ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

Energy consumption of parallel algorithms for solving linear systems on HPC architecture

Tesi di laurea in
Sistemi Operativi M

Relatore
Prof. Anna Ciampolini

Candidato
Sofia Montebugnoli

Correlatore
Prof. Daniela Loreti

Correlatore
Ing. Marcello Artioli

III Sessione di Laurea
Anno Accademico 2021-2022

Abstract

Modern High-Performance Computing HPC systems are gradually increasing in size and complexity due to the correspondent demand of larger simulations requiring more complicated tasks and higher accuracy. However, as side effects of the Dennard's scaling approaching its ultimate power limit, the efficiency of software plays also an important role in increasing the overall performance of a computation. Tools to measure application performance in these increasingly complex environments provide insights into the intricate ways in which software and hardware interact. The monitoring of the power consumption in order to save energy is possible through processors interfaces like Intel Running Average Power Limit RAPL. Given the low level of these interfaces, they are often paired with an application-level tool like Performance Application Programming Interface PAPI. Since several problems in many heterogeneous fields can be represented as a complex linear system, an optimized and scalable linear system solver algorithm can decrease significantly the time spent to compute its resolution. One of the most widely used algorithms deployed for the resolution of large simulation is the Gaussian Elimination, which has its most popular implementation for HPC systems in the Scalable Linear Algebra PACKage ScaLAPACK library. However, another relevant algorithm, which is increasing in popularity in the academic field, is the Inhibition Method. This thesis compares the energy consumption of the Inhibition Method and Gaussian Elimination from ScaLAPACK to profile their execution during the resolution of linear systems above the HPC architecture offered by CINECA. Moreover, it also collates the energy and power values for different ranks, nodes, and sockets configurations. The monitoring tools employed to track the energy consumption of these algorithms are PAPI and RAPL, that will be integrated with the parallel execution of the algorithms managed with the Message Passing Interface MPI.

Contents

List of Figures

Listings

Chapter 1

Introduction

Nowadays the reduction of energy consumption is a core issue for many companies, especially for the most energy consuming, like data centers. Since the advent of Green IT, in other words the study and practice of environmentally sustainable computing or IT, the interest in creating more efficient High Performance Computing Systems has increased. This was caused initially by Dennard ‘s scaling law approaching and reaching their ends in CMOS technology, secondly from the increasing request of exascale systems, and finally from the unpredictable substantial increase of the energy price due to the last years catastrophes. Hence, the thermal and power challenges represent a renewed issue that is disrupting the typical programming model adopted so far by the scientific community.

The breakdown of the effectiveness of Dennard's scaling around 2006 led to the inability to significantly increase the clock frequencies, therefore most of CPU manufactures has focused on multicore processors alternative way to improve performance. An increased core count benefits many workloads, but the increase in active switching elements from having multiple cores still results in increased overall power consumption and thus worsens CPU power dissipation issues [14] [19]. The result is that only some fraction of an integrated circuit can be active at any given point in time without violating power constraints, in other words the total power consumption of each device limits the practical achievable performance. Moreover, due to the overheating of the cores the cooling systems consumes more additional power, and this impacts directly on the power budget available for the data center. Until 2013 the power budget of the Top500[2] for supercomputers has increased, but in the last years raising the performances directly means increasing the power budget, and consequently the economic budget to power these HPC systems. Thus, the efficiency is a core issue if it is impossible to increase the energy consumption.

The exascale systems are considered resources of National interest for many countries, due to the progress in research reachable through the modelling of complex systems. The U.S. Department of Energy states[1] that these systems are fundamental for National security issues like: stockpile stewardship, simulation tools for assessing nuclear weapons performance, response to hostile threat environments. Moreover, they play an important part in cosmological probing of the standard model, in the analysis of molecular structure and interaction in biology, in health care, energy security and economic security. Hence, HPC systems will be the tool used to tackle all these complex issues that can be only theoretically modelled before.

Finally, the complex geopolitical situation faced in the last years, due to climate crisis, war and pandemic has showed how the cost of energy can rapidly change reaching high and low peeks. This instability affects the quantity of energy that can be consumed, because a facility like CINECA that currently absorbs 10 MW per 250 Pflops will have to dramatically increase the budget to buy the energy needed to power the supercomputers. Hence, reaching an efficient power usage effectiveness index is not only contributing to a more sustainable HPC systems, but it also helps to reduce the actual costs. To increase efficiency is important both to decrease the energy absorbed by the plants, by reaching the requirements imposed by international standard organizations like ISO 50001 Energy management systems [3] and to optimize the energy consumption of the code run on the CPUs of HPC systems.

This thesis wants to explore the energy consumption of parallel algorithms on HPC architectures, with the focus on linear systems resolution algorithms, which represent several real-world applications, spanning from the field of medicine (e.g., medical image processing, computed tomography, etc.) to that of engineering (e.g., aerodynamic design, circuit simulation, etc.).

When a program is executed on HPC systems using a parallel paradigm, improving the efficiency of the algorithm is a core issue. In fact, all those problems which are CPUs intensive and aim to solve complex problems often requires days of computation. Therefore, each instruction should be optimized and strictly necessary for the resolution of the problem, because irrelevant, repetitive, or superfluous instructions can cause a delay in the end of computation of several hours, originating a waste of energy, computational resources, and money. Some of the most complex problems can be represented through matrices and their resolution is represented by the solution of the matrix as a linear system. Hence, the study of linear system solver algorithms is important to solve a wide set of problems from many fields with real-world applications. These linear systems involve thousands of equations, that can be managed efficiently from parallel architectures like HPC systems. With the purpose of improving this class of algorithms it can be helpful to study their performances on the HPC systems by monitoring the execution of these tasks.

In particular, this thesis focuses on two different algorithms deployed for the linear systems resolution which are: the Gaussian Elimination and the Inhibition Method IMe [11]. The Gaussian Elimination also known as row reduction is the most efficient algorithm for solving systems of linear equations both in a parallel and in a sequential form with an arithmetic complexity of $2/3n^3 + O(n^2)$. Whereas, the Inhibition method in the last available version has reached a complexity of $3/2n^3 + O(n^2)$ so far. However, recently it was proved [7] that IMe has a good integrated low-cost multiple fault tolerance, which is more efficient than the checkpoint/restart technique usually applied in Gaussian Elimination linear systems resolution. Therefore, a deep analysis and understanding of IME can lead to significant contribute to the available algorithms for liner systems resolution and for further optimizations. Both the Gaussian Elimination and the Inhibition Method have a parallel version which is implemented by the Netlib organization in the ScaLAPACK library in the first case, and by ENEA and University of Bologna in the second case. To analyse the efficiency of these algorithms is useful to measure and compare the energy consumption during their execution and understand, if possible, which parts of the processor are more stressed. In this phase the linear systems solver algorithms will be tested and monitored to collect data about their energy consumption.

Thesis outline Accordingly, the reminder of this thesis is structured as follows. Chapter Chapter 2 covers the technologies and studies about the HPC systems, the energy measurements and capping, and the definition of the main algorithms linear systems resolution, with particular focus on the Inhibition Method. Chapter Chapter 3 focuses on how the Message Passing Interface and the Performance API should integrate to monitor the designated metrics, moreover it contains a categorization of the different possible approaches to sample the metrics and it explains how the monitoring program is integrated and implemented in the code of the linear systems solver algorithms, the last part of this chapter explains how everything is presented to the user with a command line interface. Chapter Chapter 4 clarifies which are the settings needed to configure the Marconi A3 environment and it points out the steps to execute the parallel code on it. Chapter chapter 5 specifies the parameters of the performed tests, and the obtained results, through charts and observations. The parameters consists in a set of different variables for the execution which are tested during the monitoring. Whilst, the results of the several executions are showed and reviewed in the second part of the chapter. Finally, Chapter 6 concludes this thesis by summarising its main contribution and presents its future developments.

Chapter 2

State of the Art

2.1 High Performance Computing

High-Performance Computing HPC is defined as the set of technologies used by a computer cluster to create a computation system capable of providing high performances through the parallel computation.

As far as the Flynn's taxonomy for computer architecture is concerned, it is possible to classify the HPC systems as a combination of the Multiple instruction streams, multiple data streams MIMD and the Single Instruction stream, Multiple Data stream SIMD. Between the MIMD architectures the most popular for HPC systems is the Distributed Memory System SMD. Consequently, in most of the HPC systems there is a cluster with distributed memory nodes connected through low-latency high-bandwidth networks, in which each node is multiprocessor.

Compute nodes are often organized in limited lifetime sub-clusters created at execution time by the system scheduler to satisfy the user's request which has exclusive access to this set of nodes. The resource request happens through a batch queue system that allows the users to submit the application jobs.

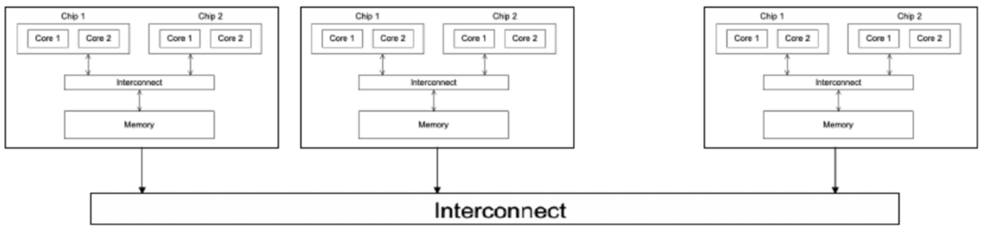

Figure 2.1: HPC schema

To develop software which can fit the HPC architecture and make the most out of the hardware parallelism, it is important to apply the concepts of parallel programming. There are two possible approaches: on one hand, the code parallelization can be automatic, especially through special compilers capable of translate sequential code into the parallel one, even though the performances are not fulfilling; on the other hand, in the explicit parallelization the developer articulates the parallelism in the code using ad-hoc programming languages and libraries like MPI for message exchange models, or OpenMP for shared memory models.

The Single Program Multiple Data SPMD is an execution model characterized by a set of cores in which each one executes the same program; what is more the conditional branching allows to differentiate the code executed by each core and, the personalization of the code is possible using the rank of the node in which the process is executing.

2.1.1 Parallelization

Since parallelization is an advanced form of programming, most of the time the first version of the program is sequential. Then the sequential version is transformed into a more efficient parallel one by splitting the execution in many processes. When a program is parallelized, it is ready to be run on an HPC system.

Ideally, the execution time of the program should decrease with the increase of the employed nodes, nevertheless parallelization process introduces an overhead and the execution time does not decline linearly. As a matter of fact, this happens because not all the algorithms are embarrassingly parallel; if an interaction depends on the previous computations, some synchronization points or message exchange are needed. To generalize the workflow to perform a parallelization, the developer should:

– Split the algorithm between the available processes in order to balance the load and minimize the interactions.

– Design the synchronization between the parallel processes

– Establish the condition for the communication between the processes.

The last two points are mutually exclusive because the choice depends on the architecture of the HPC system: in shared memory systems processes engage through synchronizations, whilst distributed memory systems are based on communication and message exchange. The metric for performance measurement is the flop (floating point operations per second). As far as HPC systems are concerned, the metrics are slightly different due to the evaluation of parallel programming.

Speedup is defined ad $S = Tseq/Tpar$; where $Tseq$ is the time taken for the execution of the program in its sequential form on a single node, while $Tpar$ is the execution time of the program in its parallel version. The speedup measures the

Listing 2.1: basic MPI structure

```
main () {
    // sequential part
    MPI_Init () ;
    //< code with calls to MPI library >
    // parallel part
    MPI_Finalize () ;
    // sequential part
}
```

difference between the sequential version and the parallel version, in other words it is the gain in the application of the parallelization.

2.1.2 Parallel programming

As mentioned before there are two possible architectures: shared memory and message exchange. MPI defines a communication protocol for processes in parallel systems. The standard is implemented in different libraries for many languages: C/C++, Fortran, Python. The standardized interface increases portability on different architectures and implementations.

MPI follows a SPMD paradigm,; the execution of a parallel program is held from different instances of the same program, where each one is executed on a different node. Moreover, MPI offers a huge set of functions to perform point-to-point and many-to-many communication, eventually with synchronous or asynchronous communication. A key feature of MPI is represented by the powerful tools for data partitioning and data collecting, with a static and implicit management of the degree of parallelism.

The block MPI Init/MPI Finalize defines the space for the usage of MPI functions. Every process in MPI runs the same program, conditional branching is used to differentiate the behaviour according to the role. The MPI communicator is an abstraction that defines a communication domain, where some processes are allowed to communicate between each other, and two processes not belonging to the same domain cannot communicate. Each MPI program has a default communicator called `MPI_COMM_WORLD`. It is possible to create sub-groups of processes, or sub-communicators and the division is made using an established criterion. They can be defined by the programmer or exposed by the MPI library like `MPI_COMM_TYPE_SHARED`, which differentiates ranks that run on the same node. There are two types of communicators:

– Intra-communicator is a collection of processes that can send messages to each other and engage in collective communication operations.

– Inter-communicator is used to send messages between processes belonging to disjoint intra-communicators.

A intra-communicator is composed of a group which is an ordered collection of processes. If a group consists of p processes, each process in the group gets assigned a unique rank, which is a non-negative integer in the range 0, 1, . . . , p-1. A context where is a system-defined object uniquely identifies a communicator. Two distinct communicators have different contexts, even if they have identical underlying groups. Finally, attributes allow to specify the topology.

2.2 Cineca Architecture

The tests of this thesis are executed on Marconi U3 partition on CINECA. Marconi is the new Tier-0 system, co-designed by Cineca and based on the Lenovo NeXtScale platform, that substitutes the former IBM BG/Q system (FERMI). MARCONI, based on the next-generation of the Intel® Xeon Phi™ family alongside with Intel® Xeon® processor E5-2600 v4 family product, offers the scientific community a technologically advanced and energy-efficient high performance computing system. The system, logically named 'MARCONI', has been designed to be gradually completed in about 18 months. Marconi A3 is the last part added to the system and it reaches a total computational power of about 20Pflop/s making use of the future generation Intel Xeon processors (Sky Lakes).

This supercomputer takes advantage of the new Intel® Omni-Path Architecture, which provides the high-performance interconnectivity required to efficiently scale out the system's thousand of servers.

A high-performance Lenovo GSS storage subsystem, that integrates the IBM Spectrum Scale™ (GPFS) file system, is connected to the Intel Omni-Path Fabric and provides data storage capacity. The progressive development of the Marconi system allows the use of the state-of-the-art processor technology, enabling an extremely high-performance system but still with a 'green' soul. One of the parameters of the project developed by the Cineca team is in fact to gradually increase the computational power up to 50Pflop/s without exceeding, at any stage, the limit of 3MWatt power consumption.

The architecture of Marconi is Intel OmniPath Cluster, with 17PB of local storage. There are eight login nodes. Each one contains two Intel Xeon Processor E5-2697 v4 with a clock of 2.30GHz and 128 GB of memory. Login nodes are shared between three partitions: A1 (BDW), A2 (KNL) and A3 (SKL). The three partitions are served by a single SLURM server. The performance of Marconi A3 comprehends forty-five racks, 3188 nodes and each one has 2 x 24-cores Intel Xeon 8160 CPU (Skylake) at 2.10 GHz, therefore the total cores per nodes are 48, whilst a 192 GB of DDR4 RAM is provided to every node. A single node can reach a peak performance of 3.2 TFlop/s, whereas the overall MARCONI A3 architecture can achieve a 10 PFlop/s peak [33].

2.3 Energy saving

Nowadays, computers include different power management techniques which support the reduction of energy consumption[13]. Examples are dynamic voltage frequency scaling DVFS, clock gating, and power gating. Moreover, the usage of special instructions and specialized coprocessors can help reducing energy consumption.

DVFS can reduce the clock frequency and voltage level of different components of the compute node (processors, DRAM memories, etc.) at the expense of some performance degradation. Currently, DVFS is broadly supported by low-power and high-performance processors provided by different manufacturers under different names (e.g. SpeedStep in Intel processors and PowerNow or Cool 'n' Quiet in AMD processors). There are three factors that need to be considered when DVFS is applied:

- the dynamic power, which has a quadratic relationship with frequency-voltage scaling;
- the static power, which increases exponentially with the voltage;
- the performance, which has a linear relationship with the frequency.

Clock Gating reduces the power consumption by disabling the clock in those parts of the circuit that are idle or like in the case of flip-flops, maintain a steady state that does not need to be refreshed. The power used to drive the clock signal can represent more than a half of the overall power consumption. Therefore, clock gating can potentially achieve a significant energy reduction. This technique can be controlled both at hardware and software level. Hardware-level approaches typically provide a finer granularity, allowing also to disable components inside a functional block. Software-level approaches are usually applied at entire functional blocks, but they allow more elaborated energy-saving policies.

Power gating is a more aggressive approach in which a functional block is disconnected from the power supply, powering of all its components. Nowadays, existing processors contain clock gating logic managed by a power reduction policy for almost every functional block. For some components clock gating is used in combination with power gating features. Given that the entire functional unit is disconnected, power gating achieves a better power reduction than clock gating. However, given that the functional unit state is erased, it is necessary to provide mechanisms for saving and restoring the states of the functional units, which increases the complexity and complicates resource utilization when applying power gating to active components that need to preserve their state.

In order to obtain the benefits offered by an Ultrascale or Exascale system, it will be increasingly important to provide system services for an effective management of the system resources on behalf of the applications. Those services can be offered to the applications through the programming environment or through specialized libraries, but they should be as transparent to the user as possible to support application porting and sustainability. As energy is a cross-layer issue, several aspects of the system software and the operating system should be involved in energy efficiency resource management, but it is also paramount to provide metrics and facilities to monitor and express energy at the processor and system level.

2.3.1 Energy metrics

In order to properly evaluate a specific system property, it is necessary to define corresponding metrics. With regard to energy, the main basic metric is usually the unit of work or amount of heat transferred, measured in Joule (J), while the power, i.e. the amount of transferred energy in time, is measured in Watt (W). In the computing system context, several initiatives related to energy measurement and management have been started, mostly grouped under the umbrella of Green IT. There are two main metrics for evaluating energy efficiency in data centers: metrics[31]: Power Usage Effectiveness (PUE), and Data center Infrastructure Efficiency (DCiE). PUE is defined as follows:

$$PUE = \frac{TotalFacilityEnergy}{ITEquipmentEnergy}$$

while DCiE is specified as its reciprocal:

$$DCiE = \frac{1}{PUE} = \frac{ITEquipmentEnergy}{TotalFacilityEnergy}x100$$

The energy for the total facility is the overall amount of energy consumed by the whole data center, including IT systems and facilities. The IT systems energy is the energy consumed by just the IT equipment only, such as processing, storage, and network components for data management and processing. The facilities include all the other subsystems, such as UPS and power management systems, cooling systems, lighting systems, etc.

2.3.2 Energy efficient algorithms

It is important to study the influence of Hardware mechanisms to reduce energy consumption on algorithms and applications. It must be investigated whether these techniques can be employed to reduce the energy consumption of algorithms, and which specific characteristics of algorithms may influence the resulting energy consumption. If the influencing factors are known and can be captured quantitatively, this information can be used to tune applications towards a smaller energy consumption by applying suitable algorithmic transformation techniques [9]. The energy consumption E of an algorithm can be described by the power consumption P of the execution resources employed and by integrating P over the execution time of the algorithm: $E = R \times t_{max} \times t = t_0 P(t)dt$. Typically, the power consumption varies during the execution time of the application, depending on the specific execution situation of the application and the resulting usage of the different execution resources. The variations of the power consumption during the execution time can be measured in detail with specialized power meters, power acquisition systems, or hardware counters (e.g. Intel RAPL interface).

The power consumption of processors comprises a dynamic and a static power consumption part[10]. The dynamic power consumption $Pdyn$ is related to the switching activity of the processor during execution and it can be expected that it is smaller during processor idle periods. The static power consumption $Pstat$ captures the leakage power, which becomes more important for processors with smaller transistor size, and it is present even if there is no switching activity of the transistors. For DVFS processors, the dynamic power consumption increases significantly with the operational frequency f, and often, a dependence $P_{dyn}(f) = \gamma f^{\alpha}$ with $2.5 \leq \alpha \leq 9$ is assumed, where is a suitable parameter. The dependence of the static power consumption $Pstat$ on f is typically quite small and is often neglected and assumed to be constant.

For parallel applications, the speedup obtained plays a role and it can be observed that applications with a larger speedup tend to have a larger power consumption than applications with a smaller speedup. This can be explained by the fact that applications with a smaller speedup typically include more idle times during which some parts of the processing cores can be powered down, thus reducing the average power consumption.

2.4 RAPL

All the Intel CPUs offer power management interfaces that are not architectural but address the power management needs of several platform's specific components. RAPL (Running Average Power Limit) interfaces provide mechanisms to enforce power consumption limit. Power limiting usages have specific usages in client and server platforms[12]. For client platform power limit control and for server platforms used in a data center, the following power and thermal related usages are desirable:

- Platform Thermal Management: Robust mechanisms to manage component, platform, and group-level thermals, either proactively or reactively (e.g., in response to a platform-level thermal trip point).
- Platform Power Limiting: More deterministic control over the system ' s power consumption, for example to meet battery life targets on rack-level or container-level power consumption goals within a data center.
- Power / Performance Budgeting: Efficient means to control the power consumed (and therefore the sustained performance delivered) within and across platforms.

The server and client usage models are addressed by RAPL interfaces, which expose multiple domains of power rationing within each processor socket. The RAPL power domain is a physically meaningful domain for power management. Each power domain informs the energy consumption of the domain, allows to limit the power consumption of that domain over a specified time window, monitors the performance impact of the power limit and provides other useful information, that is, energy measurement units, minimum or maximum power supported by the domain.

The figure 2.2 shows the hierarchy of the power domains graphically. RAPL provides the following power domains for both measuring and limiting energy consumption:

- Package: Package (PKG) domain measures the energy consumption of the processor die. It includes the consumption of all the cores, integrated graphics and also the uncore components (last level caches, memory controller).
- Power Plane 0: Power Plane 0 (PP0) domain measures the energy consumption of all processor cores on the socket.
- Power Plane 1: Power Plane 1 (PP1) domain measures the energy consumption of processor graphics (GPU) on the socket (desktop models only).
- Memory domain includes the directly attached DRAM, memory domain measures the energy consumption of random-access memory (RAM) attached to the integrated memory controller.

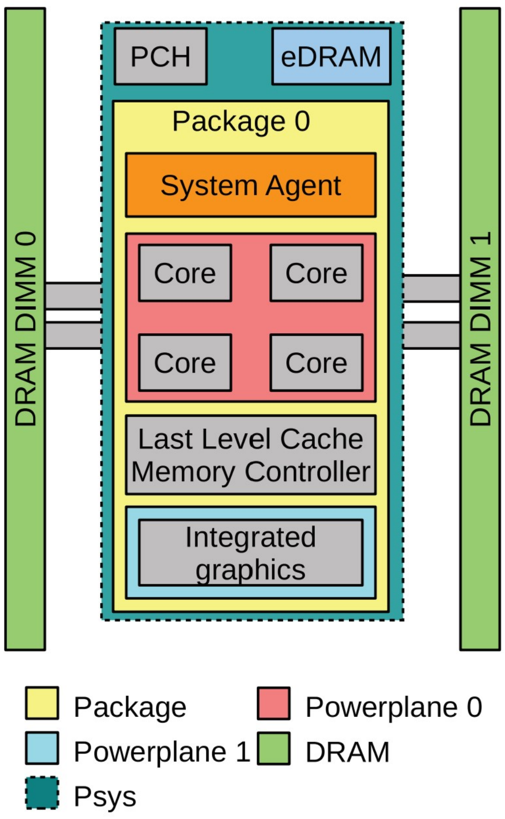

Figure 2.2: RAPL architecture, hierarchy of the power domain [23]

– PSys: Intel Skylake has introduced a new RAPL Domain named PSys. It monitors and controls the thermal and power specifications of the entire SoC and it is useful especially when the source of the power consumption is neither the CPU nor the GPU. As Figure 1 suggests, PSys includes the power consumption of the package domain, System Agent, PCH, eDRAM and a few more domains on a single socket SoC.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each processor complex. Programming specific RAPL domain across multiple sockets is not supported. As matter of facts, the SkyLake architecture contains two packages, named PK0 and PK1, and each of them is linked to a different DRAM, respectively DRAM0 and DRAM1. Therefore, the actual structure can be schematized as showed in figure 2.3.

