UNIVERSITAT JAUME I DE CASTELLÓ
E. S. DE TECNOLOGIA I CIENCIÈS EXPERIMENTALS

Performance and energy optimization of the iterative solution of sparse linear systems on multicore processors

Ph.D. Thesis

Presented by: Maria Barreda Vayá
Supervised by: José I. Aliaga Estellés
Enrique S. Quintana Ortí

Castelló de la Plana, March 2017

UNIVERSITAT JAUME I DE CASTELLÓ
E. S. DE TECNOLOGIA I CIENCIÈS EXPERIMENTALS

Performance and energy optimization of the iterative solution of sparse linear systems on multicore processors

Maria Barreda Vayá

Contents

List of Figures

List of Tables

It always seems impossible until it is done.

Nelson Mandela

Summary

Large sparse systems of linear equations are ubiquitous problems in diverse scientific and engineering applications and big-data analytics. The interest of these applications and the fact that the solution of the linear system is usually a significant time-consuming stage has promoted the design and high-performance implementation of numerous matrix storage formats, algorithms, and libraries to efficiently tackle sparse instances of these linear algebra problems in general-purpose processorss (GPPs), following the evolution of computer architectures.

High Performance Computing (HPC) architectures enable the solution of complex applications by aggregating a number of multicore processors. As a consequence, developers face the challenge of implementing parallel algorithms that efficiently exploit the concurrency of the hardware. Furthermore, the advances in the number of transistors that can be integrated in a circuit have not enjoyed a proportional reduction of the power dissipated by the CMOS technology, turning the power wall into a crucial challenge that the HPC community needs to address. Unfortunately, despite the importance of energy consumption, few software developers take it into account in their implementations.

In this dissertation we target the solution of large sparse systems of linear equations using preconditioned iterative methods based on Krylov subspaces. Specifically, we focus our efforts on ILUPACK, a library that offers multi-level Incomplete LU (ILU) preconditioners for the effective solution of sparse linear systems. The increase of the number of equations in these systems and the introduction of new HPC architectures motivates us to develop a parallel version of ILUPACK which optimizes both execution time and energy consumption on current multicore architectures and clusters of nodes built from this type of technology. Thus, the main goal of this thesis is the design, implementation and evaluation of parallel and energy-efficient iterative sparse linear system solvers for multicore processors as well as recent manycore accelerators such as the Intel Xeon Phi.

To fulfill the general objective of the thesis, we optimize ILUPACK exploiting task parallelism via the programming models underlying OmpSs, MPI and a combination of both. These implementations are also tuned for their execution on specialized architectures like Non-Uniform Memory Access (NUMA) platforms or the Intel Xeon Phi. Finally, the energy efficiency of the solver is evaluated in different multicore platforms, taking advantage of an automatic framework to detect power sinks, also developed as part of this thesis.

Resumen

Los sistemas dispersos de ecuaciones lineales aparecen en numerosas aplicaciones científicas y de ingeniería así como en procesos de análisis que involucran grandes volúmenes de datos. El interés de estas aplicaciones y el hecho de que la solución del sistema lineal sea, de manera habitual, una parte costosa de su tratamiento ha promovido el diseño y las implementaciones de alto rendimiento de formatos de almacenamiento de matrices, algoritmos y bibliotecas para tratar de manera eficiente estos problemas de álgebra lineal en procesadores de propósito general, siguiendo la evolución de las arquitecturas de computadores.

Las arquitecturas de alto rendimiento permiten la solución de aplicaciones complejas mediante la utilización de varios procesadores multinúcleo. Como consecuencia, los desarrolladores se enfrentan al reto de implementar algoritmos paralelos que exploten de manera eficaz la concurrencia del hardware. Por otro lado, los avances en el número de transistores que pueden integrarse en un circuito no han gozado de una reducción proporcional en la potencia disipada por la tecnología CMOS, postulando a la potencia como un problema fundamental que la comunidad que trabaja en temas de computación de alto rendimiento debe afrontar. Desgraciadamente, a pesar de la importancia del consumo energético, son pocos los programadores que tienen en cuenta este factor en sus desarrollos de software.

En esta tesis doctoral abordamos la solución de sistemas dispersos de ecuaciones lineales utilizando métodos iterativos precondicionados basados en subespacios de Krylov. En concreto, centramos nuestros esfuerzos en ILUPACK, una biblioteca que implementa precondicionadores de tipo ILU multinivel para la solución eficiente de sistemas lineales dispersos. El incremento en el número de ecuaciones de estos sistemas, y la aparición de nuevas arquitecturas, motiva el desarrollo de una versión paralela de ILUPACK que optimice tanto el tiempo de ejecución como el consumo energético en arquitecturas multinúcleo actuales y en clusters de nodos construidos a partir de esta tecnología. De manera general, el objetivo principal de la tesis doctoral es el diseño, implementación y evaluación de resolutores paralelos energéticamente eficientes para sistemas lineales dispersos orientados a procesadores multinúcleo así como aceleradores hardware como el Intel Xeon Phi.

Para lograr el objetivo general de la tesis doctoral, optimizamos ILUPACK aprovechando el paralelismo de tareas del método mediante los modelos de programación subyacentes en OmpSs y MPI. Estas implementaciónes están dirigidas asimismo a la ejecución sobre arquitecturas especializadas de tipo NUMA y el Intel Xeon Phi. Finalmente, la eficiencia energética de las implementaciones resultantes se evalúan sobre diferentes arquitecturas multinúcleo, haciendo uso de un entorno automático para detectar sumideros de potencia desarrollado en el marco de la tesis doctoral.

Agradecimientos

La tesis finalmente está llegando a su fin. Han sido cuatro años muy intensos, de trabajo y esfuerzo, pero también de satisfacción y recompensa. En esta etapa de mi vida he afrontado experiencias que nunca antes hubiera imaginado y que me han ayudado a mejorar, tanto a nivel personal como profesional. Sin embargo, esto no hubiese sido posible sin el respaldo de las personas que me han acompañado en cada paso de este camino. A todas ellas, quisiera expresar mi más sincero agradecimiento.

A mis directores, José I. Aliaga Estellés y Enrique S. Quintana Ortí, ya que sin su colaboración hoy no estaría escribiendo estas líneas. A José, por su dedicación, por su ayuda, por su constancia, y su ánimo en mis malos momentos. A Enrique, por haber confiado en mí desde el primer momento, por su apoyo, por ser una fuente infinita de conocimientos y un trabajador incansable.

A la Universitat Jaume I, Generalitat Valenciana y Comisión Europea por su soporte económico durante estos cuatro años, y especialmente al *Ministerio de Educación, Cultura y Deporte* por la financiación de esta investigación a través del programa FPU, sin la cual este trabajo no hubiese sido posible.

Al Centro Nacional de Supercomputación de Barcelona (BSC) por permitirme utilizar su supercomputador MareNostrum en mi investigación. Especialmente, dar las gracias a Rosa M. Badía y al personal investigador del BSC por su colaboración desinteresada.

A las personas del grupo HPC&A de la Universitat Jaume I y a las que en algún momento han estado en él: Rafa M., José Manuel, Sergio B., Asun, Maribel, Juan Carlos, Germán L., Germán F., Merche, Alfredo, Toni, Fran, Manel, Ruymán, Rafa R., José Antonio, Sandra, Sergio I., Héctor, Adrián, Sisco, Sonia, Rocío y Goran. Gracias a todos por vuestra colaboración y el buen ambiente que hemos creado. No quisiera olvidarme de los técnicos del Departamento de Ingeniería y Ciencias de la Computación, Gustavo y Vicente. Gracias por vuestra ayuda y paciencia.

