	1219 %	330 %	−269 %	−164 %	15 %	39 %	1 %	−53 %	0 %	−254 %	18 %
$8 \times 6 \times 8$ 2.0 GHz	15.6	465.2	286.4	751.1	34006	4583	2174.0	293.5	26.4	1.9	3092.5	38589	2464.6
	31 %	2 %	144 %	30 %	−26 %	−7 %	4 %	18 %	1 %	−18 %	0 %	−23 %	6 %
$8 \times 6 \times 8$ 2.2 GHz	45.9	776.0	1431.3	2203.7	88050.5	10394.5	1917.8	226.4	26.4	2.4	3098	98445	2144.5
	284 %	69 %	1121 %	283 %	−225 %	−143 %	15 %	37 %	1 %	−53 %	0 %	−214 %	18 %
$8 \times 6 \times 8$ 2.4 GHz	11.5	453.7	98.7	552.5	26073.5	3660.5	2269.	318.7	26.4	1.6	3084.5	29734	2587.4
	−4 %	−1 %	−16 %	−4 %	4 %	15 %	0 %	11 %	1 %	2 %	0 %	5 %	1 %
$8 \times 6 \times 8$ 2.6 GHz	11.5	456.7	95.2	553.1	26135	4331.5	2275.9	376.9	26.7	1.6	3079	30465.5	2652.8
	−4 %	0 %	−19 %	−4 %	3 %	−1 %	−1 %	−5 %	0 %	3 %	0 %	3 %	−1 %
$8 \times 6 \times 8$ 2.8 GHz	11.5	454.3	98.8	554.3	26313.5	3735.5	2290.8	324.1	26.4	1.6	3087	30042.5	2615.2
	−4 %	−1 %	−16 %	−4 %	3 %	13 %	−1 %	10 %	1 %	2 %	0 %	4 %	0 %
$8 \times 6 \times 8$ 1.4 GHz, CNTD 500	30.1	576.1	869.1	1445.5	56290	7385	1866.9	245.3	26.4	2.3	2515	63686.5	2111.6
	152 %	26 %	642 %	151 %	−108 %	−72 %	18 %	32 %	1 %	−43 %	18 %	−103 %	19 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 500	20.2	520.5	453.9	972.7	40248.5	5414.5	1982.6	267.6	26.4	2.0	2525.5	45663	2247.6
	69 %	14 %	287 %	69 %	−49 %	−26 %	12 %	25 %	1 %	−29 %	18 %	−46 %	14 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 500	74.6	1052.9	2533.4	3582.5	127500.5	15623.5	1711.2	209.5	26.4	2.6	2962.5	143087.5	1920.7
	524 %	130 %	2062 %	522 %	−371 %	−265 %	24 %	42 %	1 %	−60 %	4 %	−356 %	27 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 500	19.5	502.4	434.1	936.6	38678	5332	1984.7	273.6	26.4	2.0	2572	44003.5	2258.3
	63 %	10 %	270 %	63 %	−43 %	−24 %	12 %	24 %	1 %	−25 %	17 %	−40 %	14 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 500	57.6	834.1	1939.9	2768.2	99150.5	12599	1718.3	218.5	26.4	2.5	2846	111730.5	1936.3
	382 %	82 %	1555 %	381 %	−266 %	−194 %	24 %	39 %	1 %	−57 %	8 %	−256 %	26 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 500	13.6	472.4	181.3	653.1	28866.5	4040	2120.7	297.5	26.4	1.7	2711	32906.5	2417.7
	14 %	3 %	55 %	13 %	−7 %	6 %	6 %	17 %	1 %	−6 %	12 %	−5 %	8 %
$8 \times 6 \times 8$ 2.6 GHz, CNTD 500	13.2	468.6	165.4	635.3	28369	4734.5	2150.3	358.7	27.0	1.7	2797	33095.5	2509.7
	10 %	2 %	41 %	10 %	−5 %	−10 %	5 %	0 %	0 %	−4 %	9 %	−6 %	4 %
$8 \times 6 \times 8$ 2.8 GHz, CNTD 500	13.1	463.5	166.1	630.1	28605	4024	2181.5	308.0	26.4	1.6	2842.5	32632	2489.6
	10 %	1 %	42 %	9 %	−6 %	6 %	4 %	14 %	1 %	−4 %	8 %	−4 %	5 %
$8 \times 6 \times 8$ 1.4 GHz, CNTD 750	28.4	566.5	797.2	1363.8	53531	7062.5	1893.5	248.8	26.4	2.2	2458.5	60593.5	2141.0
	137 %	24 %	580 %	137 %	−98 %	−65 %	16 %	31 %	1 %	−41 %	20 %	−93 %	18 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 750	29.7	563.4	861.5	1425.7	55325	7115	1867.2	239.8	26.4	2.3	2550	62449.5	2105.9
	148 %	23 %	635 %	148 %	−104 %	−66 %	18 %	33 %	1 %	−43 %	17 %	−99 %	20 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 750	74.7	1048.8	2528.5	3586.4	127771	15647	1710.7	209.5	26.4	2.6	2974	143418	1920.2
	525 %	129 %	2057 %	523 %	−372 %	−265 %	24 %	42 %	1 %	−60 %	4 %	−357 %	27 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 750	19.3	499.8	430.5	930.0	38789	5301	2000.7	274.0	26.4	2.0	2621.5	44090	2272.7
	62 %	9 %	267 %	62 %	−43 %	−24 %	12 %	24 %	1 %	−26 %	15 %	−41 %	13 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 750	57.5	830.3	1941.9	2763.4	99347	12576	1727.0	219.0	26.4	2.5	2889	111970	1945.2
	381 %	81 %	1557 %	380 %	−267 %	−193 %	24 %	39 %	1 %	−57 %	6 %	−257 %	26 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 750	13.1	469.3	163.9	632.4	28163.5	3966	2138.5	301.6	26.4	1.7	2723.5	32140.5	2440.8
	10 %	2 %	40 %	10 %	−4 %	7 %	6 %	16 %	1 %	−6 %	12 %	−3 %	7 %
$8 \times 6 \times 8$ 2.6 GHz, CNTD 750	12.8	466.6	146.7	614.0	27555.5	4630.5	2161.2	362.8	26.7	1.6	2830.5	32196.5	2524.6
	7 %	2 %	25 %	7 %	−2 %	−8 %	5 %	−1 %	0 %	−3 %	8 %	−3 %	4 %
$8 \times 6 \times 8$ 2.8 GHz, CNTD 750	12.7	459.7	148.6	610.1	27810	3948	2194.0	311.3	26.4	1.6	2894.5	31758	2504.3
	6 %	0 %	27 %	6 %	−3 %	8 %	3 %	13 %	1 %	−3 %	6 %	−1 %	5 %

8.11.2 Experiment 8

Experiment 8 is similar to Experiment 7, however it is performed with a much higher number of iterations to evaluate the complete execution of the linear solver with COUNTDOWN.

Cubo_1772481.Ext_bin When we increase the CPU frequency to 1.6 GHz, we see a significant improvement in execution time compared to the baseline. However, this improvement comes at the cost of higher energy and power consumption. The relative execution time of the application decreases by about 50 %, indicating a significant increase in performance, but energy consumption and power usage increase by about 50 %, highlighting the trade-off between performance and energy efficiency.

Moving to a CPU frequency of 2.2 GHz, we see a further reduction in execution time, indicating improved speed. However, this increase in speed is accompanied by even higher energy consumption and power usage compared to the 1.6 GHz configuration. Despite the higher power consumption, the performance gains are substantial, with execution time reduced by approximately 60 %.

These results are supported by the graphical results of which are in appendix (fig. B.20) and all data displayed in table 8.28.

Wing_4538k.csr.Ext_bin At a lower CPU frequency of 1.6 GHz, execution times are generally longer compared to higher frequencies. This is to be expected, as lower frequencies result in slower processing speeds. However, energy and power consumption are comparatively lower at this frequency.

Increasing the CPU frequency to 2.2 GHz significantly reduces execution times in all configurations. This improvement in speed comes with a corresponding increase in energy and power consumption, as the processor runs at a higher frequency and therefore consumes more power.

Moving to the highest frequency of 2.8 GHz results in the fastest execution times of the configurations tested. However, this comes at the cost of significantly higher power and energy consumption compared to lower frequencies. The trade-off between performance and energy efficiency becomes more pronounced at this frequency, as the processor operates at its maximum potential and it consumes significant power to achieve peak performance.

These results are supported by the graphical results of which are in appendix (fig. B.19) and all data displayed in table 8.27.

Table 8.27: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 8 on the *Cubo_1772481.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	EXE time [s]	APP time [s]	MPI time [s]	TOT time [s]	Energy PKG [J]	Energy DRAM [J]	AVG Power PKG [W]	AVG Power DRAM [W]	MAX Memory usage [GB]	AVG IPC	AVG CPU frequency [MHz]	Energy [J]	AVG Power [W]
$8 \times 6 \times 8$, Baseline	19.808	780.829	167.653	952.189	45641	11227	2307.65	559.91	32.78	1.52	3080	56744	2835.9
$8 \times 6 \times 8$, 1.0 GHz	106.217	2136.306	2967.131	5099.825	203099	22619	1913.34	212.88	31.73	2.52	3099	225686	2126.91
	436 %	174 %	1670 %	436 %	−345 %	−101 %	17 %	62 %	3 %	−66 %	−1 %	−298 %	25 %
$8 \times 6 \times 8$, CNTD	19.772	782.424	170.506	951.355	45853	10522	2318.27	532.04	32.5	1.51	3070	56380	2850.08
	0 %	0 %	2 %	0 %	0 %	6 %	0 %	5 %	1 %	1 %	0 %	1 %	−1 %
$8 \times 6 \times 8$, CNTD 500	21.464	786.672	246.021	1031.988	48656	11890	2266.81	546.13	32.78	1.55	2988	60481	2799.29
	8 %	1 %	47 %	8 %	−7 %	−6 %	2 %	2 %	0 %	−2 %	3 %	−7 %	1 %
$8 \times 6 \times 8$, CNTD 750	20.5	781.831	204.091	985.733	46776	11474	2283.13	554.75	32.77	1.53	3011	58124	2819.8
	3 %	0 %	22 %	4 %	−2 %	−2 %	1 %	1 %	0 %	−1 %	2 %	−2 %	1 %
$8 \times 6 \times 8$ 1.4 GHz	97.905	1886.232	2811.3755	4700.8965	186103	22579	1899.675	230.685	31.73	2.53	3099	208681	2130.165
	394 %	142 %	1577 %	394 %	−308 %	−101 %	18 %	59 %	3 %	−66 %	−1 %	−268 %	25 %
$8 \times 6 \times 8$ 1.6 GHz	101.2085	2000.859	2841.9355	4859.725	197641	27194.5	1953.585	268.69	32.02	2.53	3099	224843	2222.51
	411 %	156 %	1595 %	410 %	−333 %	−142 %	15 %	52 %	2 %	−66 %	−1 %	−296 %	22 %
$8 \times 6 \times 8$ 1.8 GHz	70.1145	1318.6105	2046.581	3366.6675	136569	23914.5	1982.17	341.27	32.26	2.455	3098.5	160910.5	2323.305
	254 %	69 %	1121 %	254 %	−199 %	−113 %	14 %	39 %	2 %	−62 %	−1 %	−184 %	18 %
$8 \times 6 \times 8$ 2.0 GHz	101.4	1988.691	2880.37	4868.6465	199169	22943	1965.205	226.34	31.74	2.52	3099	222103.5	2191.545
	412 %	155 %	1618 %	411 %	−336 %	−104 %	15 %	60 %	3 %	−66 %	−1 %	−291 %	23 %
$8 \times 6 \times 8$ 2.2 GHz	102.197	2005.5195	2908.2755	4906.7395	194351.5	21736.5	1903.105	212.695	31.74	2.52	3099	216088	2115.765
	416 %	157 %	1635 %	415 %	−326 %	−94 %	18 %	62 %	3 %	−66 %	−1 %	−281 %	25 %
$8 \times 6 \times 8$ 2.4 GHz	86.41	1591.92	2552.9625	4148.9075	171560.5	20284.5	1985.405	234.755	31.74	2.495	3096.5	191845	2220.16
	336 %	104 %	1423 %	336 %	−276 %	−81 %	14 %	58 %	3 %	−64 %	−1 %	−238 %	22 %
$8 \times 6 \times 8$ 1.4 GHz, CNTD 500	120.4105	2077.7455	3723.708	5781.0475	208551	26823	1731.935	222.73	31.75	2.55	2925.5	235437.5	1954.7
	508 %	166 %	2121 %	507 %	−357 %	−139 %	25 %	60 %	3 %	−68 %	5 %	−315 %	31 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 500	128.2755	2209.527	3945.424	6158.6625	227429.5	33439.5	1772.34	260.63	32.005	2.57	2982.5	260827	2032.78
	548 %	183 %	2253 %	547 %	−398 %	−198 %	23 %	53 %	2 %	−69 %	3 %	−360 %	28 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 500	86.2775	1444.7645	2698.7165	4143.481	155568	28560.5	1820.445	331.16	32.265	2.515	2871.5	184128.5	2149.54
	336 %	85 %	1510 %	335 %	−241 %	−154 %	21 %	41 %	2 %	−65 %	7 %	−224 %	24 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 500	129.898	2160.2895	4086.413	6236.4105	231712.5	28216	1782.485	217.195	31.755	2.57	2985.5	259900.5	1999.79
	556 %	177 %	2337 %	555 %	−408 %	−151 %	23 %	61 %	3 %	−69 %	3 %	−358 %	29 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 500	133.404	2160.988	4264.4445	6404.932	228196.5	27145	1709.62	203.47	31.75	2.58	2955.5	255340.5	1913.455
	573 %	177 %	2444 %	573 %	−400 %	−142 %	26 %	64 %	3 %	−70 %	4 %	−350 %	33 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 500	105.2345	1696.675	3355.8395	5052.5145	191601.5	23815	1825.495	226.305	31.75	2.54	2917	215416.5	2052.735
	431 %	117 %	1902 %	431 %	−320 %	−112 %	21 %	60 %	3 %	−67 %	5 %	−280 %	28 %
$8 \times 6 \times 8$ 1.0 GHz, CNTD 750	61.1995	1270.978	1666.1375	2938.8245	113415.5	15361	1856.585	251.075	31.74	2.35	2606	128762	2106.86
	209 %	63 %	894 %	209 %	−148 %	−37 %	20 %	55 %	3 %	−55 %	15 %	−127 %	26 %
$8 \times 6 \times 8$ 1.2 GHz, CNTD 750	131.8275	2367.8675	3974.479	6329.229	230441.5	46677.5	1743.475	354.01	32.48	2.58	2983	277186.5	2097.55
	566 %	203 %	2271 %	565 %	−405 %	−316 %	24 %	37 %	1 %	−70 %	3 %	−388 %	26 %
$8 \times 6 \times 8$ 1.4 GHz, CNTD 750	113.517	1901.327	3578.208	5450.478	198967	24996	1750.28	219.62	31.75	2.55	2905	224019	1969.57
	473 %	144 %	2034 %	472 %	−336 %	−123 %	24 %	61 %	3 %	−68 %	6 %	−295 %	31 %
$8 \times 6 \times 8$ 1.6 GHz, CNTD 750	141.554	2423.057	4342.438	6796.155	244744	36354.5	1734.315	257.25	32.01	2.585	3009.5	281099.5	1990.955
	615 %	210 %	2490 %	614 %	−436 %	−224 %	25 %	54 %	2 %	−70 %	2 %	−395 %	30 %
$8 \times 6 \times 8$ 1.8 GHz, CNTD 750	118.6045	2004.116	3704.2185	5694.529	211105.5	26719.5	1776.5	222.61	31.755	2.56	2918.5	237427	1999.075
	499 %	157 %	2109 %	498 %	−363 %	−138 %	23 %	60 %	3 %	−68 %	5 %	−318 %	30 %
$8 \times 6 \times 8$ 2.0 GHz, CNTD 750	136.084	2239.209	4282.512	6533.426	235003	54230	1726.76	397.89	32.76	2.62	3017	289233	2117.05
	587 %	187 %	2454 %	586 %	−415 %	−383 %	25 %	29 %	0 %	−72 %	2 %	−410 %	25 %
$8 \times 6 \times 8$ 2.2 GHz, CNTD 750	124.386	2044.643	3886.678	5971.967	220576	32125	1774.41	261.25	32	2.57	2996	250363	2037.08
	528 %	162 %	2218 %	527 %	−383 %	−186 %	23 %	53 %	2 %	−69 %	3 %	−341 %	28 %
$8 \times 6 \times 8$ 2.4 GHz, CNTD 750	119.46	1967.651	3770.765	5735.549	211010	37683	1765.67	315.25	32.24	2.59	3002	248693	2074.74
	503 %	152 %	2149 %	502 %	−362 %	−236 %	23 %	44 %	2 %	−70 %	3 %	−338 %	27 %
$8 \times 6 \times 8$ 2.6 GHz, CNTD 750	113.4985	1880.62	3560.601	5449.5355	202567	27473	1802.34	252.305	31.865	2.555	2989.5	230040	2060.53
	473 %	141 %	2024 %	472 %	−344 %	−145 %	22 %	55 %	3 %	−68 %	3 %	−305 %	27 %

Table 8.28: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 8 on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	EXE time [s]	APP time [s]	MPI time [s]	TOT time [s]	Energy PKG [J]	Energy DRAM [J]	AVG Power PKG [W]	AVG Power DRAM [W]	MAX Memory usage [GB]	AVG IPC	AVG CPU frequency [MHz]	Energy [J]	AVG Power [W]
$8 \times 6 \times 8$, Baseline	67.5	2585.1	693.8	3244.5	158754.0	38261.5	2347.6	585.2	27.3	1.4	3068.0	200657.5	2933.8
$8 \times 6 \times 8$, 1.0 GHz	426.6	5890.7	14564.1	20476.6	784874.0	175395.0	1840.0	411.0	27.4	2.6	3099.0	960269.0	2251.2
	531 %	128 %	1999 %	531 %	−394 %	−358 %	22 %	30 %	0 %	−82 %	−1 %	−379 %	23 %
$8 \times 6 \times 8$, CNTD	72.4	2645.3	923.4	3477.4	168999.0	45754.0	2305.7	630.1	27.7	1.5	3043.0	214753.0	2937.9
	7 %	2 %	33 %	7 %	−6 %	−20 %	2 %	−8 %	−1 %	−4 %	1 %	−7 %	0 %
$8 \times 6 \times 8$, CNTD 500	66.3	2567.4	609.7	3184.3	156568.0	46040.5	2358.6	692.4	28.0	1.4	3072.0	202627.0	3047.8
	−2 %	−1 %	−12 %	−2 %	1 %	−20 %	0 %	−18 %	−2 %	4 %	0 %	−1 %	−4 %
$8 \times 6 \times 8$, CNTD 750	71.1	2549.4	854.4	3414.9	167474.0	45328.0	2353.3	634.6	27.7	1.5	3038.0	212824.0	2987.9
	5 %	−1 %	23 %	5 %	−5 %	−18 %	0 %	−8 %	−1 %	−3 %	1 %	−6 %	−2 %
$8 \times 6 \times 8$ 1.2 GHz, Baseline	354.1	4700.8	12296.8	16997.5	671398.0	114425.0	1896.2	333.0	26.9	2.5	3099.0	789321.0	2229.2
	424 %	82 %	1672 %	424 %	−323 %	−199 %	19 %	43 %	2 %	−79 %	−1 %	−293 %	24 %
$8 \times 6 \times 8$ 1.8 GHz, Baseline	418.5	5971.3	14125.4	20087.2	790687.5	95419.5	1888.1	228.5	26.5	2.5	3099.0	886017.0	2117.3
	519 %	131 %	1936 %	519 %	−398 %	−149 %	20 %	61 %	3 %	−78 %	−1 %	−342 %	28 %
$8 \times 6 \times 8$ 2.2 GHz, Baseline	459.2	7041.2	15004.1	22045.3	825830.0	272821.0	1801.4	594.8	28.5	2.6	3099.0	1097817.0	2395.9
	580 %	172 %	2063 %	579 %	−420 %	−613 %	23 %	−2 %	−4 %	−85 %	−1 %	−447 %	18 %
$8 \times 6 \times 8$ 2.4 GHz, Baseline	126.6	2800.1	3276.2	6076.3	271482.0	44780.0	2145.1	353.8	26.7	2.1	3096.0	316262.0	2498.9
	87 %	8 %	372 %	87 %	−71 %	−17 %	9 %	40 %	2 %	−48 %	−1 %	−58 %	15 %
$8 \times 6 \times 8$ 2.8 GHz, Baseline	108.2	2653.7	2567.8	5195.0	238108.5	40289.0	2199.5	372.3	26.7	2.0	3092.0	278122.0	2572.9
	60 %	3 %	270 %	60 %	−50 %	−5 %	6 %	36 %	2 %	−38 %	−1 %	−39 %	12 %
$8 \times 6 \times 8$ 2.8 GHz, CNTD 500	122.6	2740.7	3143.1	5888.0	256629.0	36901.0	2185.8	327.2	26.4	2.0	3060.0	300448.0	2485.6
	82 %	6 %	353 %	81 %	−62 %	4 %	7 %	44 %	3 %	−44 %	0 %	−50 %	15 %

8.12 Experiment 9

Experiment 9 systematically analyzes the clock frequency of individual threads within nodes to examine Linux scheduler response times to downclock requests and return to nominal frequencies. By closely monitoring how the Linux scheduler adjusts thread clock frequencies in response to varying workload demands, this experiment sheds light on the efficiency and responsiveness of the scheduler's frequency management mechanisms. To conduct the experiment, modifications were made to COUNTDOWN to systematically analyze clock frequency and perform sampling at the beginning and end of each MPI call. These changes allow for a detailed examination of thread clock frequencies and enable precise measurement of how the Linux scheduler responds to frequency adjustments. The source code modifications are available on the GitHub repository at [72].

In this experiment, we observe the following 4 configurations, specifically 3 already seen in experiments 5-8, and one new to also consider the 4^{th} possible case, not analyzed in previous experiments, but deserving a note here to analyze the differences with and without the slack algorithm even at 750 µs.

The sampling of this graph was carried out every one second, plus every MPI call, after which only a part of the plot was exclusively cropped, namely the most significant part, where there were more frequency changes.

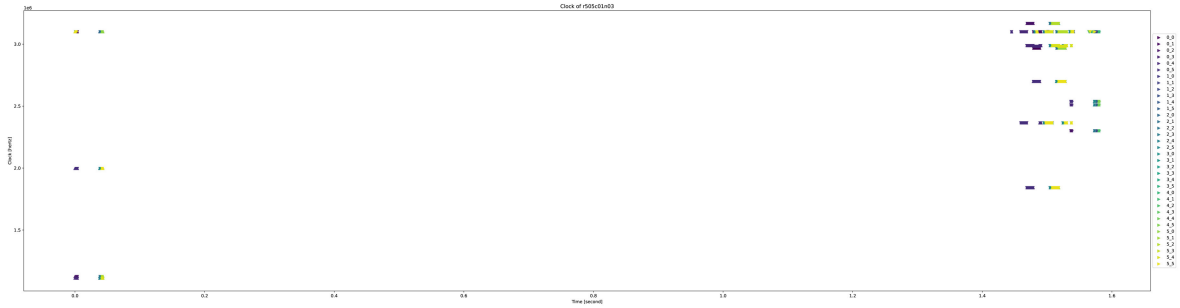
COUNTDOWN with slack and a callback delay of 500 µs **without slack (CNTD 500)** Execution of the power saving algorithm without including the slack optimization algorithm. This scenario isolated the power saving aspect and explored its effects without slack time optimization. Figures 8.7a and 8.8a show the clock plots respectively for the matrices *Cubo_1772481.Ext_bin* and *Wing_4538k.csr.Ext_bin*.