RAPL interfaces consist of non-architectural Model Specific Registers MSR. The counters are 32-bit registers that indicate the energy consumed since the processor was booted up. The counters are updated approximately once a millisecond

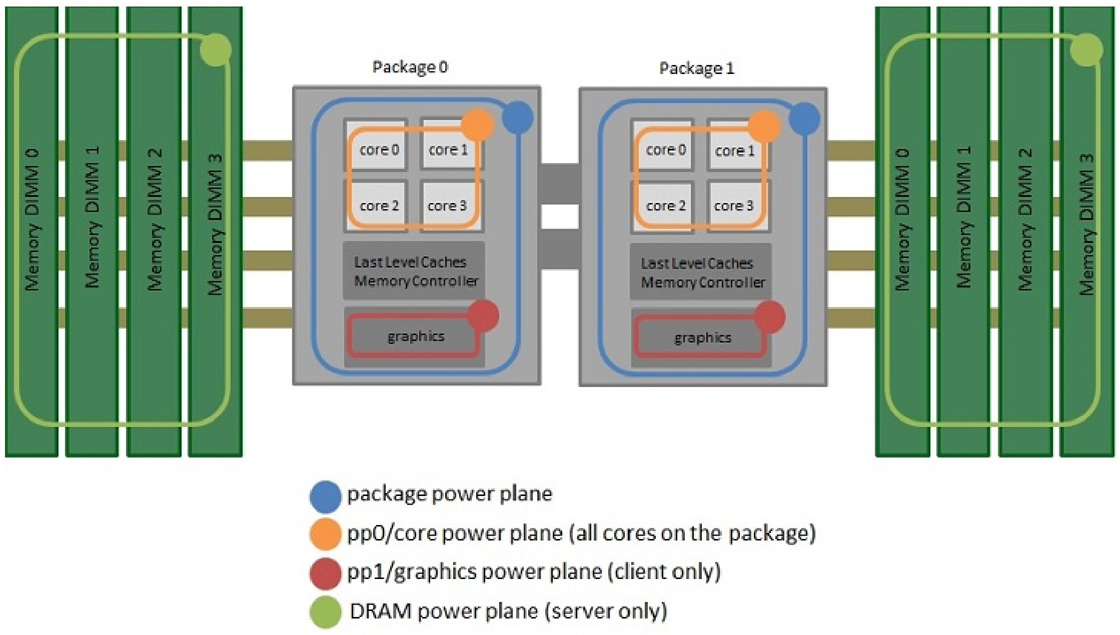

Figure 2.3: Structure of SkyLake processors [34]

(due to jitter). The MSRs can be accessed directly on Linux using the MSR driver in the kernel. For direct MSR access the MSR driver must be enabled, and the read access permission must be set for the driver. Reading RAPL domain values directly from MSRs requires detecting the CPU model and reading the RAPL energy units before reading the RAPL domain (i.e., PKG, PP0, PP1, etc.) consumption values.

Once the CPU model is detected, the RAPL domains can be read per package of the CPU by reading the corresponding 'MSR status' register. Each RAPL domain supports the following set of capabilities, some of which are optional as stated below.

– POWER_LIMIT-MSR interfaces to specify power limit, TIME_WINDOW lock bit, clamp bit etc.

– Energy Status - Power metering interface providing energy consumption information.

– PERF_STATUS (Optional) - Interface providing information on the performance effects (regression) due to power limits. It is defined as a duration metric that measures the power limit effect in the respective domain. The meaning of duration is domain specific.

– Power Info (Optional) - Interface providing information on the range of parameters for a given domain, minimum power, maximum power etc.

– Policy (Optional) - 4-bit priority information that is a hint to hardware for dividing budget between sub-domains in a parent domain.

Each of the above capabilities requires specific units in order to describe them. Power is expressed in Watts, Time is expressed in Seconds, and Energy is expressed in Joules. Scaling factors are supplied to each unit to make the information presented meaningful in a finite number of bits. Units for power, energy, and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR. Apart from directly reading MSRs, RAPL readings can also be read from sysfs interface, perf events or through the PAPI library. RAPL support for the sysfs powercap interface is enabled from Linux Kernel version 3.13 and the perf_event_open support requires Linux Kernel version 3.14. PAPI library is used for gathering performance-related data. It is platform independent, and it has a RAPL interface, which uses the MSR driver to report RAPL values.

2.4.1 Advantages in using RAPL

RAPL has several advantages due to its integration directly in the processor through the MSR. The accuracy of RAPL energy readings is high even though it is based on activity counters and the accuracy varies across different processing architectures. RAPL power values are promisingly accurate, and these values can be used to predict or model full-system power consumption. Moreover, the RAPL energy calculations are implemented in the hardware. There is no need to do any complex calculations, such as numerical integration in the software. Energy consumption can be measured on the same machine without significant overhead. No additional equipment is needed, which makes RAPL a very-low-cost option for energy measurements. It is important to control the temperature of the CPU to obtain accurate power readings through RAPL. There is a good correlation between the RAPL package power and temperature. Nevertheless, CPU temperature can influence the CPU package power draw. To obtain accurate RAPL power readings it would be good to warm up the CPU before obtaining any RAPL measurements. A 2min warm-up period should be enough for any suitable program that keeps the CPU package busy. RAPL updates the energy counters approximately once every 1ms, that is, 1,000Hz. This frequency is much higher compared to external power meters that typically measure power only once a second. It is demonstrated that many HPC benchmarks have phases with different package and DRAM power consumption characteristics. Besides, the RAPL sample rate is sufficient to distinguish different execution phases in applications. RAPL starts running as soon as the processor boots up. There is no need to configure it if one wants to measure energy consumption, which makes it very easy to use. Since RAPL is always running, there is very little additional overhead introduced by reading the energy counters.

2.4.2 Disadvantages in using RAPL

Since RAPL interface is strictly connected to the CPU[23], it is affected by some limitations. The energy counters are limited to 32bits even though the MSRs are 64bit wide. Therefore, they will eventually overflow. Fortunately, these overflows are quite rare. The overflows can be mitigated easily by sampling the energy counters more frequently than $t_{overflow} = \frac{2^32\times E_u}{P}$, where E_u is the energy units used and P is the power consumption. Frequent-enough sampling allows detecting every overflow that occurs. Generally, sampling the counters every 5 minutes should be sufficient for any CPU model.

Measurements show a time delay between updates to different energy counters, which means that the RAPL updates are not atomic. The non-atomicity introduces errors when sampling multiple counters at high sampling rates. It is possible to read both fresh and stale values of different counters. There was an issue in validating the RAPL values specifically for the RAPL DRAM domain. The initial findings suggest that RAPL DRAM values were unstable and unreliable for earlier versions of processors that included RAPL. DRAM values were only available to server grade systems prior to Haswell. Prior to Haswell, the RAPL DRAM values followed different trends depending on the benchmark, and the values also differed substantially at times with reference values. Since the introduction of Haswell, RAPL DRAM values are now more reliable and follow a strong correlation with AC reference measurements.

The update interval of the RAPL energy counters is approximately 1ms. Intel documentation states that the time unit used by RAPL is 0.976ms. Unfortunately, the updates do not have any timestamps associated with them. In addition, there seems to be no way to predict the exact timing of future RAPL updates. Nevertheless, there are several solutions to cope with this problem:

– busy polling. It is possible to determine the exact timing of the RAPL updates by busy polling the counters for updates. This technique allows the exact synchronization with the RAPL updates. Hähnel et al. [17] demonstrated that this method can be used to measure the energy consumption of short functions.

– supersampling. Every update can be observed without the timing by sampling more often than every 1ms, for example, every 0.9ms. The method will occasionally produce duplicate values, but they can be filtered out easily. The main advantage is the reduced overhead compared to busy polling. However, the relative overhead is still quite large.

- High frequency sampling. Sampling frequencies in the range 50–1,000Hz fall into this category. The advantage is a lower overhead due to a lower sampling rate. A 50Hz sampling rate is still sufficient to distinguish different execution phases.
- Low frequency sampling. At sampling rates slower than 50Hz, the relative error due to the spikes is less than 0.5%, which is small enough to be ignored. The energy values should remain accurate even with long sampling periods, since they are calculated by the hardware. Thus, the only drawback with slow sampling is the loss of temporal precision.

It is not possible to increase the sampling rate from the default of 1,000 samples per second, which limits the ability to measure the energy consumption of individual functions.

One solution is to force the application to sleep before and after the execution of a single function. In this manner only a single function is executed during one RAPL update period. Hähnel et al. [17] showed that this method works.

RAPL does not support the measurement of the power consumption of individual CPU cores. The Power Plane 0 domain only gives the total energy consumed by all cores in a single package. Each core has its own MSRs, but Khan et. al. [23] found that all cores show the same values for the RAPL counters. This prevents us from separating the power consumption of different processes or threads running concurrently on the same processor. Moreover, individual core measurement is further complicated by hyperthreading, which allows one core to simultaneously run instructions form two threads. The only possible solution is to use disable explicitly hyperthreading and to apply techniques which implies the hardware counters to model and attribute the power consumption to individual threads such as HaPPy[38], notwithstanding these models reportedly have large errors ($\approx 10\%$).

2.5 Powercap

2.5.1 Relation between frequency, power and performances

Modern hardware is constrained by power and temperature limitations, often quantified as Thermal Design Power. A popular mechanism for balancing performance and power consumption is Dynamic Voltage and Frequency Scaling DVFS. For compute-bound applications, DVFS provides a linear relationship between frequency and performance. However, power is non-linear with frequency since an increase in frequency also requires an increase in voltage.

Although the relationship between performance and power is more difficult to model, hardware can be better at optimizing voltage and frequency than software while still respecting a power cap over a time window. Power capping allows a system administrator to configure an upper limit on the power consumption of various hardware components while letting the hardware more efficiently manage voltage and frequency. Setting a power cap does not imply that the component will consume that power, only that it will not violate that limit on average over the specified time window [21].

2.5.2 Powercap framework

The power capping framework provides a consistent interface between the Linux kernel and the user space that allows power capping drivers to expose the settings to user space in a uniform way [20].

The framework exposes power capping devices to user space via `sysfs` in the form of a tree of objects. The objects at the root level of the tree represent ‘control types’, which correspond to different methods of power capping. For example, the intel-rapl control type represents the Intel “Running Average Power Limit” (RAPL) technology, whereas the ‘idle-injection’ control type corresponds to the use of idle injection for controlling power.

Power zones represent different parts of the system, which can be controlled and monitored using the power capping method determined by the control type the given zone belongs to. They each contain attributes for monitoring power, as well as controls represented in the form of power constraints. If the parts of the system represented by different power zones are hierarchical, meaning that, one bigger part consists of multiple smaller parts with their own power controls, those power zones may also be organized in a hierarchy with one parent power zone containing multiple subzones and so on, in order to reflect the power control topology of the system. In that case, it is possible to apply power capping to a set of devices together with the parent power zone; moreover, if more fine-grained control is required, it can be applied through the subzones.

2.6 PAPI

Performance Application Programming Interface PAPI is an interface for accessing performance counters on different platforms in a common way. As each processor vendor defines different processor interfaces to the performance counters, PAPI was built to solve this problem and to handle requests to these counters in a comfortable way.

As for the development of PAPI the main goal was a common and convenient way to access performance counters on different platforms: PAPI is build up on different layers for a better abstraction of different tasks found in each layer as shown in figure 2.4. The main layers are the Portable Layer which offers an API for tool and application developers and the Machine Specific Layer used to access performance counters on a given platform. A given platform consists possibly of a certain processor architecture, a certain operating system, available libraries or a combination of these.

The Portable Layer consists of the PAPI Low Level-API enabling a developer to access all core functions of PAPI and a direct interaction with the counter interface on a given platform. The PAPI High Level-API defines only a fraction of functions compared to the PAPI Low Level-API to access the counters, but these functions are enough to extract performance data using pre-sets defined by PAPI.

The Machine Specific Layer handles all direct access to a given platform. The term direct access is meant to express the access either to the counters on a platform directly or by using an operating system interface for accessing these processor specific functions. The Machine Specific Layer also limits PAPI in its functionality, as PAPI supports many different platforms where some platforms do not support specific functionalities.

Between the Portable Layer and the Machine Specific Layer is the core functionality of PAPI with support for managing the counter access. Memory allocation, thread binding and event related issues are handled here, invisible for the developer of a tool or application for performance counter instrumentation [8].

2.6.1 Events

Preset Events Pre-set events can be defined as a single event native to a given CPU or can be derived from linear combination of native events. More complex derived combinations of events can be expressed in reverse polish notation and computed at run-time by PAPI [36]. By calling `papi_avail` it is possible to examine the list of preset events on the given platform.

Native Events PAPI components contains tables of native event information allowing native events to be programmed in essentially the same way as a preset event. Each native event may have several attributes, called unit masks, that can act as filters on what gets counted. These attributes can be appended to a native event name to tell PAPI exactly what to count. Attributes can be appended in any order or combination, and are separated by colon characters [36]. By calling

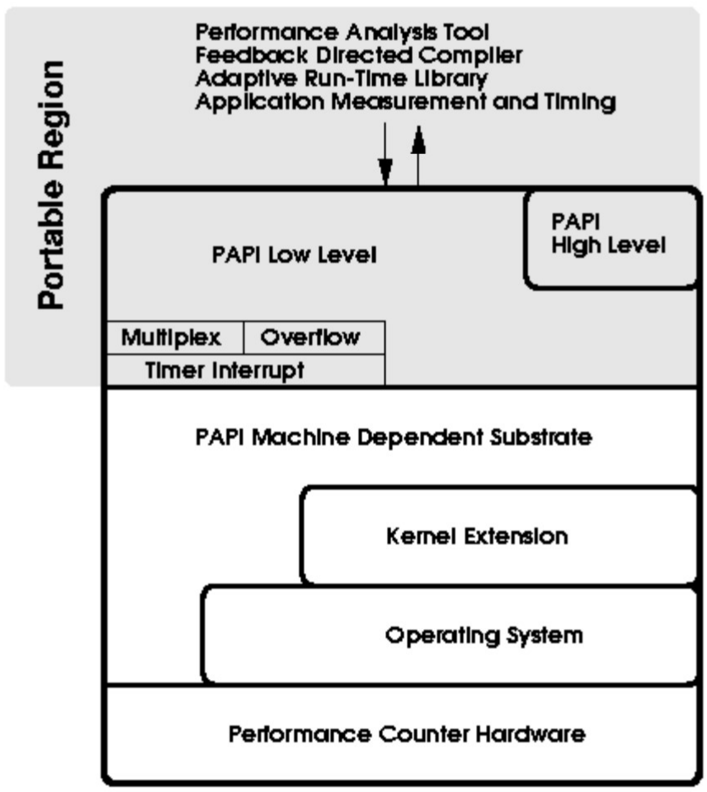
Portable Region
Performance Analysis Tool
Feedback Directed Compiler
Adaptive Run-Time Library
Application Measurement and Timing
PAPI Low Level
PAPI High Level
Multiplex
Overflow
Timer Interrupt
PAPI Machine Dependent Substrate
Kernel Extension
Operating System
Performance Counter Hardware

Figure 2.4: PAPI architecture, hierarchy of power domains [8]

`papi_native_avail` it is possible to examine the list of native events with all the correspondent available attributes.

Event Sets PAPI provides an abstraction from hardware events called EventSets. An EventSet consists of events that the user wishes to count as a group. There are two reasons for this abstraction. The first reason is efficiency in accessing the counters through the operating system. Most operating systems allow the programmer to move the counter values in bulk without the need to make a separate system call for each counter. By exposing this grouping to the user, the PAPI library can greatly reduce its overhead when accessing the counters. This efficiency is especially important when PAPI is used to measure small regions of code inside loops with large iteration counts. The second reason for EventSets is that users can evolve their own specialized counter groupings specific to their application areas [36]. The use of EventsSet is directly related to the Low Level API.

2.6.2 Low-level API

The low-level API manages hardware events in user-defined groups called Event Sets. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface. Unlike the high-level interface, it allows both PAPI pre-set and native events. Other features of the low-level API are the ability to obtain information about the executable and the hardware as well as to set options for multiplexing and overflow handling. One of the benefits of using the low-level API rather than the high-level API is that it increases efficiency and functionality.

It should also be noted that the low-level interface could be used in conjunction with the high-level interface, as long as attention is paid to ensure that the PAPI library is initialized prior to the first low-level PAPI call.

The low-level API is only as powerful as the substrate upon which it is built. Thus, some features may not be available on every platform. The converse may also be true; more advanced features may be available on every platform and defined in the header file [27]. Moreover, the low-level interface is particularly interesting for the use of RAPL and powercap.

2.6.3 High-level API

The high-level provides the ability to record performance events inside instrumented regions of serial, multi-processing (MPI, SHMEM) and thread (OpenMP, Pthreads) parallel applications. It is intended for users who want to perform simple event measurements in a very convenient way as they only must mark code sections. Events to be recorded are determined via an environment variable (PAPI_EVENTS) that lists comma separated events for any component. This enables users to perform different measurements without recompiling. In addition, users do not need to take care of printing performance events since an output is generated at the end of each measurement.

Another benefit of high-level API rather than the low-level API is that it is easier to use and requires less setup. For instance, the dynamic setting of performance events via the environment variable and the automatic detection of components makes the use of the high-level API extremely simple. It should also be noted that the high-level API can be used in conjunction with the low-level API and, in fact, it does call the low-level API [26].

2.6.4 Multiplexing

Most modern microprocessors have a very limited number of events than can be counted simultaneously. This limitation severely restricts the amount of performance information that the user can gather during a single run. As a result, large applications with many hours of run time may require days or weeks of profiling in order to gather enough information to base a performance analysis on it. This limitation can be overcome by multiplexing the counter hardware. By subdividing the usage of the counter hardware over time, multiplexing presents the user with the view that many more hardware events are countable simultaneously. This unavoidably incurs a small amount of overhead and can adversely affect the accuracy of reported counter values [36].

2.6.5 Handlers

One of the most significant features of PAPI for the tool writer is its ability to call user-defined handlers when a particular hardware event exceeds a specified threshold. This is accomplished by setting up a high-resolution interval timer and installing a timer interrupt handler. For systems that do not support counter overflow at the operating system level, PAPI uses SIGPROF and ITIMER_PROF. PAPI handles the signal by comparing the current counter value against the threshold. If the current value exceeds the threshold, then the user's handler is called from within the signal context with some additional arguments. These arguments allow the user to determine which event overflowed,how much, and in which location of the source code it overflowed [36].

2.6.6 Thread support

As very large SMP's become ever more popular in the HPC community, fully thread aware performance tools are becoming a necessity. As with any API, the interface must be re-entrant, because any number of threads may simultaneously call the same PAPI function. This means that any globally writeable structures must be locked while in use. This requirement has the potential of increasing overhead and introducing large sections of machine dependent code to the top layer. PAPI has only one global data structure, which keeps track of process wide PAPI options and thread specific pointer maps. Fortunately, this structure is only written by two API calls that are almost exclusively used during the initialization and termination of threads and the PAPI library [36].

A second problem is the accuracy of event counts as returned by threads calling the API. In order to support threaded operation, the operating system must save and restore the counter hardware upon context switches among different threads or processes. The PAPI library must keep thread-specific copies of the counter data structures and values [36].

Although PAPI attempts to introduce as little overhead as possible and thus perturb application performance to only a minor degree, some perturbation is inevitable and has yet to be measured. Counts produced for various PAPI metrics may vary from one run to another of the same program on the same inputs on some architectures, due to contention for resources with other applications or the operating system. The vendor-provided counter interfaces may occasionally have bugs that cause inaccurate reporting of hardware counter data [36].

2.6.7 Tools

The PAPI project has developed two tools that demonstrate graphical display of PAPI performance data in a useful manner for the application developer. These tools are meant to demonstrate the capabilities of PAPI rather than its production quality tools. The tool front ends are written in Java and can be run on a separate machine from the program being monitored. All that is required for real-time monitoring and display of application performance is a socket connection between the machines. The first tool, called the `perfometer`, provides a runtime trace of a chosen PAPI metric; the second tool, called the `profometer`, provides a histogram that relates to the occurrences of a chosen PAPI event to the text addressed in the program [36].

2.6.8 PAPI and RAPL

It is important to highlight the link between PAPI and RAPL. While the RAPL values can be measured in-band and consumed by the program, RAPL is system-wide, so a separate process may be used to measure energy and power. In this way the running code does not need to be instrumented and some of the PAPI overhead can be avoided. In fact, the internal circuitry of the Intel processor can estimate current energy usage based on a model driven by hardware counters, temperature, and leakage models. The results of this model are available to the user via a model specific register (MSR), with an update frequency on the order of milliseconds. Accessing MSRs requires ring-0 access to the hardware; typically, only the operating system kernel can do this. This means accessing the RAPL values requires a kernel driver.

PAPI offers many components which are a specification of a low-level API. The RAPL component enables PAPI to access Linux RAPL energy measurements. This tool can be used to gather energy measurements on a SandyBridge, IvyBridge or Haswell Intel chip using RAPL. RAPL directly access MSR registers, but this can have many significant performance and security implications. In order to encourage system administrators to give wider secure access to the MSRs on a machine, LLNL has released a Linux kernel module (`msr_safe`) which provides safer, white-listed access to the MSRs that can be tuned by the site administrator [24]. LLNL has released a library `libmsr` to provide a simple, safe, consistent interface to several of the model-specific registers (MSRs) in Intel processors via the `msr_safe` kernel module [37].

PAPI API can read the state of power consumption on a CPU socket and the current performance of the code. It can also make determinations about the desired CPU performance, and it can adjust and cap the power consumption as desired. RAPL allows users to set power limits over two specific time windows, meaning, one can have local power spikes while keeping the power low over a larger time window [22].

To enable reading RAPL counters the user needs to link against a PAPI library that was configured with the RAPL component enabled. At configuration time is necessary to perform `./configure --with-components="rapl"` and to have read permissions on the `/dev/cpu/*/msr` files. This is sufficient to enable the component. Typically, the utility `papi_components_avail` will display the components available to the user, whether they are disabled, and when they are disabled and why [29].

The RAPL component works by using PAPI to poll the RAPL stats every 100ms. It will dump each statistic to different files, which then can be plotted. The measurements are made in nJ. The frequency can be adjusted by changing the source code. The files can be elaborated and put them into a plotting program like gnu-plot.

The available RAPL events wihich are added to PAPI event set and corresponds to the available events on Marconi U3 are:

- rapl:::THERMAL_SPEC_CNT:PACKAGE0 ;
- rapl:::THERMAL_SPEC_CNT:PACKAGE1 ;
- rapl:::MINIMUM_POWER_CNT:PACKAGE0 ;
- rapl:::MINIMUM_POWER_CNT:PACKAGE1 ;
- rapl:::MAXIMUM_POWER_CNT:PACKAGE0 ;
- rapl:::MAXIMUM_POWER_CNT:PACKAGE0 ;
- rapl:::MAXIMUM_POWER_CNT:PACKAGE1 ;
- rapl:::MAXIMUM_TIME_WINDOW_CNT:PACKAGE0 ;
- rapl:::MAXIMUM_TIME_WINDOW_CNT:PACKAGE1 ;
- rapl:::PACKAGE_ENERGY_CNT:PACKAGE0 ;
- rapl:::PACKAGE_ENERGY_CNT:PACKAGE1 ;

– rapl:::DRAM_ENERGY_CNT:PACKAGE0 ;
– rapl:::DRAM_ENERGY_CNT:PACKAGE1 ;
– rapl:::PP0_ENERGY_CNT:PACKAGE0 ;
– rapl:::PP0_ENERGY_CNT:PACKAGE1 ;
– rapl:::THERMAL_SPEC:PACKAGE0 ;
– rapl:::THERMAL_SPEC:PACKAGE1 ;
– rapl:::MINIMUM_POWER:PACKAGE0 ;
– rapl:::MINIMUM_POWER:PACKAGE1 ;
– rapl:::MAXIMUM_POWER:PACKAGE0 ;
– rapl:::MAXIMUM_POWER:PACKAGE1 ;
– rapl:::MAXIMUM_TIME_WINDOW:PACKAGE0 ;
– rapl:::MAXIMUM_TIME_WINDOW:PACKAGE1 ;
– rapl:::PACKAGE_ENERGY:PACKAGE0 ;
– rapl:::PACKAGE_ENERGY:PACKAGE1 ;
– rapl:::DRAM_ENERGY:PACKAGE0 ;
– rapl:::DRAM_ENERGY:PACKAGE1 ;
– rapl:::PP0_ENERGY:PACKAGE0 ;
– rapl:::PP0_ENERGY:PACKAGE1 ;

2.6.9 PAPI and Powercap

Powercap Linux kernel interface has the purpose of this interface is to expose the RAPL settings to the user. Powercap exposes an intuitive `sysfs` tree representing all the power “zones” of a processor. These zones represent different parts of a processor that support power monitoring capabilities. The top-level zone is the CPU package that contains “subzones,”, which can be associated with core, graphics, and DRAM power attributes. Depending on the system, only a subset of the subzones may be available. For example, Sandy Bridge processors have two packages, each having core, graphics, and DRAM subzones, which, in addition, allows the calculation of uncore (last level caches, memory controller) power by simply subtracting core and graphics from package. On the other hand, KNL processors have a single package containing only core and DRAM subzones. Power measurements can be collected, and power limits enforced, at the zone or the subzone level. Applying a power limit to the package level will also affect all the subzones in that package [18].

The POWERCAP component of PAPI supports measuring and capping power usage on recent Intel architectures, using the powercap interface exposed through the Linux kernel. The powercap component exposes power attributes that can be read to retrieve their values and a smaller set of attributes that can be written to set system variables. All power attributes are mapped to events in PAPI, which matches its well-understood format.

It is possible to write a small test application that uses the component to poll the appropriate events for energy statistics at regular intervals. This information could then be logged for later analysis. Also, it is possible to create a test program that applies power limits to any of the power "zones" discussed in the previous section by specifying the appropriate event names in the component. For example, one could apply separate power limits for DRAM than for core events to a CPU package. Alternatively, a power limit could be applied one level up in the hierarchy at the CPU package level. With this added flexibility, which is the result of our development on this component, the powercap component can aid a user in finding opportunities for energy efficiency within each application [18].