A los compañeros del *Institut Computational Mathematics* de la *Technische Universität Brawnschweig*, especialmente a Matthias Bollhöfer, por ofrecerme la oportunidad de integrarme en su entorno de trabajo y por su contribución en la tesis. Asimismo, me gustaría dar las gracias por su hospitalidad a los compañeros del grupo *Alpines* de INRIA-Paris, y a Laura Grigori en particular.

A mis amigos y amigas, por estar a mi lado, servirme de distracción y apoyarme en todas mis decisiones. Aunque no os lo parezca, me habéis ayudado mucho. Gracias!

A mi familia, y muy especialmente a mis padres, Tere y Paco, y a mi hermano Francisco. A mis padres, gracias por darme la oportunidad de recibir la mejor educación, por su comprensión, ayuda y respeto de cada una de mis decisiones, así como por estar ahí siempre, tanto en lo bueno como en lo malo. A mi hermano, gracias por su optimismo, su apoyo y su generosidad.

A Alejandro, el mejor compañero que se puede tener. Gracias por su apoyo y cariño, por su paciencia y comprensión, por su confianza en mí, y en definitiva, por estar siempre a mi lado.

– ¡Gracias! · Gràcies! · Thanks! –

Castellón, diciembre de 2016.

CHAPTER 1

Introduction

1.1 Motivation

The efficient solution of large sparse systems of linear equations is a key problem arising in many scientific and engineering applications and, more recently, in data analytics. On the one hand, one of the most important engineering problems yielding to large sparse systems of equations is the discretization of finite elements for Partial Differential Equations (PDEs). This problem underlies, among others, the analysis of the resistance of concrete structures, the estimation of the electrons' orbit, the evaluation of the Earth's gravitational field, the simulation of the behavior of structural aircraft components, or the detection of occlusions in blood vessels. On the other hand, the connection between sparse linear algebra and graph algorithms has turned the former into an appealing means to mine the vast amount of information in social networks, and other data analytic processes, such as web search engines, information retrieval, or the creation of economic models. These examples are only a small fraction of the applications that involve sparse linear systems. In addition, the solution of sparse linear systems has traditionally been the bottleneck in computer simulations, and remains in that position today, when the use of 3D-models has increased the size of the systems and the simulation time and energy consumption.

The interest of the afore-mentioned applications has led to the design and high-performance implementation of numerous storage formats, algorithms, and libraries to efficiently solve sparse linear algebra problems in GPPs, following the evolution of computer architectures.

In the early 2000s, the limitations of instruction-level parallelism and the increasing performance gap between the processor and memory, among other factors, promoted the design and use of processors composed of several cores, in order to improve the performance of the computers. Thus, in 2001 IBM released the first GPP that featured multiple processing cores on the same CMOS: the IBM POWER4 processor [132, 139]. Ever since, multicore processors have become the mainstream to improve the throughput for high-end computing. More than 15 years after the introduction of this type of architectures, the performance gains delivered by each new chip generation maintain a linear growth rate as Moore's law, which states that the number of transistors in a dense integrated circuit doubles approximately every two years, still appears to be valid. However, in order to keep up with this conjecture during the last decade, the Very-Large-Scale Integration (VLSI) scaling rules for processor design had to be dramatically changed. During this period, a different variety

of multicore processors have been developed: symmetric, asymmetric, dynamic, heterogeneous, etc. Nowadays, the term multicore processor comprises two different approximations: on the one hand, GPPs, as those developed by Intel, ARM or AMD; on the other hand, hardware accelerators such as the Graphics Processing Units (GPUs) from NVIDIA and AMD/ATI, the Intel Xeon Phi, Field-Programmable Gate Arrays (FPGAs), etc. All these HPC architectures enable the solution of complex applications by aggregating a number of multicore processors. As a consequence, developers face the challenge of implementing parallel algorithms that efficiently exploit the concurrency of the hardware, usually consisting of a large number of cores.

Unfortunately, the advances in the number of transistors that can be integrated in a circuit, following Moore's law [139], have not rendered a proportional reduction of the power dissipated by the CMOS technology, leading to the end of Dennard's scaling law [66]. This fact made necessary the transition from complex-single core architectures, with high operating frequencies, to multicore processors with moderate frequencies, and more recently, it has contributed to the adoption of the hardware accelerators due to their favorable relation between power dissipation and computational performance.

Nowadays, as we progress on the road to Exascale systems, the *power wall* stands as a crucial challenge that the HPC community needs to address [53, 77], due to the inability to dissipate heat in CMOS circuits operating at a high frequency [78, 81, 77]. Figure 1.1 illustrates this situation. Although this plot shows a continuous increment in the number of cores, it also exhibits that, in general, the operating frequency is sacrificed so that the dissipated power by the processor does not overtake 150 W. Moreover, power consumption implies a high economic cost, around a million of euros per MegaWatt (MW) and year. A simple calculation shows that an Exaflop system, built from current hardware technology, would dissipate between 150 and 165 MWatts, rendering it economically unfeasible [5, 7, 70, 78, 80, 92, 129]. In other words, even with the rate of improvement in power efficiency enjoyed by supercomputers during the past few years [5], the target power consumption of 20 MWatts for an Exascale computer will still be exceeded by a factor of 8× by 2018–2020. Moreover, energy consumption results in carbon dioxide emission, a danger for the environment and public health, and heat, which reduces the reliability and lifetime of hardware components. Therefore, if we have to overcome the power barrier, a holistic power/energy-aware approach is needed, in particular one that aims at developing more energy-efficient hardware as well as energy-aware system software, communication and computational libraries, and applications.

Despite the importance of energy consumption, few software developers take it into account in their implementations. In our view, a reasonable principle to develop energy-efficient software is to put this metric on par with productivity and performance. At this point, it is worth reminding that the energy consumption results from the product between execution time and (average) power dissipation rate. Therefore, an increase of power (due, for example, to a change in the processor frequency) can result in higher or lower energy consumption, depending on the variation of the execution time.

In this dissertation we consider the solution of large sparse systems of linear equations using preconditioned iterative methods based on Krylov subspaces. Concretely, we focus on the ILUPACK[1] library, which provides multi-level ILU preconditioners for the effective solution of linear systems, especially those arising from the discretization of PDEs. ILUPACK has been successfully applied to several large-scale application problems, in particular, to the Anderson model of localization [171] or the Helmholtz equation [48]. The increase of the number of equations in these systems and the introduction of new HPC architectures motivated us to develop a parallel version of ILUPACK

[1] Available at `http://ilupack.tu-bs.de`.

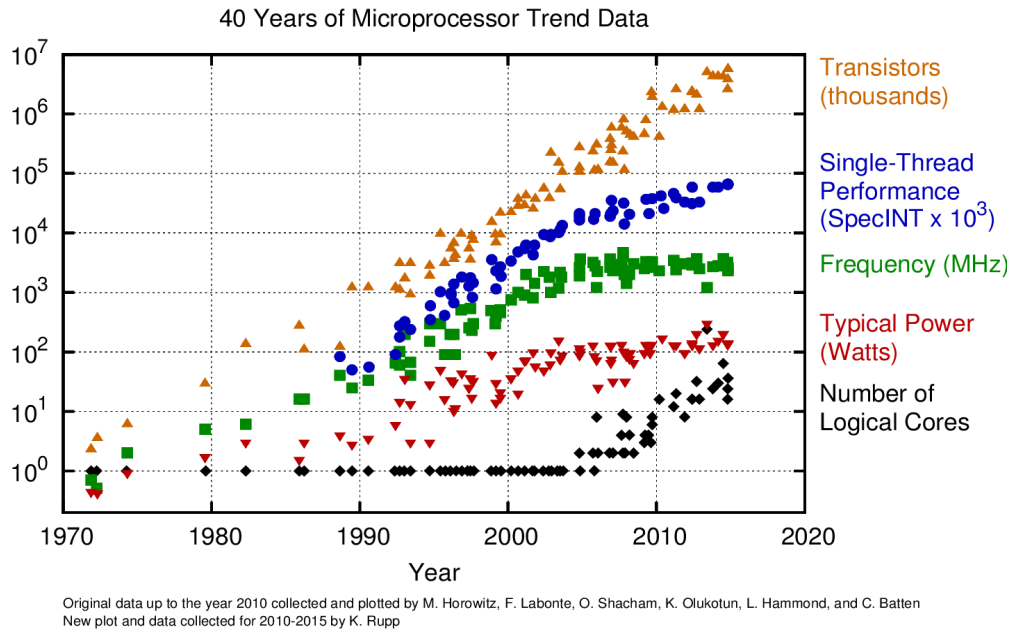


Figure 1.1: Evolution of the processors during the last 40 years in the number of transistors, single-thread performance, operating frequency, dissipated power, and number of cores. [8]