COUNTDOWN with slack and a callback delay of 750 µs **without slack (CNTD 750)** Running the power saving algorithm along with slack time optimization. This scenario is intended to examine the combined effects of power optimization and slack with a slack callback delay set to 100 µs. In previous runs, it is called CNTD SLACK. Figure 8.8b show the clock plots respectively for the matrices *Cubo_1772481.Ext_bin* and *Wing_4538k.csr.Ext_bin*.

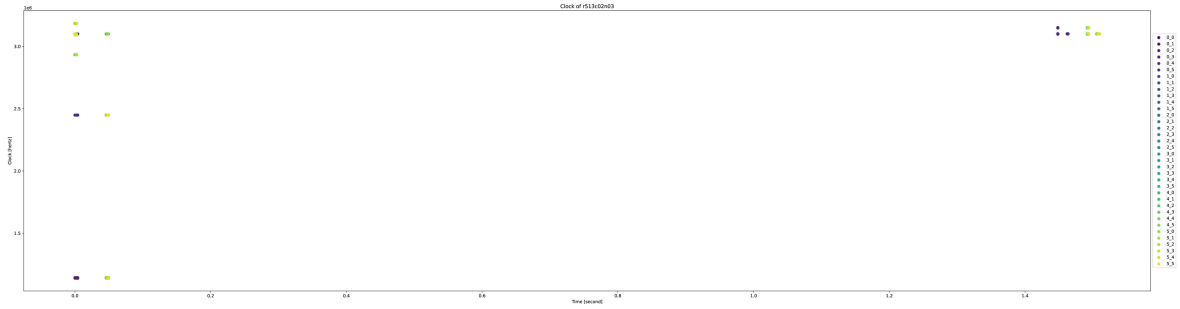
COUNTDOWN with slack and a callback delay of 500 µs **(CNTD 500)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to 500 µs. Figures 8.7c and 8.8c show the clock plots respectively for the matrices *Cubo_1772481.Ext_bin* and *Wing_4538k.csr.Ext_bin*.

COUNTDOWN with slack and a callback delay of 750 µs **(CNTD 750)** . Running the power saving algorithm along with slack time optimization. This scenario examines the combined effects of power optimization and slack with a slack callback delay set to 750 µs. Figures 8.7d and 8.8d show the clock plots respectively for the matrices *Cubo_1772481.Ext_bin* and *Wing_4538k.csr.Ext_bin*.

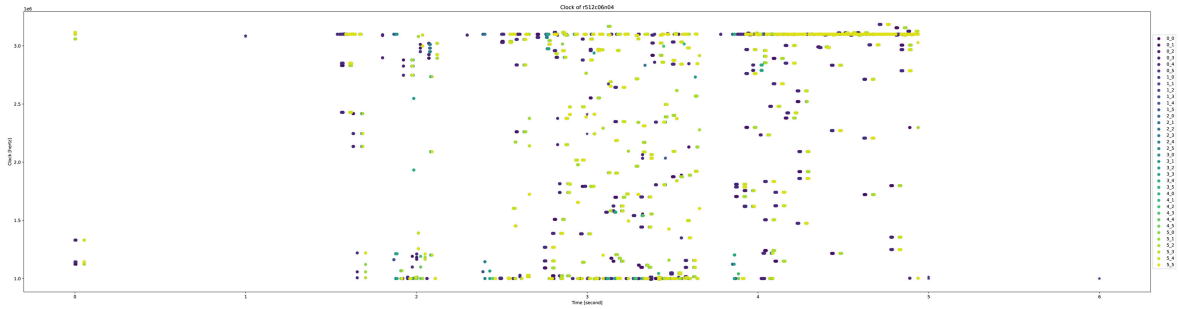
Finally, the average duration of MPI calls and the corresponding number of calls that took this long can be found in the appendix in figs. B.21 and B.22.



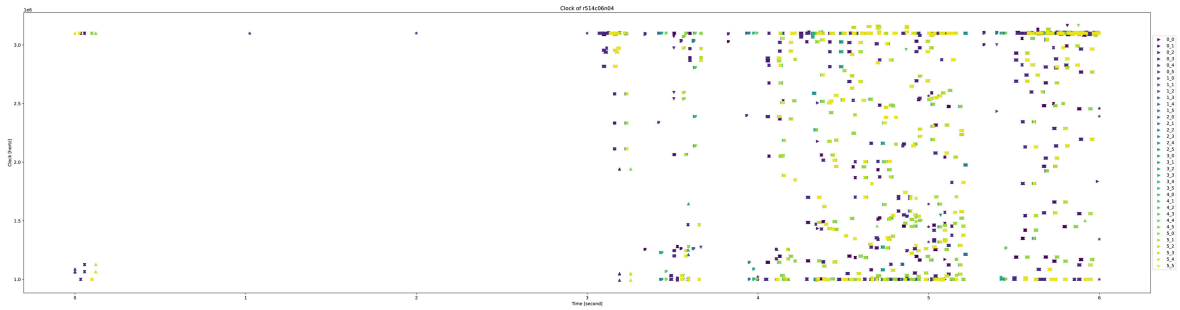
(a) Clock analysis with COUNTDOWN configuration enabled, slack disabled, COUNTDOWN callback delay set to 500 µs.



(b) Clock analysis with COUNTDOWN configuration enabled, slack enabled, COUNTDOWN callback delay set to 750 µs.



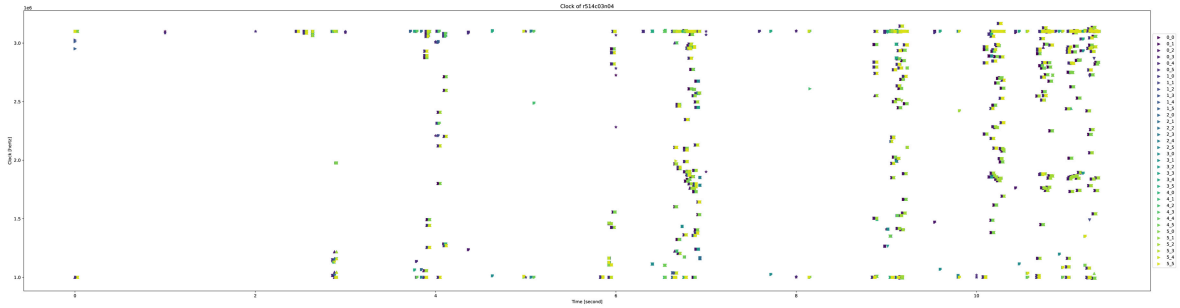
(c) Clock analysis with COUNTDOWN configuration enabled, slack disabled, COUNTDOWN callback delay set to 500 µs.



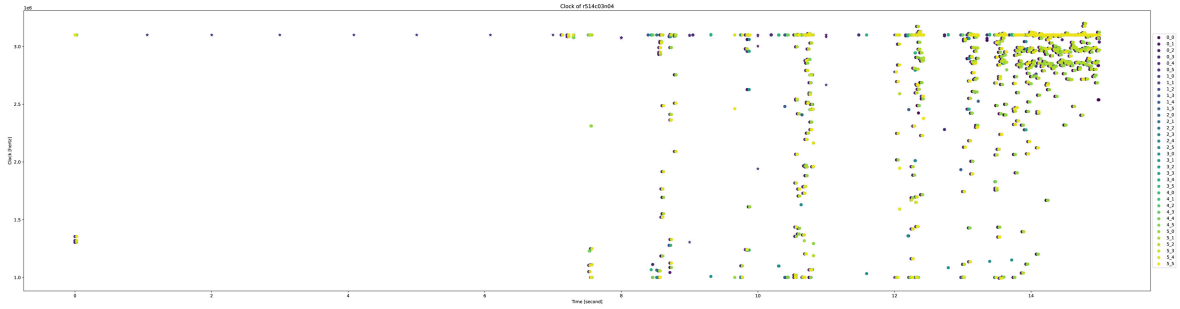
(d) Clock analysis with COUNTDOWN configuration enabled, slack enabled, COUNTDOWN callback delay set to 500 µs.

Figure 8.7: Experimental Results for Experiment 9 on the *Cubo_1772481.Ext_bin* matrix.

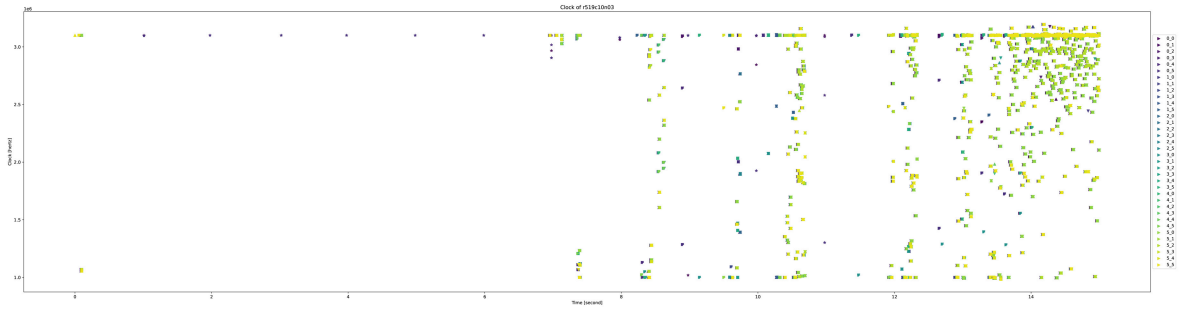
The graph shows the first 6 s, 36 threads of the 48 threads of a node in an $8 \times 6 \times 8$ configuration. Up arrow indicates start of MPI call, right arrow indicates start of callback, down arrow indicates MPI call, asterisk indicates frequency sampled every second.



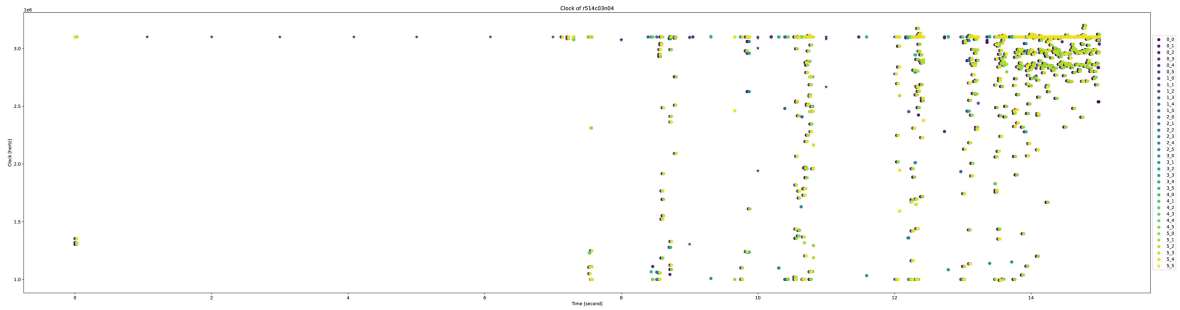
(a) Clock analysis with COUNTDOWN configuration enabled, slack disabled, COUNTDOWN callback delay set to 500 µs.



(b) Clock analysis with COUNTDOWN configuration enabled, slack enabled, COUNTDOWN callback delay set to 750 µs.



(c) Clock analysis with COUNTDOWN configuration enabled, slack disabled, COUNTDOWN callback delay set to 500 µs.



(d) Clock analysis with COUNTDOWN configuration enabled, slack enabled, COUNTDOWN timer set to 500 µs.

Figure 8.8: Experimental Results for Experiment 9 on the *Wing_4538k.csr.Ext_bin* matrix.

The graph shows the first 10 s, 36 threads of the 48 threads of a node in an $8 \times 6 \times 8$ configuration.

8.13 Experiment 10

Experiment 10 extends the modifications made to the COUNTDOWN codebase in Experiment 9, using the modified code to facilitate the calculation of the duration of MPI calls.

The primary objectives of Experiment 10 are twofold: first, to visualize the time duration of APP/MPI phases and the relative frequency for the configuration $8 \times 6 \times 8$ of Chronos with the matrices *Cubo_1772481.Ext_bin* and *Wing_4538k.csr.Ext_bin*; and second, to plot the time and average frequency of application/MPI phases for these benchmarks. These visualizations, which correspond to Figures 6 and 7 in the Cesarini et al. [36] paper, provide insight into the time and frequency characteristics of application execution on a single node.

In this experiment, we see the same 4 configurations that we saw in the previous experiment (section 8.12). Only MPI call-level sampling data is shown in these plots.

As expected from the data in figs. 8.9b, 8.9f, 8.9j, 8.9n, 8.10b, 8.10f, 8.10j and 8.10n, we would have predicted that requests below $500\,\mu s$ would have a relatively high frequency, while those above that threshold would have a lower frequency. However, as discussed in section 4.4, the scheduler's response time is not instantaneous, but rather delayed.

Nevertheless, it is interesting to observe that phases between $0\,\mu s$ and $500\,\mu s$ show a higher frequency for MPI phases and a lower frequency for application phases, contrary to the expected pattern. Consequently, I designed Experiment 11 as described in section 8.14, but it did not yield the expected results. Our hypothesis is that in phases shorter than $500\,\mu s$, the average frequency is influenced more by the frequency of the preceding phase than by the intended one.

In figs. 8.9 and 8.10 we examine the relationship between the duration of each application phase, the subsequent MPI phase duration, and their average frequency. The application phase duration is plotted on the y axis, the MPI phase duration is plotted on the x axis, and the average frequency is indicated by the color code. In figs. 8.9c, 8.9g, 8.9k, 8.9o, 8.10c, 8.10g, 8.10k and 8.10o we show the frequency of the MPI phase, while in figs. 8.9d, 8.9h, 8.9l, 8.9p, 8.10d, 8.10h, 8.10l and 8.10p we show the frequency of the application phase.

For both plots we can identify four quadrants:

APP & MPI > $500\,\mu s$ This zone is characterized by long application phases followed by long MPI phases. Instances in this zone have a low occurrence during MPI phases and a high occurrence during application phases. This pattern is considered optimal because implementing a frequency scaling strategy can reduce MPI power consumption without sacrificing application efficiency. Stages in this range are excellent candidates for detailed DVFS strategies.

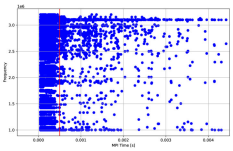
APP > $500\,\mu s$ **& MPI <** $500\,\mu s$ This zone is characterized by long application phases followed by short MPI phases. Instances in this zone have a remarkably high average frequency during both application and MPI phases. The reason for this phenomenon lies in the brevity of the MPI phases, which limits the ability of the HW power controller to quickly adjust the frequency downward in response to the scaling request (prologue) before being overridden by the directive to operate at maximum frequency (epilogue). Consequently, implementing COUNTDOWN control in this

segment is not useful because the frequency reduction during MPI phases is minimal. However, performance is not compromised because the application phases always run at the highest frequency. Given this scenario, it is advisable to exclude phases in this region from the COUNTDOWN policy and keep the frequencies at their maximum level.

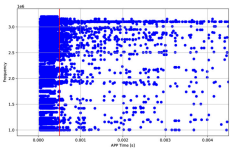
APP < $500\,\mu s$ **& MPI >** $500\,\mu s$ In this zone, short application phases are followed by long MPI phases. This is due to the short nature of the application phases, which does not allow the hardware power controller enough time to respond to the frequency increase request made at the end of the previous MPI phase before this setting is replaced by the minimum frequency request at the beginning of the next MPI phase. Although implementing precise DVFS strategies in this region can result in power savings, it may compromise overall performance by running application phases at lower frequencies. Given the significant application execution time overhead, it is advisable not to consider applying COUNTDOWN policies to phases in this region.

APP & MPI < $500\,\mu s$ This zone exhibits contrasting behavior to the Application & MPI > $500\,\mu s$ region. In this region, both the application and MPI phases operate erratically at varying high and low average frequencies because the HW power controller can not effectively handle and respond to the requested frequency adjustments. The average frequency at which the MPI and application phases operate is closely related to the nature of the preceding long phase: if it was an application phase, the subsequent short phases will operate at a high average frequency; conversely, if it was an MPI phase, the subsequent short phases will operate at a low average frequency. Implementing detailed DVFS policies in this region can lead to unexpected results that can degrade application performance. For COUNTDOWN performance managers, it is advisable not to consider any phases shorter than $500\,\mu s$.

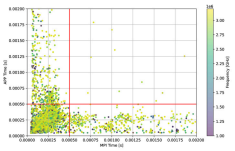
As seen in figs. 8.9o and 8.9p, only this configuration is the one where the COUNTDOWN algorithm theoretically performs best, but as we have seen in the previous results, it does not.



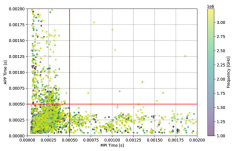
(a) CNTD 500: Average frequency and time duration of MPI phases.



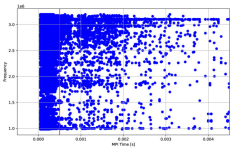
(b) CNTD 500: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



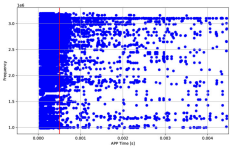
(c) CNTD 500: Time of APP/MPI phases and average frequency of MPI phases.



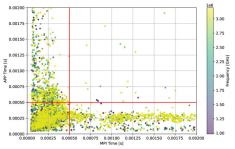
(d) CNTD 500: Time of APP/MPI phases and average frequency of APP phases.



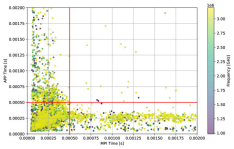
(e) CNTD SLACK 500: Average frequency and time duration of MPI phases.



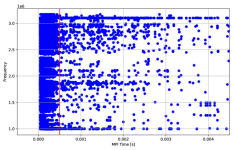
(f) CNTD SLACK 500: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



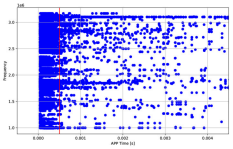
(g) CNTD SLACK 500: Time of APP/MPI phases and average frequency of MPI phases.



(h) CNTD SLACK 500: Time of APP/MPI phases and average frequency of APP phases.



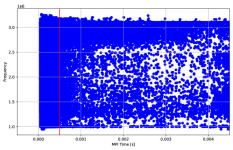
(i) CNTD 750: Average frequency and time duration of MPI phases.



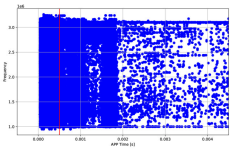
(j) CNTD 750: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.

(k) CNTD 750: Time of APP/MPI phases and average frequency of MPI phases.

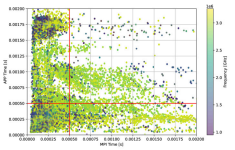
(l) CNTD 750: Time of APP/MPI phases and average frequency of APP phases.



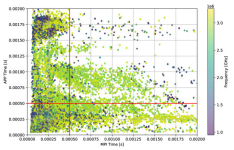
(m) CNTD SLACK 750: Average frequency and time duration of MPI phases.



(n) CNTD SLACK 750: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.

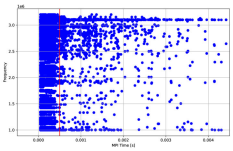


(o) CNTD SLACK 750: Time of APP/MPI phases and average frequency of MPI phases.

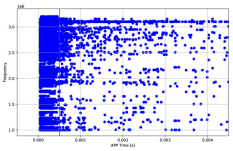


(p) CNTD SLACK 750: Time of APP/MPI phases and average frequency of APP phases.

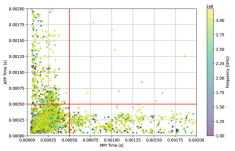
Figure 8.9: Average frequency and time of APP/MPI phases for *Cubo_1772481.Ext_bin* on Chronos configuration $8 \times 6 \times 8$.



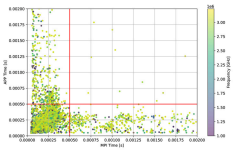
(a) CNTD 500: Average frequency and time duration of MPI phases.



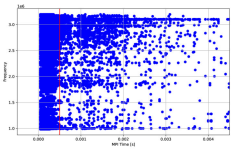
(b) CNTD 500: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



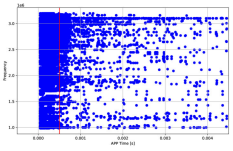
(c) CNTD 500: Time of APP/MPI phases and average frequency of MPI phases.



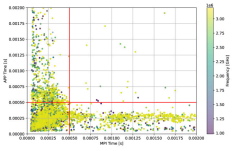
(d) CNTD 500: Time of APP/MPI phases and average frequency of APP phases.



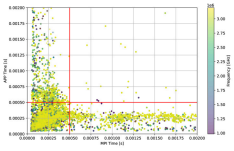
(e) CNTD SLACK 500: Average frequency and time duration of MPI phases.



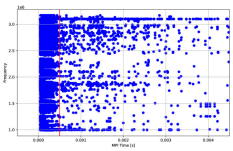
(f) CNTD SLACK 500: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



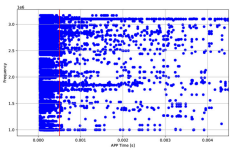
(g) CNTD SLACK 500: Time of APP/MPI phases and average frequency of MPI phases.



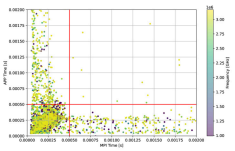
(h) CNTD SLACK 500: Time of APP/MPI phases and average frequency of APP phases.



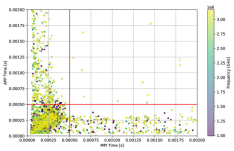
(i) CNTD 750: Average frequency and time duration of MPI phases.



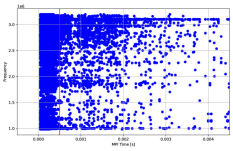
(j) CNTD 750: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



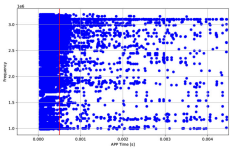
(k) CNTD 750: Time of APP/MPI phases and average frequency of MPI phases.



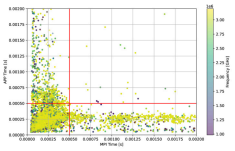
(l) CNTD 750: Time of APP/MPI phases and average frequency of APP phases.



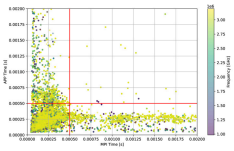
(m) CNTD SLACK 750: Average frequency and time duration of MPI phases.