To enable the reading of POWERCAP counters the user needs to link against a PAPI library that was configured with the POWERCAP component enabled. For example `./configure --with-components="powercap"` is sufficient to enable the component. The powercap `sysfs` interface exposes energy counters and R/W register-like power settings. The counters and R/W settings apply to a power domain on a system.

Typically, the utility `papi_components_avail` will display the components available to the user, whether they are disabled and why [28]. In the events listed below, Zone 0 corresponds to package 1 in figure 2.3, zone 1 corresponds to package 0 in figure 2.3, and and zone 0 subzone 0 and zone 1 subzone 0 represents respectively the DRAMs 0 and 1.

– powercap:::ENERGY_UJ:ZONE0 ;

– powercap:::MAX_ENERGY_RANGE_UJ:ZONE0 ;

– powercap:::MAX_POWER_A_UW:ZONE0 ;

– powercap:::POWER_LIMIT_A_UW:ZONE0 ;

- powercap:::TIME_WINDOW_A_US:ZONE0 ;
- powercap:::MAX_POWER_B_UW:ZONE0 ;
- powercap:::MAX_POWER_B_UW:ZONE0 ;
- powercap:::POWER_LIMIT_B_UW:ZONE0 ;
- powercap:::TIME_WINDOW_B:ZONE0 ;
- powercap:::ENABLED:ZONE0 ;
- powercap:::NAME:ZONE0 package 1;
- powercap:::ENERGY_UJ:ZONE0_SUBZONE0 ;
- powercap:::MAX_ENERGY_RANGE_UJ:ZONE0_SUBZONE0 ;
- powercap:::MAX_POWER_A_UW:ZONE0_SUBZONE0 ;
- powercap:::POWER_LIMIT_A_UW:ZONE0_SUBZONE0 ;
- powercap:::TIME_WINDOW_A_US:ZONE0_SUBZONE0 ;
- powercap:::ENABLED:ZONE0_SUBZONE0 ;
- powercap:::NAME:ZONE0_SUBZONE0 DRAM 1;
- powercap:::ENERGY_UJ:ZONE1 ;
- powercap:::MAX_ENERGY_RANGE_UJ:ZONE1 ;
- powercap:::MAX_POWER_A_UW:ZONE1 ;
- powercap:::POWER_LIMIT_A_UW:ZONE1 ;
- powercap:::TIME_WINDOW_A_US:ZONE1 ;
- powercap:::MAX_POWER_B_UW:ZONE1 ;
- powercap:::POWER_LIMIT_B_UW:ZONE1 ;
- powercap:::TIME_WINDOW_B:ZONE1 ;
- powercap:::ENABLED:ZONE1 ;
- powercap:::NAME:ZONE1 package 0;

– powercap:::ENERGY_UJ:ZONE1_SUBZONE0 ;

– powercap:::MAX_ENERGY_RANGE_UJ:ZONE1_SUBZONE0 ;

– powercap:::MAX_POWER_A_UW:ZONE1_SUBZONE0 ;

– powercap:::POWER_LIMIT_A_UW:ZONE1_SUBZONE0 ;

– powercap:::TIME_WINDOW_A_US:ZONE1_SUBZONE0 ;

– powercap:::ENABLED:ZONE1_SUBZONE0 ;

– powercap:::NAME:ZONE1_SUBZONE0 DRAM 0;

2.6.10 PAPI and MPI

PAPI supports MPI [30]. Since MPI is widely used in HPC systems that are very energy consuming, the need for a library capable of measuring the relevant events and metrics also on exascale systems is significant both for academic studies and companies. PAPI supports MPI since the very first version and there are many papers leveraging this technology to study the impact of algorithms on HPC systems. One must further bear in mind certain other considerations. In fact, when using timers in applications that contain multiplexing, profiling, and overflow, MPI uses a default virtual timer and must be converted to a real timer for the application to work properly. Otherwise, the application will exit. The very first program used to test the architecture, RAPL, Powercap events and MPI was the computation of pigreco with Montecarlo method as showed in listing Listing 2.2. The MPI initialization and Finalization are executed before the PAPI initialization, and that in this simple example all ranks monitor the designated PAPI event.

Listing 2.2: basic program with PAPI and MPI

```c
#include <papi.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "papi_test.h"

int main(int argc, char *argv[]){

  int  n, myid, numprocs, i, retval, EventSet =
     PAPI_NULL, code=-1;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x;
  char eventname[] ="powercap:::ENERGY_UJ:ZONE0";
  long_long values[1] = {(long_long) 0};

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /*Initialize the PAPI library */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT)
    test_fail(__FILE__, __LINE__, "PAPI_library_init
       failed\n", retval);

  /* Create an EventSet */
  retval = PAPI_create_eventset(&EventSet);
  if (retval != PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_create_eventset
       failed\n", retval);

   /* Translate event name to its code */
  retval = PAPI_event_name_to_code(eventname, &code);
    if (retval != PAPI_OK)
        test_fail(__FILE__, __LINE__, "
           PAPI_event_name_to_code", retval);

  /* Add Total Instructions Executed to our EventSet */
  retval = PAPI_add_event(EventSet, code);
  if (retval != PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_add_eventset
       failed\n", retval);

  /* Start counting */
  retval = PAPI_start(EventSet);
  if (retval != PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_start failed\n",
        retval);

    n=1000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16
           f\n", pi, fabs(pi - PI25DT));


   /* Read the counters */
   retval = PAPI_read(EventSet, values);
   if (retval != PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_read failed\n",
       retval);

   printf("After reading counters: %lld\n",values[0]);

   /* Stop the counters */
   retval = PAPI_stop(EventSet, values);
   if (retval != PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_stop failed\n",
       retval);

   printf("After stopping counters: %lld\n",values[0]);

   MPI_Finalize();
}
```

2.7 Fault Tolerance

As an effect of the steeply increase in size of the HPC systems, the Mean Time Between Failures MTBF has plunged. Consequently, the research on new fault tolerance techniques has a huge impact on the improvement of the run-time behaviour of these clusters, the management of the malfunctions affecting the computation is a crucial issue. To better understand how a failure occurs, it is necessary to define fault and errors. Failure is defined as a behaviour not conforming with the requirements, error is a problem that can generate an incorrect behaviour and a fault is a set of events that can cause errors. Fault is the concrete causing occurrence, error is a sequence of events, whereas failure is the visible effect of the two quoted above. Faults are classified as transient, intermittent, and permanent, failures are classified as Bohrbug: a repeatable and neat failures, often easy to be corrected, and Eisenbug: a less repeatable, hard to be understood failure, errors are classified as soft errors or fail-continue and hard-errors or fail-stop.

The fail-stop errors may anticipate a catastrophic result by aborting a long, energy-intensive computation. The phases that follow the occurrence of the error are the identification and the recovery. As matter of fact one of the first approach to the fault tolerance in HPC systems was the Checkpoint Restart, which consists in saving the state of long-running programs. One of the most relevant revolutions of this approach suggest saving the state in memory to make the recovery faster, this was proposed by Plank et al. [32] and called diskless. The checkpoint/ restart can be applied at four different levels: system, compiler, user, algorithm.

The large-scale linear algebra problems are a relevant part of the issues tackled through the use of HPC system. Since the computations are long and expansive in terms of resources, it is necessary to apply the fault tolerance techniques to the linear systems resolution. Algorithm Based Fault Tolerance ABFT is a series of algorithm-specific recovery actions that are devised to handle both soft-errors and hard-errors.

2.8 Linear systems solver algorithms

To better understand the Inhibition Method algorithm is necessary to make a premise by introducing the Gauss elimination method which is the most notorious algorithm for linear systems resolution. To enhance the importance of linear systems resolution as a key-issue in many fields, in this section is also presented a remarkable alternative to IMe: ScaLAPACK.

2.8.1 Gauss Elimination Method

Gaussian elimination, also known as row reduction, is an algorithm for solving systems of linear equations. It consists of a sequence of operations performed on the corresponding matrix of coefficients. This method can also be used to compute the rank of a matrix, the determinant of a square matrix, and the inverse of an invertible matrix[16]. The Gauss elimination applied to a matrix A consist in 4 four steps[4]:

– We consider the first non-null column of A with index j_1. If the first element of this column a_{1j1} is 0, we pass to the third step, if a_{i1j1} is not 0 with $i_1 > 1$ we pass to the second step.

– We swap the first row with the i_1 row, obtaining a new matrix where the element in $1, j_1$ position is other than zero.

– We delete in the j_1 column all the elements different from zero from the second row to the bottom, by leaving the first row unchanged, and by using it to perform the elimination from the others. In this way we obtain a new matrix that has $a_{i,j1} = 0$ for $i = 2, \dots, m$.

– By leaving the first row unchanged, now we proceed by repeating the first the second and the third step on the $(m-1) \times n$ matrix of the $m-1$ rows only if they are not completely null, in that case we stop.

After the completion of these steps the matrix is in the echelon form which means that all rows consisting of only zeroes are at the bottom, and the leading coefficient (also called pivot) of a nonzero row is always strictly to the right of the leading coefficient of the row above it.

Instead of stopping when the matrix is in echelon form, one could continue until the matrix is in the reduced row echelon form, which means that all the leading coefficients have 1 as value. The process of row reducing until the matrix is reduced is sometimes referred to as Gauss–Jordan elimination, to distinguish it from stopping after reaching the echelon form[25].

2.8.2 Inhibition method

The Inhibition Method was proposed in 1963[11] to simplify the analysis of complex electric circuit. Then it turned out to be useful also in the resolution of physical linear systems and square matrix inversion. This algorithm is general because it can be applied to every linear system or non-liner system due to one of them, and it is independent from the physical nature of the system itself.

The Inhibition Method (IME) is an iterative, exact, non-inverting method to solve any linear system. It derives from the Cross method, from which IME inherits the fundamental characteristic of decomposing the problem into easier to solve sub-problems, even though the cross method is non-exact[6].

IME produces a hierarchical sequence of sub-systems, at the end of which only elementary systems can be found. Hence, they can be solved rapidly with the minimum of knowledge, consequently only few program code lines are needed.

The first step of the algorithm consists in the splitting process. The problem is split into more simple sub-systems by inhibiting a component. Then after decomposing the problem into the most elementary element to solve, there is a recollecting process to unify all the result and obtain the resolution of the problem.

Due to the splitting and recollecting process the method introduces a series of additional computations that does not guarantee high efficiency.

IMe starts from considering the linear system with n equations and n unknowns in its matrix form: $Ax = b$, where A is the $n \times n$ matrix of coefficients, b is the vector of constant terms and x contains the unknowns. First, it prescribes to compute a matrix T (n) , called inhibition table, and a vector $h(n)$ of elements, called auxiliary quantities. $T(n)$ is built using only the $a_{i,j}$ coefficients from A as follows:

$$T^{(n)} = \left[\begin{array}{ccccc|ccccc} \frac{1}{a_{1,1}} & 0 & \dots & \dots & 0 & \frac{a_{2,1}}{a_{1,1}} & \dots & \dots & \frac{a_{n,1}}{a_{1,1}} & \\ 0 & \frac{1}{a_{2,2}} & \dots & \dots & 0 & \frac{a_{1,2}}{a_{2,2}} & 1 & \dots & \dots & \frac{a_{n,2}}{a_{2,2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & \dots & \frac{1}{a_{n-1,n-1}} & 0 & \vdots & \vdots & \dots & 1 & \frac{a_{n,n-1}}{a_{n-1,n-1}} \\ 0 & \dots & \dots & 0 & \frac{1}{a_{n,n}} & \frac{a_{1,n}}{a_{n,n}} & \dots & \dots & \frac{a_{n-1,n}}{a_{n,n}} & 1 \end{array}\right]$$

The algorithm is divided in two parts: the first part is the INITIME procedure. It handles handles the initialization of the $T(n)$ matrix. The second part reduces iteratively the number of rows and columns. The matrix $T(n)$ and the vector $h(n)$ can be seen as a decomposition of the original problem into n sub-problems (one for each row). The auxiliary quantities initialization is the result of the following formula.

$$h_i^{(n)} = \frac{1}{1 - t_{n,n+1}^{(n)} t_{i,2n}^{(n)}}$$

Along with $T(n)$ and $h(n)$, the algorithm imposes to initialize the vectors b (n) and x (n) as showed in 2.1.

The problem splitting ends the first phase. Thus, it is possible to proceed with the second part which consist in applying to each element of T(n) an equation to iteratively reduce the number of rows and columns. At each step, usually called level, $l(withl = n, ..., 1)$, $T(l)$ represents the original problem decomposed into l sub-problems. The equation, called fundamental formula, is as follows.

$$t_{i,j}^{(l-1)} = (t_{i,j}^{(l)} - t_{l,j}^{(l)} \cdot t_{i,n+l}^{(l)}) \cdot h_i^{(l)}(1)$$

with i, j, l such that $i = 1, . . ., l - 1$, $j = 1, . . .,n + l - 1$, $l = n, . . ., 1$

It should be noted that each element of the inhibition table ($t_{i,j}^{(l-1)}$ is recomputed using: its previous value ($t_{i,j}^{(l)}$, the element on the last row ($t_{l,j}^{(l)}$ and column ($t_{i,n+l}^{(l)}$ of the previous inhibition table and the corresponding auxiliary quantity ($h_i^{(l)}$. Furthermore, it should be emphasized that due to how $T^{(n)}$ was initialized, the value of the 0-entries in the first $l-1$ columns of any $T^{(l)}$ is still null in $T^{(l-1)}$ because the last elements of these columns are all 0. Hence the fundamental formula, should be applied only to those elements which change at every step, and this represents a significant optimization in terms of flops.

To summarize, the h vector should be determined at each new level;- only the element in the last row ($t_{l,n+i}^{(l)}$) and column ($t_{i,n+l}^{(l)}$) of $T^{(l)}$ are needed at each level, moreover at each level l also the $l-n$ elements of the vector $x^{(n)}$ and $l-1$ elements of $b^{(n)}$ are modified. At level $l=1$, when $T^{(1)}$ matrix has only one row and n columns, the iteration stops, and the solution of the linear system can be found in the vector $x^{(1)}$. The complexity measured in terms of flops needed for the computation of the sequential algorithm depends on the complexity of the two parts, the initialization part has a n^2 complexity, whilst the second part has a $3/2n^3 + O(n^2)$ complexity. The flops needed for the Gauss Jordan elimination and this IMe are comparable. The memory to store floating point values of the IMe algorithm is $2n^2+3n$, in this case Gauss Jordan Elimination is less memory-consuming $n^2 + O(n)$ than IMe.

2.8.3 Parallelization of IMe

One of the best ways to compute an algorithm faster is to parallelize parts which are independent from the others, ascertaining that the amount of data replicated and exchanged by the nodes is minimized. In the following procedure, we will use $t^{(l)}$ $,j$ and $t^{(l)}$ i, to address the j-th column and the i-th row of $T^{(l)}$, respectively, and N as number of nodes, considering N-1 slaves and one master. Under certain condition the fundamental formula allows an independent computation of each

	Initialization	**Update**
$\boldsymbol{b}$	$b_i^{(n)} = \begin{cases} b_n, & i=n \\ b_i - t_{n,n+i}^{(n)} b_n, & \text{o/w} \end{cases}$	$b_i^{(l-1)} = b_i^{(l)} - t_{l-1,n+i}^{(l-1)} b_l^{(l)}, \quad i = 1 \dots l-2$
$\boldsymbol{x}$	$x_i^{(n)} = \begin{cases} t_{n,i}^{(n)} \cdot b_n, & i=n \\ 0, & \text{o/w} \end{cases}$	$x_i^{(l-1)} = x_i^{(l)} + t_{l-1,i}^{(l-1)} b_l^{(l)}, \quad i = l-1 \dots n$

Table 2.1: IMe prescribed steps to compute the system's solution

element of T. Precisely, three different parallelization schemes are possible:

i) column-wise, entailing that the node computing the last column $t^{(l)}_{*,n+l}$ should make it available to all the others, and all the nodes should share $h^{(l)}$;

ii) row-wise, symmetrically, the node computing the last row $t^{(l)}_{l,*}$ should make it available to all the others and $h^{(l)}$ is shared;

iii) block-wise, combining row-wise and column-wise parallelization.

As a matter of fact, the scheme used in the Inhibition Method Parallelized IMeP is the column-wise because its characteristic fits the integration with the fault tolerance requirements better than the others. Each computing node works on a subset T' of T, by iteratively applying the fundamental formula. At every level it is also necessary to broadcast from the master to the slaves h, whilst the node in charge of the computation of the last column $t_{*,n+l}$ should broadcast it to all the other nodes, and besides only the n elements of the last row which result modified after the application of the fundamental formula must be sent to the master.

One of the concerns of parallelization is the memory usage: by spreading the execution of the algorithm in N nodes the memory occupation increases from $2n^2 + 3n$ to $moIMeP = 2n^2 + 2nN + 3n$, whereas the flops remain the same. Furthermore, the distributed environment forces to message exchange, which has a cost. The traffic generated by the exchanged messages is measured in number of messages, and volume, meaning the number of floating points. The total number and volume of messages exchanged is:

$$M_{IMeP} = n^2 + 2(N-1)n + 2(N-1)$$

$$V_{IMeP} = (N+2)n^2 + 2(N-1)n$$

Which is the sum of:

– the broadcast of the last column $t_{*,2n}$ from the master to all the slaves, for the initialization.

– h is broadcasted from the master to all slaves.

– the node that oversees of the last column $t_{*,n+l}$ broadcasts it to all the other nodes.

– all the slaves send the last entry of their columns to the master. Only n elements of the last row are exchanged, because all the others are certainly 0.

Fault tolerance applied to Inhibition method

As mentioned before there is a huge interest in the solution of linear system on HPC systems, therefore a fault tolerance approach in the execution of IMe should be adopted. To avoid the use of the Checkpoint/Restart technique, which is inefficient, it is possible to modify IMe to Implement the fault tolerance strategy directly into the algorithm. The fault tolerant Inhibition method IMeFT uses the column wise parallelization to compute a row-wise sum of all the entries of $T^{(l)}$. This checksum vector $s^{(l)}$ is stored in the memory of an additional computing processor and in case of fail-stop involving a column of $T^(l)$ it is possible to recover its entries just by solving n linear equations. Each entry of the $s^{(l)}$ vector is computer through this formula:

$$s_i^{(l)} = \sum_{k=1}^{n+l} t_{i,k}^{(l)} \forall i \in 1 \dots l \tag{2.1}$$

The application of this checksum technique to IMe comes with a notable property. Given $s^{(l)}$ checksum vector of the inhibition table$T$ at level l, the application of the fundamental formula to $s^{(l)}$ produces a vector $s^{(l-1)}$, which is again the checksum vector of $T^{(l-1)}$.

The property above allows to calculate the checksum vector only once at the beginning of the computation. The processor hosting $s^{(l)}$ can apply the fundamental formula on it just as all the other processors do on the columns of T. The vector computed in this way will continue to host the checksums of $T^{(l)}$ for any following level.

The complexity of IMe depends on the number of processors employed in the computation. To run IMeFT a processor to handle the checksum vector is necessary, hence the number of processors will be N+1. The overall complexity of the algorithm in terms of flops is:

$$flops_{IMeFT} = \frac{3}{2}n^3 + \frac{11}{2}n^2 - 4n = \frac{3}{2}n^3 + O(n^2)$$

Since the cells of the checksum vector (as well as those of the inhibition table) can be reused at each level, when enhancing IMe with fault tolerance, the memory occupation increases only of the dimension of s:

$$mo_{IMeFT} = mo_{PIMe} + n = 2n^2 + 2Nn + 4n$$

Since the communication now involves N+1 nodes the number and volume of exchanged messages can be expressed as follows.

$$M_{IMeFT} = n^2 + 2Nn + 2N$$

$$V_{IMeFT} = (N+3)n^2 + 2Nn + n$$

Applying the fault tolerance directly to IMe without the use of the C/R techniques has a huge impact on performances[7], because the algorithm is naturally resilient to errors, and it is not necessary to save periodically the state of computation on disk or on memory, which is acknowledged as a slower solution both in the saving phase and in the restart phase. Moreover, the C/R technique forces to restore the computation at the saved checkpoint; this means that many already performed operations which were not saved must be repeated. Another strength of the checksum vector is its tolerance to hard errors. A hard error can occur on a processor in which more than one column of the linear systems was computed without compromising the result of the computation. In order to guarantee this feature, the checksum vectors and the assignments of the columns to the nodes must be conveniently performed. Moreover, recent studies has showed that IMe is resilient also to multiple faults, only with slight changes to the checksum vectors of the original single fault tolerant algorithm, and this is possible without adding significant overhead to the computation.

2.8.4 Scalable LAPACK

ScaLAPACKs, acronym for Scalable Linear Algebra PACKage[5], is a library of high-performance linear algebra routines for distributed memory computers supporting MPI. ScaLAPACK solves dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems. The key ideas incorporated into ScaLAPACK include the use of:

– a block cyclic data distribution for dense matrices and a block data distribution for banded matrices, parametrizable at runtime.

– block-partitioned algorithms to ensure high levels of data reuse.

– well-designed low-level modular components that simplify the task of parallelizing the high-level routines by making their source code the same as in the sequential case.

ScaLAPACK is highly portable due to the limited dependencies from other libraries, it needs Basic Linear Algebra Subprograms (BLAS) and Basic Linear Algebra Communication Subprograms (BLACS) to run. LAPACK uses the level 3 of BLAS which includes the matrix-Matrix operations. ScaLAPACK is written in Fortran, except for a few symmetric eigenproblem auxiliary routines written in C. As far as the Gaussian Elimination is concerned, to optimize the numerical instability caused by the roundoff error it is necessary to introduce the Partial

Algorithm 1 Parallel IMe factorization with fault resilience.

Input: $\boldsymbol{A}$, matrix of coefficients; $\boldsymbol{b}$ vector of constant terms.

Output: $\boldsymbol{x}$ solution vector.

```
1: procedure IMeHP($\boldsymbol{A}, \boldsymbol{b}$)
2:     $\boldsymbol{T}, \boldsymbol{x}, \boldsymbol{b} \leftarrow$ InitIMe($\boldsymbol{A}$)
3:                                  ▷ Each processor gets a subset $\boldsymbol{T'}$ of columns in $\boldsymbol{T}$:
4:     $\boldsymbol{T'} \leftarrow$ scatterColumns($\boldsymbol{T}$, $root = 0$)
5:     for $l \leftarrow n \ldots 2$ do
6:         $q' \leftarrow$ processor holding last column of $\boldsymbol{T}$
7:                                  ▷ processors exchange last column and row of $\boldsymbol{T}$:
8:         $\boldsymbol{t}_{*,n+l} \leftarrow$ Broadcast($t_{*,n+l}$,root=$q'$)
9:         $\boldsymbol{t}_{l,*} \leftarrow$ AllGather($t_{l,*}$)
10:                                 ▷ only one processor computes the solution:
11:        if $rank == 0$ then
12:            for $i \leftarrow l \ldots n$ do
13:                $x_i \leftarrow x_i + t_{l,i} b_l$
14:            end for
15:        end if
16:        for $i \leftarrow 1 \ldots l-1$ do
17:            if $rank == 0$ then
18:                $b_i \leftarrow b_i - t_{l,n+i} b_l$
19:            end if
20:            $h_i = 1/(1 - t_{i,n+l} * t_{l,n+i})$
21:            for each $\boldsymbol{t}_{*,j} \in \boldsymbol{T'}$ do
22:                if $j == i$ or ($j \geq l$ and $j \neq n+i$) then
23:                                 ▷ apply the fundamental formula:
24:                    $t_{i,j} \leftarrow (t_{i,j} - t_{l,j} \cdot t_{i,n+l}) \cdot h_i$
25:                end if
26:            end for
27:        end for
28:    end for
29:    return $\boldsymbol{x}$
30: end procedure
```

Pivoting technique, which consists in swapping rows so that $A_{(i,i)}$ is the largest in column. In fact, the problem of numerical instability occurs when diagonal $A_{(i,i)}$ is tiny (not just zero) and the algorithm may terminate but get a completely wrong answer.

Chapter 3

Design and implementation

The objective of this work consists in monitoring the behaviour of the linear system solver algorithms through the energy consumption of the CPU package 0 and 1 and DRAM 0 and 1, which represent the total consumption of the CPU and the power consumed by the random-access memory (RAM), respectively.

3.1 Requirements

In order to monitor the whole execution, the program should be easily portable since it should adapt to different algorithms. Moreover, it should fit both a white box approach and a black box approach introducing only small changes. Another important characteristic is the efficiency of the adopted solution because the overhead caused by the monitoring libraries should not affect the performance of the algorithm itself, or at least the impact should be very low. Eventually, also the modularity plays a remarkable role. The structure of the monitoring system requires a function call to begin the monitoring and a function call at the end. These parts should not interact directly even if they will be working on the same arrays of events to modify the values.

The testing framework should support both simple and complex tests. Moreover, it is important that results are collected automatically and saved in a human readable format in order to review them. The test framework should not impact the structure of the tested algorithms and of the overall performances. The structure of tests should adapt to the different algorithms and work properly. Furthermore, since the test are run multiple nodes architecture in which each node will have different energy values, all the measurements should be collected properly, and the solution should be scalable in order to manage the execution on different nodes.