which optimizes *both time and energy on current multicore architectures and clusters of nodes built from this type of technology.*

1.2 State-of-the-art

In this section we revisit some of the most relevant parallel software for solving sparse linear systems. Although there is a vast variety of packages for this purpose, we focus on the most popular parallel methods, grouping them into direct solvers, iterative solvers, and a mix of both. All the methods in these collections are targeted to massively-parallel HPC computers. In addition, as the energy consumption is an important issue that we consider in our research, at the end of this section we revisit different energy optimization techniques which can be applied in scientific computing in general and the solution of sparse linear systems in particular.

1.2.1 Direct Solvers

Let us consider a linear system of the form $Ax = b$ where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix, $b \in \mathbb{R}^n$ is the vector of independent terms, and $x \in \mathbb{R}^n$ is the sought after solution. Direct methods for the solution of linear systems compute a decomposition of A into two triangular factors L and U, of the same dimension as A, so that $A = LU$. This transforms the initial problem into a simpler one whose solution is easy to obtain. Later in the dissertation, we explain the intrinsics of these methods. Table 1.1 lists different software packages for solving sparse linear systems using direct methods, and we provide a short description of the most relevant ones next.

PaStiX

PaStiX [160, 101], developed by the Bacchus team from INRIA, is a multi-threaded library for the solution of huge linear systems of equations. The solution of the system is computed

Name	Types of matrices accepted	Language
PaStiX	SPD, unsymmetric	C/Fortran
PARDISO	Symmetric, unsymmetric	C/C++/Fortran/Matlab
SuperLU	Unsymmetric	C
MUMPS	SPD, symmetric and sym-pat	C/Fortran/Matlab/Scilab

Table 1.1: List of software based on direct solvers for the solution of sparse linear systems. sym-pat denotes a problem with symmetric nonzero pattern but unsymmetric data values.

using different methods depending on the matrix nature. For example, if the matrix is symmetric positive definite (SPD), the Cholesky or Cholesky-Crout with or without numerical pivoting is employed [88]. However, if the matrix is unsymmetric, it uses the LU decomposition with static pivoting to solve the system.

This library solves the system following a sequence of steps. First it reorders the unknowns in order to reduce the fill-in induced by the decomposition, and then applies a symbolic factorization to predict the structure of the factors. Upon completion of these steps, it distributes the matrix blocks among the processors, applies the decomposition of A, and solves the system (top-down). Finally, if necessary, it refines the solution using different iterative methods such as GMRES or CG [88]. This refinement is only applied if the precision of the result is insufficient.

PARDISO

The package PARDISO [172, 173, 121] contains high-performance, robust, memory-efficient and easy-to-use software for solving large sparse symmetric and nonsymmetric linear systems of equations on shared-memory and distributed-memory architectures. In order to improve the performance of the sequential and parallel sparse numerical factorization, the algorithms are based on a Level-3 BLAS update, and pipelining is exploited with a combination of left- and right-looking supernode techniques. The parallel pivoting methods allow complete supernode pivoting in order to balance numerical stability and scalability during the factorization process. The approach relies on Open Multi-Processing (OpenMP) directives and MPI parallelization, and has been successfully tested on many shared-memory parallel systems.

PARDISO calculates the solution of a set of sparse linear equations with multiple right-hand sides, using a parallel LU, LDL^T or LL^T factorization. Moreover, PARDISO 5.0.0 computes the exact bit-identical solution on multicores and cluster of multicores. This package performs different analysis steps depending on the structure of the input matrix A. If the matrix is symmetric, the solver first computes a symmetric fill-in reducing permutation based on either the minimum degree algorithm or the nested dissection algorithm from the METIS package, followed by the parallel left/right-looking numerical Cholesky factorization. The solver uses diagonal pivoting or 1×1 and 2×2 Bunch-Kaufman pivoting for symmetric indefinite matrices and an approximation of the solution is found by forward and backward substitution and iterative refinement. The process for unsymmetric matrices is more complex and is described in [173].

SuperLU

SuperLU [125] contains a set of direct solvers for the solution of large sets of linear equations. This solver is particularly appropriate for matrices with very unsymmetric structure and it contains three different libraries for sequential (SuperLU), shared-memory multi-processors (SuperLU_MT,

Name	Types of matrices accepted	Language
pARMS	Symmetric, unsymmetric	C/Fortran
PETSc	Symmetric, unsymmetric	C/C++/Fortran/Matlab/Python
Hypre	Symmetric, unsymmetric	C/C++/Fortran/Python
ILUPACK	Symmetric and/or Hermitian, general	C/C++/Fortran/Matlab

Table 1.2: List of software based on iterative solvers for the solution of sparse linear systems.

based on Pthreads) and message-passing architectures (SuperLU_DIST, using MPI). All the libraries use variations of the LU factorization optimized to take advantage of the sparsity pattern and the target computer architecture.

MUMPS

MUMPS [6] is a package for solving systems of linear equations, where the coefficient matrix is sparse and can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed-memory computers. This package implements a direct method based on a multifrontal approach that computes an LU factorization. If the matrix is symmetric then the package computes a factorization $A = LDL^T$, where D is block diagonal matrix with blocks of order 1 or 2 on the diagonal. The system is solved in three main steps: analysis, factorization, and solution. The first step includes a preprocessing and a symbolic factorization. In the second, the original matrix is first distributed onto the processors depending on the mapping obtained during the analysis, and the numerical decomposition is computed as a sequence of dense factorization steps on so-called frontal matrices (multifrontal approach). After the factorization, the factor matrices are kept distributed (in memory or on disk). In the third step, the solution is obtained by a forward elimination step followed by a backward elimination step. The first elimination step can be performed during the factorization, so that only one of the factors has to be stored, and the third step only requires the backward elimination step. This solution is finally postprocessed, using iterative refinement or backward error analysis, to obtain the solution of the original system.

1.2.2 Iterative Solvers

Iterative methods solve the linear system $Ax = b$ through a sequence of operations which are applied to an initial solution, so that the final solution is progressively approximated. These methods are commonly used to solve large systems of equations and will be described later in this document in quite more detail. Some well-known iterative solvers are introduced in Table 1.2 and are explained below.

pARMs

pARMS [154, 126] is a library of parallel iterative solvers for sparse linear systems of equations developed at the University of Minnesota. The library offers a large selection of parallel preconditioners based on a Recursive Multi-level ILU factorization and solvers based on Krylov subspace approach, using a domain decomposition viewpoint, ranging from simple Additive Schwarz with or without overlapping, to more complex Schur complement techniques. The communications between processes are implemented using MPI.

PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) [35, 155], developed at Argonne National Laboratory, is a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on parallel (and serial) computers, especially those arising from PDEs. PETSc uses MPI for all message-passing communication. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem, and it uses object-oriented programming techniques which provide enormous flexibility for users.

PETSc includes efficient implementations of Krylov subspace methods (GMRES, CG, CGS, Bi-CG-Stab, TFQMR, Richardson,...) and popular preconditioners such as ILU factorization with level of fill-in or based on a magnitude threshold, Jacobi, Additive Schwarz, etc. Moreover, it provides interfaces to access external software as, for example, Trilinos/ML or Hypre.

Hypre

The Hypre [161, 109] library, developed at Lawrence Livermore National Laboratory, offers high performance preconditioners and solvers for large and sparse linear systems on massively-parallel computers. It provides conceptual interfaces which give users a more natural means to describe their linear systems, and provide access to methods such as geometric multigrid which require additional information beyond just the matrix. Its object model is more general and flexible than those in other solver libraries, contributing to the robustness, ease of use, and interoperability. Multigrid preconditioners are a major focus of the library (Algebraic MultiGrid (AMG), Approximate INVerse (AINV),...), but it also provides several of the most commonly-used solvers, such as CG or GMRES, to be used in conjunction with the preconditioners.

ILUPACK

ILUPACK [52, 110] is a software library for the iterative solution of large sparse linear systems. The package, written in C and Fortran, implements a multi-level incomplete factorization approach (multi-level ILU) based on an "inverse-based pivoting" strategy combined with Krylov subspace iterative methods. ILUPACK supports single and double precision arithmetic for real and complex numbers, and it works for symmetric and/or Hermitian matrices that may or not be positive definite and general square matrices. Apart form the standard reordering routines to reduce fill-in, this library also includes ARMS multi-level reordering strategies [168], as well as reordering routines to locate the greater elements in the main diagonal [49, 73, 99]. These strategies are very useful to improve the numerical stability of the incomplete factorization of symmetric indefinite and general matrices. Prior to our work, there existed two parallel versions of this library: one for shared-memory using OpenMP and an alternative for distributed-memory using MPI.

1.2.3 Direct and Iterative Solvers

The packages described next implement direct and iterative methods to solve large sparse linear systems. The type of method which will be used is chosen depending on the features of the matrices and the needs of the user. In addition, some of them combine both methods.

HiPS

HiPS [82] (Hierarchical Iterative Parallel Solver) is a C-library for solving large sparse linear systems on parallel platforms, using techniques based on the Schur complement. The code, devel-

Name	Types of matrices accepted	Language
HiPs	Symmetric, unsymmetric	C/C++/Fortran
Trilinos	Symmetric, unsymmetric	C/C++
WSMP	Symmetric Positive Definite (SPD), quasi-definite, and indefinite	C/Fortran
PARASPAR	Unsymmetric	C/Fortran

Table 1.3: List of software that combines direct and iterative solvers for the solution of sparse linear systems.

oped by the ScAlApplix team of INRIA Bordeaux, provides a hybrid method which blends direct and iterative solvers. HIPS exploits the partitioning and multistage ILU techniques to enable a highly parallel scheme.

Trilinos

Trilinos [102], developed by the Sandia National Laboratory (EE.UU.), provides a wide-variety of solution methods for linear and eigen systems. The main objective of the Trilinos project is to facilitate the design, development, integration and ongoing support of mathematical software libraries within an object-oriented framework for the solution of large-scale problems. Trilinos is composed of several libraries which have been developed independently and can be also used independently or together with other packages because the software provides interfaces to promote the interoperability of independently developed software.

Trilinos uses a two-level software structure designed around collections of *packages*. A Trilinos *package*, in the first level, is an integral unit usually developed by a small team of experts in particular areas such as algebraic preconditioners, nonlinear solvers, etc. In addition, there is a second level composed of the packages that exist underneath the top level, which provide a common look-and-feel, including configuration, documentation, licensing, and bug-tracking. The Ifpack package provides object-oriented interfaces for Jacobi preconditioners based on the ILU factorization; the ML package for AMG-based preconditioners based on smoothed aggregation; and the Belos package contains implementations based on the Krylov subspace methods. The main computational parts are written in C, the interfaces to access these parts in C++, and globally, Trilinos uses MPI as the message-passing library.

WSMP

Watson Sparse Matrix Package (WSMP) [95, 94, 96], developed by the IBM T. J. Watson Research Center, is a collection of algorithms for large sparse systems of linear equations. This high-performance, robust, and easy-to-use software can be used as a serial package, or in shared-memory and distributed-memory environments. In the distributed-memory environment each process can be either serial or multithreaded. WSMP can be used for symmetric systems, as well as general systems, using direct and iterative methods. For symmetric positive definite systems, WSMP leverages a modified version of the multifrontal algorithm for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm. For the solution of general sparse systems, WSMP employs a modified version of the multifrontal algorithm for matrices with an unsymmetric pattern of nonzeros. Moreover, WSMP supports threshold partial pivoting for general matrices with a user-defined threshold.

PARASPAR

PARASPAR [188] is a parallel package for large and sparse linear systems of unsymmetric matrices. Both direct and preconditioned iterative methods are used. The direct methods are based on the Gaussian elimination with three pivoting strategies, and the iterative methods employed are a modified ORTHOMIN algorithm, CGS, BI-CGSTAB and TFQMR. The preconditioners are computed via an approximate LU factorization, which is obtained by dropping small non-zero elements during the Gaussian elimination. The accuracy of the preconditioners can be automatically improved if they are not sufficiently accurate.

1.2.4 Energy Optimization

In general, most of the parallel solvers described above are designed to improve performance, but they are oblivious about energy efficiency. This is due to the common belief that an improvement in performance implies also an upgrade in energy efficiency. However, this fact is not always true, because the energy consumed by an application not only depends on the execution time. There are several factors to take into account in order to optimize energy efficiency.

We next describe different techniques that can be used to improve the energy efficiency of a scientific application, while maintaining the performance:

Dynamic Voltage-Frequency Scaling (DVFS). A number of energy reduction strategies act on the processor frequency, leveraging slack periods in the computation [31, 128, 163, 169, 187]. Dynamic Voltage and Frecuency Scaling (DVFS) is a framework to change the processing frequency and/or operating voltage of a processor, based on system performance requirements at the given point of time, to achieve the best performance or lowest power dissipation. Unfortunately, the consumption adjustment by acting on the voltage-frequency provide limited improvements in energy efficiency, in general.

Undervoltage. An alternative to the use of DVFS consists in operating in the undervoltage band [10, 33], reducing the voltage while maintaining a constant frequency, with the consequent reduction of dissipated power. However, this approximation can unleash errors which decrease performance and increase of energy consumption due to their treatment. The balance between these factors has been initially studied in [180, 29].

Communication-avoiding strategies. In applications with irregular access patterns, the data movements consume significant amounts of time and energy, which can exceed the computational cost [78]. The reduction of these costs can be achieved by developing 2.5D/3D memory technologies and exploiting near-data computing [100]. From the application point of view, during last decade, the communication costs have been reduced in numerous basic dense matrix computing operations, direct and iterative methods for sparse linear systems, and tensors [65]. Theoretical studies consider parallel message-passing systems, with a hierarchy of shared-memory or, in some cases, a mixed configuration. Communication-avoiding algorithms have been developed for a subset of the above-mentioned applications, which accomplish the communication theoretical levels, on a variety of these architectures [65]. In the iterative solution of sparse linear systems it is possible to reduce the amount of communications with a reorganization of the resolution schemes, that change the numerical properties of the method [105], or by applying segmentation techniques [179]. An efficient alternative for iterative methods hides the temporal cost of the communications by overlapping them with other computations [105].