(n) CNTD SLACK 750: Time of APP phase and and relative count of the frequency at which it appears in an execution of APP phases.



(o) CNTD SLACK 750: Time of APP/MPI phases and average frequency of MPI phases.



(p) CNTD SLACK 750: Time of APP/MPI phases and average frequency of APP phases.

Figure 8.10: Average frequency and time of APP/MPI phases for *Wing_4538k.csr.Ext_bin* on Chronos configuration $8 \times 6 \times 8$.

8.14 Experiment 11

Experiment 11 extends the modifications of Experiments 9 and 10 by adding a small delay between the end of the slack time and the communication, as shown in fig. 8.11, this small modification which, according to the literature, in particular Mazouz et al. [73], to be less than 100 µs, this value is of course in addition to the fact that the Linux kernel only accepts this command once every 500 µs, as described in section 4.4. This would therefore result in a maximum overhead in execution time of about 10 s if we consider a run of 100 s and 10 000 000 MPI call, since in this run we have only the 10 % of calls that are subject to actions by COUNTDOWN.

The results of Experiment 11, shown in table 8.29 for the matrix *Wing_4538k.csr.Ext_bin*, however, say that the execution times of this delay are much higher, about 20 times as long, i.e. 10 ms, which effectively makes good prospects of this idea unfeasible, since reading the actual CPU frequency is a time-consuming task in itself.

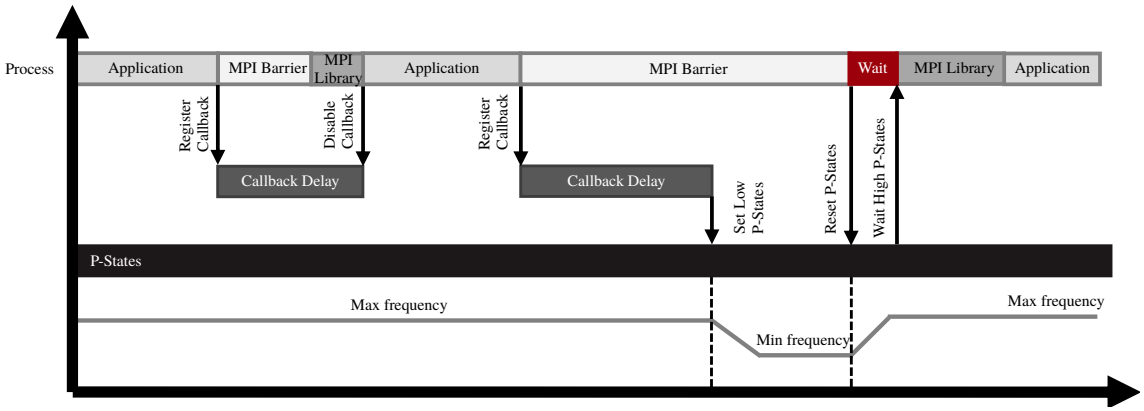


Figure 8.11: Proposed approach for Experiment 11.

Table 8.29: Median values depicting the comparison between profiles with and without COUNTDOWN for Experiment 11 on the *Wing_4538k.csr.Ext_bin* matrix.

The percentage results compare the baseline run to the current run. Positive time percentages indicate overhead, and positive energy or power percentages indicate savings.

Run config	**EXE time [s]**	**APP time [s]**	**MPI time [s]**	**TOT time [s]**	**Energy PKG [J]**	**Energy DRAM [J]**	**AVG Power PKG [W]**	**AVG Power DRAM [W]**	**MAX Memory usage [GB]**	**AVG IPC**	**AVG CPU frequency [MHz]**	**Energy [J]**	**AVG Power [W]**
$8 \times 6 \times 8$, Baseline	12.0	457.9	116.9	575.2	27056	4285	2264.2	358.3	26.7	1.6	3084.0	31341	2622.4
$8 \times 6 \times 8$, CNTD 500	12.2 2 %	456.9 0 %	131.0 12 %	587.6 2 %	26484 2 %	3595 16 %	2167.9 4 %	294.5 18 %	26.4 1 %	1.6 1 %	2973.0 4 %	30088 4 %	2463.2 6 %
$8 \times 6 \times 8$, wait	86.8 627 %	4736.8 934 %	474.0 305 %	5211.3 806 %	247345 −814 %	43365 −912 %	2856.2 −26 %	500.2 −40 %	16.3 39 %	1.2 24 %	3066.5 1 %	290730 −828 %	3357.3 −28 %

8.15 Experiments Conclusions

To conclude these experiments with a general consideration of energy savings, we now consider as a base case the fastest point $32 \times 6 \times 8$, and analyze with percentages the remaining cases $16 \times 6 \times 8$ and $8 \times 6 \times 8$.

As for *Cubo_1772481.Ext_bin* in the case of low iterations, a tripling of execution times, i.e. a $200\,\%$ increase, can lead to a reduction of up to $30\,\%$ energy, as it can be seen in table 8.30a. While for the case of high iterations (table 8.30b) one can save up to $20\,\%$ with a $200\,\%$ increase in execution time.

Regarding *Wing_4538k.csr.Ext_bin* in the case of low iterations, doubling the execution times, i.e. an increase of $120\,\%$, can lead to a reduction in energy of up to $40\,\%$, as it can be seen in table 8.30c. While in the case of high iterations (table 8.30d) one can save up to $4\,\%$ with a $270\,\%$ increase in execution time.

While weighing the data as described in Table 3 of Cesarini et al. [35], we summarize the data obtained in our previous experiments in table 8.31. In these data, we see that COUNTDOWN does not bring advantages in many cases, for example for the *Wing_4538k.csr.Ext_bin* matrix, there is not any benefit from using COUNTDOWN. Whereas if we set the frequency to $1.0\,\text{GHz}$ on each MPI call and let the operating system return to the frequency it likes when it thinks it is best, we get savings between $6\,\%$ and $12\,\%$ in an ideal configuration with a lower iteration, however, this does not allow a solution with acceptable accuracy.

The situation in the *Wing_4538k.csr.Ext_bin* matrix is much better. In the case of low iterations there is a general reduction of consumption up to $20\,\%$ both with the classical callback delay $500\,\mu\text{s}$ and with the longer callback delay of $750\,\mu\text{s}$, while in the case of full iter, that is, when the problem converges, we have that the only case where COUNTDOWN brings benefits is $16 \times 6 \times 8$.

We have observed that the COUNTDOWN algorithm demonstrates its maximum effectiveness in specific cases, particularly when the number of threads is high, and synchronization times can be significantly extended, leading to substantial energy savings, up to $19\,\%$ in one instance, closely approaching the $25\,\%$ mentioned by the COUNTDOWN's authors. However, this results in increased variance in the distribution of execution times, power, and energy, highlighted by a much wider standard deviation compared to the configuration without COUNTDOWN.

It is crucial to note that, especially in the case of the *Cubo_1772481.Ext_bin* matrix, COUNTDOWN does not lead to improvements, but rather to longer execution times and higher energy consumption. These findings underscore the importance of carefully assessing the specific conditions under which COUNTDOWN can provide tangible benefits, balancing energy savings with potential increases in execution times and the complexity of parallelism management.

Although the results on the *Wing_4538k.csr.Ext_bin* matrix are generally good, they are not comparable to those described in [36], except in one special case and not with the default settings. Also, the fact that $750\,\mu\text{s}$ is more performant makes sense in my opinion, since it is clear from the literature that $500\,\mu\text{s}$ is the average synchronization time of the various threads, so COUNTDOWN should only be used if it is greater than the average.

Table 8.30: Summary mirror of Experiments 3–8 performed with COUNTDOWN and Chronos: general reflection on energy savings.

Positive percentages represent a time overhead and/or energy/power saving. Negative percentages represent time and/or energy/power savings.

(a) *Cubo_1772481.Ext_bin* low iteration result compared with the base case $32 \times 6 \times 8$.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$32 \times 6 \times 8$, 1.0 GHz	−9.25	11.67	2.53
$32 \times 6 \times 8$, CNTD	−4.87	9.53	4.52
$32 \times 6 \times 8$, CNTD 500	18.42	−11.35	6.07
$32 \times 6 \times 8$, CNTD 750	13.51	−5.60	6.54
$16 \times 6 \times 8$	47.43	25.06	49.01
$16 \times 6 \times 8$, 1.0 GHz	1090.99	−387.65	58.93
$16 \times 6 \times 8$, CNTD	60.32	17.21	48.47
$16 \times 6 \times 8$, CNTD 500	60.63	21.18	50.82
$16 \times 6 \times 8$, CNTD 750	58.32	21.99	50.66
$8 \times 6 \times 8$	174.97	25.06	72.68
$8 \times 6 \times 8$, 1.0 GHz	169.33	29.46	73.75
$8 \times 6 \times 8$, CNTD	176.63	23.23	72.10
$8 \times 6 \times 8$, CNTD 500	194.90	21.34	73.26
$8 \times 6 \times 8$, CNTD 750	187.58	23.29	73.26

(b) *Cubo_1772481.Ext_bin* full iteration result compared with the base case $32 \times 6 \times 8$.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$32 \times 6 \times 8$, 1.0 GHz	1496.34	−1128.12	22.71
$32 \times 6 \times 8$, CNTD	3.84	−2.07	1.10
$32 \times 6 \times 8$, CNTD 500	29.75	−20.89	6.48
$32 \times 6 \times 8$, CNTD 750	10.93	−5.75	4.05
$16 \times 6 \times 8$	73.33	8.31	47.60
$16 \times 6 \times 8$, 1.0 GHz	1450.35	−503.35	60.81
$16 \times 6 \times 8$, CNTD	75.00	5.69	46.19
$16 \times 6 \times 8$, CNTD 500	101.44	−2.03	49.78
$16 \times 6 \times 8$, CNTD 750	86.36	5.18	48.90
$8 \times 6 \times 8$	198.09	18.89	72.92
$8 \times 6 \times 8$, 1.0 GHz	1498.45	−222.61	79.69
$8 \times 6 \times 8$, CNTD	197.55	19.41	72.79
$8 \times 6 \times 8$, CNTD 500	223.01	13.54	73.27
$8 \times 6 \times 8$, CNTD 750	208.50	16.91	73.08

(c) *Wing_4538k.csr.Ext_bin* low iteration result compared with the base case $32 \times 6 \times 8$.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$32 \times 6 \times 8$, 1.0 GHz,	−33.81	32.55	−0.85
$32 \times 6 \times 8$, CNTD	−27.86	28.03	1.14
$32 \times 6 \times 8$, CNTD 500	−17.08	19.59	4.10
$32 \times 6 \times 8$, CNTD 750	−16.12	17.57	2.53
$16 \times 6 \times 8$	73.68	17.39	52.38
$16 \times 6 \times 8$, 1.0 GHz	14.52	40.42	48.33
$16 \times 6 \times 8$, CNTD	51.94	22.62	49.33
$16 \times 6 \times 8$, CNTD 500	37.26	30.07	49.44
$16 \times 6 \times 8$, CNTD 750	28.97	34.05	48.93
$8 \times 6 \times 8$	116.91	41.08	73.06
$8 \times 6 \times 8$, 1.0 GHz	109.68	43.61	73.35
$8 \times 6 \times 8$, CNTD	135.81	38.03	73.91
$8 \times 6 \times 8$, CNTD 500	121.71	43.44	74.70
$8 \times 6 \times 8$, CNTD 750	119.85	42.32	73.97

(d) *Wing_4538k.csr.Ext_bin* full iteration result compared with the base case $32 \times 6 \times 8$.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$32 \times 6 \times 8$, 1.0 GHz	2421.45	−1643.32	30.95
$32 \times 6 \times 8$, CNTD	5.95	−2.45	3.14
$32 \times 6 \times 8$, CNTD 500	26.82	−20.51	5.00
$32 \times 6 \times 8$, CNTD 750	12.40	−8.57	3.87
$16 \times 6 \times 8$	83.84	4.24	48.25
$16 \times 6 \times 8$, 1.0 GHz	2053.78	−666.16	64.51
$16 \times 6 \times 8$, CNTD	78.98	11.98	50.89
$16 \times 6 \times 8$, CNTD 500	112.01	−6.58	50.91
$16 \times 6 \times 8$, CNTD 750	78.33	12.69	51.17
$8 \times 6 \times 8$	246.07	9.50	74.22
$8 \times 6 \times 8$, 1.0 GHz	2085.36	−333.09	80.22
$8 \times 6 \times 8$, CNTD	270.95	3.14	74.18
$8 \times 6 \times 8$, CNTD 500	239.71	8.61	73.22
$8 \times 6 \times 8$, CNTD 750	264.33	4.01	73.74

Table 8.31: Summary mirror of Experiments 3–8 performed with COUNTDOWN and Chronos: indidual reflection on energy savings.

Positive percentages represent a time overhead and/or energy/power saving. Negative percentages represent time and/or energy/power savings.

(a) *Cubo_1772481.Ext_bin* low iteration result.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$8 \times 6 \times 8$, 1.0 GHz	−2.05	5.88	3.88
$8 \times 6 \times 8$, CNTD	0.60	−2.44	−2.13
$8 \times 6 \times 8$, CNTD 500	7.25	−4.96	2.10
$8 \times 6 \times 8$, CNTD 750	4.58	−2.35	2.09
$16 \times 6 \times 8$, 1.0 GHz	707.82	−550.71	19.46
$16 \times 6 \times 8$, CNTD	8.74	−10.47	−1.06
$16 \times 6 \times 8$, CNTD 500	8.95	−5.17	3.57
$16 \times 6 \times 8$, CNTD 750	7.38	−4.10	3.25
$32 \times 6 \times 8$, 1.0 GHz	−9.25	11.67	2.53
$32 \times 6 \times 8$, CNTD	−4.87	9.53	4.52
$32 \times 6 \times 8$, CNTD 500	18.42	−11.35	6.07
$32 \times 6 \times 8$, CNTD 750	13.51	−5.60	6.54

(b) *Cubo_1772481.Ext_bin* full iteration result.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$8 \times 6 \times 8$, 1.0 GHz	436.23	−297.73	25.00
$8 \times 6 \times 8$, CNTD	−0.18	0.64	−0.50
$8 \times 6 \times 8$, CNTD 500	8.36	−6.59	1.29
$8 \times 6 \times 8$, CNTD 750	3.49	−2.43	0.57
$16 \times 6 \times 8$, 1.0 GHz	794.47	−558.04	25.22
$16 \times 6 \times 8$, CNTD	0.97	−2.86	−2.68
$16 \times 6 \times 8$, CNTD 500	16.22	−11.27	4.16
$16 \times 6 \times 8$, CNTD 750	7.52	−3.42	2.48
$32 \times 6 \times 8$, 1.0 GHz	1496.34	−1128.12	22.71
$32 \times 6 \times 8$, CNTD	3.84	−2.07	1.10
$32 \times 6 \times 8$, CNTD 500	29.75	−20.89	6.48
$32 \times 6 \times 8$, CNTD 750	10.93	−5.75	4.05

(c) *Wing_4538k.csr.Ext_bin* low iteration result.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$8 \times 6 \times 8$, 1.0 GHz	−3.33	4.29	1.07
$8 \times 6 \times 8$, CNTD	8.71	−5.18	3.15
$8 \times 6 \times 8$, CNTD 500	2.21	4.00	6.07
$8 \times 6 \times 8$, CNTD 750	1.36	2.10	3.38
$16 \times 6 \times 8$, 1.0 GHz	−34.06	27.88	−8.49
$16 \times 6 \times 8$, CNTD	−12.52	6.34	−6.40
$16 \times 6 \times 8$, CNTD 500	−20.97	15.35	−6.16
$16 \times 6 \times 8$, CNTD 750	−25.75	20.17	−7.23
$32 \times 6 \times 8$, 1.0 GHz	−9.25	11.67	2.53
$32 \times 6 \times 8$, CNTD	−27.86	28.03	1.14
$32 \times 6 \times 8$, CNTD 500	−17.08	19.59	4.10
$32 \times 6 \times 8$, CNTD 750	−16.12	17.57	2.53

(d) *Wing_4538k.csr.Ext_bin* full iteration result.

Run config	Time Overhead [%]	Energy Saving [%]	AVG Power Saving [%]
$8 \times 6 \times 8$, CNTD	7.19	−7.02	−0.14
$8 \times 6 \times 8$, 1.0 GHz	531.49	−378.56	23.27
$8 \times 6 \times 8$, CNTD 500	−1.84	−0.98	−3.89
$8 \times 6 \times 8$, CNTD 750	5.28	−6.06	−1.85
$16 \times 6 \times 8$, CNTD	−2.64	8.08	5.09
$16 \times 6 \times 8$, 1.0 GHz	1071.57	−700.05	31.41
$16 \times 6 \times 8$, CNTD 500	15.32	−11.30	5.13
$16 \times 6 \times 8$, CNTD 750	−3.00	8.83	5.63
$32 \times 6 \times 8$, CNTD	5.95	−2.45	3.14
$32 \times 6 \times 8$, 1.0 GHz	2421.45	−1643.32	30.95
$32 \times 6 \times 8$, CNTD 500	26.82	−20.51	5.00
$32 \times 6 \times 8$, CNTD 750	12.40	−8.57	3.87

9

Conclusions

In this work, we have explored state-of-the-art techniques for energy-efficient execution of parallel programs on HPC clusters. This endeavor represents a significant challenge for the future: achieving energy savings while maintaining a comparable level of service. To meet this challenge, the development of more tools such as COUNTDOWN, Adagio, MPI inside, and truly efficient MPI communication implementations is imperative. These tools facilitate the prediction and programming of CPU frequency to minimize busy wait times. However, achieving optimal energy efficiency also requires programmers to refactor parallel programs to minimize branch divergence, as this can significantly impact execution times.

Although theoretical models indicate substantial energy savings, as evidenced by up to $40\,\%$ reductions in energy consumption and a doubling of execution times in low-iteration scenarios, practical implementations present a more nuanced picture. In high-iteration scenarios, where execution times increase by $400\,\%$, energy savings are only about $15\,\%$.

COUNTDOWN, while capable of achieving up to $25\,\%$ energy savings, demonstrates its effectiveness in specific scenarios, such as the full iteration with the *Wing_4538k.csr.Ext_bin* matrix and $16 \times 6 \times 8$ node configuration. However, it is critical to consider the overall energy consumption associated with performing these calculations. The cumulative energy consumption on the CINECA machines, estimated as the sum of all recorded values in $1\,727\,761\,944\,\mathrm{J}$ ($480\,\mathrm{kW\,h}$, about half the consumption of a fully electric house for one year) with $15\,000$ runs of Chronos, underscores the importance of evaluating energy optimizations holistically to ensure they contribute to overall sustainability goals.

In addition, the emergence of big.LITTLE architectures warrants more attention. These architectures typically use CPU hardware instructions to dynamically shift workloads between high performance and energy-efficient cores. MPI could leverage this architecture to use energy-efficient cores for communication tasks and high performance cores for computational tasks, potentially further optimizing energy consumption.

In summary, while theoretical models provide promising insights into energy-efficient parallel execution, practical implementations present complex challenges. Through continued research and development of tools and techniques, coupled with thoughtful architectural considerations, we can strive to achieve sustainable and efficient use of HPC resources in the future.

It is important to note that while it may seem obvious that different code will produce different results, variation in data inputs can also have a significant impact on performance.

Even with data inputs that are not very different in size or structure, COUNTDOWN's performance can vary, indicating the sensitivity of its optimizations to subtle variations in input data.

It's also worth noting that the CINECA supercomputers are often offline for maintenance, at least once a month, making it difficult to run large experiments and requiring careful advance planning.

The HPC sector now surpasses the airline industry in terms of carbon footprint. A single data center can use as much electricity as $50\,000$ houses, highlighting the significant environmental impact of high performance computing. In addition to its significant carbon footprint, the HPC sector also faces challenges related to waste generation and rising temperatures that contribute to climate change.

These findings highlight that achieving optimal energy efficiency and sustainability in HPC requires addressing not only technical challenges, but also environmental considerations and operational constraints.

Appendices

A

Top500 and Green500 November 2023

In this appendix, you will find a concise overview of the top 10 supercomputers from both the Top500 and Green500 lists. These lists, contained in tables A.1 and A.2 respectively, showcase the leading supercomputing systems globally. The Top500 list highlights the top performers based on raw computational power, while the Green500 list emphasizes energy-efficient computing solutions.

Table A.1: Top 10 supercomputers in the Top500 list, november 2023.

Source: https://www.top500.org/lists/top500/2023/11/.

Rank	System	Manufacturer	Organization and Location	Cores	Rmax [PFlop/s]	Rpeak [PFlop/s]	Power [kW]
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	DOE/SC/Oak Ridge National Laboratory, United States	8699904	1194.0	1679.8	22 703
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11	Intel	DOE/SC/Argonne National Laboratory, United States	4742808	585.3	1059.3	24 687
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR	Microsoft	Microsoft Azure, United States	1123200	561.2	846.8	
4	**Supercomputer Fugaku** - A64FX 48C 2.2GHz, Tofu interconnect D	Fujitsu	RIKEN Center for Computational Science, Japan	7630848	442.0	537.2	29 899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	EuroHPC/CSC, Finland	2752704	379.7	531.6	7107
6	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband	EVIDEN	EuroHPC/CINECA, Italy	1824768	238.7	304.5	7404
7	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	IBM	DOE/SC/Oak Ridge National Laboratory, United States	2414592	148.6	200.8	10 096
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 40C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR200	EVIDEN	EuroHPC/BSC, Spain	680960	138.2	265.6	2560
9	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400	Nvidia	NVIDIA Corporation, United States	485888	121.4	188.6	
10	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	IBM / NVIDIA / Mellanox	DOE/NNSA/LLNL, United States	1572480	94.6	125.7	7438

Table A.2: Top 10 supercomputers in the Green500 list, november 2023.

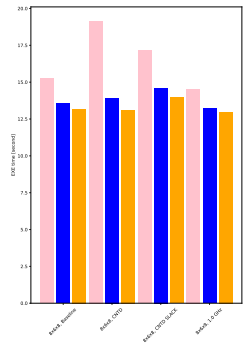
Source: https://www.top500.org/lists/green500/2023/11/.