3.2 Naive solution

The first proposed solution entails the redefinition of the MPI primitives for initialization and finalization. The new functions are wrappers that also contains the initialization and the termination of PAPI monitored events in addition to the MPI primitives. The redesigned MPI initialization function has a first part in which are performed the PAPI operation for: library initialization, thread initialization, event set creation with the specification of all the desired events. These functions calls are followed by the actual begin of the parallel program represented by the MPI initialization. As far as the MPI finalization function is concerned, it should stop the monitoring performed throughout the PAPI events, then it should gather the values of the PAPI event set from all the MPI ranks. Finally, it is possible to save the values of the event set in a new file opened by the master rank. After persisting the values of the PAPI events it is possible to perform the MPI finalization of the parallel part of the program.

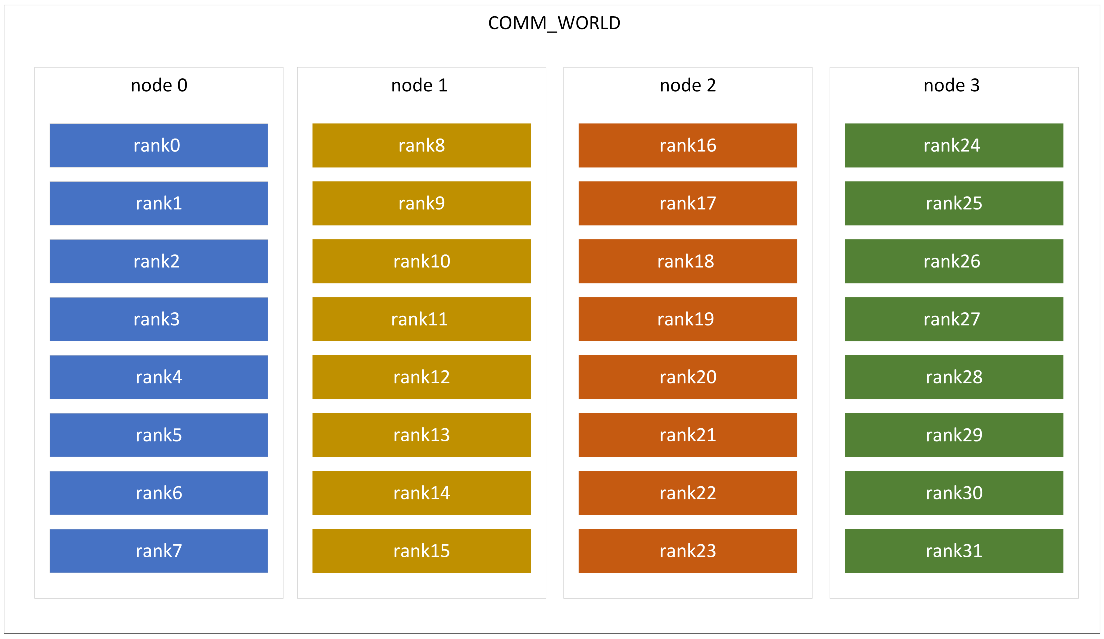

Figure 3.1: Structure of the MPI communicator in the naïve solution, with the execution on four different processors

The followed approach is black box, because through the redefinition of the MPI primitives it is possible to keep the linear system solver algorithm as it is. The only significant edit to apply to the original file is the incorporation of the dependency from the header file in which the functions are defined.

The whole program is launched from the main function in which the linear system solver algorithm is written between the redefined function for MPI initialization and MPI finalization. Since the monitoring is injected at the beginning and at the end of the algorithm, the computation of the metrics is coarse-grained. As a matter of fact, the measures are sampled at the beginning and at the end without considering the distinction between the initialization of the matrix and the actual computation of the solution. This coarse-grained solution respects the requirements of modularity and portability, although the efficiency is particularly compromised by the execution of the monitoring on each rank of the MPI program. Hence, each rank included in the execution will perform both the monitoring and the linear system solver algorithm. This creates a slight overhead in the computation designated for each node, but with regards to the whole execution, the repetition of the monitoring on each node is redundant. Therefore, the second approach should aim to the optimization of the monitoring by assigning only one rank per node to complete this computation.

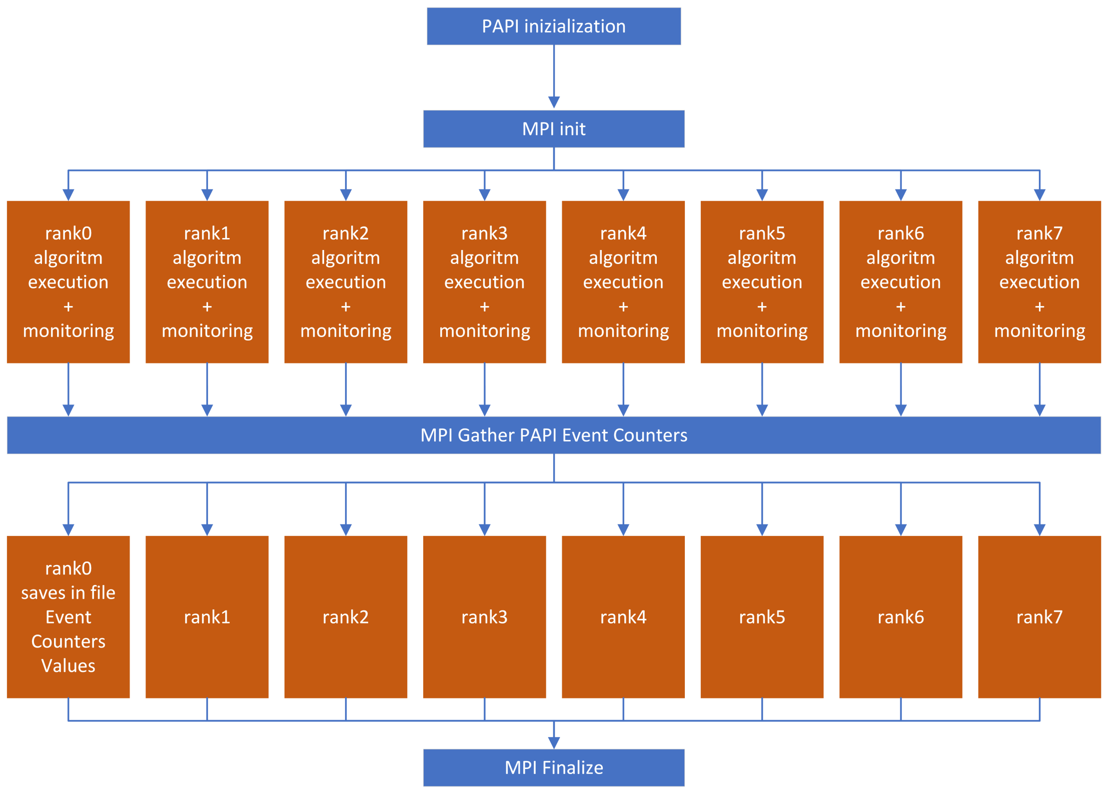
PAPI inizialization
MPI init
rank0 algoritm execution + monitoring
rank1 algoritm execution + monitoring
rank2 algoritm execution + monitoring
rank3 algoritm execution + monitoring
rank4 algoritm execution + monitoring
rank5 algoritm execution + monitoring
rank6 algoritm execution + monitoring
rank7 algoritm execution + monitoring
MPI Gather PAPI Event Counters
rank0 saves in file Event Counters Values
rank1
rank2
rank3
rank4
rank5
rank6
rank7
MPI Finalize

Figure 3.2: Structure of the MPI program in the naïve solution

3.2.1 Implementation

The first proposed solution entails the redefinition of `MPI_Init()` and `MPI_Finalize()`. The new function for MPI initialization should call the functions of the `papi.h` library and it should perform the library initialization; and the thread initialization, it should create the event set and add all the desired events. All these operations are performed before the `PMPI_Init()` which is the last instruction of `MPI_Init()`. Whilst the `MPI_Finalize()` should call `PAPI_stop()` in the first place, and then through `MPI_Gather()` it can collect the values of the PAPI event set from all the MPI ranks. Finally, it is possible to save the values of the event set in a new file. The last instruction is `PMPI_Finalize()`.

3.3 Common node solution

The second solution improves the efficiency of the first naïve solution by designating only one rank in each node to run the monitoring task. As a matter of fact, the collected metrics regards the overall energy consumption of the CPU package, of the processor die and of the DRAM. Therefore, since by simplifying each node represents a CPU and each rank represents a core, only one rank per processor should run the monitoring. MPI communicators can establish which ranks belongs to one node, thus it is possible to group ranks which shares the same processor (or node) by creating new sub-communicators. After the implementation of the sub-communicators, it is possible to appoint one rank for the monitoring task. These nominated ranks will differentiate their execution from the others. Firstly, they begin the monitoring by initializing PAPI, they start the measurements of the chosen metrics, then they execute the assigned part of the linear system solver algorithm like all the other ranks, finally after all the ranks on the same node have finished the computation, they stop the measurements. Hence, the monitoring ranks should wait all the processing ranks in their communicator before stopping the measurements, to achieve this it is necessary to introduce an MPI barrier of synchronization that grants the alignment at the same point of the execution for all ranks in the same group. It is important to stress that the begin and the end of the monitoring are always preceded by a MPI synchronization barrier for the ranks on the same node to improve the correctness of the measurements.

The followed approach is white box because the monitoring part performed by PAPI is injected in a specific rank per each node, thus it is necessary to group the ranks first and then to appoint one rank for the monitoring.

This solution maintains the modular and portable structure of the naïve solution, but the efficiency is significantly improved by the differentiated execution of the ranks. This solution compromises with the time spent for the synchronization

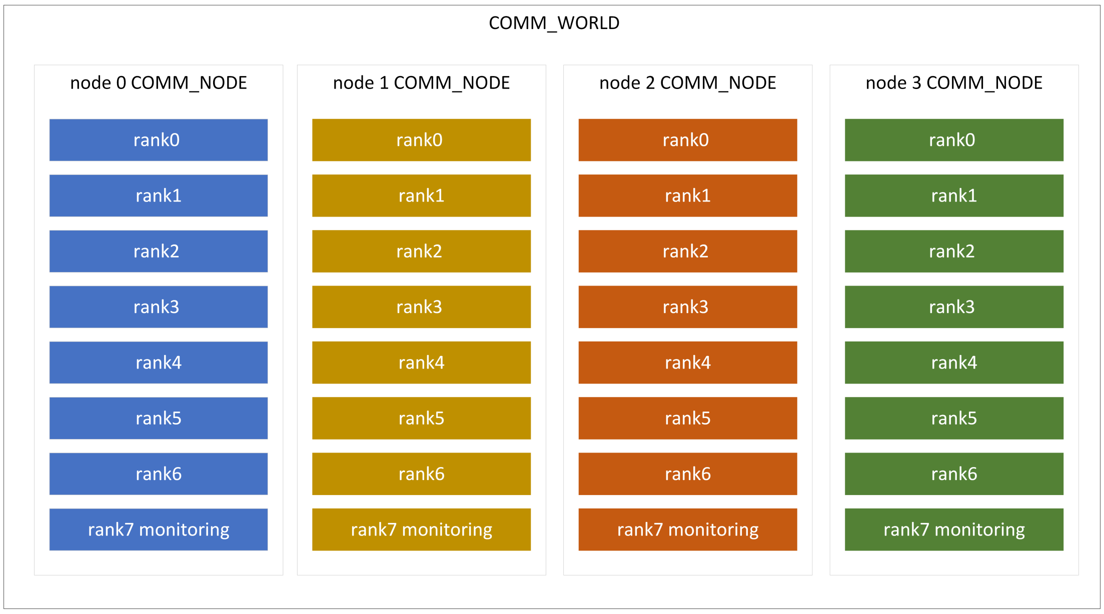

Figure 3.3: Structure of the four MPI node communicators in the second solution, with the execution on four different processors

in order to achieve accurate measurements, as a matter of fact the synchronization slows the execution of the program and adds some overhead, not directly to the linear system solver algorithm but to the whole execution. Another remarkable difference between the naïve solution and the parallel one regards the PAPI initialization: in the first one it occurs before the MPI initialization, when the program is still sequential, whilst in the second solution the PAPI initialization take place only in the ranks designated for monitoring.

3.3.1 Implementation

The second proposed solution entails the designation of one rank per each node where to perform the monitoring. After `MPI_init()` a new communicator for each is created through the method `MPI_Comm_split_type()`, through the constant split type MPI_COMM_TYPE_SHARED the ranks are automatically divided in group based on their rank. The following step is to designate the monitoring rank, which is always the one which has the highest rank value in the communicator. If the rank is designated for monitoring it calls `start_monitoring()` from `papi_monitoring.h`. The values are passed as reference because also the fuction `end_monitoring()` needs the events values to process the monitoring measurements. The method `start_monitoring()` uses `PWCAP_plot_init()` to initialize

PAPI. The functions for PAPI initialization perform the library initialization, the thread initialization, the creation of the event set and the addition of all the desired events. Eventually, `start_monitoring()` calls `PAPI_start_AND_time()` which starts PAPI monitoring. The algorithm is executed between `start_monitoring()` and`stop_monitoring()`. Before stopping the whole monitoring, ranks which run on the same node are synchronized to the `MPI_Barrier()`.When all the ranks has terminated the execution of the linear system solver algorithm the PAPI monitoring procedure is ended by the monitoring ranks. The function `end_monitoring()` from `papi_monitoring.h` stops PAPI event counters with `PAPI_stop_AND_time()` , then it creates one file for each processor with `file_management()`. In each file are saved the values of PAPI event counters for the processor in which the node waa run. The function `PAPI_term()` cleans up and destroys PAPI Event set. After that the monitoring is completed and `MPI_Finalize()` is performed.

Since most of the RAPL events of interest are included in powercap event set, which also adds the power capping functionalities, the monitored events will belong only to powercap event set. Therefore, the array `event_names` which is used to list the names of the monitored events in `papi_monitoring.h` will contain all the powercap event set displayed by PAPI. This array is a parameter of `papi_event_name_to_code` which translates the names into the macros of the PAPI library.

3.4 Tester: command line interface

3.4.1 Tester.c

The solutions discussed above are integrated in a more complex system that is capable of launching all the different versions of IMe, ScaLAPACK and LAPACK, with many different options. This application is called tester, and it offers a command line interface where to specify the version of the linear system solver, the options and the commands needed for the execution.

User can launch tester by specifying four different commands:

– `--help` prints information about the usage of the command line interface;

– `--list` prints the list of testable routines;

– `--save` saves the generated matrices to files;

– `--run` runs the specified tests. The name specified by the user after this command will be parsed from the routine mapper that will launch the right test routine and execute the correct version of the linear solver algorithm.

Furthermore many options can be specified:

- `-v <integer number>` specifies the verbosity level for 0 to 3, in which 0 correspond to quiet;
- `-nm <integer number>` specifies the matrix rank;
- `(-nrhs <integer number>` is the Right Hand Side (RHS) of the equation, considering the expression $Ax = b$ RHS number represents b;
- `-seed <integer number>` is the seed of the random generation of the linear system coefficients;
- `-cnd <integer number>` is the condition number of the input matrix;
- `-no-cnd-set` disables matrix pre-conditioning;
- `-no-cnd-readback` disables condition number checking after generation;
- `-no-nre-readback` disable normwise relative error checking due to computation with float and not fractions;
- `-mat-gen <string [par or ser] >` is the type of the random generation parallel or sequential;
- `-r <integer number>` run multiple times corresponded to the specified number of repetitions;
- `-o <file path>` saves the output to CSV file
- `-i <file path>` takes as input matrices base the ones saved at the specified the file path name (.A, .X, .B auto appended)
- `-ft <integer number>` is the fault-tolerance level [0-..] (0=none);
- `-fr <integer number>` simulates a correspondent number faulty MPI ranks;
- `-fl <integer number>` simulates faulty IMe inhibition level;
- `-npf <integer number>` is the number of simulated faults [0-..] (0=none);
- `-nps <integer number>` number of spare processes for recovery [0-..] (0=none);
- `-cp <integer number>` checkpointing interval;
- `-spk-nb <integer number>` ScaLAPACK blocking factor;

- `-ime-nb <integer number>` IMe blocking factor;
- `-ma <integer number>`monitors the execution by measuring the energy values, =0 monitors the execution, , =1 monitors the allocation; =2 monitors the allocation and the execution;
- `-head <string>` appends to the head of the monitoring file a string where are specified the hardware parameters of the execution.

Since the version of the linear systems solver algorithm is specified in the command line after the `-run` command, there is a method that maps the string name specified by the user with the correspondent algorithm test routine in order to launch the correct version of the algorithm. This method is launched directly from tester during the parse of user's input.

Moreover, there are two more helpers: one is a container for the structures used both from tester and from the algorithm test routines, and the other one is a list of define directives which contains the name of all the available versions.

The structure container defines: a structure with the required parameters for the algorithms that runs on a parallel environment, a structure with the required parameters for the fault tolerant algorithms, a structure with the input matrices, a structure for the output that contains the start and the end time of the tests, and finally a result structure for the total time spent for the execution and the result code.

3.4.2 Test routines

Each version of the linear systems solver algorithm depends on a routine which is a sort of algorithm test wrapper. Into each routine, the linear system is initialized, respecting all the options specified in the command line interface, the resolution algorithm is performed, and the matrix representing the linear system is deallocated. The routines are wrappers of the actual the linear systems solver. The routine of correspondent versions of IMe, ScaLAPACK and LAPACK, have common signatures: e.g., if a version IMe supports multiple faults, the correspondent ScaLAPACK test routine that wraps the multiple faults version of ScaLAPACK will have the same signature of the function.

Since these routines are wrappers of the linear systems solver algorithms the code for monitoring the execution is injected there. If the option `-ma` is specified with code 2, `start_monitoring` and `end_monitoring` includes the allocation and the monitoring. If the option `-ma` has code 0 only the execution of the code is monitored, whilst if `-ma` is 1 only the allocation is monitored. This allows to collect data about different parts of the execution. To maintain a more efficient code a new test routine is created for each level of monitoring to avoid the injection of many if statements and repeated parts. The monitoring is only possible for the IMe version `pbDGESV_CO.bf1.h` and the ScaLAPACK version `ScaLAPACK_pDGESV.h`. In the figure 3.5 are showed all the dependencies between the classes.

The `--head` option is added with the aim of specifying the parameters of the execution on Marconi A3. These parameters are saved with the monitoring information because they must be considered for the analysis of the energy consumption. Hence, it is fundamental to save them with the collected data.

3.4.3 Algorithms versions

The tested version of IME is `pbDGESV_CO.bf1.h` that is a parallel double precision general solver. CO stands for compression which is the new technique introduced in the last version of IMe with the aim of improving the performances. Since this is the last version of the algorithm and it is undergoing a patent approval, it will not be further discussed in this work. The correspondent version of ScaLAPACK is `ScaLAPACK_pDGESV.h` which is a parallel double precision general version of the algorithm. Both IMe and ScaLAPACK algorithms are tested in their non-fault-tolerant version.

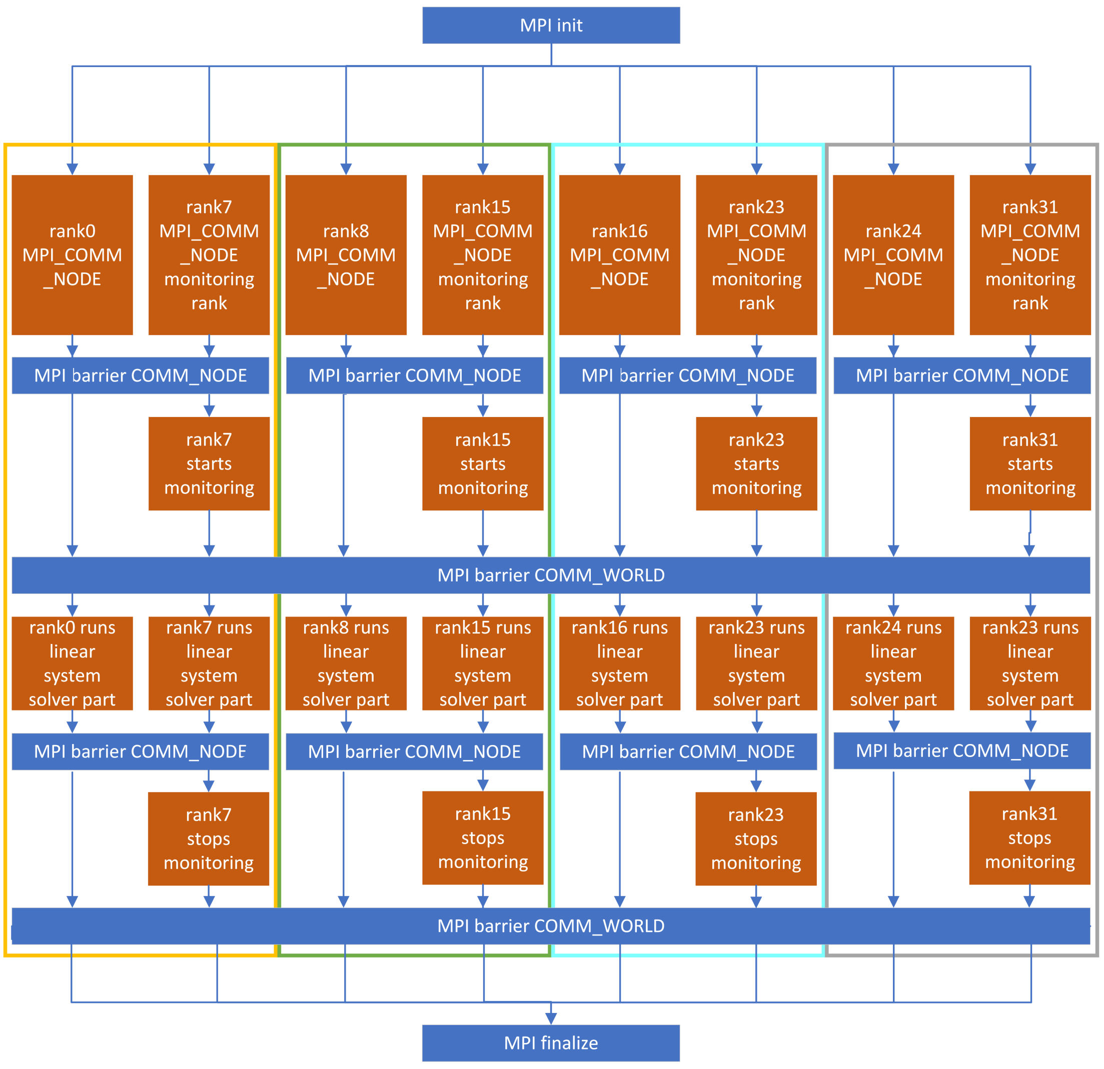
MPI init
rank0 MPI_COMM_NODE
rank7 MPI_COMM_NODE monitoring rank
rank8 MPI_COMM_NODE
rank15 MPI_COMM_NODE monitoring rank
rank16 MPI_COMM_NODE
rank23 MPI_COMM_NODE monitoring rank
rank24 MPI_COMM_NODE
rank31 MPI_COMM_NODE monitoring rank
MPI barrier COMM_NODE
rank7 starts monitoring
rank15 starts monitoring
rank23 starts monitoring
rank31 starts monitoring
MPI barrier COMM_WORLD
rank0 runs linear system solver part
rank7 runs linear system solver part
rank8 runs linear system solver part
rank15 runs linear system solver part
rank16 runs linear system solver part
rank23 runs linear system solver part
rank24 runs linear system solver part
rank23 runs linear system solver part
MPI barrier COMM_NODE
rank7 stops monitoring
rank15 stops monitoring
rank23 stops monitoring
rank31 stops monitoring
MPI barrier COMM_WORLD
MPI finalize

Figure 3.4: Structure of the MPI program in the common nodes solution

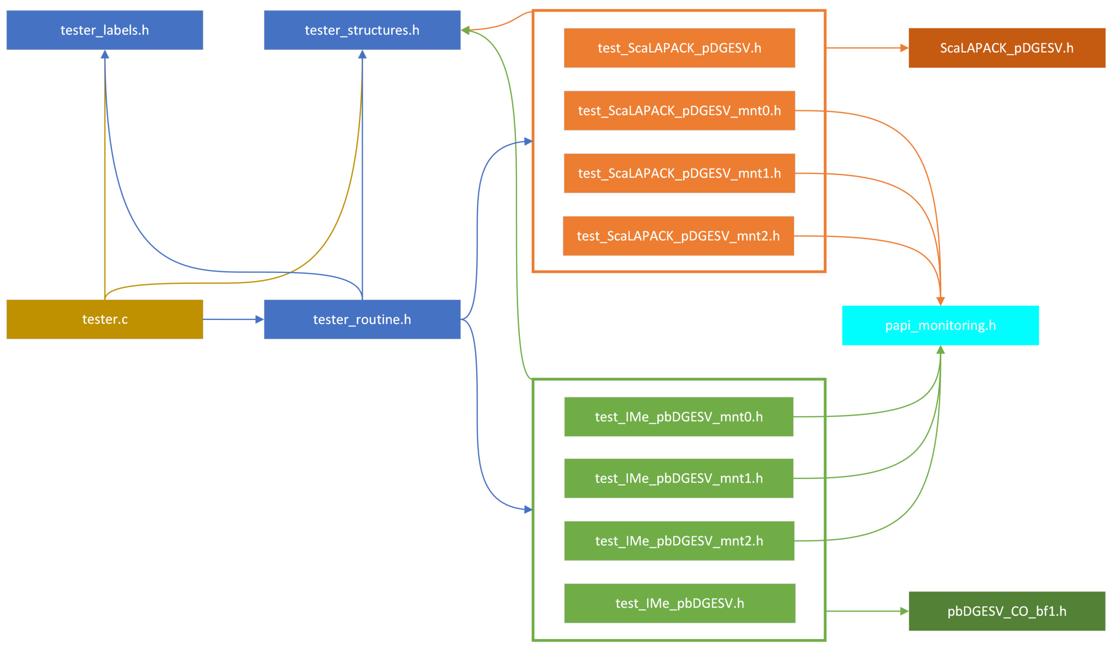

Figure 3.5: Dependencies of the C files

Chapter 4

Configuration and Execution

4.1 Configuration

4.1.1 Machine configuration

To achieve the correct environment for the execution of the tests, at every access to MARCONI via `ssh` it is necessary to load the required modules. As mentioned before the architecture of Marconi A3 is Intel Skylake; therefore, a specific compiler option is needed to generate an optimized code for the architecture and the supported features as AVX-512 instruction sets. This is possible by loading the module env-skl/1.0 and the last available version of the GNU compiler. Since the tests are designed to run on a parallel architecture, it is necessary to load the `openmpi` module in the version openmpi/3.0.0–gnu–7.3.0 with the gnu compiler. Also, PAPI requires its module thus it is loaded with previous ones.