Approximate computing. Contemporary processors lack the necessary power to exactly model numerous real-world phenomena. For example, weather simulations or climate change studies are based on approximate models of the real phenomenon. Besides, the amount of information managed by the data centers will increase by a factor of 50 in the next years, while the computational capacity of the centers will only increment in a factor of 10 [83]. Under these conditions, approximate computing exploits the differences between the level of accuracy required by the application and the precision supported by the circuitry (IEEE 754 standard) to reduce the energy consumption [137, 185]. Among the variety of approximate computing techniques, we point out precision scaling, memoization and task elimination. Iterative refinement is a well-known case of approximate computing strategy to solve linear algebra problems using simple-precision arithmetic (32 bits), in most of the computations, to obtain a double-precision result (64-bits).

1.3 Objectives

The main goal of this thesis is *the design, implementation and evaluation of parallel and energy-efficient iterative sparse linear system solvers for multicore processors as well as recent manycore accelerators such as the Intel Xeon Phi.*

Prior to our work, there existed a parallel version of ILUPACK for shared-memory multiprocessors which used an *ad-hoc* runtime based on OpenMP [131]. However, this solution strongly coupled the numerical algorithm with an ad-hoc runtime in charge of exploiting task-parallelism. To address this problem, we aim to develop a task-parallel version of ILUPACK which can exploit considerable levels of thread-concurrency and requires minor changes in the numerical algorithm. The advantages of this proposal reside in that it limits the amount of modifications in ILUPACK's legacy code and enables a parallelization scheme which can be leveraged to implement different versions of the solver based on the runtime framework OmpSs [149], MPI, and a combination of both. Besides, the parallelization scheme can be also applied to easily parallelize other ILU-type iterative solvers.

To achieve the main purpose, the work is divided in several specific objectives which are summarized as follows:

Develop an automatic power-performance analysis framework. As energy consumption is an important issue to tackle nowadays in our implementations, we develop a library, called PMLib, which traces the use of power made by the applications. This is completed with a powerful inspection tool that automatically detects the power sinks of the applications. This tool will help developers to identify the power bottlenecks in the applications, so that they will be able to focus their efforts to address them. We will use this tool in our dissertation to analyze the power consumption of ILUPACK.

Study and analyze ILUPACK. As a starting point, we investigate ILUPACK, a library for the iterative solution of sparse linear systems based on Krylov subspaces, which exploits inverse-based ILUs to control the growth of the inverse triangular factors. In this study we go in detail to investigate the possibilities to efficiently extract additional concurrency for multicore processors.

Leverage task-parallelism in ILUPACK with OmpSs. The initial goal is the design, implementation, and evaluation of an optimized version of ILUPACK, which will be used in NUMA platforms and manycore architectures such as the Intel Xeon Phi. This implementation exploits task-parallelism using OmpSs to improve performance.

Exploit the interoperability between message-passing and OmpSs. In order to execute ILUPACK on distributed-memory clusters of multicore processors, we design, implement, and evaluate a concurrent version which exploits the interoperability between MPI and OmpSs, efficiently leveraging the resources inside each processing node.

Evaluate the energy-efficiency. We analyze the energy costs of different implementations of ILUPACK, using the PMLib library. In the evaluation we may consider several factors: hardware, DVFS, Dynamic Concurrency Throttling (DCT), etc.

1.4 Structure of the Document

The remainder of this thesis describes the research that has been undertaken to fulfill the goals stated above. Concretely, this manuscript is organized in six chapters. We next provide a brief introduction to each of them:

Chapter 1 introduces the thesis. This chapter provides the motivation, and reviews the state-of-the-art, objectives and organization of the document.

Chapter 2 presents an integrated framework for power-performance analysis of parallel scientific workloads (PMLib). Moreover, this chapter describes an inspection tool that uses the PMLib framework to automatically identify power sinks in the applications.

Chapter 3 summarizes different methods for solving sparse linear systems and the most significant preconditioning techniques. Furthermore, this chapter introduces ILUPACK, the numerical package for solving large systems of equations that is targeted in this thesis.

Chapter 4 describes how to extract task-parallelism in the PCG method in ILUPACK. In addition, this chapter presents different parallel implementations of the solver using OmpSs, MPI and a combination of both.

Chapter 5 analyzes the energy-efficiency of the parallel versions of ILUPACK developed as part of the thesis in different hardware architectures. This study considers a variety of factors that can influence in the energy consumption.

Chapter 6 presents the main conclusions derived from this research, and provides future directions to extend the work presented in this dissertation. Finally, it offers a list of publications derived from this thesis.

CHAPTER 2

Automatic Power-Performance Analysis Framework

The considerable benefits that Exascale computing are expected to yield for crucial scientific disciplines, from biology to nuclear engineering, are also anticipated to exert a positive impact on industrial competitiveness that outweighs the costs of this technology by far [30, 43]. However, a number of recent studies [30, 70, 75] have identified power consumption as one of the key challenges to be able to assemble efficient hardware scaling beyond Petascale. The power wall is now recognized as a crucial challenge that the HPC community will have to face [70, 78, 80]. A variety of signs of this trend range from the energy efficiency regulatory requirements set by the US Environmental Protection Agency to the biannual elaboration of the Green500 list [5] and the ongoing standardization effort of this ranking. While in the past decade, HPC facilities have enjoyed considerable improvements in the power-performance ratio [5] —mostly due to the deployment of heterogeneous platforms equipped with hardware accelerators (e.g., NVIDIA and AMD graphics processors, Intel Xeon Phi) or the adoption of low-power processors (IBM PowerPC A2, ARM chips, etc.)— much remains to be done in terms of energy efficiency to render Exascale systems feasible by 2020.

In the last years, power-saving technologies and mechanisms originally designed for embedded and mobile appliances have been increasingly embraced by the designers of desktop and server systems. Nevertheless, awareness in software power efficiency still lags much behind [11, 170], despite the energy waste that an ill-behaved application can infer. Indeed, tracing the consumption of power made by scientific applications and workloads is key to detect energy bottlenecks and understand power distribution. However, as of today, the number of fully integrated tools for this purpose is insufficient to satisfy a rapidly increasing demand.

In this chapter, we present an integrated framework for power-performance analysis of parallel scientific workloads which has been developed as part of this dissertation [25, 36]. The framework leverages recent advances from vendors of desktop and server systems that enhance their systems with on-board sensors to obtain fine-grain energy measurements of on-core hardware components. Intel introduced these sensors, called RAPL [113, 114], with their Sandy Bridge microarchitecture. Following this trend of exposing the power measurements from the systems, NVIDIA presented the NVML interface for monitoring and managing various states of GPU devices, and Intel also defined the MIC Management Library for controlling and configuring several metrics of the Intel Xeon Phi coprocessor platform. Concretely, in this work we have designed a power-performance analysis framework to obtain power/energy samples from RAPL sensors, GPU devices and Xeon

Phi coprocessors. Furthermore, the key contribution that complements the framework is a powerful inspection tool that automatically identifies power sinks [37].

This chapter is organized as follows. In Section 2.2, we describe the performance-power analysis framework, the instrumentation and visualization tools integrated with it, and the PMLib library [36]. In Section 2.3 we describe the contributions of this work related with the framework. Section 2.4 introduces a major extension of the framework consisting of a tool to automatically detect the power sinks. There we also illustrate the information provided by the framework and the potential of the automatic tool to detect power bottlenecks with two detailed examples. Finally, we provide some concluding remarks in Section 2.5.