Rank	TOP500 Rank	System	Manufacturer	Organization and Location	Cores	Rmax [PFlop/s]	Rpeak [PFlop/s]	Power [kW]
1	293	**Henri** - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR	Lenovo	Flatiron Institute, United States	8288	2.9	44	65.4
2	44	**Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	DOE/SC/Oak Ridge National Laboratory, United States	120 832	19.2	309	62.7
3	17	**Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES), France	319 072	46.1	921	58.0
4	25	**Setonix** - GPU - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	Pawsey Supercomputing Centre, Kensington, Western Australia, Australia	181 248	27.2	477	57.0
5	92	**Dardel GPU** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	KTH - Royal Institute of Technology, Sweden	52 864	9.0	146	56.5
6	8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 40C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR200	EVIDEN	EuroHPC/BSC, Spain	680 960	138.2	2560	53.984
7	5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	EuroHPC/CSC, Finland	2 752 704	379.7	7107	53.4
8	1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	HPE	DOE/SC/Oak Ridge National Laboratory, United States	8 699 904	1194.0	22 703	52.6
9	84	**Goethe-NHR** - Supermicro AS-4124GS-TNR, AMD EPYC 7452 32C 2.4GHz, AMD Instinct MI210 64 GB, Mellanox InfiniBand EDR	MEGWARE / Supermicro	Universitaet Frankfurt, Germany	96 768	9.1	195	46.5
10	496	**Olaf** - Lenovo ThinkSystem SR675 V3, AMD EPYC 9334 32C 2.7GHz, NVIDIA H100, Infiniband NDR 400	Lenovo	Science Institute, South Korea	3936	2.0	45	45.1

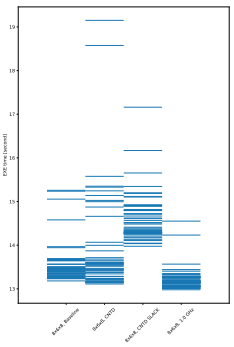
B

Graphical Results

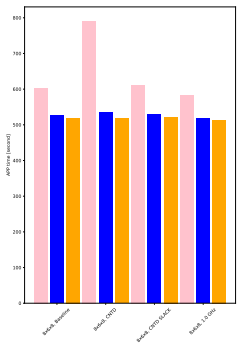
In this appendix, we present the graphs depicting the results of Experiments 3, 4, 5, 6, 7, 8 and 9 (sections from 8.7 to 8.12). For a more detailed description, see section 8.3.



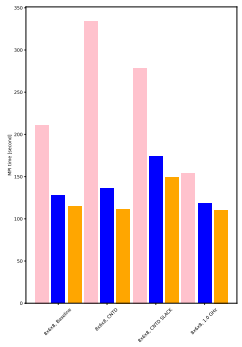
(a) EXE time: variations as COUNTDOWN configuration.



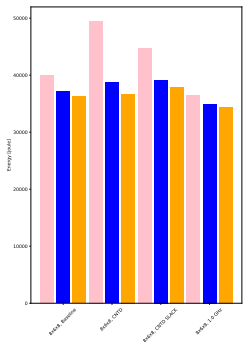
(b) EXE time: distribution of execution times for the 51 runs per instance.



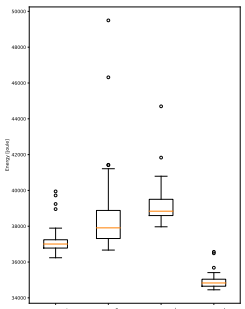
(c) APP time: distribution of execution times for the 51 runs per instance.



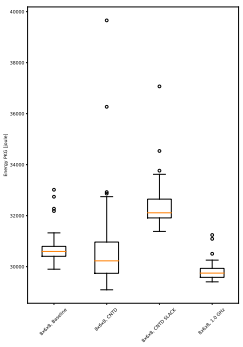
(d) MPI time: distribution of execution times for the 51 runs per instance.



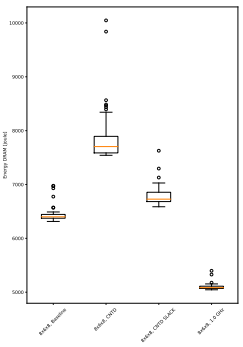
(e) Energy: variations as COUNTDOWN configuration.



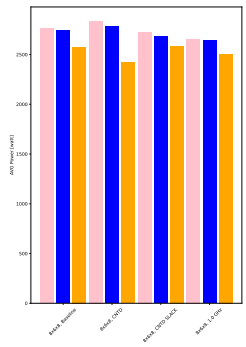
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



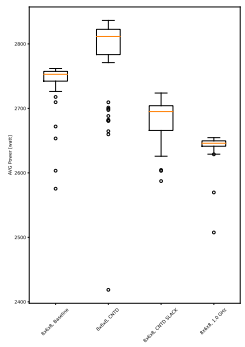
(g) Energy: boxplot of distribution of the energy consumed by CPU.



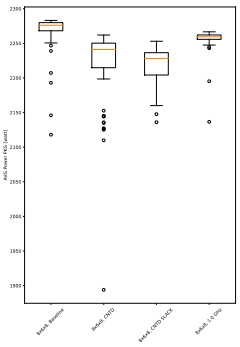
(h) Energy: boxplot of distribution of the energy consumed by RAM.



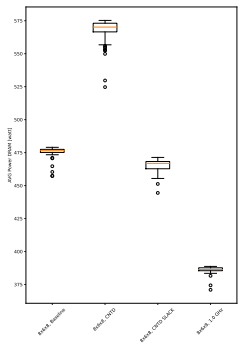
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

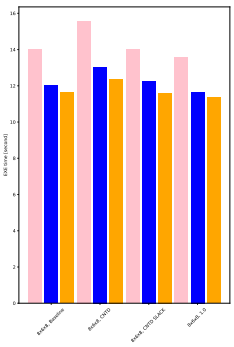


(k) Power: boxplot of distribution of the power consumed by CPU.

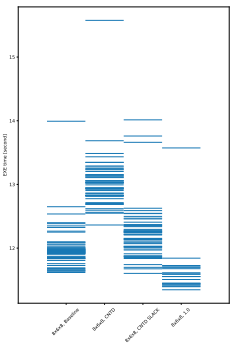


(l) Power: boxplot of distribution of the power consumed by RAM.

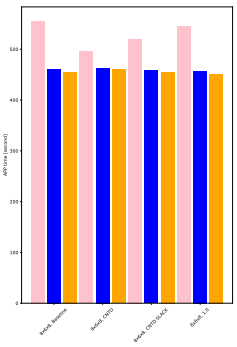
Figure B.1: Experimental Results for Experiment 3 on the *Cubo_1772481.Ext_bin* matrix.



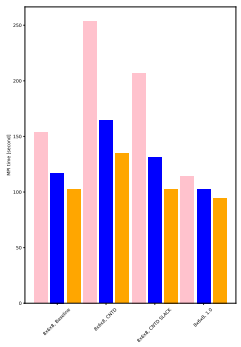
(a) EXE time: variations as COUNTDOWN configuration.



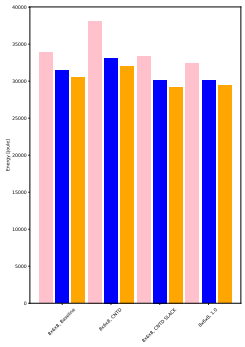
(b) EXE time: distribution of execution times for the 51 runs per instance.



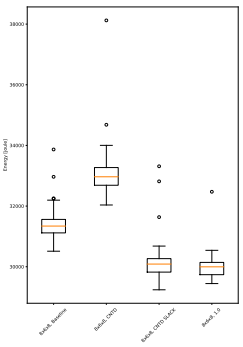
(c) APP time: distribution of execution times for the 51 runs per instance.



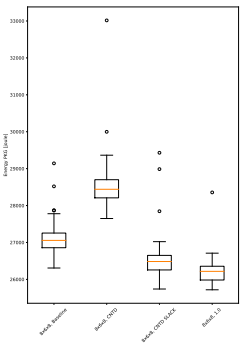
(d) MPI time: distribution of execution times for the 51 runs per instance.



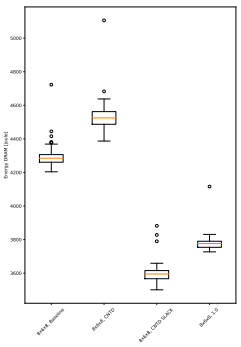
(e) Energy: variations as COUNTDOWN configuration.



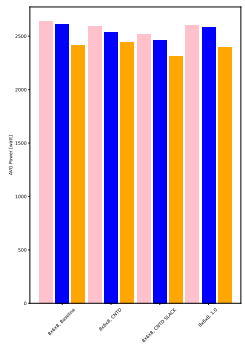
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



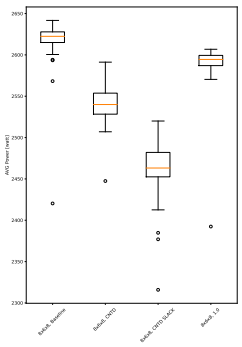
(g) Energy: boxplot of distribution of the energy consumed by CPU.



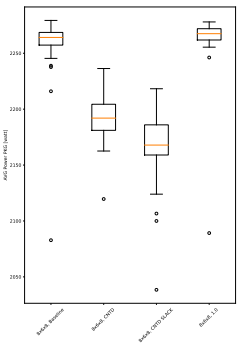
(h) Energy: boxplot of distribution of the energy consumed by RAM.



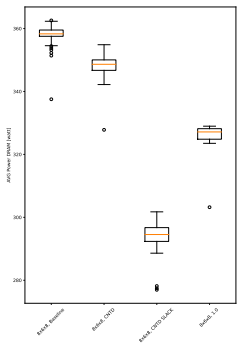
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

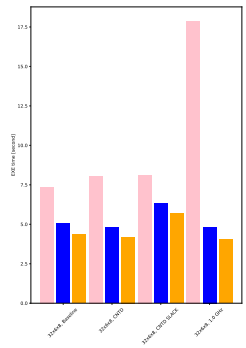


(k) Power: boxplot of distribution of the power consumed by CPU.

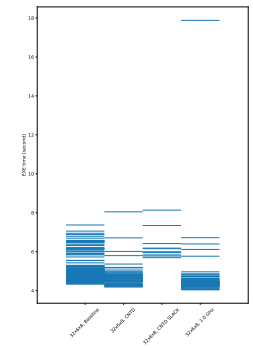


(l) Power: boxplot of distribution of the power consumed by RAM.

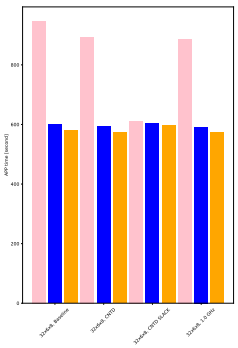
Figure B.2: Experimental Results for Experiment 3 on the *Wing_4538k.csr.Ext_bin* matrix.



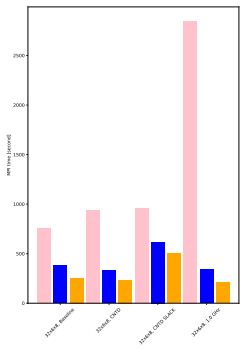
(a) EXE time: variations as COUNTDOWN configuration.



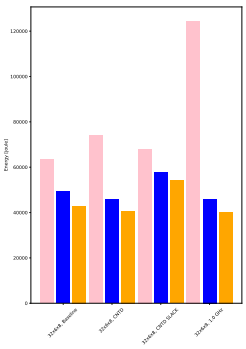
(b) EXE time: distribution of execution times for the 51 runs per instance.



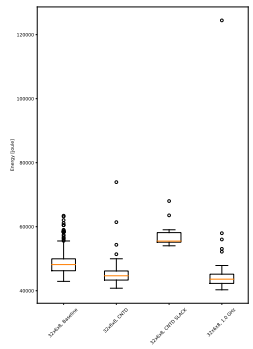
(c) APP time: distribution of execution times for the 51 runs per instance.



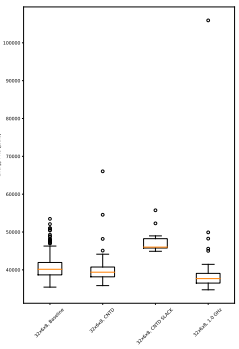
(d) MPI time: distribution of execution times for the 51 runs per instance.



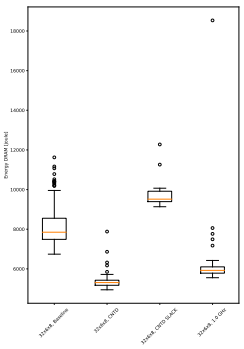
(e) Energy: variations as COUNTDOWN configuration.



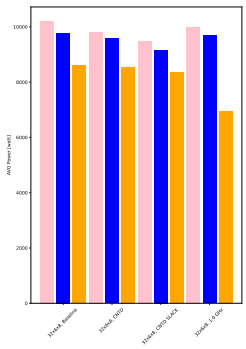
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



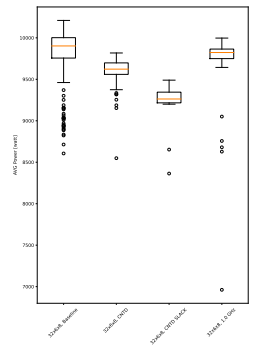
(g) Energy: boxplot of distribution of the energy consumed by CPU.



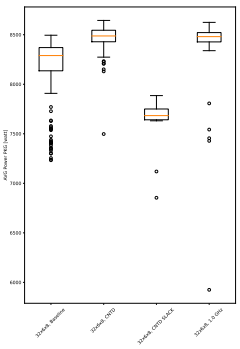
(h) Energy: boxplot of distribution of the energy consumed by RAM.



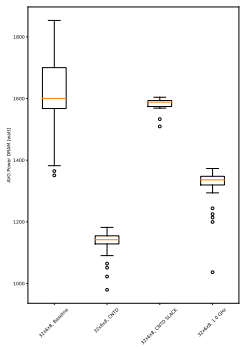
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

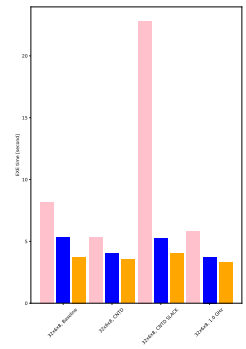


(k) Power: boxplot of distribution of the power consumed by CPU.

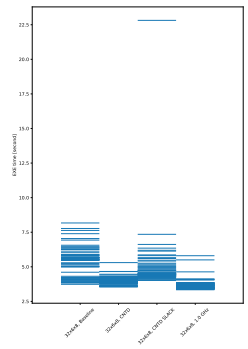


(l) Power: boxplot of distribution of the power consumed by RAM.

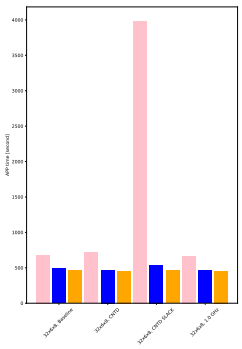
Figure B.3: Experimental Results for Experiment 4 on the *Cubo_1772481.Ext_bin* matrix.



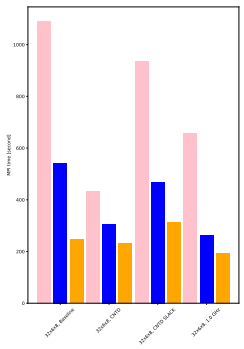
(a) EXE time: variations as COUNTDOWN configuration.



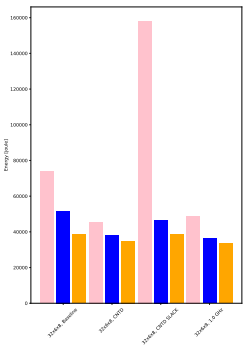
(b) EXE time: distribution of execution times for the 51 runs per instance.



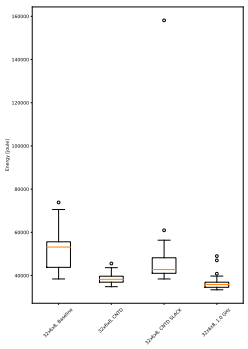
(c) APP time: distribution of execution times for the 51 runs per instance.



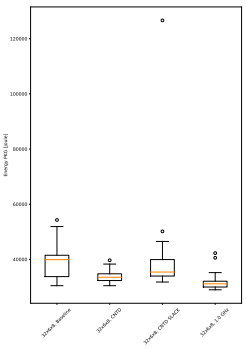
(d) MPI time: distribution of execution times for the 51 runs per instance.



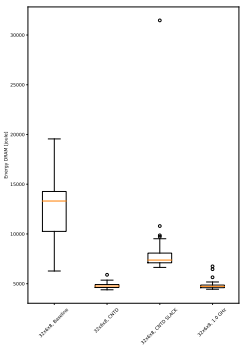
(e) Energy: variations as COUNTDOWN configuration.



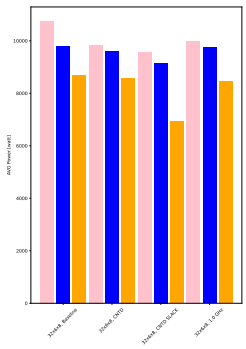
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



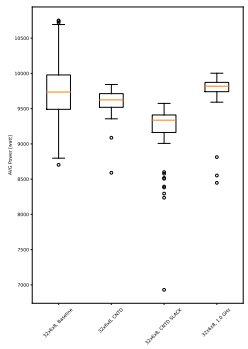
(g) Energy: boxplot of distribution of the energy consumed by CPU.



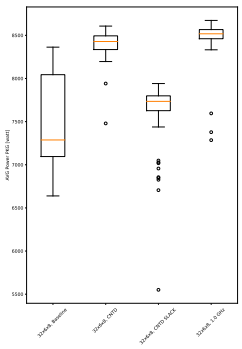
(h) Energy: boxplot of distribution of the energy consumed by RAM.



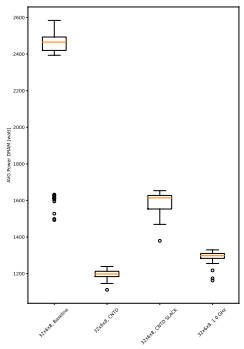
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

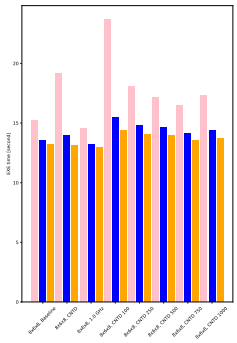


(k) Power: boxplot of distribution of the power consumed by CPU.

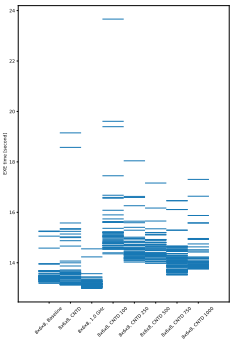


(l) Power: boxplot of distribution of the power consumed by RAM.

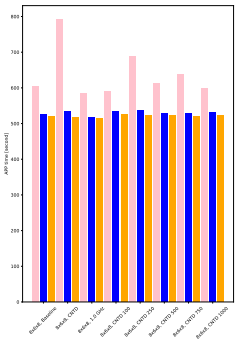
Figure B.4: Experimental Results for Experiment 4 on the *Wing_4538k.csr.Ext_bin* matrix.



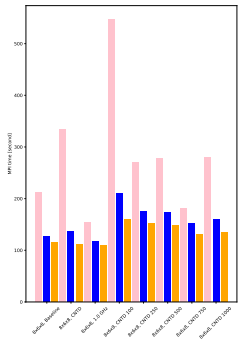
(a) EXE time: variations as COUNTDOWN configuration.



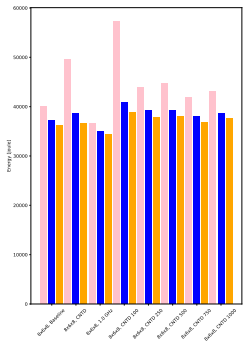
(b) EXE time: distribution of execution times for the 51 runs per instance.



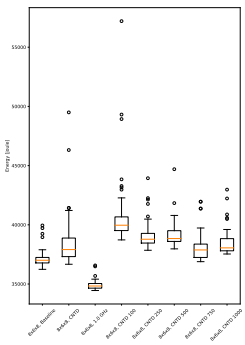
(c) APP time: distribution of execution times for the 51 runs per instance.



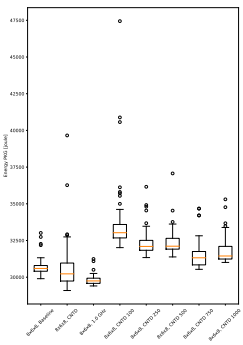
(d) MPI time: distribution of execution times for the 51 runs per instance.



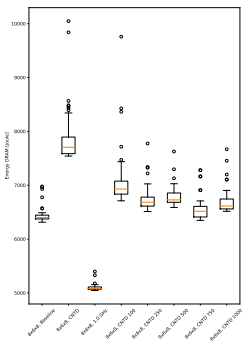
(e) Energy: variations as COUNTDOWN configuration.



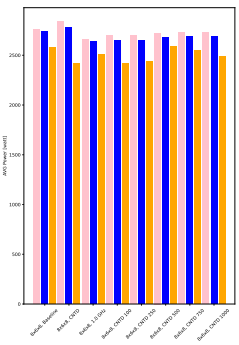
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



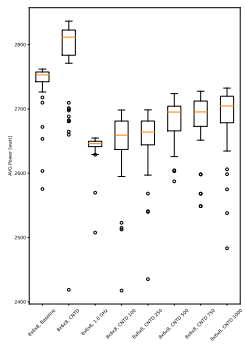
(g) Energy: boxplot of distribution of the energy consumed by CPU.



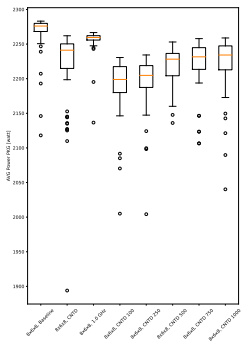
(h) Energy: boxplot of distribution of the energy consumed by RAM.



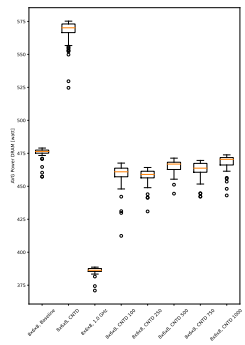
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

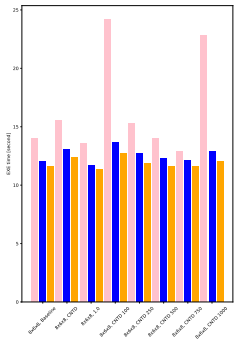


(k) Power: boxplot of distribution of the power consumed by CPU.

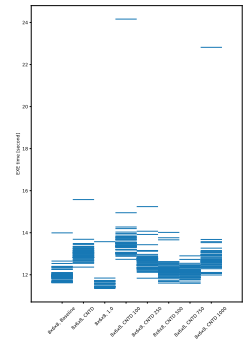


(l) Power: boxplot of distribution of the power consumed by RAM.

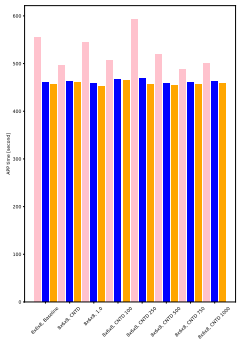
Figure B.5: Experimental Results for Experiment 5a on the *Cubo_1772481.Ext_bin* matrix.



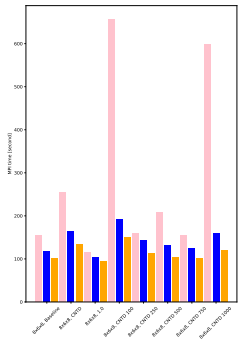
(a) EXE time: variations as COUNTDOWN configuration.



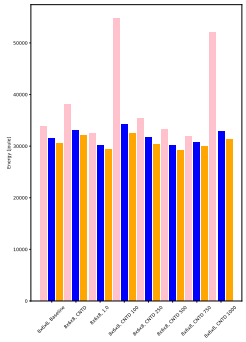
(b) EXE time: distribution of execution times for the 51 runs per instance.