4.1.2 Building LAPACK and BLAS

To compare the performance of IMe and the optimized Gaussian Elimination contained in the ScaLAPACK library, some parts of this opensource library were modified to integrate the fault tolerance with the restart/checkpointing technique into the original algorithm. Since the dependencies within LAPACK SCALAPACK and BLACS are strong, it is necessary to recompile the whole library, even though there is already a general installation of it provided by MARCONI A3. Through a makefile BLAS Basic Linear Algebra Subprograms, and LAPACK Linear Algebra PACKage are recompiled in the 3.9.0 version.

The make file requires to specify the type of compiler used for the tests and the library used for basic linear algebra which is Blaslib as showed in Listing 4.1.

Listing 4.1: Compile script for LAPACK library

```
#!/bin/bash
cd ime-papi-master/ime/src/testers/LAPACK/lapack-3.9.0/
make CC=mpicc FC=mpif77 blaslib
```

4.1.3 Building IMe and ScaLAPACK

After building the libraries it is necessary to compile the whole project, which contains several versions of IMe and ScaLAPACK implemented over the years. This is made with a Makefile compatible with numerous architectures: Cineca Galileo, Marconi and Marconi 100, Enea Cresco and Ubuntu, and many MPI implementations: openmpi, intelmpi, ch mpi, and spectrucm mpi. Moreover, since the math library is widely used, it is possible to specify the source of the libraries: source code, math kernel library or system. The final instruction used to build the project on Marconi A3 architecture is `mpicc`. When compiling the C files, it should be called with some options:

- `-march=skylake-avx512` this allows to have optimized compiled programs for the Skylake architecture;
- `-O3` is the CFLAG optimization strongly recommended by the Cineca user support to compile files with gcc;
- `-g` builds the file in debug mode;
- `--lgfortran` links the GNU compiler for Fortran;
- `-Wall` enables all the warnings generated by the compiler;
- `-Wextra` enables some extra warning flags that are not enabled by `-Wall`;
- `-fPIC` generates position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts, and it avoids any limit on the size of the global offset table;
- `-o` writes the build output to an output file, in this case is will be `$(BIN_DIR)/tester`;
- `-I` adds includes directory of header files `-I$(PAPI_INC)` and `-I$(PAPI_HOME)/share/papi`
- `-L` looks in directory for library files `-L$(PAPI_LIB) -L$(PAPI_HOME)/share/papi/testli`
- `-ltestlib -papi` are used to link the static libraries `papi_test.h` and `papi.h` to the program.

The result of building all those C files should produce the output file tester which allows to execute any kind of IMe or ScaLAPACK version compiled before.

4.2 Execution

4.2.1 Launch tests on MARCONI

Marconi A3 allows the submission of both interactive and batch jobs, but in this work are used batch jobs only. Therefore, the standard error and standard output are saved into a result file. Marconi uses SLURM workload manager for Linux clusters, hence, in order to execute a program in batch mode a script with `sbatch launcher.sh` should be submitted. The script usually contains `sbatch` directive for the execution on Marconi that specify information about the user, the partition, the project linked to the execution, the resources, and the type of usage of the resources. The SLURM script which is launched at each test contains:

- the specification of the project account `try22_Loreti2`;
- the partition of Marconi A3 where the program should be executed which is `skl_usr_prod` for general executions;
- the name of the submitted job `SKL_batch_job`;
- the number of nodes deployed for the execution;
- the number of tasks per node deployed for the execution, the number of tasks times nodes should be the same of the number of MPI ranks;
- the name of the file in which the standard output `job.out` will be printed;
- the name of the file in which the standard error `job.err` will be printed;
- in the option `--gres` there can be specified resources, for the execution `msrsafe` is a plugin module which manages the MSR-SAFE kernel driver and restores the MSR registers in a HPC production environment;
- the option `exclusive` specifies that the job allocation cannot share nodes with other running jobs.
- all the modules are loaded again because the CPUs of Marconi U3 are independent from the login ones. Therefore, since the environment is different is necessary to load PAPI, OpenMPI, GNU and SKL again;

– `srun` is the standard SLURM command to start an MPI program, it should be followed by the name of the executable file and its arguments, which is `tester`;

As far as the parameters for tester are concerned, they are configured as follows:

– the output file is named after the matrix dimension (e.g output_8640.csv) with the `-o` option;

– the number of repetitions `-r` is always set to 10, with the only exception for the executions on 144 ranks for 34.560 and 25.920 matrix dimensions due to the long time spent for their execution. In these cases, `-r` is set to 5;

– `-no-cnd-readback`, `-no-cnd-set`, `-no- nre-readback` are set to avoid unnecessary checks on the input and output matrices;

– the matrix dimension is specified with `-nm` and can be set to 8.640, 17.280, 25.920 or 34.560, which are the tested dimensions;

– the input files are specified with `-i` option, each input file has three different extensions that identify the matrices A, B and x, and they are specific for each matrix dimension.

– `-spk -nb 32` specifies the dimension of the blocks of computations for the ScaLAPACK algorithm;

– the fault tolerance option `-ft` is always set to zero since the versions tested are not fault tolerant;

– the monitoring option `-ma` is set to 0 or 2 depending on the type of monitoring chosen for the execution (0 = only the algorithm execution, 2 = allocation, deallocation and execution of the algorithm);

– the option `-head` allows to insert an execution specification at the beginning of each monitoring file, in order to save also the SLRUM configuration of the current job.

– the option `--run` is set to `IMe-PB-SV-CO-bf1` or to `SPK-SV` depending on the tested algorithm. The script launched at each execution on Marconi U3 is showed in listing Listing 4.2.

Listing 4.2: Batch script submitted at each execution to SLURM

```
#!/bin/bash
#SBATCH -A try22_Loreti2
```

```
#SBATCH --partition=skl_usr_prod
#SBATCH --job-name=SKL_batch_job
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=48
#SBATCH -o job.out
#SBATCH -e job.err
#SBATCH --gres=msrsafe
#SBATCH --exclusive

module load autoload openmpi/3.0.0--gnu--7.3.0 env-skl
   /1.0 papi/6.0.0

srun ./tester -o "output_8640.csv" \
-r 10 -no-cnd-readback -no-cnd-set -no-nre-readback \
-nm 8640 \
-i "/marconi_work/try22_Loreti2/test_matrices/
   rank8640_cnd1_seed1" \
-spk-nb 32 \
-ft 0 \
-ma 0 \
-head "dim_matrix: 8640   ranks: 144   nodes:3   rXn:48
     soc:2" \
--run IMe-PB-SV-CO-bf1
```

Chapter 5

Monitoring IMe and ScaLAPACK

5.1 Parameters of the tests

The monitoring focuses on the measurements of the energy status values. However, there are many combinations of the parameters listed in the next section that will affect the measurements. By combining all the possibilities for each parameter there are 108 different tested configurations.

Optimization level of the compiler The optimization level of the compiler (CFLAG) is set for all the tests at O3, which is the maximum achievable level of optimization. The time spent to perform the compilation and the efficiency of the executable generally increases with the level assigned to CFLAG (-O0, -O1, -O2, -O3). On the other hand, the time spent for the execution of the compiled file decreases significantly.

Matrices allocation The matrices allocation is tested in a contiguous form. This means that the matrix is stored in the RAM in contiguous cells of memory. The advantage of contiguous memory allocation is the increase in the processing speed. As the operating system uses the buffered I/O and reads the process memory blocks consecutively it reduces the head movements. This speed ups the processing.

Characteristics of the input linear systems The input linear system is not generated by the algorithm at run-time, but it is loaded from a file with the `-i` option, since it is not convenient to compute every time matrices of the same dimension. With a fixed input, it is possible to repeat the measurements with the guarantee that the input data is the same. In fact, this experimental approach is highly recommended during the execution of large amount of tests.

There are four different matrices dimensions 8640, 17280, 25920 and 34560 which are multiple between each other's. The different dimensions are used to understand the trend of the energy values with fixed dimensions for the ranks and nodes. As explained in the following section 5.1.2 there is a strong dependency between matrices dimensions and number of ranks.

Algorithms and repetitions As far as the linear system solver algorithms are concerned, the chosen version of IMe for these tests is IMe_pbDGESV_CO_bf1, whilst the chosen one for ScaLAPACK is ScaLAPACK_pDGESV. All the tests are performed at the same conditions for both the algorithms.

Moreover, in order to achieve more realistic values which allows to truly compare the metrics, the `-r` option of `tester` is set to ten. Therefore, for each job scheduled are performed ten repetitions of the execution of the linear systems solver.

5.1.1 Monitored phases: allocation and execution

The algorithm can be differentiated in two parts, the allocation of the matrix and the effective execution. The algorithm is monitored during the execution of the linear system resolution, and in its general execution which comprehends the allocation, the deallocation, and the execution. The monitoring of the allocation phase is avoided because it does not consider the deallocation part. Hence it is preferrable to monitor the whole execution and then produce an esteem of the energy consumption of the allocation and deallocation by computing the gap between the execution of the linear system algorithm and the whole test. Moreover, this approach saves resources and energy that are not strictly necessary for the aim of this work.

5.1.2 Nodes, ranks, sockets

Given a fixed matrix dimension the number of nodes and ranks heavily affects the computation. The less the number of ranks, the higher the load for each of them; therefore, there is a compromise between the time spent for the execution (which increases with the load per rank), and the energy consumed to power more ranks on different nodes. It should be evaluated whether to increase the energy consumption by using more nodes to achieve a faster execution, or instead less nodes for a longer period dealing with a slower execution.

Certainly, if the aim is to save energy it is important to exploit wisely the resources. Thus, it should be assessed if it is more energy consuming to assign exactly 48 ranks per node, or to assign only 24 ranks per processor by doubling the total number of nodes. In the first case the level of parallelism of the processor which is 48 cores corresponds with the number of ranks assigned to the node. Whilst, in the second case only half of the cores are used, and the other half is powered, but it remains idle, because all the executions are performed with the SLURM directive `--exclusive`.

Another aspect which is explored in these tests concerns the sockets inside the processor. As explained in section 2.2, each processor of Marconi A3 has 48 cores and it is divided into 2 sockets of 24 cores each. Therefore, when the algorithms are executed only on 24 cores per processor, the ranks can be distributed in the two sockets with 12 cores each, or it is possible to fill just one socket and keep the other one idle, as showed in 4.2.1.

These tests are concerned especially on the observation of strong scalability to study how the energy consumption varies with the number of processors for a fixed total problem size. For each matrix dimension there are three possible values for ranks, which are 144, 576 and 1296. These values are strictly related to the matrix dimension, and they are not casual. In fact, IMe needs a square number of ranks, so that the matrix dimension is a multiple of the square root of the number of ranks. Moreover, there are limitations to the number ranks used for computation which is affected also from the available resources at CINECA.

Table 5.1 summarizes the configuration of the hardware, by defining the number of processors, the ranks and the sockets included. To compute the smallest number of nodes for a given value of ranks, the ranks should be divided by 48. Then to obtain the number of nodes for 24 cores per processor, the result should be multiplied by two. The computation that requires less processors is the first one with 144 ranks, here only three nodes are involved with 48 ranks assigned each, whilst the most parallelized involves 54 nodes with 24 ranks each.

5.2 Data collection

Since the total number of tests is 144, the total number of files is 8250 and there are many aspects to manage, some helpers were implemented.

5.2.1 Directories hierarchy

All the tests are organized in a hierarchical tree of directories with the same structure for each matrix dimension. Into the leaves are contained a personalized launcher with the correct parameters for the execution and a copy of tester.

Ranks	Nodes	Ranks per Node	Sockets	Ranks x Socket	
144	3	48	2	24	24
	6	24	1	24	0
	6	24	2	12	12
576	12	48	2	24	24
	24	24	1	24	0
	24	24	2	12	12
1296	27	48	2	24	24
	54	24	1	24	0
	54	24	2	12	12

Table 5.1: test configurations for nodes, ranks and sockets

The directories are organized on five levels:

– The first level classifies the matrix dimension (e.g. 8.640, 17.280, 25.920, 34.560);

– The second level states the number of ranks (e.g. 144, 576, 1.296);

– The third level indicates three nodes configuration per number of ranks, the number of nodes changes depending on the ranks (e.g. 3 nodes 48 cores, 6 nodes 24 cores 1 socket, 6 nodes 24 cores 2 sockets);

– The fourth level states the algorithm type which is IMe or ScaLAPACK;

– The fifth level indicates the monitored part of the execution (e.g. execution, general);

This organization was necessary to distinguish the files generated at each execution and to achieve a better management.

5.2.2 Launching the tests and collecting the results

Launcher for multiple jobs Since for each matrix dimension there are 36 tests to execute distributed in the different directories, it was implemented a general launcher script. The script `launch-etor.sh` recursively explores the hierarchy of directories. Whenever it founds a leaf, it changes the `launcher.sh` and `tester` permissions to executable and it executes the `launcher.sh`, which is the SLURM script for the submission of the batch job listed in 4.2. The directory to explore is requested as argument, if it is not specified the script fails.

Elimination of multiple files in case of wrong executions At the begging some executions were interrupted due to time limit. Thus, the text files generated were less than the requested number. This did not allow to proceed with the generation of the CSV, since the number of repetitions did not correspond to the target. It is important that the repetitions of the same test are run on the same processor to grant equal conditions of the environment for the tests. The `elimin-etor.sh` recursively deletes all the text files in the leaves. As argument is should be stated the directory in which is necessary to delete the files, since the script is recursive directories of any level are accepted.

Elimination of batch jobs in case of wrong executions For the same reasons listed above, it was also necessary to delete large amounts of batch jobs. Hence, it was implemented a bash script called `s-scancel-etor.sh` that deletes all the jobs in queue. This script does not need any argument.

Update tester version Since the versions of IMe and ScaLAPACK are periodically updated it is necessary to change the tester version in the leaves directories. In fact, to reach more modularity and avoid relative paths problems each leaf contains the tester program to run IMe and ScaLAPACK. Therefore, it was implemented `copy-etor.sh` which is a script for distributing the new versions of tester to all the hierarchy. This script takes as arguments the file to distribute, and the directory tree where to copy it `./copy-etor.sh <dir/file> <path>`.

5.2.3 Composition of multiple CSV files

Generator of multiple CSV files When a job is run it generates one file for each processor at each execution. Hence, the maximum number of text files generated for one job can reach 540. The automation of the extraction of the relevant data from these files is fundamental. To carry out this job, it was implemented a text file parser and a bash script. The bash script recursively runs the parser on each leaf, and it also creates the right file name by extracting the information from the file path. The parser should be called with the file name and the path where to find the files as arguments. Since the parser is written in Kotlin[35] it should be run as `.jar` file, thus it needs the `java -jar` command to execute. The arguments requested from `csv-etor.sh` to create the CVSs files is `./csv-etor.sh <directory> <command>`, the directory represent the place where to search for the text files, and as command it should be specified `java -jar ParserTxtCsv.jar`.

Parser of the text files The parser of the text files has two tasks. The first is to read and parse each file in the stated directory, the second task consists in the CSV writing.

The structure of the data is organized into three classes. The `Data` class, describes two parameters: a key which is the name of the powercap event, and a float value, which is the value of the powercap event. The `Repetition` class describes all the columns of the CSV output file, which are the time duration, the repetition number, and the list of the powercap events as Data objects. The `Processor` class describes all the repetitions made on one processor; therefore, its parameters are the name of the processor and a list of `Repetions` objects.

Given this structure the `TxtReader` class reads all the lines of the file and loads them on the map of `processor_name` and `Processor` objects, this map is called `powercapDB` and each entry represents one file. Each file contains three header lines and then the powercap events list. The first line is saved as comment because it contains the SLURM parameters, from the second line are saved the processor name and the current repetition, and from the third line it is extracted the duration of the execution. From all the other lines it is extracted the powercap event name and its value which is converted in `float`.

Once the map `powercapDB` is filled with all the files into the directory. Every `Processor` object in the map is written on the CSV file through the class `CsvWriter`. The `CsvWriter`, iterated on the `Processor` objects, writes for each of them in the first line the processor name, then it creates one column for each event name contained in the `Data` list and it fills the following rows with the powercap values of the 5 or 10 repetitions. The class iterates this for all the processors, and it closes the file. Due to some problems with the settings of the decimals on Excel it also replaces the period divider with comma.

The `Main` class checks the input directory; it opens it and executes the `readData` method from class `TxtReader` on all the files found in the subdirectories. After `powercapDB` map is filled, the `Main` class calls the method `writeData` from `CsvWriter` that writes the CSV file, and it terminates.

5.2.4 Data aggregation from Comma-Separated Files to Excel

The result of the parsing phase is a CSV file which contains data from all the execution with the same configuration. This means that for each processor there are five or ten rows, with the values of all the powercap events as columns. Clearly, it is essential that all data needs to be aggregated. Thus, with Visual Basic programming language is created an Excel macro which is transformed in Add-ins to run it on whatever CSV file. As matter of facts, since the number of nodes also determines the number of rows, there is one macro and one Add-ins for each node dimension (3, 6, 12, 24, 27, 54). The macro executes the following steps:

– for the first processor it computes the median and the maximum of the time duration, the average of the power and energy consumption of package 0, of the power and energy consumption of package 1, of the power and energy consumption of DRAM 0, and of the power and energy consumption of DRAM 1 ;

– it copies the functions above at the end of each block of rows correspondent to one processor, this will be iterated as many times as the number of nodes minus one;

– at the end of the files it computes the maximum duration between all the nodes, the median of the median of the durations, and the sum of the averages for the power consumption of package 0, package 1, DRAM 0 and DRAM 1.

– it computes the sum of the values obtained for the total power and energy consumption of package 0, package 1, DRAM 0 and DRAM 1. In this way, it is obtained the total power consumption for one execution of the algorithm with the set configuration.

The Add-ins is an Excel construct generated from an Excel macro which can be added to the menu-bar of the Excel application, and that allows to execute the linked macro in any kind of Excel file. In this way, to obtain the total values it is simply necessary to open the CSV file, click the correspondent Add-in button in the Excel menu and all the required values are computed.

The values obtained from each CSV file are aggregated in four more readable files correspondent to the tested matrix dimensions. From here, it is possible to extract all the graphs and the comments of the results.

5.3 Significant charts

The aim of this section is to emphasize the trends of energy consumption and duration for the different configurations, and to remark the differences between IMe and ScaLAPACK. Since the results did not show noticeable differences between the monitored phases, the graphs only contain the general phase values of IMe and

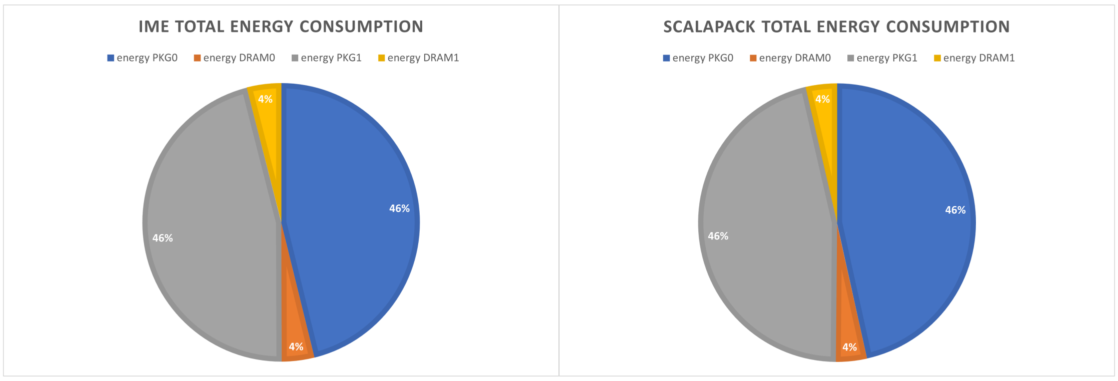

Figure 5.1: IMe and ScaLAPACK breakdown on the total energy consumption

ScaLAPACK. Due to the large amount of the different tests and configurations, the detailed description of them is showed in appendix A. Here is showed a more significant data aggregation with combined charts.

Total energy consumption breakdown The charts 5.1 shows the total energy consumption breakdown for IMe and ScaLAPACK. The percentage values are very similar for all the executions, this means that the number of CPU intensive instructions and the number I/O intensive instructions are strictly related. This bond which is given from `tester` is respected through all the computation and there are not anomalies. The DRAMs consumes a 8% of the computation, the remaining part is equally divided between package 0 and package 1.

IMe vs ScaLAPACK full loaded processors and half loaded processors The charts displayed at 5.2 show the behaviour of energy consumption in the full and half loaded processor. The full loaded processor involves all the 48 cores into the computation, each of them is assigned to one core. Whereas, the half loaded is deployed in two different ways: in one case one socket is full with 24 cores running 24 ranks and the other socket is empty, in the other case there are 12 ranks per sockets. From this graph is possible to note the difference in terms of energy consumption between the three configurations for IMe and ScaLAPACK. The full load configuration always consumes less that the other ones. Moreover, there are only slight differences between the configuration that deploys 24 cores on one socket or the one that distributes 24 cores on two sockets. In fact, the lines overlap multiple times and for both IMe and ScaLAPACK, thus is impossible to determine the best one.