2.1 Integrated Tools

The power-performance analysis framework leverages several tools described in this section. In particular, the framework interacts with the `Extrae` [79] instrumentation tool to produce performance traces; and these traces can be visualized, simultaneously with the power traces, with `Paraver` [153]. Besides, the framework obtains traces of power-related states defined by the Advanced Configuration and Power Interface specification (ACPI) [107].

2.1.1 Instrumentation and visualization tools

The use of `Extrae` and `Paraver` offers an enormous potential of analysis, both qualitative and quantitative, allowing to identify the actual performance bottlenecks of parallel applications. The microscopic view of the program behaviour that the tools provide is very useful to optimize the parallel program performance.

Paraver

`Paraver` [153], developed at Barcelona Supercomputing Center (BSC), is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif Graphical User Interface (GUI). `Paraver` was developed responding to the basic need of having a qualitative global perception of the application behaviour by visual inspection, to be able to focus on the detailed quantitative analysis of the problem. `Paraver` provides a large amount of information that directly improves the decisions on whether and where to invest the programming effort to optimize an application. The result is a reduction of the development time as well as the minimization of the human resources. Some features of `Paraver` are its support for the following [153]

- Detailed quantitative analysis of program performance.
- Concurrent comparative analysis of multiple traces.
- Fast analysis of very large traces.
- Mixed support for message passing and shared memory.
- Easy personalization of the semantics of the visualized information.

One of the main features of `Paraver` is the flexibility to represent traces coming from different environments. Traces are composed of state transitions, events, and communications with an associated timestamp. These three elements can be used to build traces that capture the behaviour along time of very different kinds of systems. The `Paraver` distribution includes, either in its

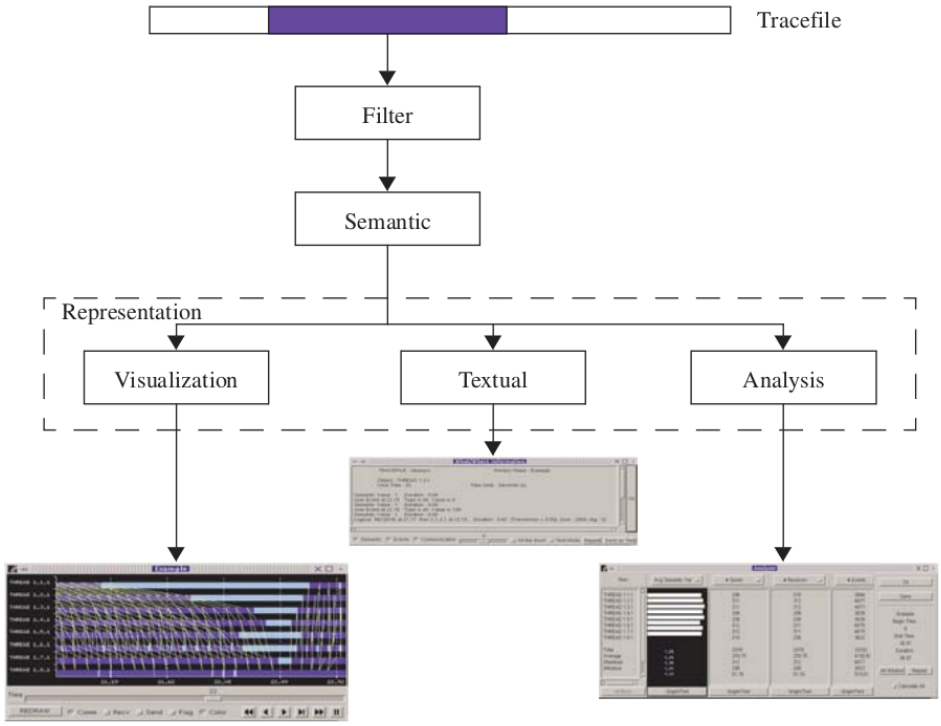


Figure 2.1: Paraver Internal Structure.

own distribution or as an additional package, the instrumentation tools for sequential and parallel application tracing, as well as for system activity tracing in a multiprogrammed environment.

`Paraver` allows users to develop their own tracing facilities according to their own interests and requirements. The possibilities offered by the visualization, semantic and quantitative analyzer modules are powerful, enabling users to analyze and understand the behaviour of the traced system. `Paraver` also allows to customize some of its parts as well as to plug-in new functionalities.

Expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of `Paraver`. The clear and modular structure of this tool plays a significant role towards achieving these targets.

The structure of `Paraver` consists of three levels of modules (see Figure 2.1). At the top, the **Filter Module** (FM) works onto the trace file. This module offers a partial view of the trace file to the next level. In the middle, the **Semantic Module** (SM) receives the trace file filtered by the previous module and interprets it. This module transforms the record traces to time-dependent values which will be passed to the **Representation Module** (RM). The SM is the most important level because it extracts and gives sense to the record values in the trace file. This file contains a significant amount of information that this module can selects (the semantics of the trace file). Finally, at the bottom, the RM receives the time-dependent values computed by the SM and displays it in different ways. The RM drives thus the whole process and offers a graphical display of the trace file.

Extrae

`Extrae` [79] is a dynamic instrumentation and measurement package developed at BSC. It is devoted to generate `Paraver` trace-files for a post-mortem analysis. This package traces programs

Supported programming models	Supported platforms
MPI	Linux clusters (x86 and x86-64)
OpenMP*	BlueGene/Q
CUDA*	Cray
OpenCL*	nVidia GPUs
pthread*	Intel Xeon Phi
OmpSs*	ARM
Java	Android
Python	

Table 2.1: Programming models and systems supported by `Extrae`. * Also available in conjunction with MPI.

compiled and ran in the shared memory model (like OpenMP and pthreads), the MPI, or both programming models (different MPI processes using OpenMP or pthreads within each MPI process). It is currently available for different platforms and operating systems; see Table 2.1.

`Extrae` uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance.

Interposition mechanisms. `Extrae` takes advantage of multiple interposition mechanisms to add monitors into the application. Independently of which mechanism is employed, the goal is the same: collect performance metrics at known application points to finally provide the performance analyst a correlation between performance and the application execution. The interposition mechanisms used by `Extrae` are [79]:

- Linker preload (`LD_PRELOAD`). Most of the current operating systems allow injecting a shared library into an application before the application is actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries, such symbols can be wrapped in order to inject code in these calls. In Linux systems this technique is commonly implemented by using the `LD_PRELOAD` environment variable. `Extrae` contains substitution symbols for many parallel runtimes, such as OpenMP (either Intel, GNU or IBM runtimes), pthread, Compute Unified Device Architecture (CUDA) accelerated applications, and MPI applications.
- DynInst. DynInst is an instrumentation library that allows to modify the application by injecting code at specific code locations. Although originally it allowed modifying the application code when the application was run, now it supports rewriting the binary of the application so that the code injection is required only once. `Extrae` uses DynInst to instrument different parallel programming runtimes, such as OpenMP (either for Intel, GNU or IBM runtimes), CUDA accelerated applications, and MPI applications. DynInst also offers `Extrae` the possibility to instrument user functions by simply listing them in a file.
- Additional instrumentation mechanisms. `Extrae` also takes advantage of some parallel programming runtimes that have their own instrumentation (or profile) mechanisms available for performance tools. These include MPI, which provides the Profile-MPI (PMPI) layer, as well as the CUPTI infrastructure to get information from CUDA devices, OmpSs or even the Open Computing Language (OpenCL) profiling capabilities.

There are some compilers that allow instrumenting application routines by using special compilation flags during the compilation and link phases.

- `Extrae` Application Programming Interface (API). Finally, `Extrae` offers the user the possibility to manually instrument the application and emit its own events if the previous mechanisms do not fulfill the user's needs.