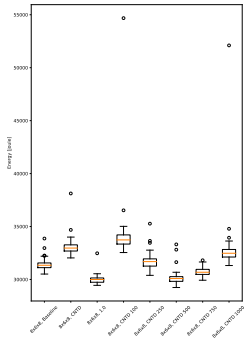
(c) APP time: distribution of execution times for the 51 runs per instance.



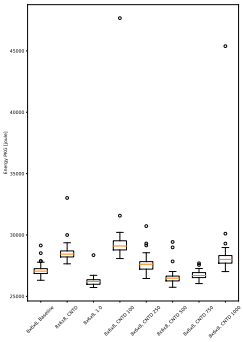
(d) MPI time: distribution of execution times for the 51 runs per instance.



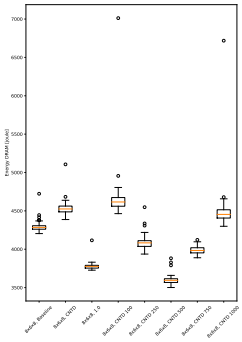
(e) Energy: variations as COUNTDOWN configuration.



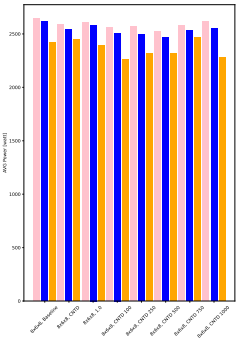
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



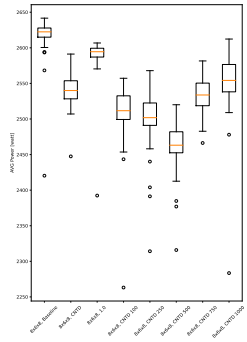
(g) Energy: boxplot of distribution of the energy consumed by CPU.



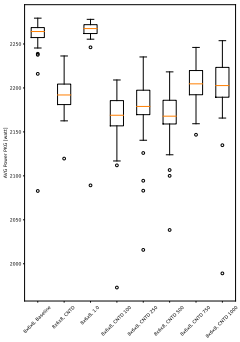
(h) Energy: boxplot of distribution of the energy consumed by RAM.



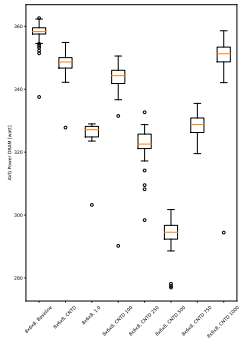
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

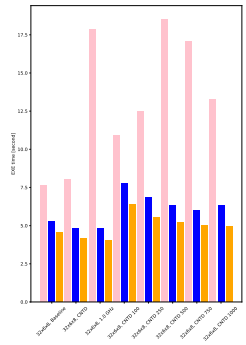


(k) Power: boxplot of distribution of the power consumed by CPU.

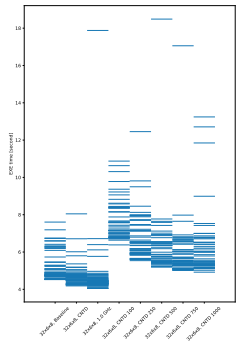


(l) Power: boxplot of distribution of the power consumed by RAM.

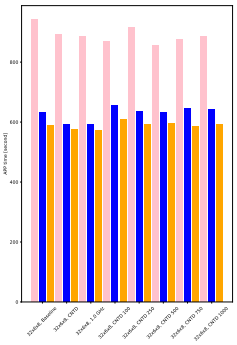
Figure B.6: Experimental Results for Experiment 5b on the *Wing_4538k.csr.Ext_bin* matrix.



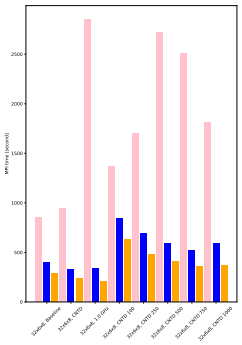
(a) EXE time: variations as COUNTDOWN configuration.



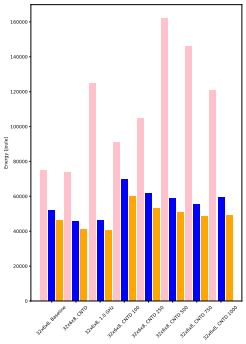
(b) EXE time: distribution of execution times for the 51 runs per instance.



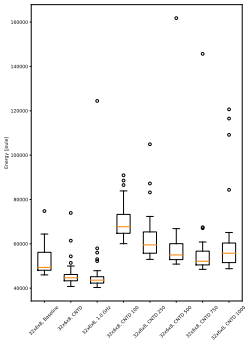
(c) APP time: distribution of execution times for the 51 runs per instance.



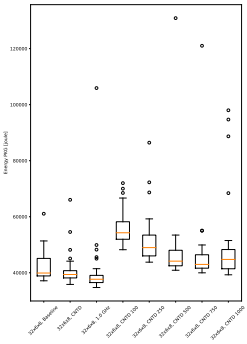
(d) MPI time: distribution of execution times for the 51 runs per instance.



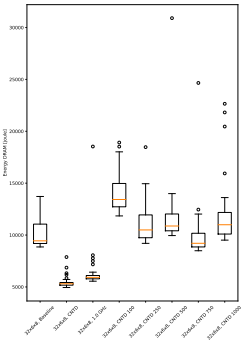
(e) Energy: variations as COUNTDOWN configuration.



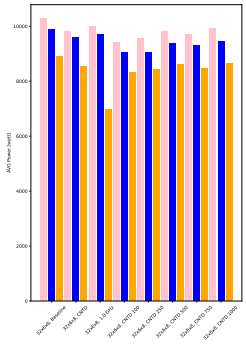
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



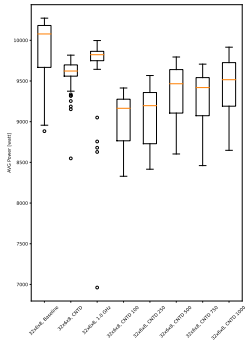
(g) Energy: boxplot of distribution of the energy consumed by CPU.



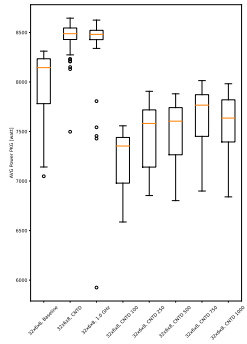
(h) Energy: boxplot of distribution of the energy consumed by RAM.



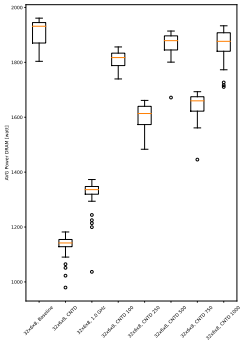
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

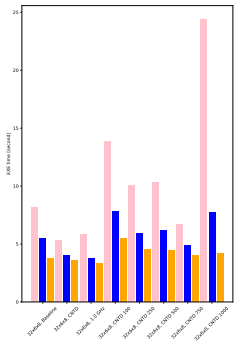


(k) Power: boxplot of distribution of the power consumed by CPU.

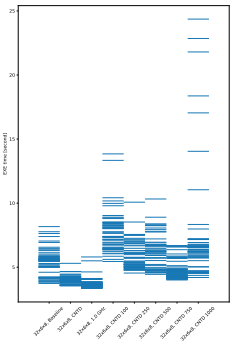


(l) Power: boxplot of distribution of the power consumed by RAM.

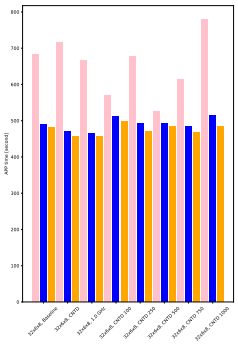
Figure B.7: Experimental Results for Experiment 5b on the *Cubo_1772481.Ext_bin* matrix.



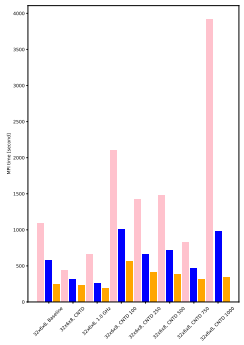
(a) EXE time: variations as COUNTDOWN configuration.



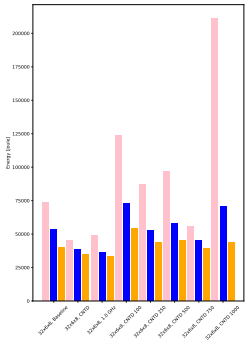
(b) EXE time: distribution of execution times for the 51 runs per instance.



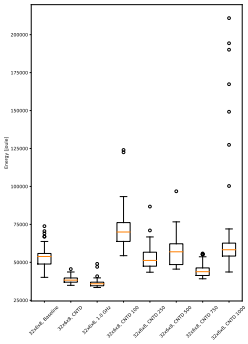
(c) APP time: distribution of execution times for the 51 runs per instance.



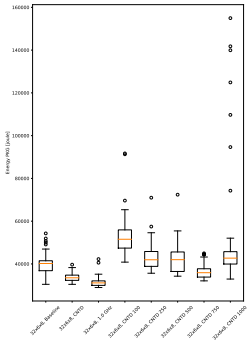
(d) MPI time: distribution of execution times for the 51 runs per instance.



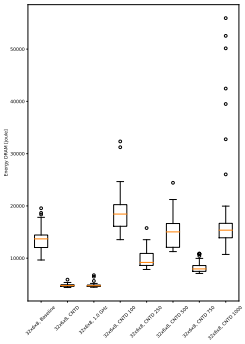
(e) Energy: variations as COUNTDOWN configuration.



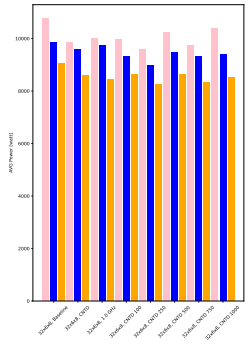
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



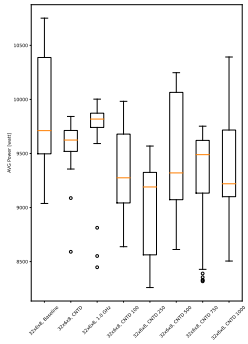
(g) Energy: boxplot of distribution of the energy consumed by CPU.



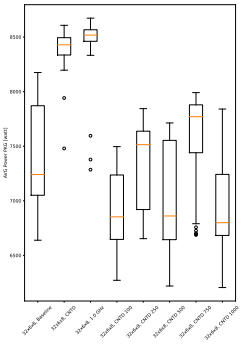
(h) Energy: boxplot of distribution of the energy consumed by RAM.



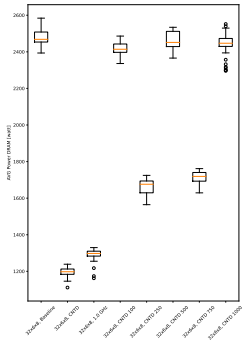
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

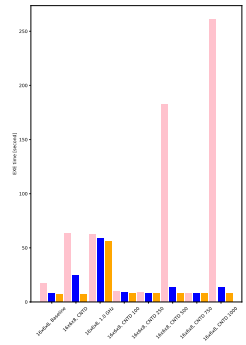


(k) Power: boxplot of distribution of the power consumed by CPU.

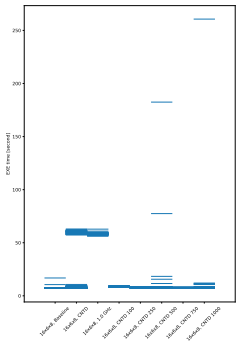


(l) Power: boxplot of distribution of the power consumed by RAM.

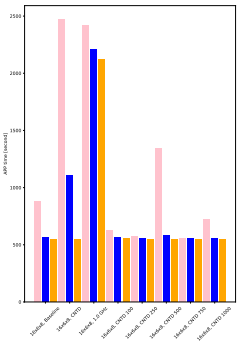
Figure B.8: Experimental Results for Experiment 5b on the *Wing_4538k.csr.Ext_bin* matrix.



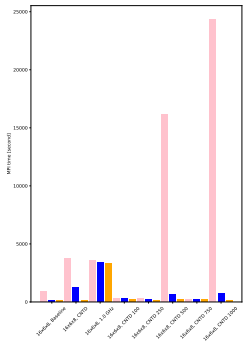
(a) EXE time: variations as COUNTDOWN configuration.



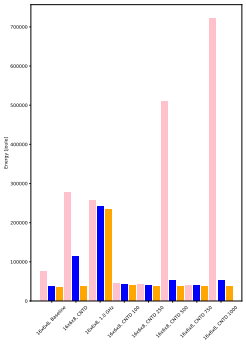
(b) EXE time: distribution of execution times for the 51 runs per instance.



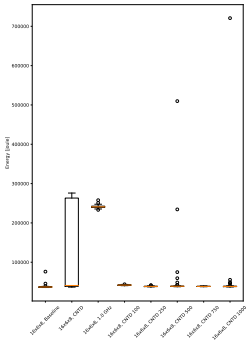
(c) APP time: distribution of execution times for the 51 runs per instance.



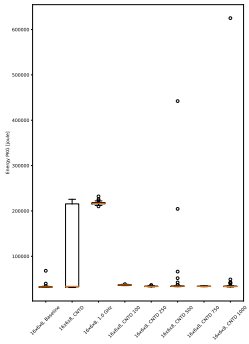
(d) MPI time: distribution of execution times for the 51 runs per instance.



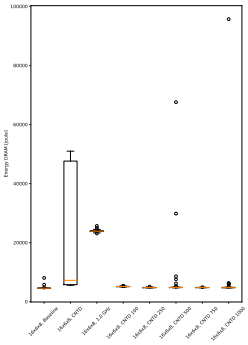
(e) Energy: variations as COUNTDOWN configuration.



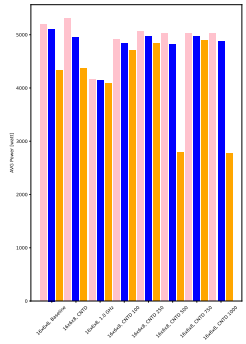
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



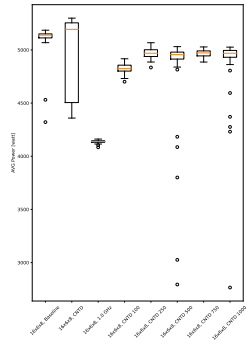
(g) Energy: boxplot of distribution of the energy consumed by CPU.



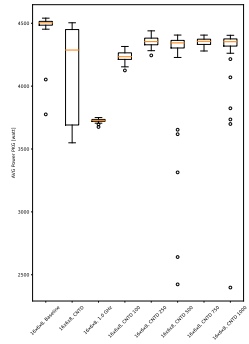
(h) Energy: boxplot of distribution of the energy consumed by RAM.



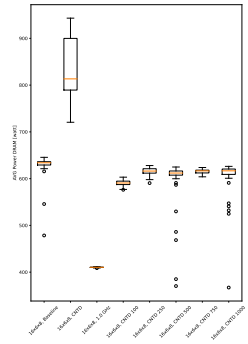
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

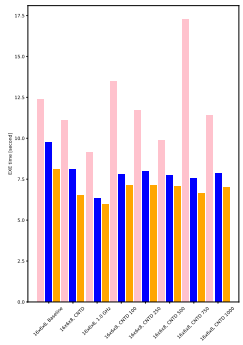


(k) Power: boxplot of distribution of the power consumed by CPU.

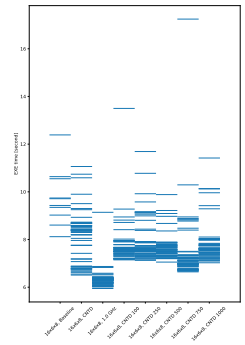


(l) Power: boxplot of distribution of the power consumed by RAM.

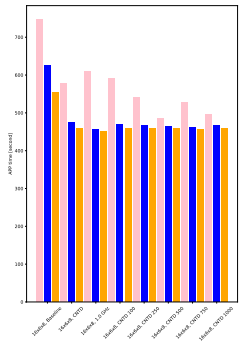
Figure B.9: Experimental Results for Experiment 5c on the *Cubo_1772481.Ext_bin* matrix.



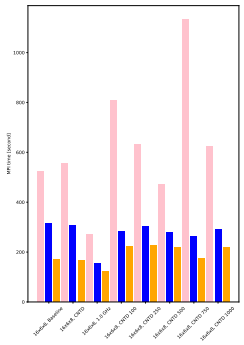
(a) EXE time: variations as COUNTDOWN configuration.



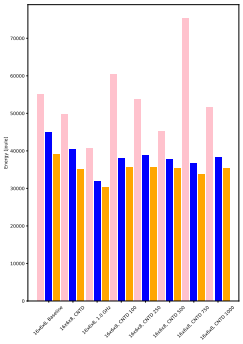
(b) EXE time: distribution of execution times for the 51 runs per instance.



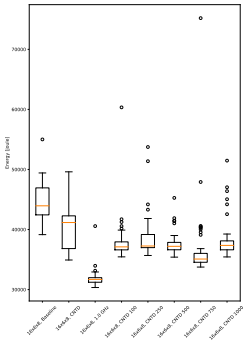
(c) APP time: distribution of execution times for the 51 runs per instance.



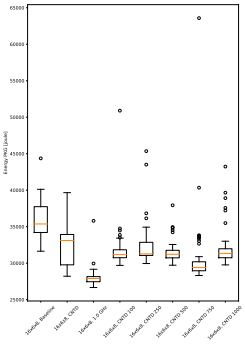
(d) MPI time: distribution of execution times for the 51 runs per instance.



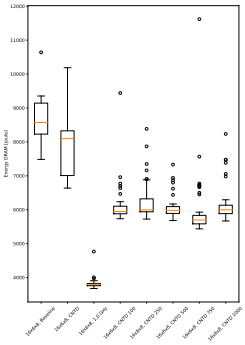
(e) Energy: variations as COUNTDOWN configuration.



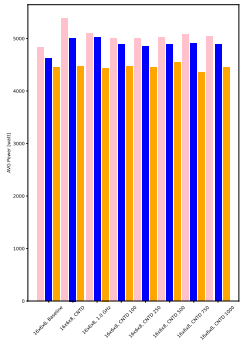
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



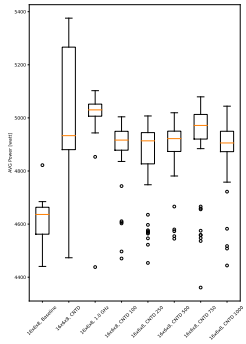
(g) Energy: boxplot of distribution of the energy consumed by CPU.



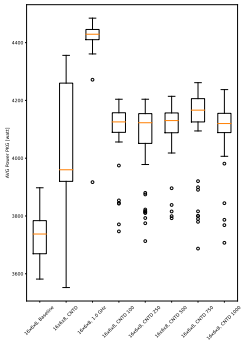
(h) Energy: boxplot of distribution of the energy consumed by RAM.



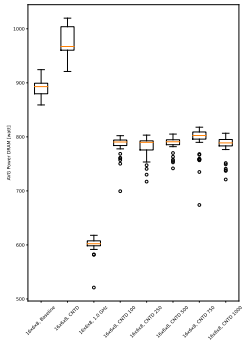
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

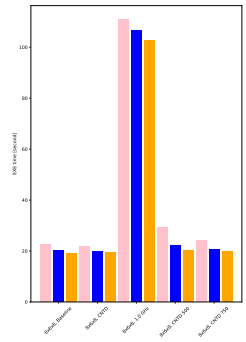


(k) Power: boxplot of distribution of the power consumed by CPU.

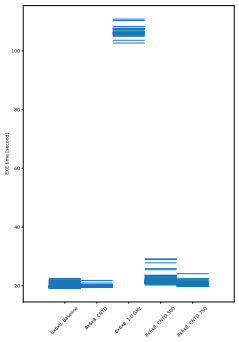


(l) Power: boxplot of distribution of the power consumed by RAM.

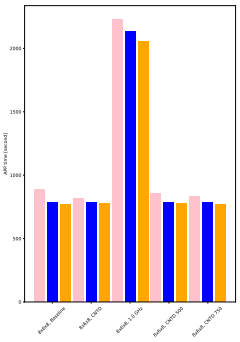
Figure B.10: Experimental Results for Experiment 5c on the *Wing_4538k.csr.Ext_bin* matrix.



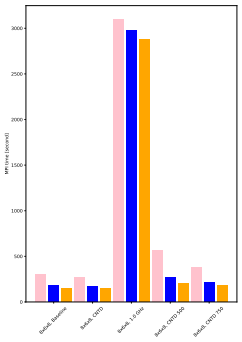
(a) EXE time: variations as COUNTDOWN configuration.



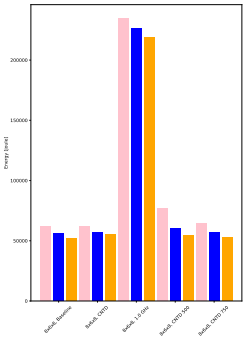
(b) EXE time: distribution of execution times for the 51 runs per instance.



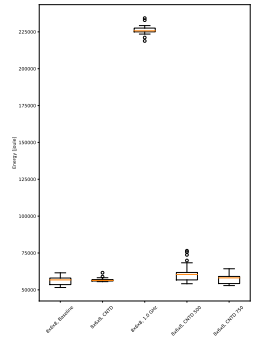
(c) APP time: distribution of execution times for the 51 runs per instance.



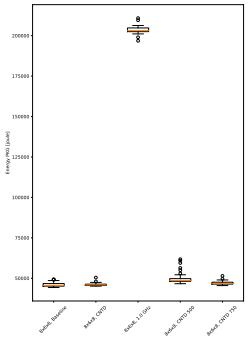
(d) MPI time: distribution of execution times for the 51 runs per instance.



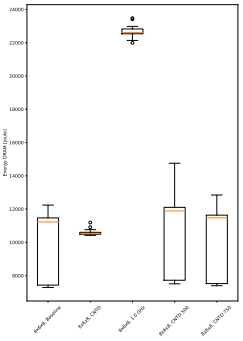
(e) Energy: variations as COUNTDOWN configuration.



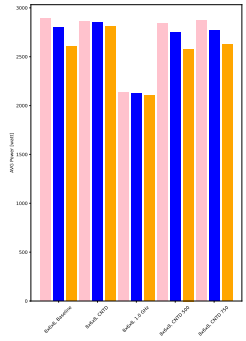
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



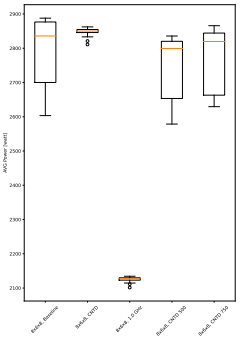
(g) Energy: boxplot of distribution of the energy consumed by CPU.



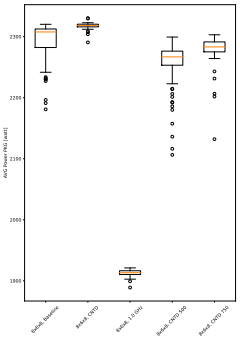
(h) Energy: boxplot of distribution of the energy consumed by RAM.