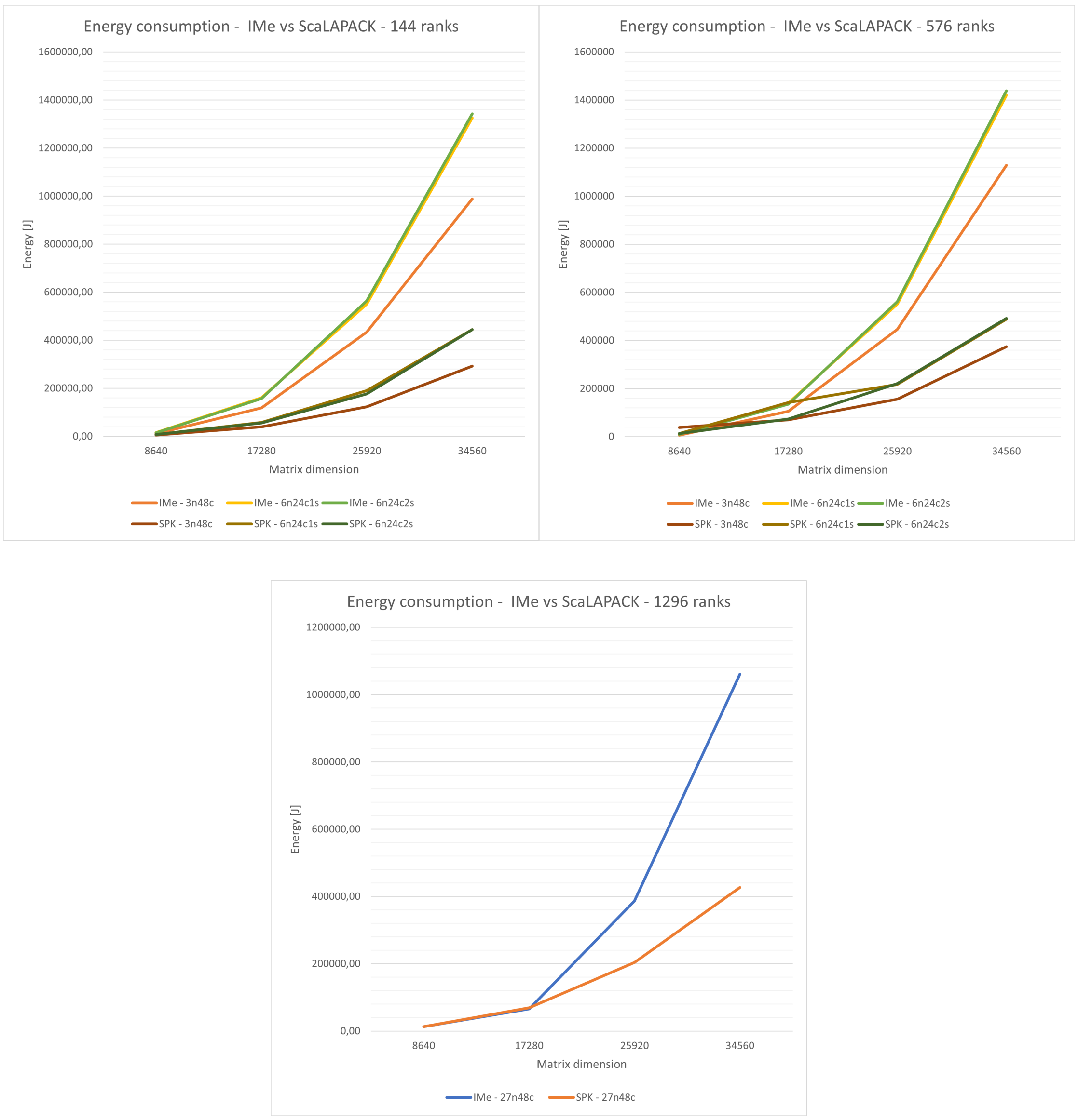

Figure 5.2: Comparison between full loaded processors and half loaded processors

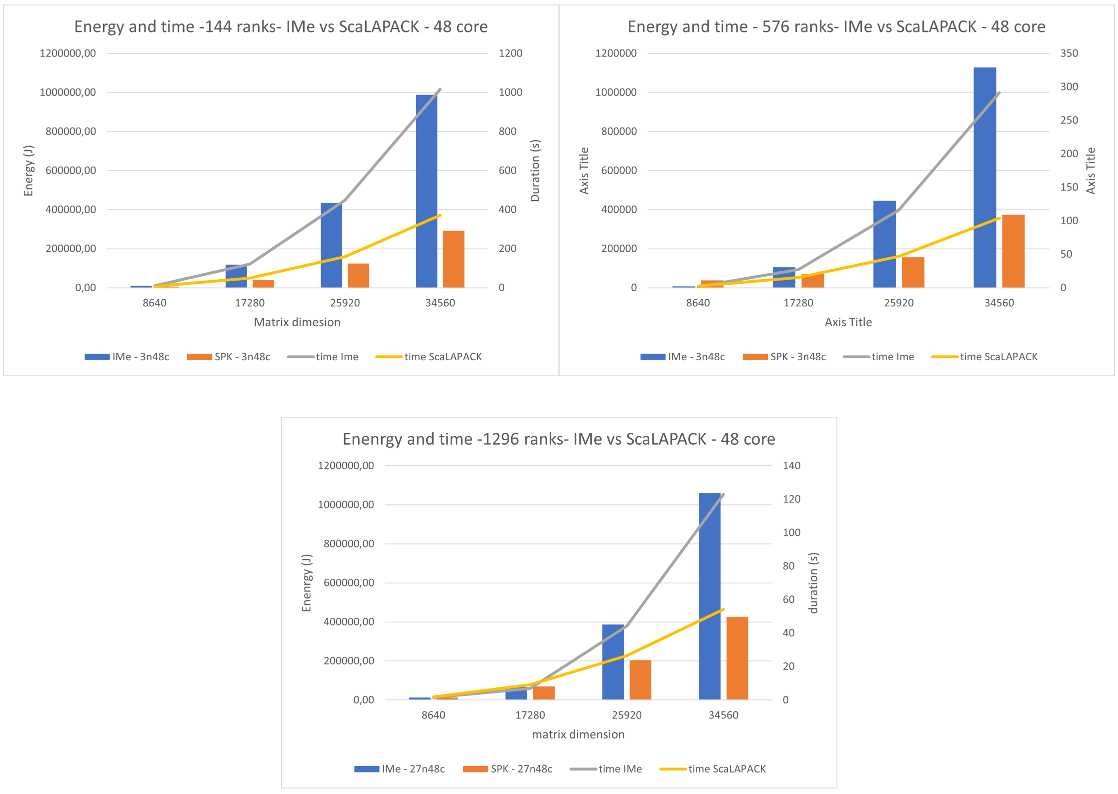

Figure 5.3: IMe and ScaLAPACK energy and time at fixed ranks size

Total energy consumption and time duration for fixed ranks sizes The total energy consumption and the duration of the execution increase with the dimension of the input matrix. In particular, in these charts there are the values obtained for the 48 cores deployments on 3 nodes, 12 nodes and 27 nodes. It is evident that the energy consumption of IMe is always equal or higher than ScaLAPACK. The trend of the energy consumption seems to be exponential, it increases faster with the linear progression of the matrix dimensions, for both the algorithms. From these charts 5.3 is possible to note the dependency between the energy consumption and the duration, that clearly follows the same course for all the ranks deployments.

Total Energy consumption and time for different matrices dimensions These charts 5.4 compares the power consumption of the three different nodes configurations for a given matrix size. In particular, in these charts there are the values obtained for the 48 cores deployments on 3 nodes, 12 nodes and 27 nodes. In this case there is not a clear pattern for the energy values. On one hand, ScaLA-

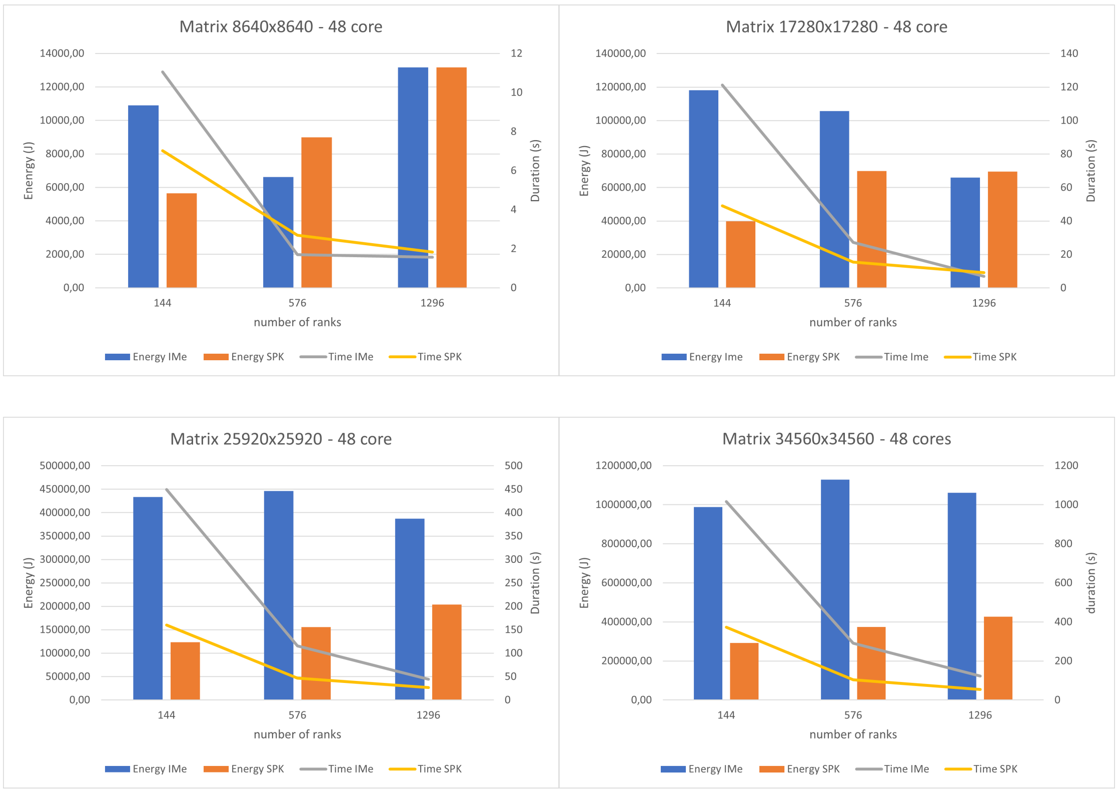

Figure 5.4: IMe and ScaLAPACK energy and time at fixed matrix size

PACK tend to assume a linear trend, whereas IMe values do not follows a specific trend. However, these charts clearly display the strong scalability behaviour of the problem for both the algorithms. In fact, the time duration decreases with the increase of the number of ranks on which is deployed the algorithm. The course of the duration is inversely proportional. As far as a comparison between IMe and ScaLAPACK are concerned, it is clear that ScaLAPACK is faster in the more dense computations, whilst IMe is faster than ScaLAPACK in more distributed computations, like for 576 an 1296 ranks for matrix dimensions 8640 and 17280.

Total energy consumption and power for different ranks These charts 5.5 compare the power consumption and the energy consumption of the three different ranks configurations, by varing the matrix dimension. Since the power consumption is obtained by dividing the energy in Joules, with the duration of the execution, the result is a constant almost horizontal line between the various matrices sizes. As matter of facts, power values, represented by the lines, reveal the actual difference between IMe and ScaLAPACK per each second. With reference to

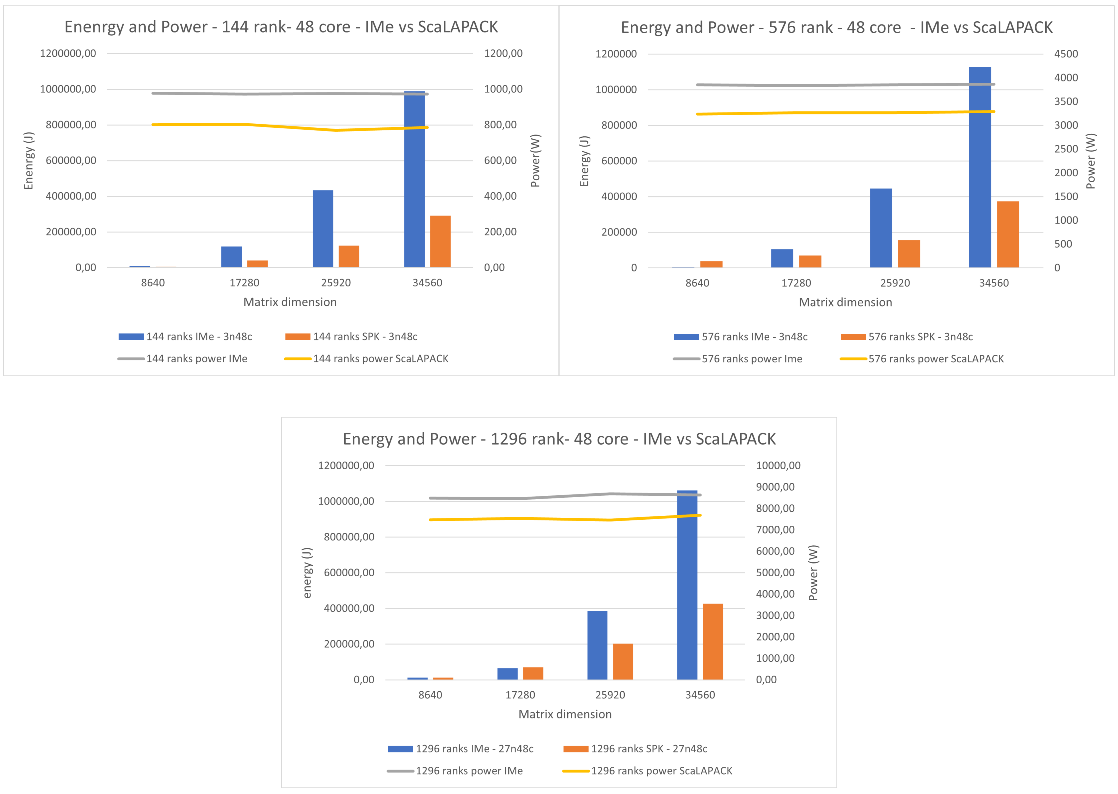

Figure 5.5: Energy and power consumption of IMe and ScaLAPACK at fixed ranks size

the values of the secondary vertical axis the power values of IMe and ScaLAPACK differs of the 12% 18%.

Total energy consumption and power for different matrices dimensions The charts in 5.6 shows the trend of the energy and power consumption by varing the number of ranks for a fixed matrix dimension. The energy consumption in this case do not show a clear trend. However it is clear the dependency of power from the deployed number of ranks. The values of power consumptions of IMe and ScaLApack are similar for the different ranks values and strongly follows a directly proportional course. Hence, it can be noticed that the power values enhance the real trend of energy consumption.

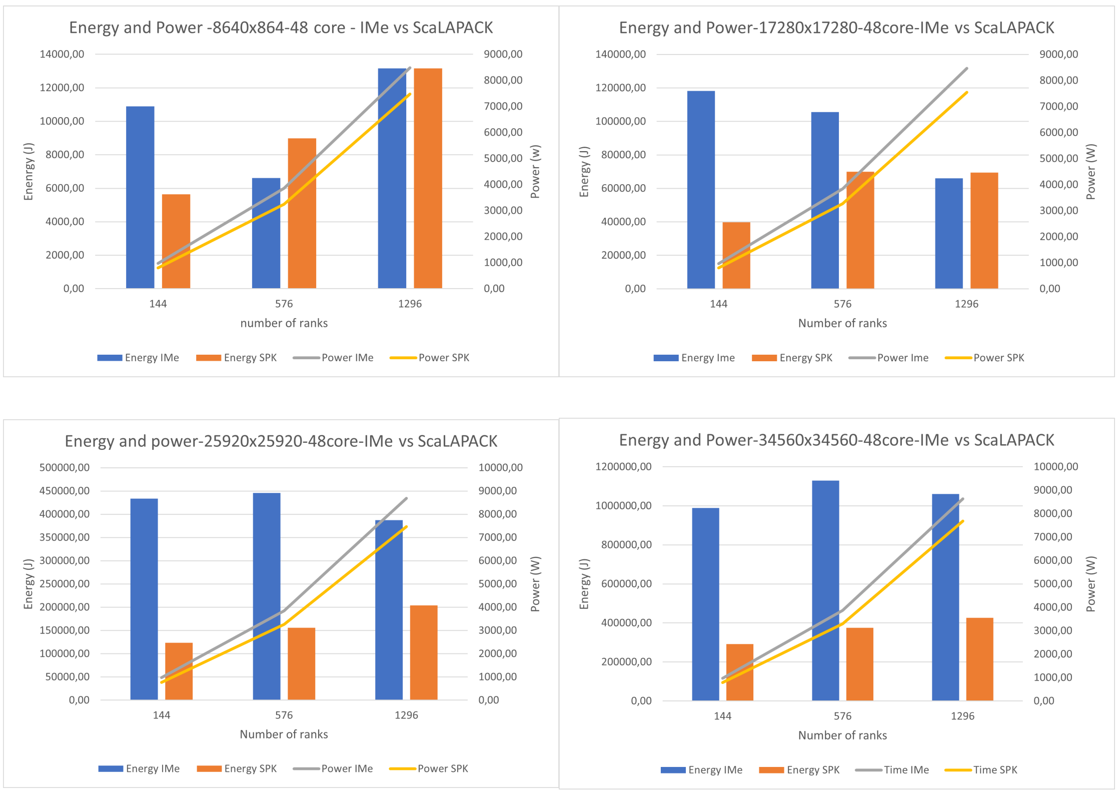

Figure 5.6: Energy and power consumption of IMe and ScaLAPACK at fixed matrix size

5.4 Concluding remarks

5.4.1 General observations

The data about the general execution and only the computation part of the algorithm does not show remarkable difference. As matter of facts, in some cases the execution of the mere algorithm is even more energy consuming than the whole execution. This is probably caused by the fact that the processors on which it is executed varies at each execution. Thus, this does not grant high precision of the measurements for this kind of comparison. To achieve more precise measurements, it could have been helpful to work always on the same nodes, and to control also other parameters of the surrounding environment. Furthermore, since the allocation and deallocation of the matrices especially impacts the DRAMs, the expected result was a significant difference between the DRAMs energy values of the general monitoring and of the computational phase. However, this trend is not detectable from the measurements.

The computations on 48 cores saves more energy if compared to the 24 cores per node execution. Furthermore, the expected behaviour in the 48 cores per node deployments was a considerable increase in the time spent for the computation due to the interactions of multiple tasks on the same resources. As matter of facts, this behaviour can be detected only from the time durations of IMe in the matrix 34560x34560 in the 144 ranks deployment. In the other scenarios, data does not confirm this aspect, sometimes the values of the 48 cores computations are more similar to the 24 cores deployments. However, in most of the cases working with a full load node saves more energy than an half load per node deployment.

Another aspect which was not expected is the behaviour of the deployments on one socket or two sockets for the 24 cores solutions. The two values tend to be very similar, and the variations are meaningless. Since the one socket deployment uses just one package of the processor, the energy consumption of the second one was expected to be low. However, this does not happen, instead the value of one socket is 50-60% lower than the other one. The trend is confirmed also for the two sockets deployments, the energy measurement of package 0 always doubles package 1. This was unforeseen and it raises some doubts about the effectiveness of the Slurm directives. In fact, the energy values for package 0 and package 1 in the 12 cores per socket solution were expected to be similar and way less than the energy consumption of package 0 in the 1 socket deployment.

5.4.2 Summary comparison between IMe and ScaLAPACK

As far as the duration is concerned, the values for IMe and ScaLAPACK strongly depends on the number of ranks. Whenever the matrix is distributed above many ranks, and each of them receives only a small part of the problem, IMe has better performances if compared to ScaLAPACK. Instead, if each task on each rank has a greater dimension ScaLAPACK clearly prevails.

As far as the total energy consumption is concerned, ScaLAPACK saves more energy than IMe. There are only some exceptions, in which the values are very similar. However, the main trend is a gap of 50% to 60% between the two algorithms. The gap tends to close with the increase in the number of ranks and the decrease of the matrix dimension, but the evidence is that ScaLAPACK is less energy consuming than IMe.

Even if the main contribution to the total power consumption comes from the power consumed from package 0 and package 1, it also should be noted that the gap between the power consumption of the DRAMs is even higher than 12-18%. ScaLAPACK seems to better respond to less ranks deployments. Whenever, the ranks are set to 144 the gap between the power consumption of the DRAMs of IMe and ScaLAPACK reaches 42%. In the 24 cores deployments both for one socket and two sockets the gap is around 33%.

Furthermore, both the algorithms seem to be affected from a significant gap between package 0 and package 1 in any configuration that involves a 24 cores deployment. Whereas this behaviour impacts ScaLAPACK only on packages, for IMe it also involves the DRAMs. Nonetheless, the difference between the energy consumption of package 0 and package 1 is very similar for both the algorithms. In fact, it is around 50%.

The energy consumption of IMe is clearly higher than the energy consumption of ScaLAPACK, but this is also due to the longer durantions of the executions for IMe. Therefore, it was computed also the power consumption, that actually confirms the gap between the two. However, the gap decreases and for the power consumption it is around 12-18%.

Chapter 6

Conclusions

The current research aimed to monitor the energy consumption during the execution on HPC systems of two linear systems solver algorithms: the Inhibition Method and ScaLAPACK. The central questions for this work were to determine which algorithm best behave in terms of energy consumption, and to explore in what ways the number of ranks, nodes and sockets impacts the energy consumption.

In the first place, two different strategies were implemented to obtain the measurements: the naïve solution and the common node solution. The second one resulted to be more fine-grained, and more adaptable to the requirements. Thus, it was integrated with the two analysed algorithms IMe and ScaLAPACK, accordingly to a white box approach that allowed the monitoring of the different phases of the algorithm.

To monitor the execution, `tester` was adapted to add the dependencies to PAPI library that can monitor the RAPL and powercap event set. The implementation of the monitoring led to revision of the `tester` command line interface, in which were included two more options. The first one to declare if the monitoring is requested and which phase should be monitored. The second one concerns the header of the text files produced during the monitoring. Since the deployment with a full loaded processor (48 core), rather than the half loaded processor (24 cores distributed on one socket or on two sockets) cannot be inferred from `tester` program, the header option was used to report these configurations on the monitoring output files.

The following phase regarded the identification and evaluation of the tested parameters (number of ranks, number of nodes, number of cores per socket and per node, phases of the algorithm, and the versions of the algorithms). After the declaration and preparation of the environment for all the 144 different configurations of the tests, the monitoring phase got started. The data was gathered through the use of some helpers to automate the process, especially through a text file parser and some Visual Basic macros.

The results showed some unexpected behaviours. The data about the just the execution phase and the whole algorithm with the allocation and deallocation of the matrices did not show substantial differences in terms of energy consumption. The full loaded processors deployments are always significantly less energy consuming than the half loaded ones, and this is evident especially for the DRAMs energy values. Instead, the distribution of the ranks on the half loaded processor on one socket or two sockets did not show remarkable patterns, however, the first socket is always more energy consuming than the second one. As far as the duration is concerned, the full load deployment takes less time than the half load one for the smallest matrices. However, this is not true for the largest matrices in which the time spent for the computation of the full load deployment is higher than half loaded ones. This is probably due to numerous interactions on the same data that slow the execution and give advantage to the more distributed deployments.

ScaLAPACK consumes less energy than IMe in most of the scenarios. IMe performances for time and energy seem to suffer more in lower ranks deployments. In terms of total energy consumption ScaLAPACK uses less energy and the gap with IMe is wide due to longer executions. However, by comparing the power consumption this gap between the two decrease.

This research clearly illustrates the differences between IMe and ScaLAPACK and the results of the various configurations. Nonetheless, it should be questioned the repeatability of these experiments. Indeed, these tests inevitably depends from the architecture and from the nodes in which they are deployed which changed at every test. Hence, on one hand the substantial differences and the clear trends can be considered an evidence, whilst on the other hand the values that show mild differences are not reliable and they should be investigated in further researches.

6.1 Future developments

The monitoring phase of this thesis discovers the energy consumptions of the two algorithms. The following step is the application of power caps to limit the power consumption during the execution, in order to reach more efficient computations, and to investigate the behaviour of IMe and ScaLAPACK with different power configurations.

Furthermore, it should be explored the precision of the energy values contained in the MSR registers, which are read by RAPL, powercap and PAPI. Indeed as Fahad et. al. [15] have showed, the precision of those measurements might be significantly inaccurate when compared with the energy measurements provided by external power meters.

A further development involves the implementation of the nodes release for IMe. In fact, as explained in appendix A, the nodes which run the ranks with the higher code finish before the other ones, due to an unbalanced distribution of the computation [7]. In order to obtain more accurate measurements and to increase the energy efficiency of the algorithm itself, a future implementation of IMe can introduce the nodes release.

Bibliography

[1] High performance computing. https://www.energy.gov/science/high-performance-computing.

[2] Top500. https://www.top500.org/lists/top500/.

[3] Iso 50001 - energy management. https://www.iso.org/iso-50001-energy-management.htm, Dec 2020.

[4] Alessandro Gimigliano Alessandra Bernardi. *Algebra lineare e geometria analitica, Città Studi, 2018.* Città Studi Elezioni, 2018.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[6] Marcello Artioli and F. Filippetti. Ime: a general method to analyse linear systems and electric circuits. In *Transactions on Engineering Sciences vol 31, WIT Press, 2001*, 2001.

[7] Marcello Artioli, Daniela Loreti, and Anna Ciampolini. Fault tolerant high performance solver for linear equation systems. *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019.

[8] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[9] Jesus Carretero, Salvatore Distefano, Dana Petcu, Daniel Pop, Thomas Rauber, Gudula Rünger, and David Singh. Energy-efficient algorithms for ultrascale systems. *Supercomputing Frontiers and Innovations*, 2(2), 2015.

[10] Jesus Carrettero, salvatore Distefano, diana petcu, daniel pop, thomas rauber, gudula Rünger, and David E. Singh. Energy-efficient algorithms for ultrascale systems. *Supercomputing Frontiers and Innovations*, 2(2), 2015.

[11] Filippo Ciampolini. Un metodo di soluzione dei circuiti lineari,". In *L'Elettrotecnica, vol. L, no. 10, 1963*, 1063.

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation, August 2007.

[13] Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. Energy-aware high-performance computing: Survey of state-of-the-art tools, techniques, and environments. *Scientific Programming*, 2019:1–19, 2019.

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, 2011.

[15] Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. A comparative study of methods for measurement of energy of computing. *Energies*, 12(11), 2019.

[16] Joseph F. Grcar. How ordinary elimination became gaussian elimination. *Historia Mathematica*, 38(2):163–218, 2011.

[17] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[18] Azzam Haidar, Heike Jagode, Phil Vaccaro, Asim YarKhan, Stanimire Tomov, and Jack Dongarra. Investigating power capping toward energy-efficient scientific applications. *Concurrency and Computation: Practice and Experience*, 31(6), 2018.

[19] Joel Hruska. The death of cpu scaling: From one core to many - and why we're still stuck - page 3 of 3. `https://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-s` 3, Jun 2012.

[20] Connor Imes and Alex Shpilkin. Power capping framework. `https://www.kernel.org/doc/html/latest/power/powercap/powercap.html`, 2017.