Sampling mechanisms. `Extrae` can be leveraged to instrument the application code, and it also offers sampling mechanisms to gather performance data. While adding monitors into a specific location of the application provides insights which can be easily correlated with the source code, the resolution of such data is directly related with the application control flow. Adding sampling capabilities into `Extrae` produces performance information for regions of code which have not been instrumented.

Currently, `Extrae` supports two different sampling mechanisms. The first mechanism is based on signal timers, which fire the sampling handler at a specified time interval. The second sampling mechanism uses the processor performance counters to trigger the sampling handler at a specified interval of events. While the first mechanism can provide samples that are totally uncorrelated with the application code, the second mechanism, using the appropriate performance counters, can provide insights of the application behaviour while still presenting some correlation with the application code/performance.

Performance data gathered. The monitors added by `Extrae` collect different types of information. Depending on the placement, each monitor can be instructed to collect specific information. The most common information is:

- Timestamp. When analyzing the behaviour of an application, it is important to have a fine timing granularity (up to nanoseconds). `Extrae` provides a set of clock functions that are specifically implemented for different target machines in order to provide the most accurate possible timing. On systems with daemons that inhibit the usage of these timers, or systems that do not have a specific timer implementation, `Extrae` still uses advanced Portable Operating System Interface (POSIX) clocks to provide nanosecond resolution timestamps with low cost.
- Performance and other counter metrics. `Extrae` uses the Performance API (PAPI) [141] and the Performance Monitor API (PMAPI) interfaces to collect information on the microprocessor performance. With the advent of the components in the PAPI software, `Extrae` is not only able to collect information regarding how the microprocessor behaves, but also allows studying multiple components of the system (disk, network, operating system, among others) and extend the study to the microprocessor (power consumption and thermal information). `Extrae` mainly collects these counter metrics at the parallel programming calls and at the samples. It also allows capturing such information at the entry and exit points of the user routines that were instrumented.
- References to the source code. Analyzing the performance of an application requires to study the code that is responsible for such performance. This way the analyst can locate the performance bottlenecks and suggest improvements on the application code. `Extrae` provides information on the source code that was executed (in terms of name of function, file name and line number) at specific location points such as programming model calls or sampling points.

State	Name	Description	CPUs
C0	Operating State	CPU fully turned on	All CPUs
C1	Halt	Stops CPU main internal clocks via software; bus interface unit and APIC are kept running at full speed	486DX4 and above
C1E	Enhanced State	Stops CPU main internal clocks via software and reduces CPU voltage; bus interface unit and APIC are kept running at full speed	All socket LGA775 CPUs
C1E	–	Stops all CPU internal clocks	Turion 64, 65-nm Athlon X2 and Phenom CPUs
C2	Stop Grant	Stops CPU main internal clocks via hardware; bus interface unit and APIC are kept running at full speed	486DX4 and above
C2	Stop Clock	Stops CPU internal and external clocks via hardware	Only 486DX4, Pentium, Pentium MMX, K5, K6, K6-2, K6-III
C2E	Extended Stop Grant	Stops CPU main internal clocks via hardware and reduces CPU voltage; bus interface unit and APIC are kept running at full speed	Core 2 Duo and above (Intel only)
C3	Sleep	Stops all CPU internal clocks	Pentium II, Athlon and above, but not on Core 2 Duo E4000 and E6000
C3	Deep Sleep	Stops all CPU internal and external clocks	Pentium II and above, but not on Core 2 Duo E4000 and E6000; Turion 64
C3	AltVID	Stops all CPU internal clocks and reduces CPU voltage	AMD Turion 64
C4	Deeper Sleep	Reduces CPU voltage	Pentium M and above, but not on Core 2 Duo E4000 and E6000 series; AMD Turion 64
C4E/C5	Enhanced Deeper Sleep	Reduces CPU voltage even more and turns off the memory cache	Core Solo, Core Duo and 45-nm mobile Core 2 Duo only
C6	Deep Power Down	Reduces the CPU internal voltage to any value, including 0 V Re	45-nm mobile Core 2 Duo only

Table 2.2: Available processor power states (extracted from [184]).

2.1.2 Advanced Configuration and Power Interface

Most current processors, from those designed for mobile devices to other conceived for desktop and HPC servers, adhere now to the ACPI [107], which defines an open standard for device configuration and power management from the operating system.

For our power monitoring purposes, the ACPI specification defines a series of *CPU-processor power states*, collectively known as C-states, that are valid on a per-core basis. They reveal the capability of an idle processor to turn off idle components in order to save power. Processor power states are designated as C0, C1, C2, C3, ..., Cn. When a processor runs in the C0 state, it is executing instructions. In contrast, the C1 through Cn power states are sleeping states where the processor consumes less power and dissipates less heat than in the C0 state. The higher the state is, the deeper the Central Processing Unit (CPU) sleep mode, which means that more components are shut down to save power. Deeper sleep states save more power, but the downside is that they incur higher latency to become operative again (the time the CPU needs to go back to C0). Some states also have submodes with different power saving latency levels. Which C-states and submodes are supported depends on the processor (see Table 2.2), but C1 is always available. Modes C1 to C3 basically cut clock signals inside the CPU, while modes C4 to C6 reduce the CPU voltage. "Enhanced" modes can apply both techniques at the same time.

To preserve power, the Operating System-directed configuration and Power Management (OSPM) places the processor into one of its supported sleeping states when idle. Moreover, in the C0 state, ACPI allows the performance of the processor to be altered through a defined "throttling" process, defining transitions into multiple performance states (P-states), which will be explained later. A diagram of processor power states is provided in Figure 2.2.

When a processor operates in C0 state, it can be in one of several *CPU-performance states* (P-states). While all C-states except C0 are idle states, the P-states are operational states related to the CPU frequency and voltage. Concretely, a high P-state is associated with a low frequency and voltage. The number of P-states is processor-specific and the implementation differs across various types, but P0 is always the highest-performance state, while higher P-states represent slower processor speeds and lower power consumption. For example, a processor in P3 state runs more

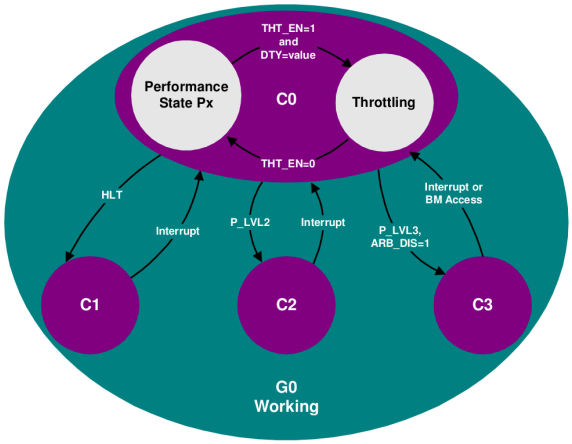


Figure 2.2: Processor power states [107].

slowly and dissipates less power than a processor running at P1 state. To operate at any P-state, the processor must be in C0 state, which implies it is active and not idling. A lower power dissipation does not necessarily imply higher energy savings, as power combines with time to determine energy consumption.

2.2 The PMLIB Framework

PMLIB [25, 36] is a framework of easy-to-use and scalable tools to analyze the power dissipation and the energy consumption of parallel MPI and/or multi-threaded scientific applications.

The framework is composed of two modules/types of daemons: A single external-PMLIB daemon runs in a separate system (the power tracing server), to avoid interfering with the application while collecting samples from different wattmeters attached to the target platform where the application is executed. Furthermore, an internal-PMLIB daemon runs in each node of the target platform, collecting information (like, e.g., the core C-states [107]) that is only accessible locally.