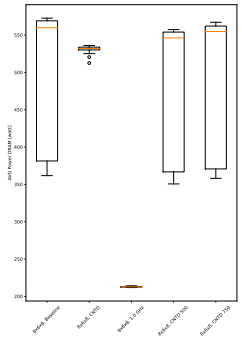
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

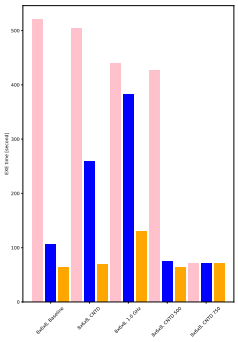


(k) Power: boxplot of distribution of the power consumed by CPU.

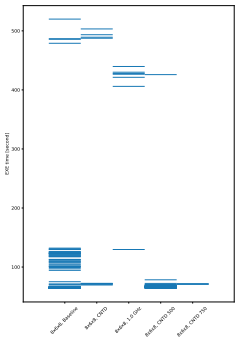


(l) Power: boxplot of distribution of the power consumed by RAM.

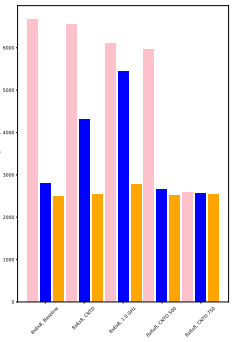
Figure B.11: Experimental Results for Experiment 6a on the *Cubo_1772481.Ext_bin* matrix.



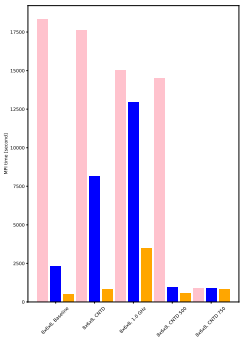
(a) EXE time: variations as COUNTDOWN configuration.



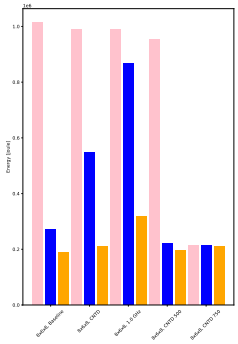
(b) EXE time: distribution of execution times for the 51 runs per instance.



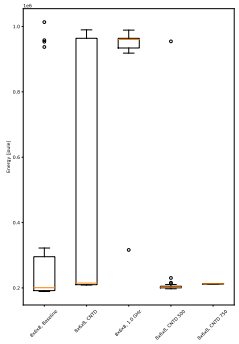
(c) APP time: distribution of execution times for the 51 runs per instance.



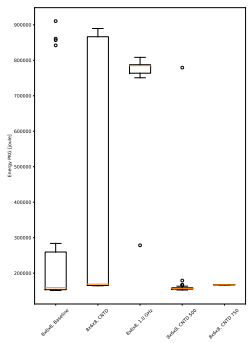
(d) MPI time: distribution of execution times for the 51 runs per instance.



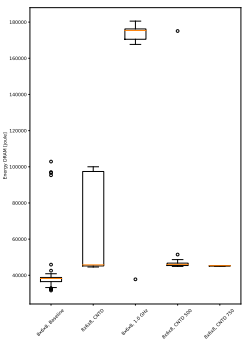
(e) Energy: variations as COUNTDOWN configuration.



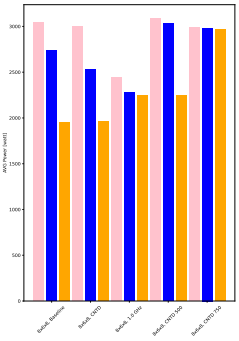
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



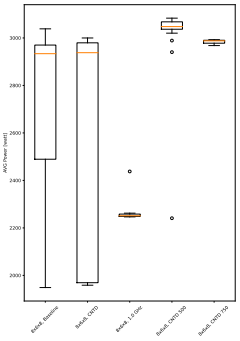
(g) Energy: boxplot of distribution of the energy consumed by CPU.



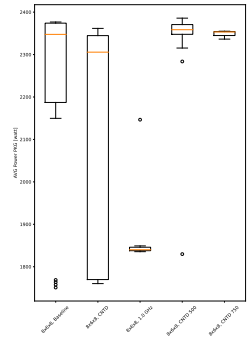
(h) Energy: boxplot of distribution of the energy consumed by RAM.



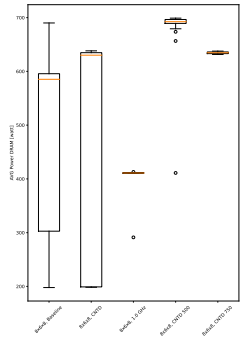
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

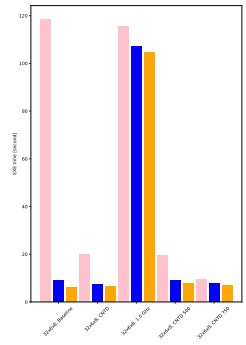


(k) Power: boxplot of distribution of the power consumed by CPU.

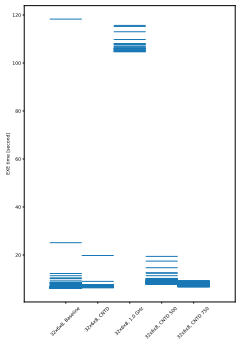


(l) Power: boxplot of distribution of the power consumed by RAM.

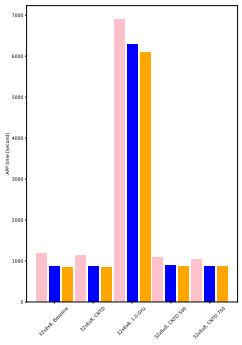
Figure B.12: Experimental Results for Experiment 6a on the *Wing_4538k.csr.Ext_bin* matrix.



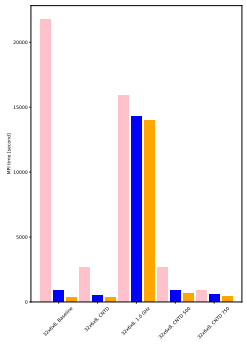
(a) EXE time: variations as COUNTDOWN configuration.



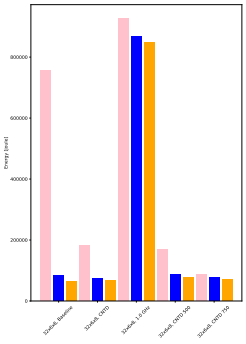
(b) EXE time: distribution of execution times for the 51 runs per instance.



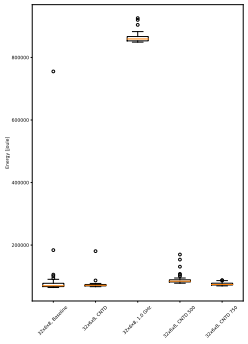
(c) APP time: distribution of execution times for the 51 runs per instance.



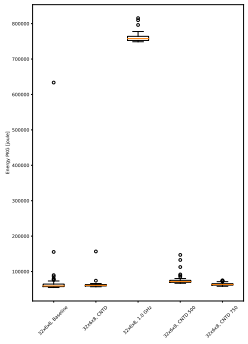
(d) MPI time: distribution of execution times for the 51 runs per instance.



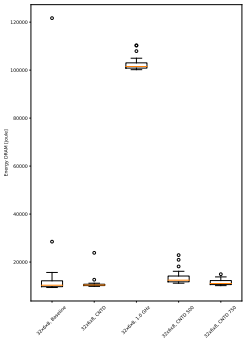
(e) Energy: variations as COUNTDOWN configuration.



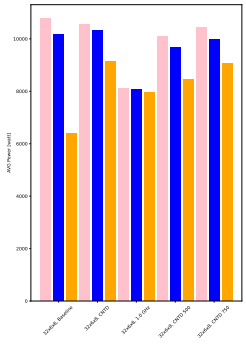
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



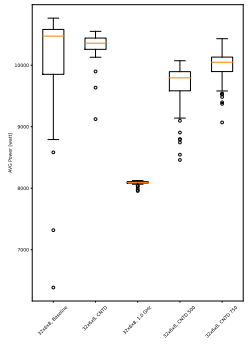
(g) Energy: boxplot of distribution of the energy consumed by CPU.



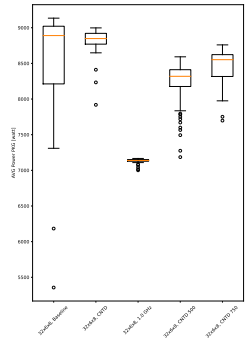
(h) Energy: boxplot of distribution of the energy consumed by RAM.



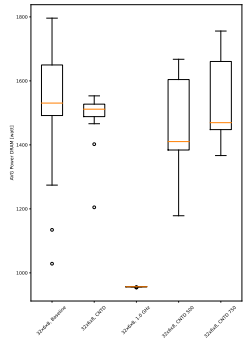
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

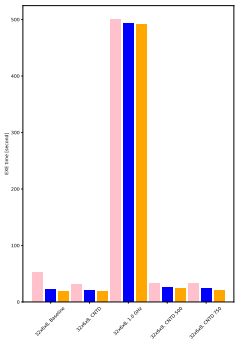


(k) Power: boxplot of distribution of the power consumed by CPU.

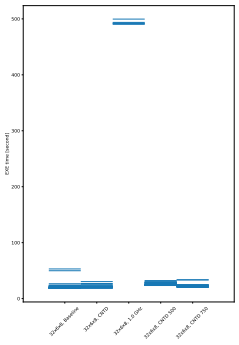


(l) Power: boxplot of distribution of the power consumed by RAM.

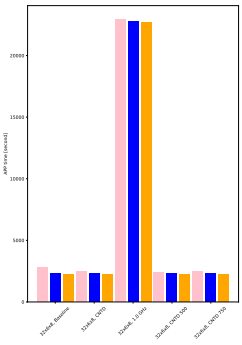
Figure B.13: Experimental Results for Experiment 6b on the *Cubo_1772481.Ext_bin* matrix.



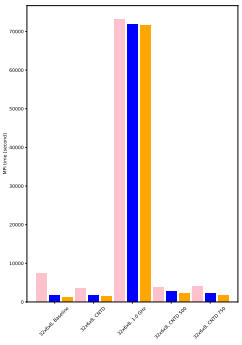
(a) EXE time: variations as COUNTDOWN configuration.



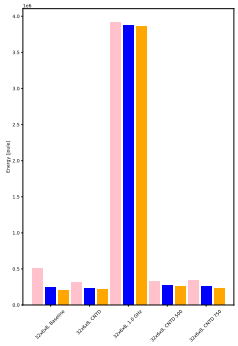
(b) EXE time: distribution of execution times for the 51 runs per instance.



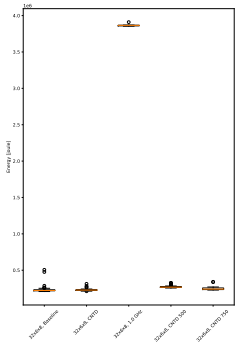
(c) APP time: distribution of execution times for the 51 runs per instance.



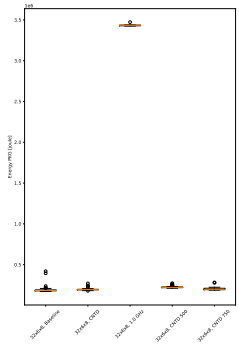
(d) MPI time: distribution of execution times for the 51 runs per instance.



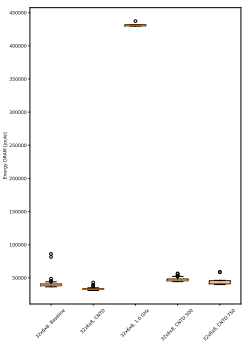
(e) Energy: variations as COUNTDOWN configuration.



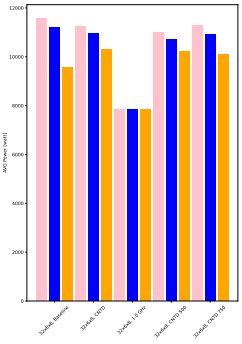
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



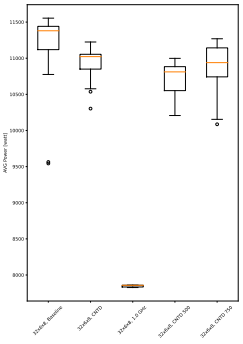
(g) Energy: boxplot of distribution of the energy consumed by CPU.



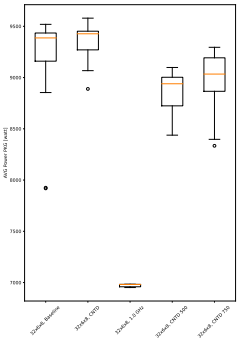
(h) Energy: boxplot of distribution of the energy consumed by RAM.



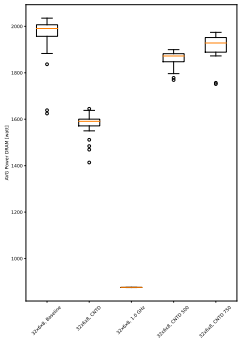
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

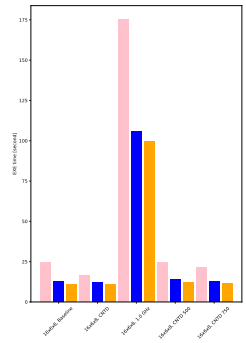


(k) Power: boxplot of distribution of the power consumed by CPU.

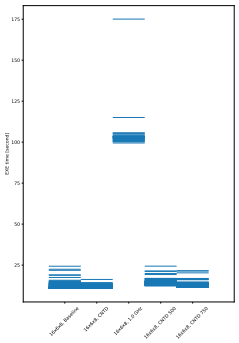


(l) Power: boxplot of distribution of the power consumed by RAM.

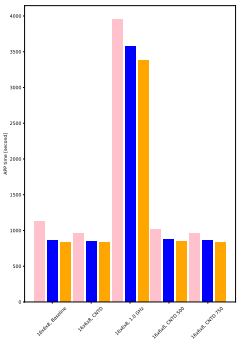
Figure B.14: Experimental Results for Experiment 6b on the *Wing_4538k.csr.Ext_bin* matrix.



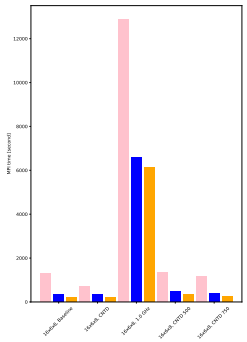
(a) EXE time: variations as COUNTDOWN configuration.



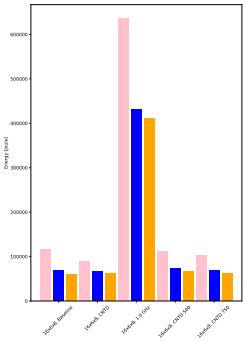
(b) EXE time: distribution of execution times for the 51 runs per instance.



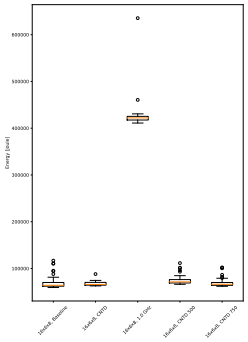
(c) APP time: distribution of execution times for the 51 runs per instance.



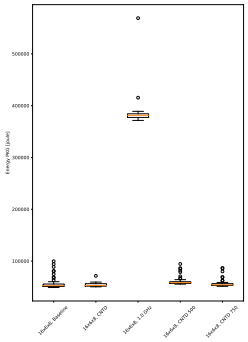
(d) MPI time: distribution of execution times for the 51 runs per instance.



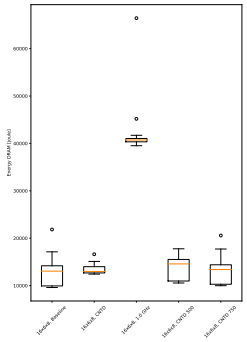
(e) Energy: variations as COUNTDOWN configuration.



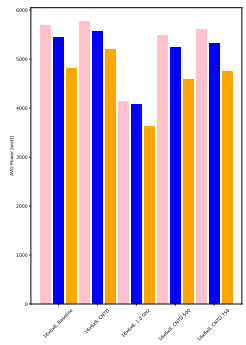
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



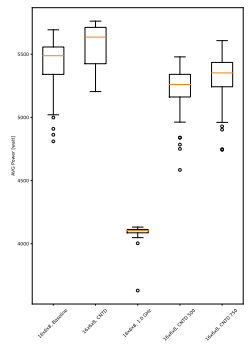
(g) Energy: boxplot of distribution of the energy consumed by CPU.



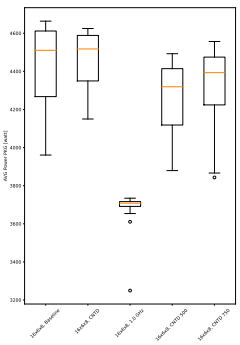
(h) Energy: boxplot of distribution of the energy consumed by RAM.



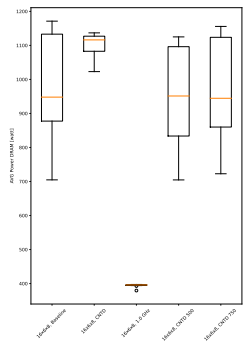
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

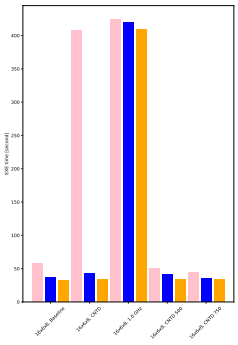


(k) Power: boxplot of distribution of the power consumed by CPU.

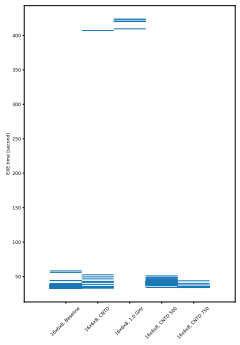


(l) Power: boxplot of distribution of the power consumed by RAM.

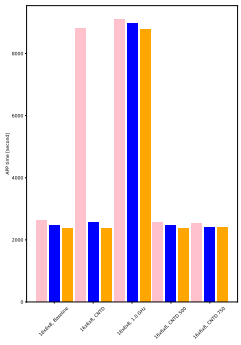
Figure B.15: Experimental Results for Experiment 6c on the *Cubo_1772481.Ext_bin* matrix.



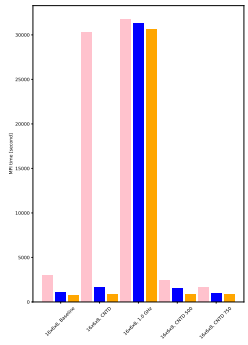
(a) EXE time: variations as COUNTDOWN configuration.



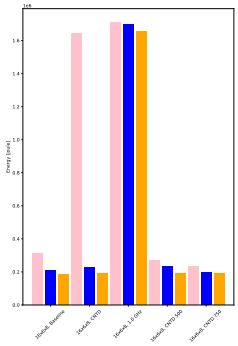
(b) EXE time: distribution of execution times for the 51 runs per instance.



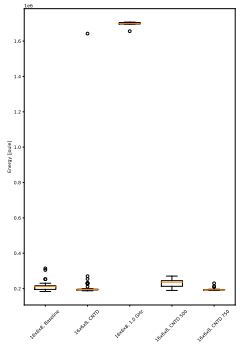
(c) APP time: distribution of execution times for the 51 runs per instance.



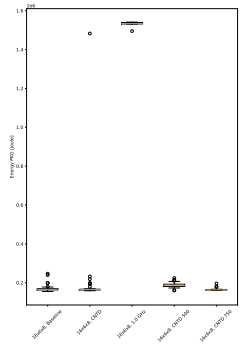
(d) MPI time: distribution of execution times for the 51 runs per instance.



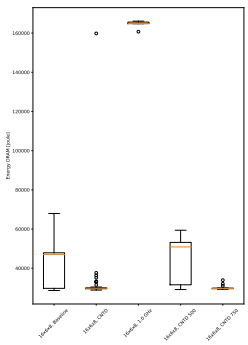
(e) Energy: variations as COUNTDOWN configuration.



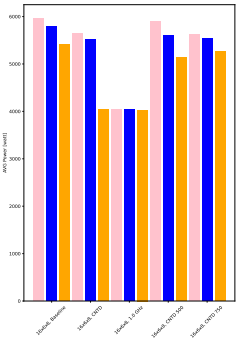
(f) Energy: boxplot of distribution as COUNTDOWN configuration.



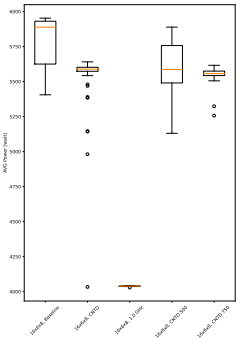
(g) Energy: boxplot of distribution of the energy consumed by CPU.



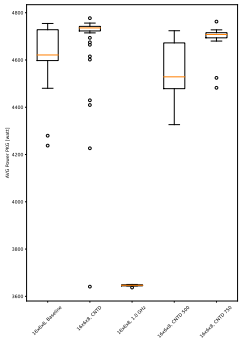
(h) Energy: boxplot of distribution of the energy consumed by RAM.



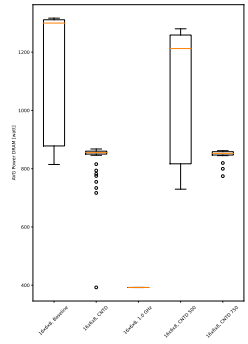
(i) Power: variations as COUNTDOWN configuration.



(j) Power: boxplot of distribution as COUNTDOWN configuration.

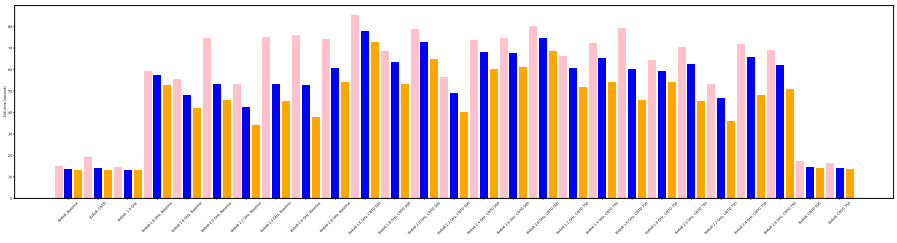


(k) Power: boxplot of distribution of the power consumed by CPU.

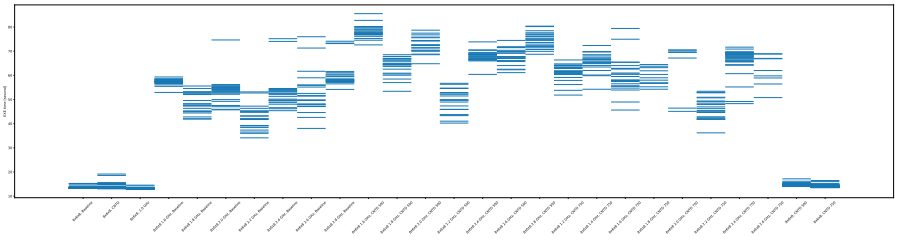


(l) Power: boxplot of distribution of the power consumed by RAM.