[21] Connor Imes and Alex Shpilkin. Powercap/powercap: C bindings to the linux power capping framework in sysfs. `https://github.com/powercap/powercap`, 2017.

[22] Heike Jagode, Asim YarKhan, Anthony Danalis, and Jack Dongarra. Power management and event verification in papi. *Tools for High Performance Computing 2015*, page 41–51, 2016.

[23] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 3(2):1–26, 2018.

[24] M. McFadden, K. Shoga, and B. Rountree. Llnl/msr-safe: Allows safer access to model specific registers (msrs). `https://github.com/LLNL/msr-safe`, 2015.

[25] Carl Dean Meyer. *Matrix analysis and applied linear algebra.* SIAM, 2000.

[26] Phil Mucci, Daniel Barry, Guiseppe Congiu, Anthony Danalis, Jack Dongarra, Heike Jagode, and Anustuv Pal. High level api. `https://bitbucket.org/icl/papi/wiki/PAPI-HL`, 2018.

[27] Phil Mucci, Daniel Barry, Guiseppe Congiu, Anthony Danalis, Jack Dongarra, Heike Jagode, and Anustuv Pal. Low level api. `https://bitbucket.org/icl/papi/wiki/PAPI-LL`, 2018.

[28] Phil Mucci, Daniel Barry, Guiseppe Congiu, Anthony Danalis, Jack Dongarra, Heike Jagode, and Anustuv Pal. Powercap component. `https://bitbucket.org/icl/papi/src/master/src/components/powercap/`, 2018.

[29] Phil Mucci, Daniel Barry, Guiseppe Congiu, Anthony Danalis, Jack Dongarra, Heike Jagode, and Anustuv Pal. Rapl component. `https://bitbucket.org/icl/papi/src/master/src/components/rapl/`, 2018.

[30] Phil Mucci, Daniel Barry, Guiseppe Congiu, Anthony Danalis, Jack Dongarra, Heike Jagode, and Anustuv Pal. Papi parallel programs. `https://bitbucket.org/icl/papi/wiki/PAPI-Parallel-Programs`, Feb 2020.

[31] Anne-Cecile Orgerie, Marcos Dias Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys*, 46(4):1–31, 2014.

[32] J.S. Plank, Kai Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[33] Elda Rossi. Ug3.1: Marconi userguide. `https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide`, 2016.

[34] Mário Santos, João Saraiva, Zoltán Porkoláb, and Dániel Krupp. Energy consumption measurement of c/c++ programs using clang tooling. In *SQAMIA*, 2017.

[35] Maxim Shafirov, Roman Elizarov, Grace Kloba, Jeffrey van Gogh, and Werner Dietl. Get started with kotlin/native in intellij idea: Kotlin. `https://kotlinlang.org/docs/home.html`, 2012.

[36] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. *Tools for High Performance Computing 2009*, page 157–173, 2010.

[37] S. Walker, K. Shoga, B. Rountree, and L Morita. Llnl/libmsr: Wrapper library for model-specific registers. apis cover rapl, performance counters, clocks and turbo. `https://github.com/LLNL/libmsr`, 2015.

[38] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. HaPPy: Hyperthread-aware power profiling dynamically. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 211–217, Philadelphia, PA, June 2014. USENIX Association.

Appendix A

Results in detail

All the energy values showed in this section were subtracted from the mean baseline energy consumption in order to have more truthful and accurate values. Furthermore, the IMe structure is affected from an unbalanced time of computation between the different ranks and nodes. In fact, the node which contains the ranks with the higher code finishes the computation after $time \times \frac{total_nodes-node_number}{total_nodes}$ of the total time spent for the computation. Therefore, after the computation this node remains idle. Since the release of the node at the end of the computation is still not implemented it can be estimated the real energy consumption by subtracting the energy consumption that corresponds to the idle time of the node.

A.1 Results for matrix 8640 x 8640

A.1.1 Deployed ranks: 144

This section refers to tables A.1 and A.2.

Phases of the algorithm As far as IMe and ScaLAPACK algorithms are concerned, there are small differences in terms of power between the measurements of the whole execution and just the algorithm part. The fact that the allocation is almost irrelevant can be inferred from the power values which are very similar. As matter of facts, in some cases, maybe due to the different processor that runs them, the power value for the execution of the algorithm is even higher than the general execution. The only relevant difference can be seen in the execution that involves 6 nodes with 24 ranks each and 1 socket, in that case ScaLAPACK the power consumption of the execution is 1,1% higher than the general monitoring.

matrix 8640		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	4999,427	9708,715	10031
		energy DRAM0	389,4627	742,0743	748,3722
		energy PKG1	4967,996	4458,927	4493,697
		energy DRAM1	408,5181	428,9076	458,9856
		power PKG0	452,8225	871,7673	882,1338
		power DRAM0	35,2835	66,6388	65,8813
		power PKG1	449,9865	400,6329	396,0141
		power DRAM1	37,011	38,5121	40,3418
		MAX duration (s)	11,4154	11,4808	12,7119
		MED duration (s)	10,94445	11,13138	11,08338
	General	energy PKG0	5072,6	9966,221	7908,229
		energy DRAM0	395,556	749,7272	618,2003
		energy PKG1	5043,2	4232,807	5154,447
		energy DRAM1	392,8134	427,3212	466,6834
		power PKG0	454,7698	893,2041	782,3499
		power DRAM0	35,468	67,2326	61,1516
		power PKG1	452,0943	379,6779	509,6863
		power DRAM1	35,222	38,2925	46,1396
		MAX duration (s)	11,645	11,895	10,4303
		MED duration (s)	11,0619	11,08118	10,06055
ScaLAPACK	Execution	energy PKG0	2689,834	4705,798	5008,216
		energy DRAM0	109,3605	256,7688	254,8007
		energy PKG1	2668,829	3085,045	2779,599
		energy DRAM1	143,2606	285,4054	254,336
		power PKG0	382,975	666,3411	703,6859
		power DRAM0	15,5703	36,3529	35,7981
		power PKG1	379,9833	436,6563	390,4813
		power DRAM1	20,3967	40,4055	35,7321
		MAX duration (s)	7,1422	7,2459	7,2593
		MED duration (s)	7,0129	7,03035	7,106325
	General	energy PKG0	2738,564	5284,993	5323,561
		energy DRAM0	145,0118	256,849	252,6403
		energy PKG1	2615,088	2492,14	2492,003
		energy DRAM1	146,6082	250,9515	269,0657
		power PKG0	389,3057	743,3531	750,305
		power DRAM0	20,6145	36,1283	35,6079
		power PKG1	371,7512	350,6072	351,2693
		power DRAM1	20,8413	35,2981	37,9221
		MAX duration (s)	7,1374	7,2292	7,2045
		MED duration (s)	7,0182	7,08415	7,083425

Table A.1: table for 8640x8640 matrix with 144 ranks partial results

matrix 8640		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	10765,40	15338,62	15732,05
		total power	975,1035	1377,551	1384,371
		MED duration (s)	10,94445	11,13138	11,08338
	general	total energy	10904,17	15376,08	14147,56
		total power	977,55	1378,41	1399,33
		MED duration (s)	11,0619	11,08118	10,06055
ScaLAPACK	execution	total energy	5611,28	8333,02	8296,95
		total power	798,9253	1179,756	1165,697
		MED duration (s)	7,0129	7,03035	7,106325
	general	total energy	5645,27	8284,93	8337,27
		total power	802,51	1165,39	1175,10
		MED duration (s)	7,0182	7,08415	7,083425

Table A.2: table for 8640x8640 matrix with 144 ranks total results

Cores deployed per processor The cores deployed per processor evidently impact the power consumption. The deployment of 48 cores per processor above 3 nodes, is the best solution in terms of energy saving, because for both the algorithms it has always the lowest power consumption, if compared with the 6 nodes deployment. The gap between the two is around 30% depending on the monitored phases and on the algorithm. In particular, ScaLAPACK is the algorithm which best highlights this difference. From data It seems that there is also a slight difference between the deployment of 24 cores on two sockets or just on one socket. In general, the deployments above two sockets are more energy consuming. The energy consumption of the DRAM doubles with the employment of 6 nodes instead of 3 nodes. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes, this is not impressive since having more nodes means also to have a more distributed data and more I/O instructions.

IMe versus ScaLAPACK The performances of ScaLAPACK are better than IMe in whatever test, however in some cases there are only slight differences. In all the executions the energy consumption of the DRAM 0 is almost the same and the maximum difference between IMe and ScaLAPACK of the total power consumption is 18%. In particular, the executions on 3 nodes show a sharp difference between the two linear system algorithms. As far as the duration of the execution is concerned, both the versions amount to a constant value, which is around 11 seconds for IMe and 7 seconds for ScaLAPACK.

A.1.2 Deployed ranks: 576

The deployment on 12 nodes and 24 nodes is approximately 4 times more power consuming if compared to the deployment on 3 and 6 nodes, this data confirms a linear trend of power consumption that follows the number of processors involved. This section refers to tables A.3 and A.4.

Phases of the algorithm Also, in this case the measurements of the two phases of the algorithms are not meaningful, there is always a slight difference between the two, and in some cases the values of the general monitoring are less than the algorithm execution itself.

Cores deployed per processor Also, this test confirms that the cores deployed per processor evidently impact the power consumption. The deployment of 24 cores per processor above 24 nodes has always the highest power consumption if compared with the 12 nodes deployment. The gap between the two reaches the 30% depending on the monitored phases and on the algorithm. ScaLAPACK is

matrix 8640		576 rank	12 nodes 48 ranks	24 nodes 24 ranks 1 socket	24 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	3207,029	5946,363	6259,775
		energy DRAM0	122,4757	232,7866	244,8755
		energy PKG1	3143,572	2679,299	2751,949
		energy DRAM1	137,7732	256,9251	276,6214
		power PKG0	1883,844	3527,825	3514,735
		power DRAM0	71,9439	138,1064	137,4853
		power PKG1	1846,552	1591,76	1545,722
		power DRAM1	80,9198	152,4047	155,3
		MAX duration (s)	1,9391	2,0164	2,0729
		MED duration (s)	1,667325	1,6456	1,757325
	General	energy PKG0	3213,42	5879,71	6299,929
		energy DRAM0	121,4517	237,5042	239,1561
		energy PKG1	3145,057	2981,004	2510,66
		energy DRAM1	134,2734	268,0601	272,8937
		power PKG0	1874,108	3404,153	3592,123
		power DRAM0	70,8315	137,4215	136,3663
		power PKG1	1834,249	1724,495	1432,076
		power DRAM1	78,301	155,0784	155,5826
		MAX duration (s)	1,9547	2,0737	2,0564
		MED duration (s)	1,6924	1,69445	1,693175
ScaLAPACK	Execution	energy PKG0	18393,38	8546,071	8528,592
		energy DRAM0	686,686	368,8279	365,3599
		energy PKG1	18256,88	3914,146	3743,026
		energy DRAM1	844,6595	427,8914	423,2024
		power PKG0	1707,253	3080,171	3103,439
		power DRAM0	65,9069	132,929	132,9535
		power PKG1	1689,281	1411,027	1362,339
		power DRAM1	80,866	154,2167	154,0021
		MAX duration (s)	17,2139	2,9877	2,9357
		MED duration (s)	14,51673	2,75375	2,726625
	General	energy PKG0	4345,022	8288,457	8288,12
		energy DRAM0	185,5889	358,2907	362,9113
		energy PKG1	4242,234	3666,391	3835,852
		energy DRAM1	213,919	415,2588	420,8571
		power PKG0	1565,626	3062,866	3021,974
		power DRAM0	66,9042	132,4091	132,3233
		power PKG1	1528,589	1355,301	1398,641
		power DRAM1	77,1122	153,4636	153,4433
		MAX duration (s)	3,5348	2,9142	2,9732
		MED duration (s)	2,691775	2,68415	2,7277

Table A.3: table for 8640x8640 matrix with 576 ranks partial results

matrix 8640		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	6610,85	9115,374	9533,22
		total power	3883,26	5410,096	5353,242
		MED duration (s)	1,667325	1,6456	1,757325
	general	total energy	6614,202	9366,278	9322,64
		total power	3857,49	5421,147	5316,15
		MED duration (s)	1,6924	1,69445	1,693175
ScaLAPACK	execution	total energy	38181,6	13256,94	13060,18
		total power	3543,307	4778,343	4752,733
		MED duration (s)	14,51673	2,75375	2,726625
	general	total energy	8986,763	12728,4	12907,74
		total power	3238,232	4704,039	4706,38
		MED duration (s)	2,691775	2,68415	2,7277

Table A.4: table for 8640x8640 matrix with 576 ranks total results

the algorithm which best highlights this difference. It seems from data that there is also a slight difference between the deployment of 24 cores on two sockets or just on one socket for IMe. In this case, the deployments above two sockets are more energy consuming then the other one. The course of the energy consumption of the DRAM is confirmed with respect to the previous case. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes and the values are very similar for ScaLAPACK and for IMe.

IMe versus ScaLAPACK Whilst this configuration confirms that ScaLAPACK is always less energy consuming than IMe. The unexpected result concerns the duration of the execution: IMe is always at least 1 second faster than ScaLAPACK. The difference in terms of energy consumption involves especially the CPUs, meaning the power plane 0, the values of the DRAM are indeed almost respectively the same for any test.

A.1.3 Deployed ranks: 1296

Due to limitations on the number of nodes available per account on CINECA, it was possible only to test the 27 nodes deployment, not the 54 one. The gap between the power consumption of IMe and ScaLAPACK in this case is only of 12%. The energy values doubles if compared to the 576 ranks deployments. Moreover, also in this case IMe results to be faster than ScaLAPACK. This section refers to tables A.5 and A.6.

A.2 Results for matrix 17280 x 17280

With respect to the previous matrix dimension the power values remains almost the same. In fact, if the correspondent tests are compared, the differences are slights and the links are respected.

A.2.1 Deployed ranks: 144

This section refers to tables A.7 and A.8.

Phases of the algorithm As far as IMe and ScaLAPACK algorithms are concerned, there are mild differences in terms of power between the measurements of the whole execution and just the algorithm part. The fact that the allocation is almost irrelevant can be inferred from the power values which are very similar. However, the execution phase is always less consuming than the general monitoring, there is only one exception for IMe.

matrix 8640		1296 rank	27 nodes 48 ranks
IMe	Execution	energy PKG0	5967,502
		energy DRAM0	228,2905
		energy PKG1	5912,082
		energy DRAM1	249,3755
		power PKG0	4108,317
		power DRAM0	157,3874
		power PKG1	4069,473
		power DRAM1	171,5181
		MAX duration (s)	2,3103
		MED duration (s)	1,281
	General	energy PKG0	6361,986
		energy DRAM0	242,67
		energy PKG1	6295,61
		energy DRAM1	266,93
		power PKG0	4101,50
		power DRAM0	156,57
		power PKG1	4058,075
		power DRAM1	172,0135
		MAX duration (s)	1,9503
		MED duration (s)	1,5673
ScaLAPACK	Execution	energy PKG0	6716,84
		energy DRAM0	281,29
		energy PKG1	6632,11
		energy DRAM1	321,97
		power PKG0	3610,72
		power DRAM0	151,20
		power PKG1	3565,20
		power DRAM1	173,06
		MAX duration (s)	2,24
		MED duration (s)	1,82
	General	energy PKG0	6714,22
		energy DRAM0	281,64
		energy PKG1	6633,49
		energy DRAM1	323,12
		power PKG0	3597,03
		power DRAM0	150,87
		power PKG1	3553,754
		power DRAM1	173,073
		MAX duration (s)	2,2215
		MED duration (s)	1,8303

Table A.5: table for 8640x8640 matrix with 1296 ranks partial results

matrix 8640		1296 rank	27 nodes 48 ranks
IMe	execution	total energy	12357,25
		total power	8506,696
		MED duration (s)	1,281
	general	total energy	13167,19
		total power	8488,15
		MED duration (s)	1,5673
ScaLAPACK	execution	total energy	13952,20
		total power	7500,178
		MED duration (s)	1,82
	general	total energy	13167,19
		total power	7474,73
		MED duration (s)	1,8303

Table A.6: table for 8640x8640 matrix with 1296 ranks total results

Cores deployed per processor The cores deployed per processor heavily impact the power consumption. The deployment of 48 cores per processor above 3 nodes is not considerably power consuming if compared to the 6 nodes deployments, because is always a 30% less than the 24-cores executions. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one socket. The trend of the energy consumption of the DRAM is constant. This difference of the DRAM values is quite evident between ScaLAPACK and IMe. An interesting aspect is the difference between the measurements of package 0 and package 1, and DRAM 0 and DRAM 1. The energy consumption of zone 0 is two times higher than zone 1. This happens for both configurations of the sockets: the values of the measurements are very similar. Moreover, packages consume on average ten times the power with respect to the DRAMs

IMe versus ScaLAPACK The performances of ScaLAPACK are better than IMe in whatever test. The executions on 6 nodes show a sharp difference between the linear system algorithms, whereas the variations between the 3 nodes deployments are minor. As far as the duration of the execution is concerned, both the versions have a quite constant value for the 6 nodes deployments and for the 3 nodes deployments. IMe spends more time for the execution on 6 nodes than 3 nodes, whilst on the opposite ScaLAPACK spends half the time for the execution on 6 nodes with respect to the 3 nodes deployment.

matrix 17280		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	54903,92	101031,6	102929,39
		energy DRAM0	4598,123	8155,317	8180,53
		energy PKG1	54843,11	48154,96	51043,48
		energy DRAM1	4859,935	4504,646	5675,11
		power PKG0	449,1165	871,9561	857,10
		power DRAM0	37,6141	70,4119	68,15
		power PKG1	448,6228	417,0288	426,15
		power DRAM1	39,7552	38,9786	47,33
		MAX duration (s)	123,5368	125,6231	128,99
		MED duration (s)	122,3288	113,5992	118,9049
	General	energy PKG0	54379,12	98356,57	96777,58
		energy DRAM0	4586,099	7695,85	7744,77
		energy PKG1	54472,84	49993,09	48293,51
		energy DRAM1	4709,852	5018,98	4821,40
		power PKG0	447,8926	859,52	864,48
		power DRAM0	37,7733	67,27	69,23
		power PKG1	448,6635	437,2808	431,6138
		power DRAM1	38,7929	43,8829	43,0517
		MAX duration (s)	122,5814	120,4826	128,1133
		MED duration (s)	121,254	114,2972	110,2472
ScaLAPACK	Execution	energy PKG0	19162,28	34495,82	35471,22
		energy DRAM0	865,8354	1828,60	1886,603
		energy PKG1	18534,21	19936,20	19570,77
		energy DRAM1	1060,988	1777,42	1844,942
		power PKG0	387,7472	697,49	720,4668
		power DRAM0	17,5197	36,97	38,3193
		power PKG1	375,0445	403,09	397,5198
		power DRAM1	21,4693	35,94	37,4732
		MAX duration (s)	50,6403	49,76	49,4122
		MED duration (s)	49,10255	49,42	49,2116
	General	energy PKG0	18753,80	33989,14	32760,75
		energy DRAM0	1047,94	1697,693	1712,787
		energy PKG1	18827,15	20175,84	18817,44
		energy DRAM1	1143,65	1908,967	1928,499
		power PKG0	378,46	690,1759	663,6771
		power DRAM0	21,15	34,4717	34,6979
		power PKG1	379,9527	409,645	381,1956
		power DRAM1	23,08	38,7612	39,067
		MAX duration (s)	51,3401	49,8565	49,6934
		MED duration (s)	49,1212	49,1835	49,27575

Table A.7: table for 17280x17280 matrix with 144 ranks partial results

matrix 17280		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	119205,09	161846,52	167828,50
		total power	975,1086	1398,375	1398,729
		MED duration (s)	122,3288	113,5992	118,9049
	general	total energy	118147,92	161064,49	157637,26
		total power	973,12	1407,95	1408,37
		MED duration (s)	121,254	114,2972	110,2472
ScaLAPACK	execution	total energy	39623,31	58038,04	58773,54
		total power	801,7807	1173,486	1193,779
		MED duration (s)	49,10255	49,42	49,2116
	general	total energy	39772,54	57771,64	55219,47
		total power	802,64	1173,05	1118,64
		MED duration (s)	49,1212	49,1835	49,27575

Table A.8: table for 17280x17280 matrix with 144 ranks total results

matrix 17280		576 rank	12 nodes 48 ranks	24 nodes 24 ranks 1 socket	24 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	49482,12	86007,12	86544,24
		energy DRAM0	3622,842	6139,787	6035,649
		energy PKG1	49330,7	41648,91	45491,48
		energy DRAM1	3942,547	3949,276	4149,74
		power PKG0	1797,669	3447,344	3378,925
		power DRAM0	131,6889	246,0722	235,7915
		power PKG1	1792,161	1674,499	1777,012
		power DRAM1	143,2991	158,3182	161,8245
		MAX duration (s)	29,6658	28,2047	28,4955
		MED duration (s)	27,32055	24,61065	24,83208
	General	energy PKG0	49285,13	85748,01	82765,45
		energy DRAM0	3595,216	6063,196	5869,871
		energy PKG1	48936,76	43639,99	42090,15
		energy DRAM1	3821,68	4182,517	3949,895
		power PKG0	1788,731	3383,13	3417,073
		power DRAM0	130,5987	239,3024	242,4499
		power PKG1	1776,116	1724,401	1742,015
		power DRAM1	138,8075	164,7462	163,0539
		MAX duration (s)	30,4778	28,6241	28,1265
		MED duration (s)	27,17293	24,83848	23,59005
ScaLAPACK	Execution	energy PKG0	24642,8	45273,48	47194,52
		energy DRAM0	1113,182	2192,012	2205,975
		energy PKG1	23913,96	24153,2	22086,73
		energy DRAM1	1309,285	2353,932	2343,279
		power PKG0	1593,295	2915,233	3039,053
		power DRAM0	71,9745	141,1358	142,0545
		power PKG1	1546,17	1555,184	1422,608
		power DRAM1	84,65	151,5575	150,8945
		MAX duration (s)	16,05	16,4685	16,2308
		MED duration (s)	15,39	15,40835	15,4701
	General	energy PKG0	33763,73	46093,75	44890,02
		energy DRAM0	1451,25	2244,756	2222,605
		energy PKG1	33074,90	23169,23	24299,69
		energy DRAM1	1634,50	2438,138	2427,721
		power PKG0	1577,45	2924,036	2867,402
		power DRAM0	70,92	142,4083	141,9639
		power PKG1	1539,426	1470,024	1552,109
		power DRAM1	79,6845	154,6685	155,0635
		MAX duration (s)	67,7754	16,2433	16,121
		MED duration (s)	15,4731	15,67765	15,63225

Table A.9: table for 17280x17280 matrix with 576 ranks partial results

A.2.2 Deployed ranks: 576

Also for this matrix, the deployment on 12 nodes and 24 nodes is approximately 4 times more power consuming compared to the deployment on 3 and 6 nodes, this data confirms a linear trend of energy consumption that follows the number of processors involved. The energy consumption is constant and varies in a range between 2188 and 5564 Watts, however most of the values are fluctuates in a 1500-1600 Watt range. With respect to the deployment of 144 ranks the duration for the execution decreases of 4 times as expected since the number of ranks quadruples. This section refers to tables A.9 and A.10.

matrix 17280		576 rank	12 nodes 48 ranks	24 nodes 24 ranks 1 socket	24 nodes 24 ranks 2 socket
IMe	execution	total energy	106378,2	137745,1	142221,1
		total power	3864,818	5526,23	5553,553
		MED duration (s)	27,32055	24,61065	24,83208
	general	total energy	105638,79	139633,7	134675,36
		total power	3834,25	5511,58	5564,59
		MED duration (s)	27,17293	24,83848	23,59005
ScaLAPACK	execution	total energy	50979,23	73972,62	73830,51
		total power	3296,096	4763,11	4754,609
		MED duration (s)	15,39	15,40835	15,4701
	general	total energy	69924,38	142221,11	73840,04
		total power	3267,48	5553,55	4716,54
		MED duration (s)	15,4731	15,67765	15,63225

Table A.10: table for 17280x17280 matrix with 576 ranks total results

Phases of the algorithm Also, in this case the measurements of the of the different phases of the algorithm are not meaningful, there is always a slight difference between the two and in some cases for IMe the values of the general monitoring are less than execution of the algorithm itself.

Cores deployed per processor The cores deployed per processor weakly impact the power consumption. The deployment of 48 cores per processor above 12 nodes is considerably less energy consuming if compared to the 24 nodes deployments. The gap between the power consumption of the 12 nodes deployment and the 24 nodes reaches the 60%. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one, there is only a 1% deviation. The course of the energy consumption of the DRAM decreases with the number of nodes. This difference is quite evident both for ScaLAPACK and for IMe.

IMe versus ScaLAPACK The performances of ScaLAPACK and IMe are similar, from a total power consumption point of view in some cases there are only slight differences. In the executions that involves 12 nodes for the energy consumption of the DRAM, there are marked variations between IMe and ScaLAPACK. The energy consumption of the DRAM varies a lot between IMe and ScaLAPACK, the second one consumes way less power. As far as the duration of the execution is concerned, both the versions amount to a constant value, which is around 29 seconds for IMe and 16 seconds for ScaLAPACK.

A.2.3 Deployed ranks: 1296

Due to limitations on the number of nodes available per account on CINECA, it was possible only to test the 27 nodes deployment. The gap between the power consumption of IMe and ScaLAPACK in this case is only 12%. The energy values doubles if compared to the 576 ranks deployments. Moreover, also in this case

IMe results to be faster than ScaLAPACK. This section refers to tables A.11 and A.12.

A.3 Results for matrix 25920 x 25920

The energy values for this matrix are 3,5 times the energy values for the matrix dimension 8640. This does not correspond to a linear trend of the energy consumption.

A.3.1 Deployed ranks: 144

As stated before, the duration of the tests on 144 ranks with ten repetitions takes too much, and the scheduling of the job is delayed by SLURM of days. Therefore, for this deployment the number of repetitions is reduced to five. This section refers to tables A.13 and A.14.

Phases of the algorithm Also, in this case the measurements of the of the different phases of the algorithms are not meaningful, there is always a slight difference between the two and in some cases for IMe the values of the general monitoring are less than jets the algorithm itself.

Cores deployed per processor The deployment of 48 cores per processor above 3 nodes is considerably more energy consuming if compared to the 6 nodes deployments. As matter of fact, the difference is higher between the energy values of the 24 cores and 48 cores deployments of package 0. In this case it reaches the 50% for ScaLAPACK, whilst for package 1 the values are similar. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one. The course of the energy consumption of the DRAM goes in a reverse direction. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes. This difference is quite evident both for ScaLAPACK and for IMe.

IMe versus ScaLAPACK The performances of ScaLAPACK and IMe are not comparable, for the 144 ranks deployment IMe is always more energy consuming than ScaLAPACK. Also the differences between IMe and ScaLAPACK for the energy consumption of the DRAM are marked. As far as the duration of the execution is concerned, both the versions amount to a constant value, which is around 980 seconds for IMe and 770 seconds for ScaLAPACK.

matrix 17280		1296 rank	27 nodes 48 ranks
IMe	Execution	energy PKG0	31561,14
		energy DRAM0	1379,125
		energy PKG1	31511,08
		energy DRAM1	1525,586
		power PKG0	4047,462
		power DRAM0	179,0256
		power PKG1	4045,192
		power DRAM1	197,3295
		MAX duration (s)	12,239
		MED duration (s)	6,888
	General	energy PKG0	31561,14
		energy DRAM0	1379,125
		energy PKG1	31511,08
		energy DRAM1	1525,586
		power PKG0	4047,462
		power DRAM0	179,0256
		power PKG1	4045,192
		power DRAM1	197,3295
		MAX duration (s)	12,239
		MED duration (s)	6,888
ScaLAPACK	Execution	energy PKG0	33468,1
		energy DRAM0	1408,903
		energy PKG1	33109,29
		energy DRAM1	1622,742
		power PKG0	3589,996
		power DRAM0	151,125
		power PKG1	3551,509
		power DRAM1	174,0567
		MAX duration (s)	9,8544
		MED duration (s)	9,16675
	General	energy PKG0	33490,26
		energy DRAM0	1394,37
		energy PKG1	33012,75
		energy DRAM1	1605,95
		power PKG0	3634,82
		power DRAM0	151,33
		power PKG1	3582,997
		power DRAM1	174,2952
		MAX duration (s)	9,5698
		MED duration (s)	9,18645

Table A.11: table for 17280x17280 matrix with 1296 ranks partial results

matrix 17280		1296 rank	27 nodes 48 ranks
IMe	execution	total energy	65976,94
		total power	8469,009
		MED duration (s)	6,888
	general	total energy	65976,94
		total power	8469,01
		MED duration (s)	6,888
ScaLAPACK	execution	total energy	69609,04
		total power	7466,687
		MED duration (s)	9,16675
	general	total energy	69503,33
		total power	7543,45
		MED duration (s)	9,18645

Table A.12: table for 17280x17280 matrix with 1296 ranks total results

A.3.2 Deployed ranks: 576

This section refers to tables A.15 and A.16.

Phases of the algorithm Also, in this case the measurements of the of the different phases of the algorithms are not meaningful, there is always a slight difference between the two.

Cores deployed per processor The deployment of 48 cores per processor above 12 nodes is 20-30% more energy consuming, if compared to the 24 nodes deployments. As matter of fact, the difference is higher between the energy values of the 24 cores and 48 cores deployments of package 0. In this case it reaches the 50% for ScaLAPACK, whilst for package 1 the values are similar. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one. The course of the energy consumption of the DRAM goes in a reverse direction. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes. This difference is quite evident both for ScaLAPACK and for IMe.

IMe versus ScaLAPACK The performances of ScaLAPACK and IMe are not comparable, for the 576 ranks deployment IMe is always 3 times more energy consuming than ScaLAPACK. Also the differences between IMe and ScaLAPACK for the energy consumption of the DRAM are marked. As far as the duration of the execution is concerned, both the versions amount to a constant value, which

matrix 25920		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	191195,5142	357745,6808	319650,3682
		energy DRAM0	15909,2	29528,07	27878,58
		energy PKG1	188880,6	177043,2	175463,9
		energy DRAM1	16610,84	18743,99	17584,86
		power PKG0	443,8648	865,4678	802,322
		power DRAM0	36,9338	71,4384	69,9054
		power PKG1	438,4906	429,1044	441,5614
		power DRAM1	38,562	45,4228	44,3904
		MAX duration (s)	433,5214	429,7922	451,0632
		MED duration (s)	430,3192	419,211	390,5878
	General	energy PKG0	199353,8	328690,8	342691,66
		energy DRAM0	17533,74	29061,53	29671,26
		energy PKG1	199133,4	175766,7	174071,68
		energy DRAM1	17595,42	17532,77	17899,59
		power PKG0	448,8448	812,485	818,30
		power DRAM0	39,5026	71,8698	70,85
		power PKG1	448,3536	435,3782	415,8016
		power DRAM1	39,6434	43,4324	42,7686
		MAX duration (s)	467,9868	428,6868	435,9664
		MED duration (s)	449,4491	399,4946	423,0895
ScaLAPACK	Execution	energy PKG0	58597,66	114940,3	107775,5
		energy DRAM0	3275,586	5697,277	5796,984
		energy PKG1	54811,21	63242,1	56759,84
		energy DRAM1	3307,908	6635,145	6510,911
		power PKG0	364,3882	718,0768	674,8168
		power DRAM0	20,3692	35,593	36,2968
		power PKG1	340,8404	395,1026	355,3898
		power DRAM1	20,571	41,4524	40,7668
		MAX duration (s)	163,5846	160,3337	160,1347
		MED duration (s)	160,0423	160,1732	159,5494
	General	energy PKG0	57655	114751,9	105456,7
		energy DRAM0	3187,135	6292,817	5740,042
		energy PKG1	59425,3	62833,55	59303,91
		energy DRAM1	3408,921	6888,643	6537,494
		power PKG0	359,0748	716,951	660,5124
		power DRAM0	19,8498	39,3166	35,9518
		power PKG1	370,1004	392,5632	371,4348
		power DRAM1	21,2312	43,0388	40,9466
		MAX duration (s)	162,5537	160,836	160,5558
		MED duration (s)	160,1248	160,0033	159,459

Table A.13: table for 25920x25920 matrix with 144 ranks partial results

matrix 25920		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	412596,15	583060,90	540577,66
		total power	957,8512	1411,433	1358,179
		MED duration (s)	430,3192	419,211	390,5878
	general	total energy	433616,39	551051,86	564334,18
		total power	976,34	1363,17	1347,72
		MED duration (s)	449,4491	399,4946	423,0895
ScaLAPACK	execution	total energy	119992,37	190514,78	176843,25
		total power	746,1688	1190,225	1107,27
		MED duration (s)	160,0423	160,1732	159,5494
	general	total energy	123676,36	190766,86	177038,11
		total power	770,26	1191,87	1108,85
		MED duration (s)	160,1248	160,0033	159,459

Table A.14: table for 25920x25920 matrix with 144 ranks total results

matrix 25920		576 rank	12 nodes 48 ranks	24 nodes 24 ranks 1 socket	24 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	208847,5349	358821,7704	359344,1331
		energy DRAM0	15984,93	26694,67	27388,76
		energy PKG1	207011,3	181073,8	179371,4
		energy DRAM1	16837,19	16689,21	16664,68
		power PKG0	1796,458	3482,34	3447,04
		power DRAM0	137,5333	259,2662	262,7388
		power PKG1	1780,706	1760,177	1724,26
		power DRAM1	144,8611	162,0359	160,0084
		MAX duration (s)	120,3205	116,9724	119,0979
		MED duration (s)	115,8465	102,7518	102,0824
	General	energy PKG0	207781,9	336650,7	337481,78
		energy DRAM0	15973,95	26516,09	26110,81
		energy PKG1	205339,8	171629,8	181178,44
		energy DRAM1	16918,6	16700,23	16953,59
		power PKG0	1795,155	3351,504	3337,58
		power DRAM0	138,0157	264,2312	258,36
		power PKG1	1774,039	1715,06	1794,276
		power DRAM1	146,1777	166,524	167,7485
		MAX duration (s)	118,084	115,5302	115,3469
		MED duration (s)	115,8295	99,25725	99,38948
ScaLAPACK	Execution	energy PKG0	76837,71	136377,35	135750,68
		energy DRAM0	3514,224	6968,44	6999,59
		energy PKG1	73689,71	72128,64	72961,32
		energy DRAM1	3997,07	6966,22	6988,81
		power PKG0	1601,437	2927,75	2895,18
		power DRAM0	73,2928	149,60	149,29
		power PKG1	1535,866	1548,68	1556,23
		power DRAM1	83,3574	149,56	149,06
		MAX duration (s)	53,3455	46,85	47,17
		MED duration (s)	47,03168	46,55	46,84735
	General	energy PKG0	75670,32	130745,94	136357,34
		energy DRAM0	3538,339	7028,25	6942,21
		energy PKG1	72808,74	72890,15	70976,49
		energy DRAM1	4001,644	7091,87	7048,32
		power PKG0	1584,944	2802,40	2924,60
		power DRAM0	74,1484	150,65	148,90
		power PKG1	1525,066	1562,441	1522,513
		power DRAM1	83,8588	152,0088	151,1801
		MAX duration (s)	51,9586	47,6863	47,061
		MED duration (s)	46,85898	46,5483	46,56905

Table A.15: table for 25920x25920 matrix with 576 ranks partial results

matrix 25920		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket
IMe	execution	total energy	448680,93	583279,47
		total power	3859,558	5663,82
		MED duration (s)	115,8465	102,7518
	general	total energy	446014,28	551496,81
		total power	3853,39	5497,32
		MED duration (s)	115,8295	99,25725
ScaLAPACK	execution	total energy	158038,72	222440,66
		total power	3293,953	4775,592
		MED duration (s)	47,03168	46,55
	general	total energy	156019,04	217756,22
		total power	3268,02	4667,50
		MED duration (s)	46,85898	46,5483

Table A.16: table for 25920x25920 matrix with 576 ranks total results

is around 110 seconds for IMe and 45 seconds for ScaLAPACK, thus ScaLAPACK takes half the the time of IMe for computation.

A.3.3 Deployed ranks: 1296

Due to limitations on the number of nodes available per account on CINECA, it was possible only to test the 27 nodes deployment. The gap between the power consumption of IMe and ScaLAPACK in this case is only 15%. The energy values doubles if compared to the 576 ranks deployments. Moreover, also in this case ScaLAPACK results to be faster than IMe. This section refers to tables A.17 and A.18.

A.4 Results for matrix 34560 x 34560

A.4.1 Deployed ranks: 144

As stated before, the duration of the tests on 144 ranks with ten repetitions takes too much, and the scheduling of the job is delayed by SLURM of days. Therefore, for this deployment the number of repetitions is reduced to five. This section refers to tables A.19 and A.20.

Phases of the algorithm Also, in this case the measurements of the of the different phases of the algorithms are not meaningful, there is always a slight difference between the two and in some cases for ScaLAPACK the values of the general monitoring are less than jets the algorithm itself.

matrix 25920		1296 rank	27 nodes 48 ranks
IMe	Execution	energy PKG0	177985
		energy DRAM0	13304,89
		energy PKG1	178245
		energy DRAM1	13451,14
		power PKG0	4013,356
		power DRAM0	300,1592
		power PKG1	4019,219
		power DRAM1	303,4483
		MAX duration (s)	46,9033
		MED duration (s)	43,965
	General	energy PKG0	180278,5
		energy DRAM0	12984,03
		energy PKG1	180229,7
		energy DRAM1	13639,54
		power PKG0	4047,401
		power DRAM0	291,5735
		power PKG1	4046,33
		power DRAM1	306,2935
		MAX duration (s)	46,0328
		MED duration (s)	44,32525
ScaLAPACK	Execution	energy PKG0	90395,86
		energy DRAM0	4388,992
		energy PKG1	89281,4
		energy DRAM1	4527,122
		power PKG0	3556,354
		power DRAM0	172,6761
		power PKG1	3512,508
		power DRAM1	178,1101
		MAX duration (s)	26,1144
		MED duration (s)	25,24645
	General	energy PKG0	97586,91
		energy DRAM0	4679,521
		energy PKG1	96606,8
		energy DRAM1	4833,37
		power PKG0	3578,371
		power DRAM0	171,8051
		power PKG1	3542,213
		power DRAM1	177,4468
		MAX duration (s)	30,9493
		MED duration (s)	26,48155

Table A.17: table for 25920x25920 matrix with 1296 ranks partial results

matrix 25920		1296 rank	27 nodes 48 ranks
IMe	execution	total energy	382986,1
		total power	8636,183
		MED duration (s)	43,965
	general	total energy	387131,8
		total power	8691,598
		MED duration (s)	44,32525
ScaLAPACK	execution	total energy	188593,4
		total power	7419,648
		MED duration (s)	25,24645
	general	total energy	203706,6
		total power	7469,836
		MED duration (s)	26,48155

Table A.18: table for 25920x25920 matrix with 1296 ranks total results

Cores deployed per processor The deployment of 48 cores per processor above 3 nodes is 27% more energy consuming if compared to the 6 nodes deployments. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one. The course of the energy consumption of the DRAM goes in a reverse direction. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes. This difference is quite evident both for ScaLAPACK and for IMe. In this case probably due to the jamming of the different ranks on the same resources on the DRAM the access time increase. Thus, the duration of the execution for the 48 cores deployment is 6-10

IMe versus ScaLAPACK The performances of ScaLAPACK and IMe are not comparable, for the 144 ranks deployment IMe is always 3 times more energy consuming than ScaLAPACK. This is due both to a longer execution and to a substantial difference between the two algorithms. Also the differences between IMe and ScaLAPACK for the energy consumption of the DRAM are marked. As far as the duration of the execution is concerned, both the versions amount to a constant value, which is around 110 seconds for IMe and 45 seconds for ScaLAPACK, thus ScaLAPACK takes half the the time of IMe for computation.

matrix 34560		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	461136,046	836827,34	813532,01
		energy DRAM0	38068,11	72436,05	65918,17
		energy PKG1	461080,144	429360,43	415796,33
		energy DRAM1	41655,0818	45333,78	45113,37
		power PKG0	448,7516	855,14	841,87
		power DRAM0	37,0456	74,04	68,22
		power PKG1	448,6974	439,92	430,21
		power DRAM1	40,5364	46,45	46,69
		MAX duration (s)	1029,677	1067,05	985,56
		MED duration (s)	1026,834	944,91	973,893
	General	energy PKG0	456235,8	782158,82	801347,15
		energy DRAM0	37694,42	64126,56	67135,18
		energy PKG1	455378,99	436351,98	430227,52
		energy DRAM1	39479,10	42526,53	44291,39
		power PKG0	448,75	835,75	832,62
		power DRAM0	37,08	68,52	69,78
		power PKG1	447,9036	466,7264	447,7482
		power DRAM1	38,831	45,5328	46,1138
		MAX duration (s)	1017,528	1011,078	1030,68
		MED duration (s)	1016,551	919,0793	954,5491
ScaLAPACK	Execution	energy PKG0	134559	241362,00	267323,3
		energy DRAM0	7680,157	15547,64	13843,78
		energy PKG1	139743,6	143240,21	143874,9
		energy DRAM1	8297,612	13017,73	14015,06
		power PKG0	361,5992	653,53	725,089
		power DRAM0	20,639	42,10	37,5498
		power PKG1	375,5316	387,87	390,2464
		power DRAM1	22,2982	35,25	38,0144
		MAX duration (s)	373,528	369,93	368,7529
		MED duration (s)	373,4933	369,08	368,6901
	General	energy PKG0	138348,43	260640,1	260719,01
		energy DRAM0	7689,90	15685,62	13614,48
		energy PKG1	136933,10	152477,4	154891,06
		energy DRAM1	8844,43	15101,04	15201,50
		power PKG0	372,51	706,2264	706,29
		power DRAM0	20,71	42,5018	36,88
		power PKG1	368,701	413,1608	419,6048
		power DRAM1	23,8142	40,9186	41,1808
		MAX duration (s)	373,1166	369,4935	370,0708
		MED duration (s)	372,1874	369,0296	369,088

Table A.19: table for 34560x34560 matrix with 1296 ranks partial results

matrix 34560		144 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	1001939,39	1383957,60	1340359,88
		total power	975,031	1415,554	1386,976
		MED duration (s)	1026,8337	944,91	973,893
	general	total energy	988788,34	1325163,89	1343001,25
		total power	972,56	1416,53	1396,26
		MED duration (s)	1016,551	919,0793	954,5491
ScaLAPACK	execution	total energy	290280,39	413167,58	439057,01
		total power	780,068	1118,75	1190,9
		MED duration (s)	373,4933	369,08	368,6901
	general	total energy	291815,87	443904,14	444426,06
		total power	785,73	1202,81	1203,96
		MED duration (s)	372,1874	369,0296	369,088

Table A.20: table for 34560x34560 matrix with 1296 ranks total results

matrix 34560		576 rank	12 nodes 48 ranks	24 nodes 24 ranks 1 socket	24 nodes 24 ranks 2 socket
IMe	Execution	energy PKG0	510493,9082	880501,14	849788,43
		energy DRAM0	39572,94	68877,53	66797,88
		energy PKG1	508007,7	456169,51	457088,37
		energy DRAM1	41836,2428	43565,59	43887,70
		power PKG0	1771,733	3369,39	3362,02
		power DRAM0	137,3519	263,81	264,26
		power PKG1	1763,113	1748,76	1809,24
		power DRAM1	145,2071	167,01	173,87
		MAX duration (s)	296,8525	286,30	290,93
		MED duration (s)	287,8548	257,70	250,6986
	General	energy PKG0	524387	859299,82	857608,41
		energy DRAM0	40380,12	66459,05	67478,22
		energy PKG1	522758,9	450153,62	468519,80
		energy DRAM1	41519,68	44184,52	44055,44
		power PKG0	1795,2	3406,55	3325,76
		power DRAM0	138,2584	263,61	261,92
		power PKG1	1789,624	1787,531	1822,084
		power DRAM1	142,1573	175,3595	171,3208
		MAX duration (s)	305,0388	281,7142	302,1371
		MED duration (s)	291,1813	250,3271	256,0835
ScaLAPACK	Execution	energy PKG0	161005,14	297188,54	294171,97
		energy DRAM0	8841,89	15271,77	15577,96
		energy PKG1	161798,64	167234,88	166632,50
		energy DRAM1	8933,12	15874,35	15848,79
		power PKG0	1523,35	2847,43	2829,45
		power DRAM0	83,67	146,33	149,84
		power PKG1	1530,86	1602,68	1602,93
		power DRAM1	84,53	152,11	152,45
		MAX duration (s)	109,66	105,04	104,82
		MED duration (s)	104,96	104,27	103,8036
	General	energy PKG0	180918,44	289555,3	295034,61
		energy DRAM0	8372,76	15725,33	15558,36
		energy PKG1	175922,78	167713,3	165732,25
		energy DRAM1	9470,66	15417,73	15853,52
		power PKG0	1589,10	2772,524	2839,03
		power DRAM0	74,13	150,5733	149,72
		power PKG1	1544,224	1605,89	1594,958
		power DRAM1	83,8184	147,6282	152,5584
		MAX duration (s)	184,9414	105,3941	104,6439
		MED duration (s)	104,0461	104,2888	103,8479

Table A.21: table for 34560x34560 matrix with 1296 ranks partial results

A.4.2 Deployed ranks: 576

The energy consumption of the 144 ranks and 576 ranks are similar, probably because the duration of the execution is one quarter of the mean duration of 144 ranks. Thus, also power consumption quadruplicates for both IMe and ScaLAPACK with respect to the previous deployment. This section refers to tables A.21 and A.22.

Phases of the algorithm Also, in this case the measurements of the of the different phases of the algorithms are not meaningful, there is always a slight difference between the two and in some cases for ScaLAPACK the values of the

matrix 34560		576 rank	3 nodes 48 ranks	6 nodes 24 ranks 1 socket	6 nodes 24 ranks 2 socket
IMe	execution	total energy	1099910,79	1449113,77	1417562,40
		total power	3817,405	5548,962	5609,391
		MED duration (s)	287,8548	257,70	250,6986
	general	total energy	1129045,67	1420097,01	1437661,87
		total power	3865,24	5633,05	5581,09
		MED duration (s)	291,1813	250,3271	256,0835
ScaLAPACK	execution	total energy	340578,79	495569,54	492231,22
		total power	3222,401	4748,553	4734,67
		MED duration (s)	104,96	104,27	103,8036
	general	total energy	374684,64	488411,68	492178,73
		total power	3291,27	4676,61	4736,26
		MED duration (s)	104,0461	104,2888	103,8479

Table A.22: table for 34560x34560 matrix with 1296 ranks total results

general monitoring are less than jets the algorithm itself.

Cores deployed per processor The deployment of 48 cores per processor above 12 nodes is 23% more energy consuming if compared to the 14 nodes deployments. Moreover, there is not any difference between the deployment of 24 cores on two sockets or just on one. The course of the energy consumption of the DRAM goes in a reverse direction. As matter of fact, the energy consumption of the DRAM decreases with the number of nodes. This difference is quite evident both for ScaLAPACK and for IMe. In this case probably due to the jamming of the different ranks on the same resources on the DRAM the access time increase. Thus, the duration of the execution for the 48 cores deployment is 9% higher than the 24 core deployments for IMe. This is more evident for IMe.

IMe versus ScaLAPACK The performances of ScaLAPACK and IMe are not comparable. For the 576 ranks deployment IMe is always 3 times more energy consuming than ScaLAPACK. This is due both to a longer execution and to a substantial difference between the two algorithms. Also the differences between IMe and ScaLAPACK for the power consumption of the DRAM are marked. As far as the duration of the execution is concerned, both the versions amount to a constant value, which is around 270 seconds for IMe and 103 seconds for ScaLAPACK, thus ScaLAPACK takes more half the the time of IMe for computation.

A.4.3 Deployed ranks: 1296

Due to limitations on the number of nodes available per account on CINECA, it was possible only to test the 27 nodes deployment. The gap between the power consumption of IMe and ScaLAPACK in this case is 55%, and it descreases with respect to the 576 ranks deployment. The energy values doubles if compared to the 576 ranks deployments. Moreover, also in this case ScaLAPACK results to be faster than IMe. This section refers to tables A.23 and A.24.

matrix 34560		1296 rank	27 nodes 48 ranks
IMe	Execution	energy PKG0	503146,46
		energy DRAM0	38130,97
		energy PKG1	504039,2
		energy DRAM1	38668,785
		power PKG0	4009,361
		power DRAM0	304,0412
		power PKG1	4016,486
		power DRAM1	308,3043
		MAX duration (s)	133,3568
		MED duration (s)	124,549
	General	energy PKG0	491899,8
		energy DRAM0	37902,06
		energy PKG1	492714,31
		energy DRAM1	38496,20
		power PKG0	4000,48
		power DRAM0	308,28
		power PKG1	4007,116
		power DRAM1	313,1089
		MAX duration (s)	126,2068
		MED duration (s)	122,9609
ScaLAPACK	Execution	energy PKG0	202783,35
		energy DRAM0	9189,63
		energy PKG1	199788,61
		energy DRAM1	10190,79
		power PKG0	3654,92
		power DRAM0	165,71
		power PKG1	3601,14
		power DRAM1	183,76
		MAX duration (s)	59,22
		MED duration (s)	54,75
	General	energy PKG0	205628,84
		energy DRAM0	9222,70
		energy PKG1	201621,13
		energy DRAM1	10217,90
		power PKG0	3704,47
		power DRAM0	166,20
		power PKG1	3632,303
		power DRAM1	184,1304
		MAX duration (s)	59,5932
		MED duration (s)	54,24235

Table A.23: table for 34560x34560 matrix with 1296 ranks partial results

matrix 34560		1296 rank	27 nodes 48 ranks
IMe	execution	total energy	1083985,45
		total power	8638,193
		MED duration (s)	124,549
	general	total energy	1061012,39
		total power	8628,98
		MED duration (s)	122,9609
ScaLAPACK	execution	total energy	421952,39
		total power	7605,535
		MED duration (s)	54,75
	general	total energy	426690,56
		total power	7687,10
		MED duration (s)	54,24235

Table A.24: table for 34560x34560 matrix with 1296 ranks total results