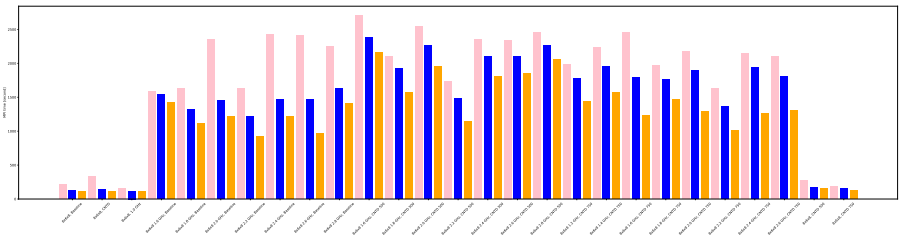
Figure B.16: Experimental Results for Experiment 6c on the *Wing_4538k.csr.Ext_bin* matrix.



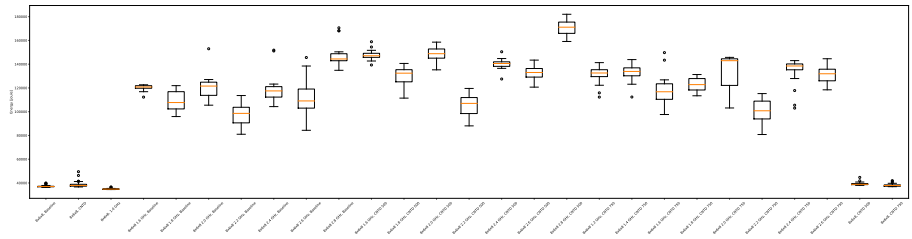
(a) EXE time: variations as COUNTDOWN configuration.



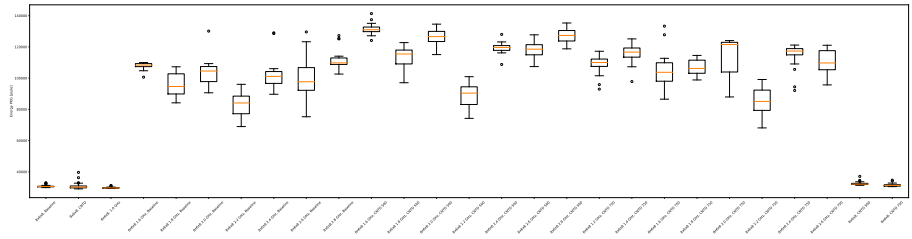
(b) EXE time: distribution of execution times for the 51 runs per instance.



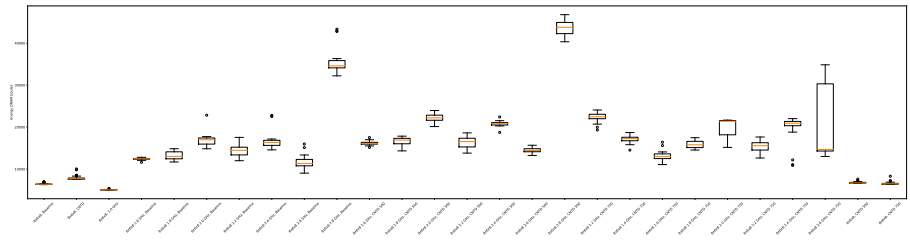
(c) MPI time: distribution of execution times for the 51 runs per instance.



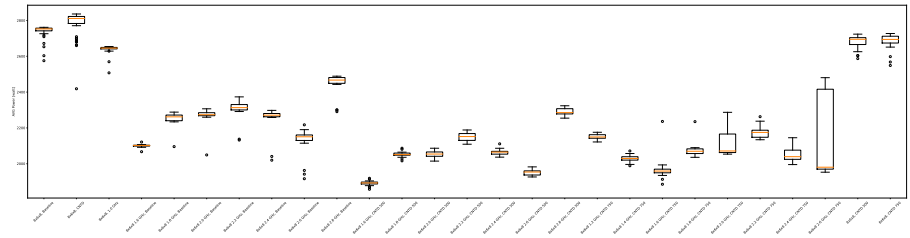
(d) Energy: boxplot of distribution as COUNTDOWN configuration.



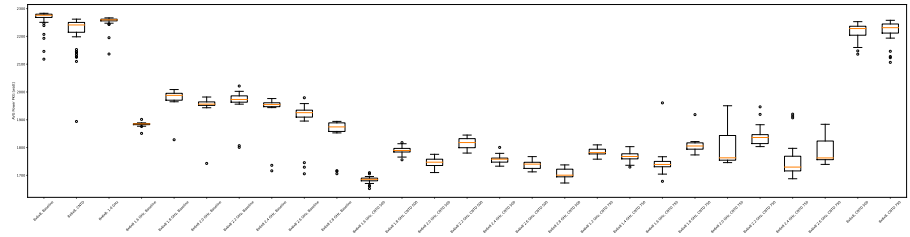
(e) Energy: boxplot of distribution of the energy consumed by CPU.



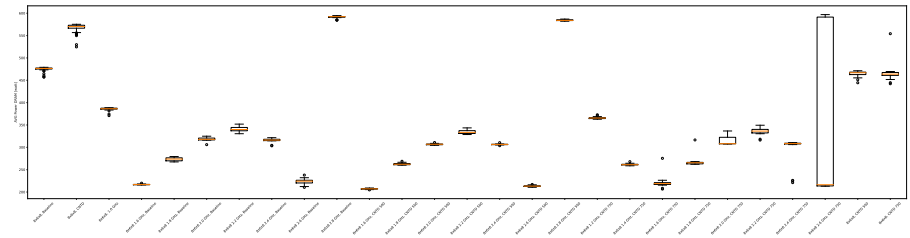
(f) Energy: boxplot of distribution of the energy consumed by RAM.



(g) Power: boxplot of distribution as COUNTDOWN configuration.

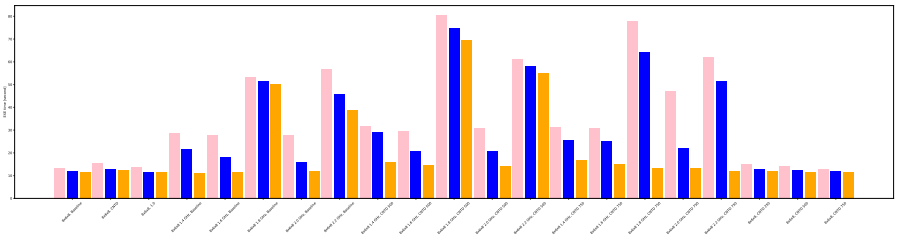


(h) Power: boxplot of distribution of the power consumed by CPU.

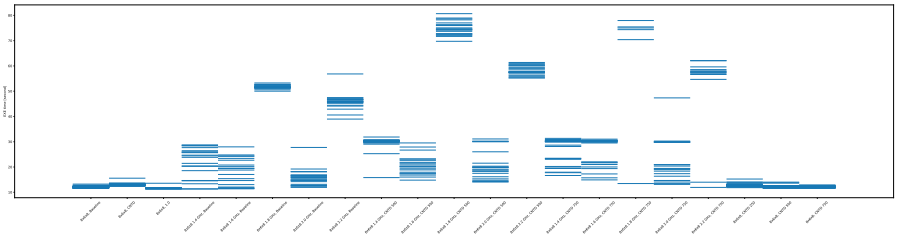


(i) Power: boxplot of distribution of the power consumed by RAM.

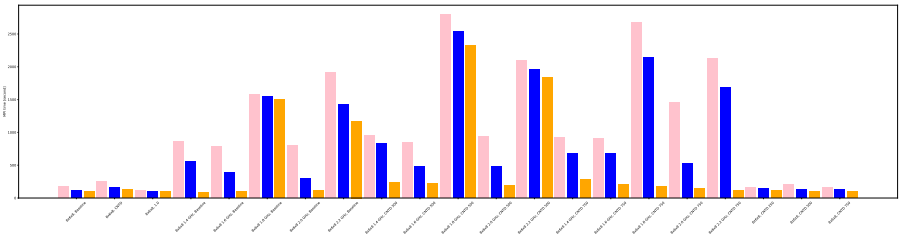
Figure B.17: Experimental Results for Experiment 7 on the *Cubo_1772481.Ext_bin* matrix.



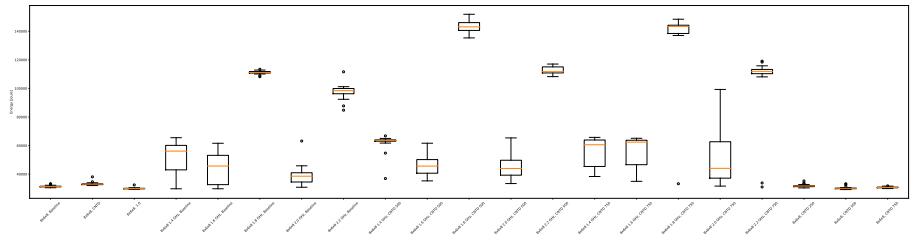
(a) EXE time: variations as COUNTDOWN configuration.



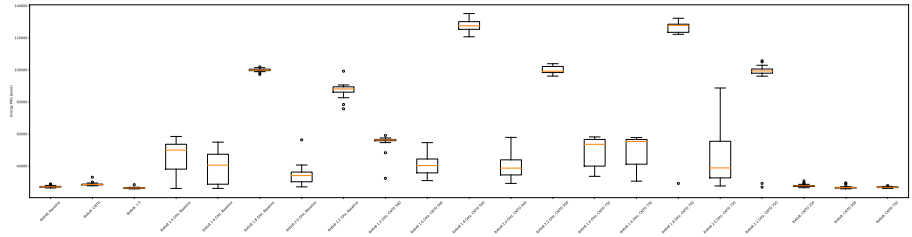
(b) EXE time: distribution of execution times for the 51 runs per instance.



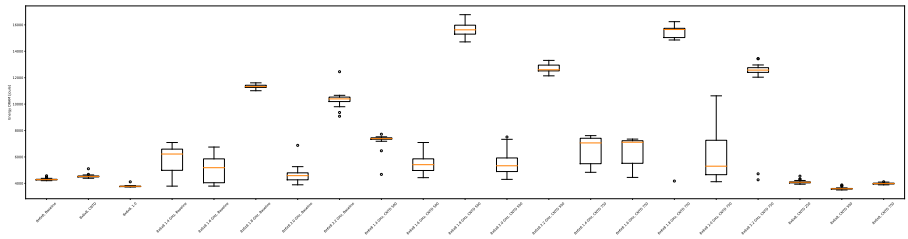
(c) MPI time: distribution of execution times for the 51 runs per instance.



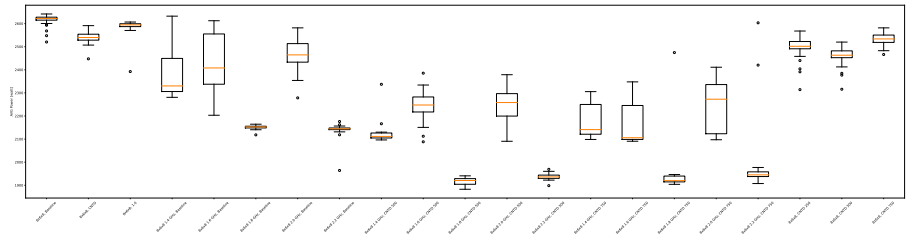
(d) Energy: boxplot of distribution as COUNTDOWN configuration.



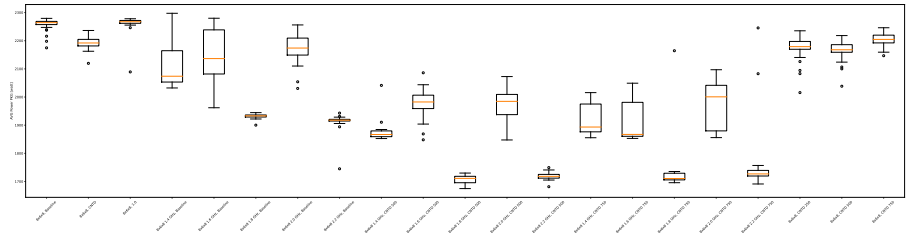
(e) Energy: boxplot of distribution of the energy consumed by CPU.



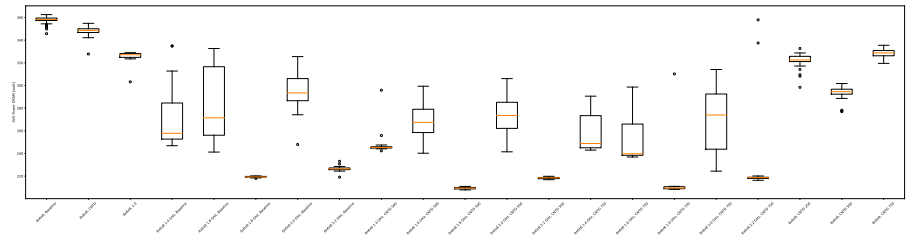
(f) Energy: boxplot of distribution of the energy consumed by RAM.



(g) Power: boxplot of distribution as COUNTDOWN configuration.

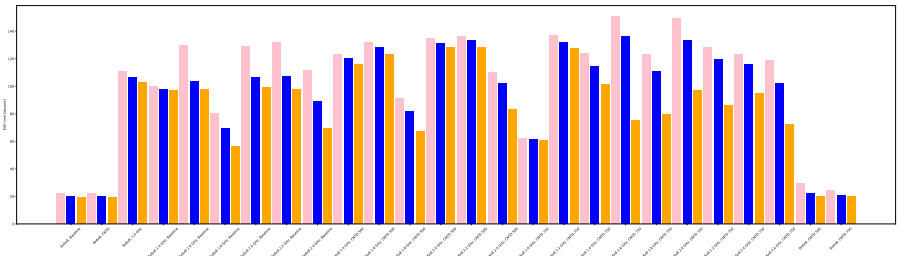


(h) Power: boxplot of distribution of the power consumed by CPU.

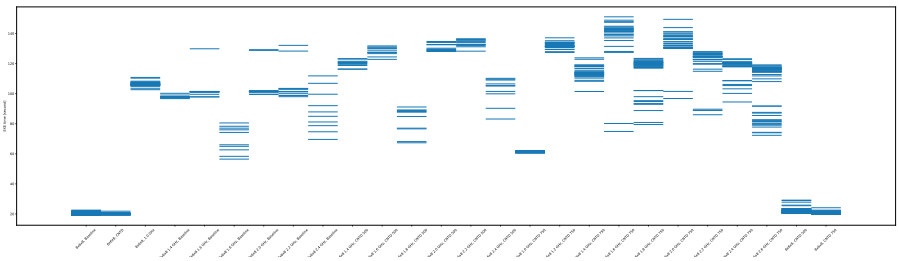


(i) Power: boxplot of distribution of the power consumed by RAM.

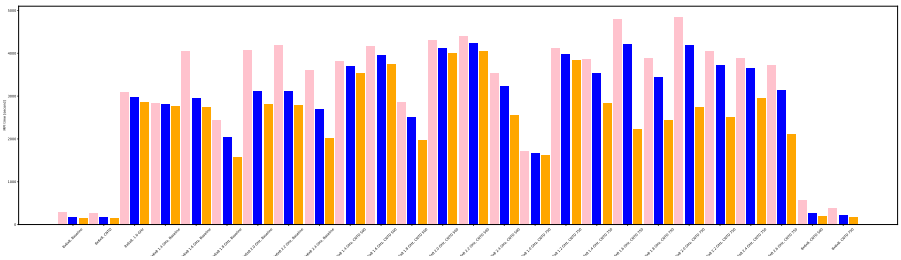
Figure B.18: Experimental Results for Experiment 7 on the *Wing_4538k.csr.Ext_bin* matrix.



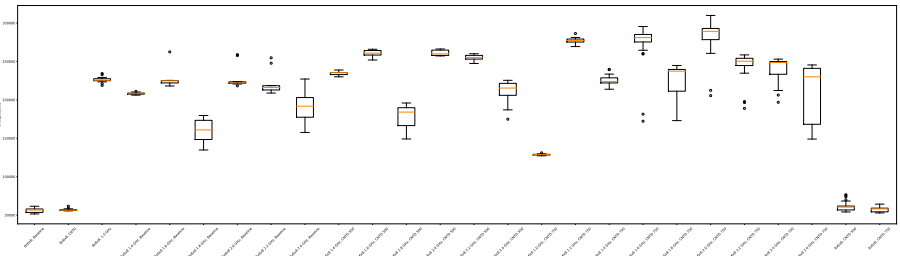
(a) EXE time: variations as COUNTDOWN configuration.



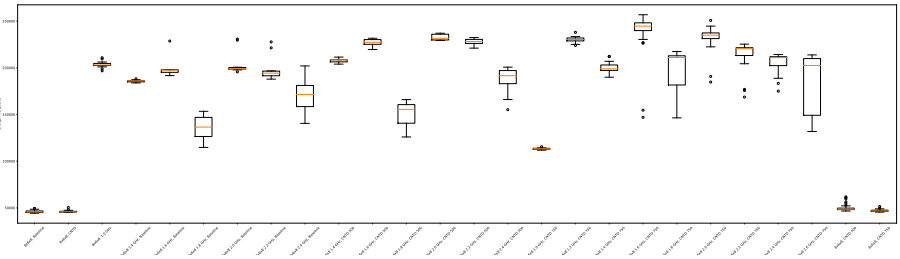
(b) EXE time: distribution of execution times for the 51 runs per instance.



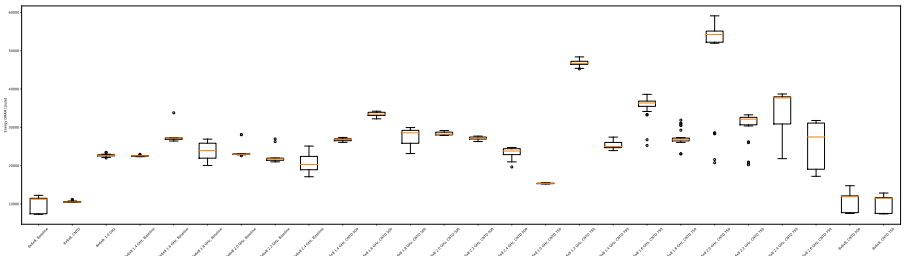
(c) MPI time: distribution of execution times for the 51 runs per instance.



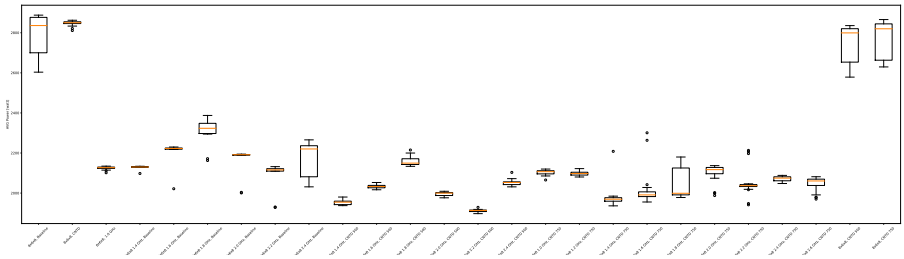
(d) Energy: boxplot of distribution as COUNTDOWN configuration.



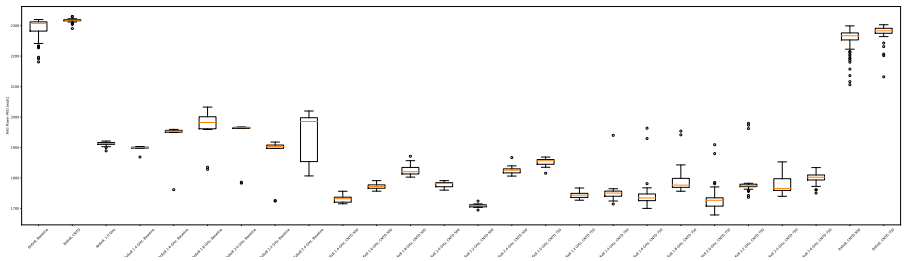
(e) Energy: boxplot of distribution of the energy consumed by CPU.



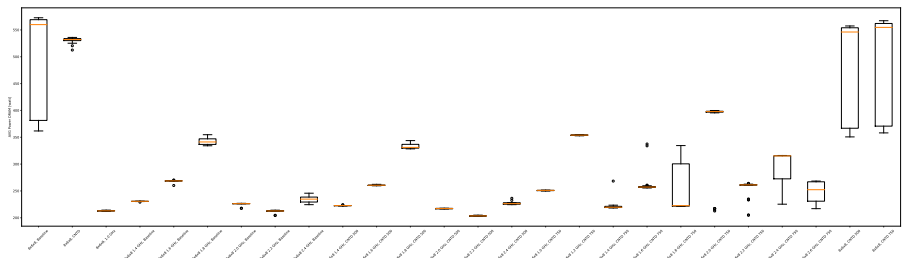
(f) Energy: boxplot of distribution of the energy consumed by RAM.



(g) Power: boxplot of distribution as COUNTDOWN configuration.

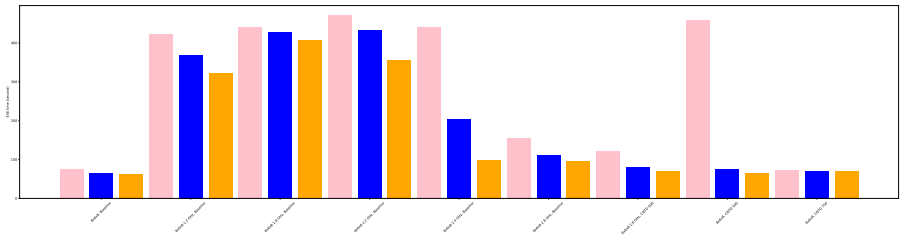


(h) Power: boxplot of distribution of the power consumed by CPU.

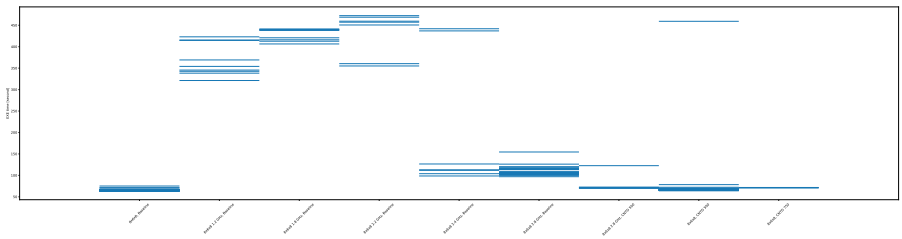


(i) Power: boxplot of distribution of the power consumed by RAM.

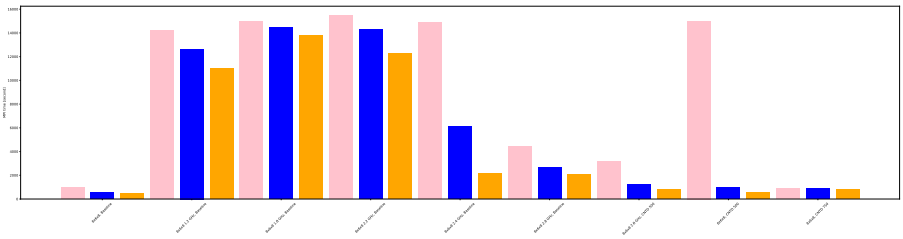
Figure B.19: Experimental Results for Experiment 8 on the *Cubo_1772481.Ext_bin* matrix.



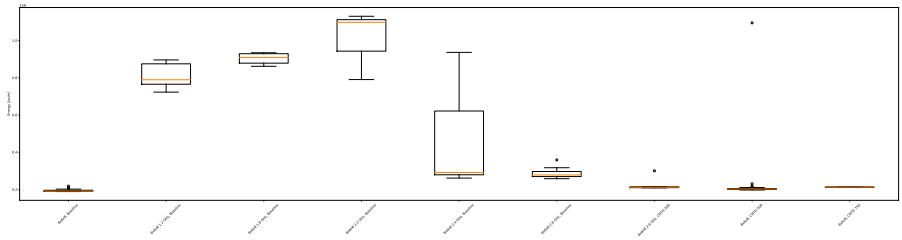
(a) EXE time: variations as COUNTDOWN configuration.



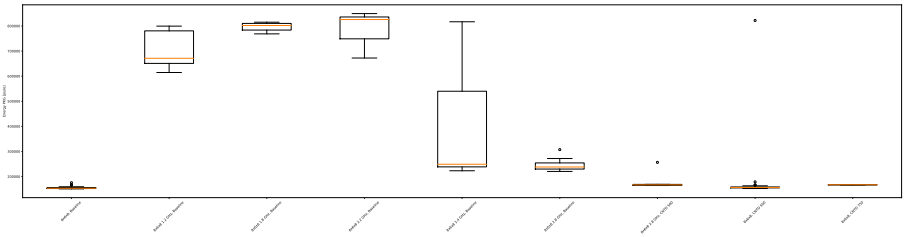
(b) EXE time: distribution of execution times for the 51 runs per instance.



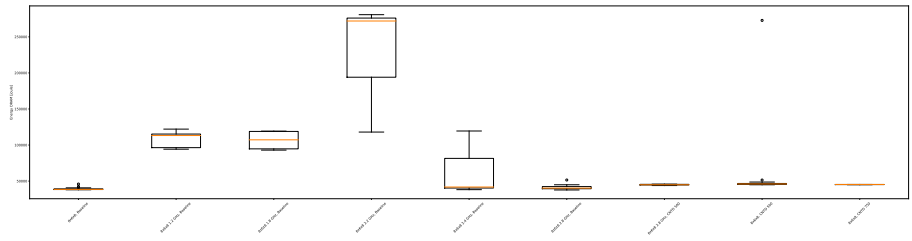
(c) MPI time: distribution of execution times for the 51 runs per instance.



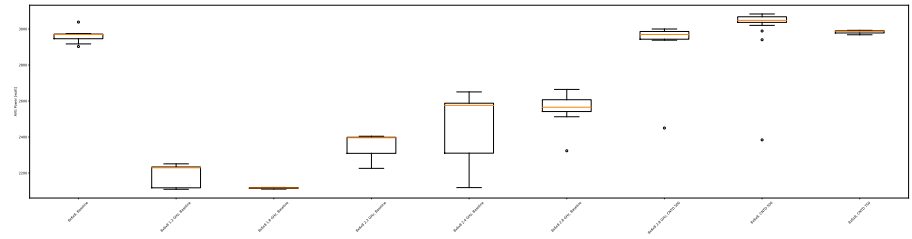
(d) Energy: boxplot of distribution as COUNTDOWN configuration.



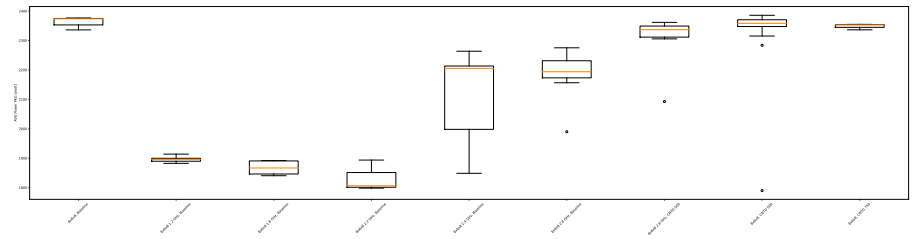
(e) Energy: boxplot of distribution of the energy consumed by CPU.



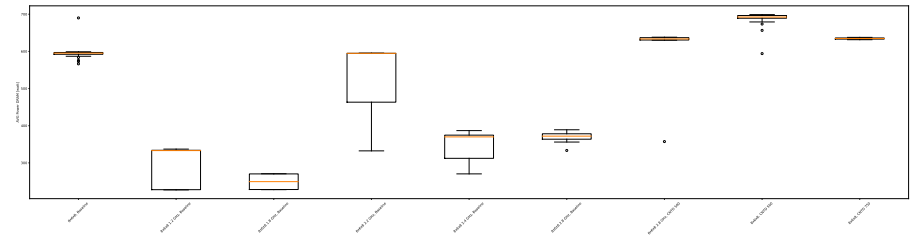
(f) Energy: boxplot of distribution of the energy consumed by RAM.



(g) Power: boxplot of distribution as COUNTDOWN configuration.

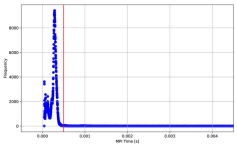


(h) Power: boxplot of distribution of the power consumed by CPU.

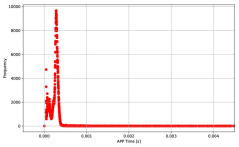


(i) Power: boxplot of distribution of the power consumed by RAM.

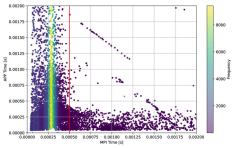
Figure B.20: Experimental Results for Experiment 8 on the *Wing_4538k.csr.Ext_bin* matrix.



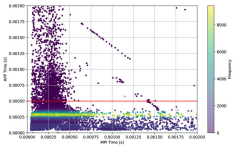
(a) CNTD 500: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



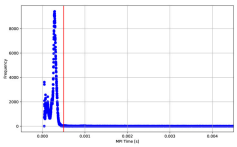
(b) CNTD 500: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



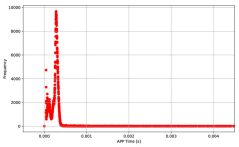
(c) CNTD 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



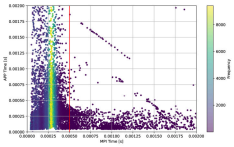
(d) CNTD 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



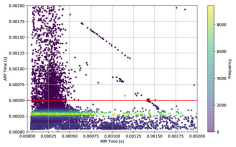
(e) CNTD SLACK 500: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



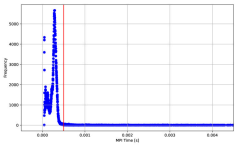
(f) CNTD SLACK 500: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



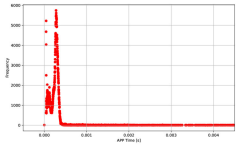
(g) CNTD SLACK 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



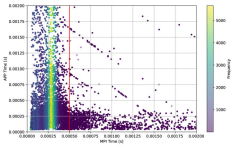
(h) CNTD SLACK 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



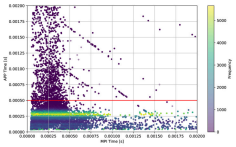
(i) CNTD 750: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



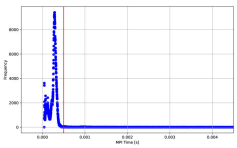
(j) CNTD 750: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



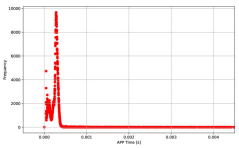
(k) CNTD 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



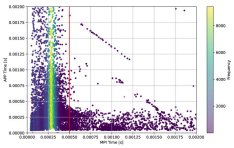
(l) CNTD 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



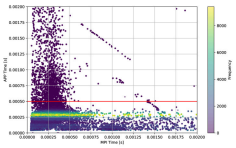
(m) CNTD SLACK 750: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



(n) CNTD SLACK 750: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.

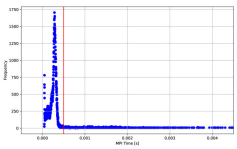


(o) CNTD SLACK 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.

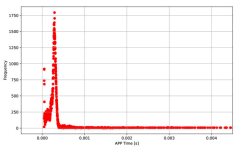


(p) CNTD SLACK 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.

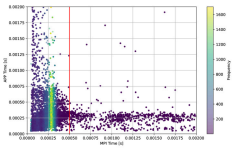
Figure B.21: Time of APP/MPI phases for *Cubo_1772481.Ext_bin* on Chronos configuration $8 \times 6 \times 8$ and relative count of the frequency at which it appears.



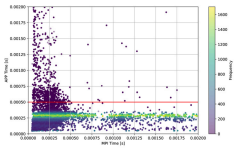
(a) CNTD 500: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



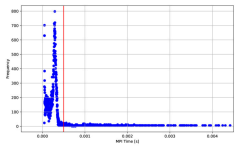
(b) CNTD 500: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



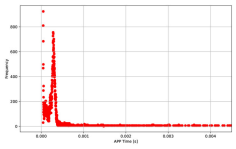
(c) CNTD 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



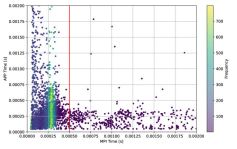
(d) CNTD 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



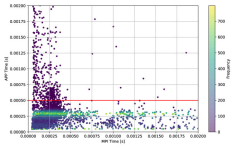
(e) CNTD SLACK 500: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



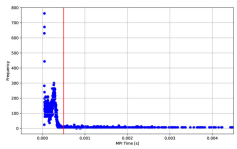
(f) CNTD SLACK 500: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



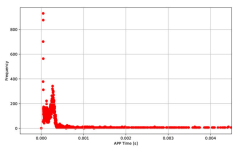
(g) CNTD SLACK 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



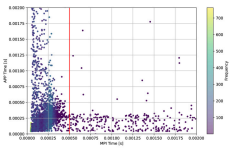
(h) CNTD SLACK 500: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



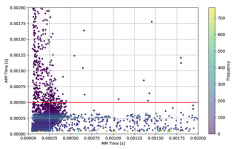
(i) CNTD 750: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



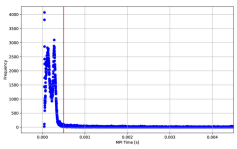
(j) CNTD 750: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



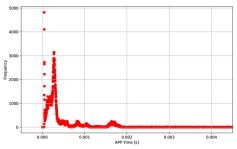
(k) CNTD 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



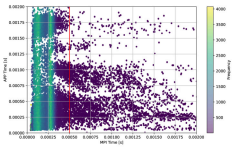
(l) CNTD 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.



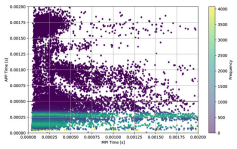
(m) CNTD SLACK 750: Time of the MPI phase and relative count of the frequency with which it occurs in an execution of MPI phases.



(n) CNTD SLACK 750: Time of the APP phase and and relative count of the frequency with which it occurs in an execution of APP phases.



(o) CNTD SLACK 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of MPI phases.



(p) CNTD SLACK 750: Time of APP/MPI phases and relative count of the frequency at which it appears in an execution of APP phases.

Figure B.22: Time of APP/MPI phases for *Wing_4538k.csr.Ext_bin* on Chronos configuration $8 \times 6 \times 8$ and relative count of the frequency at which it appears.

Bibliography

[1] S. G. Monserrate, "The Cloud Is Material: On the Environmental Impacts of Computation and Data Storage," *MIT Case Studies in Social and Ethical Responsibilities of Computing*, no. Winter 2022, jan 27 2022, https://mit-serc.pubpub.org/pub/the-cloud-is-material.

[2] J. Kim, W.-K. Jeong, and B. Nam, "Exploiting Massive Parallelism for IndexingMulti-Dimensional Datasets on the GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, 2015.

[3] "Parallel computing," [Accessed 04-03-2024]. [Online]. Available: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/x2d2f703b37b450a3:parallel-and-distributed-computing/a/parallel-computing

[4] R. Oshana, *Embedded Multicore Software Development*. Newnes, 2019.

[5] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 67–70, mar 1996. [Online]. Available: https://doi.org/10.1145/234313.234345

[6] Wikimedia Commons. Media in category "Flynn's taxonomy". [Accessed 04-03-2024]. [Online]. Available: https://commons.wikimedia.org/wiki/Category:Flynn%27s_taxonomy

[7] "Parallel computing," [Accessed 04-03-2024]. [Online]. Available: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/x2d2f703b37b450a3:parallel-and-distributed-computing/a/distributed-computing

[8] S. Sharma, C.-H. Hsu, and W. Feng, "Making a case for a Green500 list," in *Proc. of the 20th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, 1 2006.

[9] W. Feng and K. Cameron, "The Green500 List: Encouraging Sustainable Supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, dec 2007.

[10] (2018, Apr) The causes of the differences between European and US residential electricity rates. [Accessed 04-03-2024]. [Online]. Available: https://euanmearns.com/the-causes-of-the-differences-between-european-and-us-residential-electricity-rates/

[11] G. Isotton, M. Frigo, N. Spiezia, and C. Janna, "Chronos: A General Purpose Classical AMG Solver for High Performance Computing," *SIAM Journal on Scientific Computing*, vol. 43, pp. C335–C357, 2021.

[12] G. Mele, E. Ringh, D. Ek, F. Izzo, P. Upadhyaya, and E. Jarlebring, *Preconditioning for Linear Systems*. KD Publishing, 2020.

[13] J. Pool, A. Sawarkar, and J. Rodge. (2021) Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/. [Accessed 04-03-2024].

[14] L. Bergamaschi, M. Ferronato, G. Isotton, C. Janna, and A. Martínez, "Parallel matrix-free polynomial preconditioners with application to flow simulations in discrete fracture networks," *Computers & Mathematics with Applications*, vol. 146, pp. 60–70, 2023.

[15] M. Ferronato, C. Janna, and G. Gambolati, "A Novel Factorized Sparse Approximate Inverse Preconditioner with Supernodes," *Procedia Computer Science*, vol. 51, 12 2015.

[16] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, pp. 657–673, 1994, message Passing Interfaces.

[17] C.-H. Hsu and W.-C. Feng, "A Power-Aware Run-Time System for High-Performance Computing," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, p. 1.

[18] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Roşu, and R. Strong, "Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations," in *Power-Aware Computer Systems*, vol. 40. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 19–34.

[19] V. V. Zharikov, A. A. Paznikov, K. V. Pavsky, and V. A. Pavsky, "Adaptive Barrier Algorithm in MPI Based on Analytical Evaluations for Communication Time in the LogP Model of Parallel Computation," in *2018 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, 2018, pp. 1–5.

[20] H. Chen, Y. Carrasco, and A. Apon, "MPI Collective Operations over IP Multicast," in *Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, vol. 1800, 05 2000, pp. 51–60.

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. V. Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93. Association for Computing Machinery, 1993, pp. 1–12.

[22] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation," *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, vol. 44, pp. 71–79, 1997.

[23] W. G. Chen, J. D. Zhai, J. Zhang, and W. M. Zheng, "LogGPO: An accurate communication model for performance prediction of MPI programs," *Science in China, Series F: Information Sciences*, vol. 52, pp. 1785–1791, 2009.

[24] Y. Lin, Y. Tang, X. Guo, and X. Xu, "LoGPX: A New Communication Model for Message-Passing Programs," in *2011 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2011, pp. 213–219.

[25] M. I. Frank, A. Agarwal, and M. K. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," in *Proc. of the SIXTH ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[26] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, 1998, pp. 254–263.

[27] K. Al-Tawil and C. A. Moritz, "Performance Modeling and Evaluation of MPI," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 202–223, 2001.

[28] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An Energy Efficiency Feature Survey of the Intel Haswell Processor," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. Institute of Electrical and Electronics Engineers Inc., 9 2015, pp. 896–904.

[29] G. A. Abandah and E. S. Davidson, "Modeling the Communication Performance of the IBM SP2," in *Proceedings of International Conference on Parallel Processing*, 1996, pp. 249–257.

[30] Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2," *IEEE Concurrency*, vol. 4, pp. 9–23, 1996.

[31] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A.-C. Orgerie, and M. Quinson, "Predicting the Energy Consumption of MPI Applications at Scale Using a Single Node," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 92–102.

[32] L. Benini, A. Bogliolo, and G. D. Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management," in *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, vol. 8, 2000, p. 299.

[33] E. J. Hogbin. ACPI: Advanced Configuration and Power Interface. https://tldp.org/HOWTO/ACPI-HOWTO/. [Accessed 04-03-2024].

[34] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "COUNTDOWN – A run-time library for application-agnostic energy saving in MPI communication primitives," in *ACM International Conference Proceeding Series*. Association for Computing Machinery, 11 2018.

[35] D. Cesarini, A. Bartolini, A. Borghesi, C. Cavazzoni, M. Luisier, and L. Benini, "COUNTDOWN slack: A run-time library to reduce energy footprint in large-scale mpi applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 2696–2709, 11 2020.

[36] D. Cesarini, A. Bartolini, P. Bonfa, C. Cavazzoni, and L. Benini, "COUNTDOWN: A Run-Time Library for Performance-Neutral Energy Saving in MPI Applications," *IEEE Transactions on Computers*, vol. 70, pp. 682–695, 5 2021.

[37] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 11 2005, p. 33.

[38] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Guide*, 2023, 248966-048.

[39] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding Energy Consumption in Large-Scale MPI Programs," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. Association for Computing Machinery, 2007, pp. 1–9.

[40] B. Rountree, D. Lowenthal, B. Supinski, M. Schulz, V. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, 06 2009, pp. 460–469.

[41] B. Rountree, "Theory and Practice of Dynamic Voltage/Frequency Scaling in the High Performance Computing Environment," Ph.D. dissertation, University of Arizona, Tucson, USA, 2010.

[42] R. J. Wysocki. Working-State Power Management | The Linux Kernel documentation | kernel.org. https://www.kernel.org/doc/html/latest/admin-guide/pm/working-state.html. [Accessed 04-03-2024].

[43] ——. CPU Performance Scaling | The Linux Kernel documentation | kernel.org. https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html. [Accessed 04-03-2024].

[44] A. Venkatesh, K. Kandalla, and D. K. Panda, "Evaluation of Energy Characteristics of MPI Communication Primitives with RAPL," in *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, 2013, pp. 938–945.

[45] D. H. Bailey, *NAS Parallel Benchmarks*. Springer US, 2011, pp. 1254–1259.

[46] D. Thomas, J.-P. Panziera, and J. Baron, "MPInside: A Performance Analysis and Diagnostic Tool for MPI Applications," in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, ser. WOSP/SIPEW '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 79–86.

[47] M. Giles and W. Armour, "Lecture notes in Course on CUDA Programming on NVIDIA GPUs," https://people.maths.ox.ac.uk/gilesm/cuda/, 2023, [Accessed 04-03-2024].

[48] A. Venkatesh, A. Vishnu, K. Hamidouche, N. Tallent, D. Panda, D. Kerbyson, and A. Hoisie, "A case for application-oblivious energy-efficient MPI runtime," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[49] V. W. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer, "Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[50] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh, "Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 230–238.

[51] Hewlett Packard, *HPE Performance Software – Message Passing Interface MPInside Reference Guide*, 2017, 007-5780-006.

[52] D. Cesarini, A. Bartolini, and L. Benini, *Energy-Efficiency Run-time: the COUNTDOWN Approach*, 2018, pp. 10.1–10.8.

[53] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, p. 14.

[54] K.-W. Kim, H.-W. Jin, and E.-K. Byun, "Core-idling on MPI Intra-node Communication Channels for Energy Efficiency," 2021.

[55] D. Cesarini and F. Tesser, "COUNTDOWN," https://github.com/EEESlab/COUNTDOWN/, 2021, [Accessed 04-03-2024].

[56] D. Cesarini, "Power and Thermal Management Runtimes for HPC Applications in the Era of Exascale Computing," Ph.D. dissertation, alma, Aprile 2019. [Online]. Available: http://amsdottorato.unibo.it/8983/

[57] A. Bartolini, F. Beneventi, A. Borghesi, D. Cesarini, A. Libri, L. Benini, and C. Cavazzoni, "Paving the way toward energy-aware and automated data center," in *ACM International Conference Proceeding Series*. Association for Computing Machinery, 8 2019.

[58] Lawrence Livermore National Laboratory, "msr-safe," https://github.com/LLNL/msr-safe, 2017, [Accessed 04-03-2024].

[59] D. Cesarini, "slurm-msrsafe," https://gitlab.hpc.cineca.it/dcesari1/slurm-msrsafe, 2017, [Accessed 04-03-2024].

[60] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2023, 325462-082US.

[61] AMD, *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 70h-7Fh Processors*, 2018, 55072.

[62] S. Eranian, "libpfm4," https://github.com/wcohen/libpfm4, 2011, [Accessed 04-03-2024].

[63] Intel Corporation. Intel® Xeon® Platinum 8260 Processor Specifications. https://www.intel.com/content/www/us/en/products/sku/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz/specifications.html. [Accessed 04-03-2024].

[64] N. Besker and D. D. Bari. UG3.3: GALILEO100 User Guide. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.3%3A+GALILEO100+UserGuide. Cineca Consortium. [Accessed 04-03-2024].

[65] G. Isotton, M. Ferronato, G. Gambolati, and C. Janna. M3E Matrix Collection. [Accessed 04-03-2024]. [Online]. Available: https://www.m3eweb.it/matrixcollection/

[66] C. Janna, A. Franceschini, J. B. Schroder, and L. Olson, "Parallel Energy-Minimization Prolongation for Algebraic Multigrid," vol. 45, no. 5, 2023, pp. A2561–A2584.

[67] G. Isotton, A. Franceschini, and C. Janna, "On the construction of AMG prolongation through energy minimization," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '22. New York, NY, USA: Association for Computing Machinery, 2022.

[68] M. Galarnyk. (2019, 11) Understanding Boxplots. [Accessed 04-03-2024]. [Online]. Available: https://www.kdnuggets.com/2019/11/understanding-boxplots.html

[69] S. Bortolin, "HPI stress test with MPI," https://github.com/simonebortolin-msc-thesis/hpc-stress-tester/, 2024, [Accessed 04-03-2024].

[70] V. Rodriguez, "Advanced Vector Extensions (AVX) stress generator: AVX-SG," https://github.com/VictorRodriguez/AVX-SG/, 2018, [Accessed 04-03-2024].

[71] M. Bugli, L. Iapichino, and F. Baruffa, "ECHO-3DHPC: Advance the performance of astrophysics simulations with code modernization," vol. abs/1810.04597, 10 2018.

[72] S. Bortolin, D. Cesarini, and F. Tesser, "COUNTDOWN," https://github.com/simonebortolin-msc-thesis/countdown, 2023, [Accessed 04-03-2024].

[73] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of CPU frequency transition latency," *Computer Science – Research and Development*, vol. 29, 08 2013.