Figure 2.3 illustrates the interaction of the PMLIB framework, a performance tracing suite, and a graphical visualization tool with the target application. The starting point is a concurrent scientific application, instrumented with the PMLIB software, that runs on a parallel target platform (e.g., a cluster, a multicore architecture, or a hybrid computer equipped with one or several GPUs), yielding a certain power consumption. Attached to the application node(s), there may be several wattmeter devices (either internal Direct Current (DC) or external Alternating Current (AC)) that the external-PMLIB daemon steadily samples from the power tracing platform. Calls from the users' code running on the target platform, using the PMLIB API, communicate with the external-PMLIB module in order to instruct the tracing server to start/stop collecting the data captured by the wattmeters, dump the power traces into disk files, etc. Upon completion of the application's execution, the power trace can be inspected, optionally side-by-side with a performance trace, using the appropriate visualization tool.

This framework allows a smooth integration of the PMLIB power-related traces and the performance traces obtained with `Extrae` [79], while the result can be visualized using `Paraver` [182]; see Figure 2.4 for example. However, the modular design of the framework can easily accommodate other tracing suites as, e.g., TAU [164], VampirTrace [9], HDTrace [120], etc.

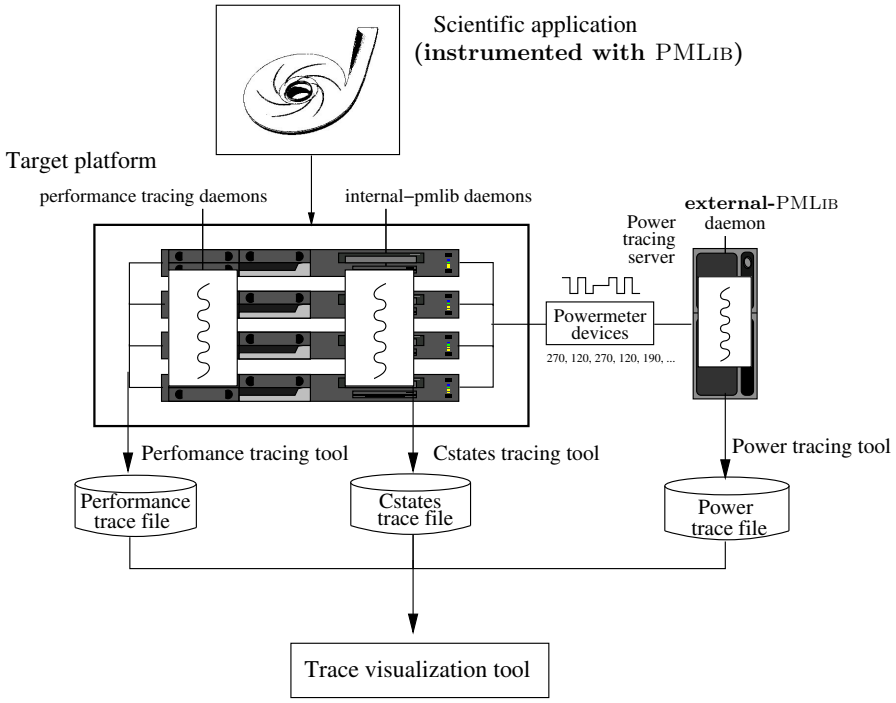


Figure 2.3: Interaction of PMLIB and performance/visualization tracing tools with a parallel scientific workload, producing traces on application performance and power dissipation that become inputs to the visualization tool.

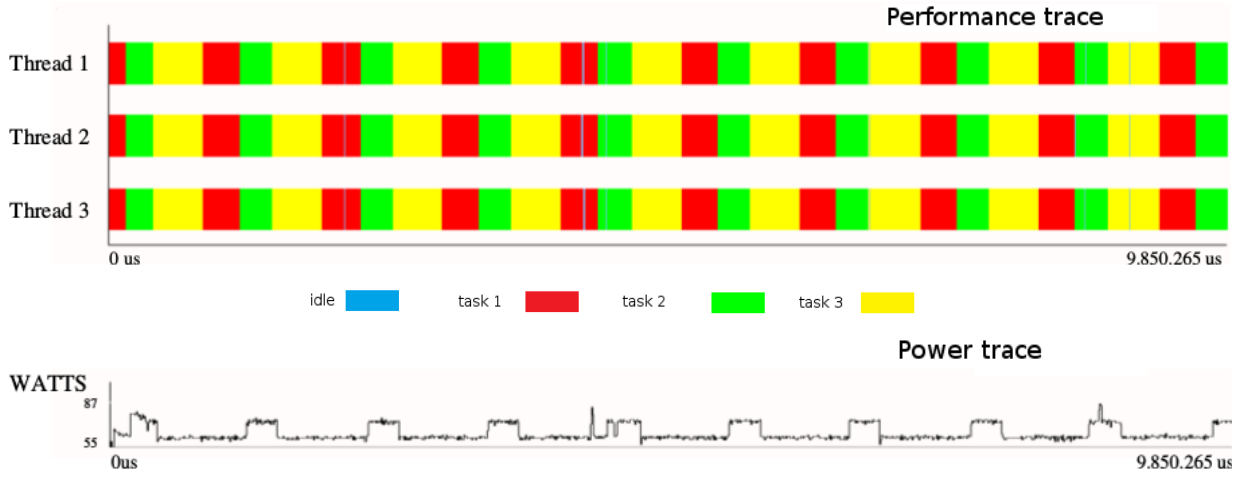


Figure 2.4: Information captured with the PMLIB framework and visualized with `Paraver`: application performance and power traces (top and bottom, respectively).

2.2.1 Hardware power sampling devices

The PMLib package interacts with a number of power sampling devices. The power sampling devices include external commercial products, such as *APC 8653* Power Distribution Unit (PDU) and *WattsUp? Pro .Net*, which are directly attached to the wires that connect the electric socket to the computer Power Supply Unit (PSU), measuring the external AC for the full platform. They also include some internal DC wattmeter designs, consisting of an appropriate choice of current transducers that produce data for a commercial Data Acquisition System (DAS) from NI and, alternatively, other designs that use a microcontroller to sample transducer data. All these devices are described in more detail next, and the connection points are illustrated in Figure 2.5:

External AC wattmeters. The APC 8653 PDU has 24 outlets and operates at a sampling rate of 1 Hz, employing the Simple Network Management Protocol (SNMP) to communicate with the tracing server via Ethernet. The WattsUp? Pro .Net works at 1 Hz, and returns samples to the server through an Universal Serial Bus (USB) 2.0 line.

Powermeter using NI DAS. Measurement devices were developed taking into account that they had to measure currents ranging from 1 to 15 A, without introducing significant voltage drops. That is why the *LEM HXS 20-NP* Hall effect current sensor was selected. The device exhibits high accuracy and linearity, and a very low internal resistance, while being able to measure current in the required ranges.

A set of designs include several channels with each one comprising a transducer that is connected to one of the power lines leaving from the PSU. The final system is a modular design, based on stackable 8-channel components that share power and reference voltage, for a total of 32 current channels.

The DAS is composed of the NI9205 module and the NIcDAQ-9178 chassis. The module features 32 16-bit resolution Analog-to-Digital (AD) channels which can sample data at 7,000 Hz. In principle, the `LabView` software from NI runs in the tracing server, reading the data captured by the DAS from a USB 2.0 port in the chassis. For convenience, a daemon/software was implemented to interact with the chassis, without the need of LabView, enabling a better integration of the device with PMLib.

Microcontroller-based wattmeters. The initial designs featured 10 and 25 channels, plus a Peripheral Interface Controller (PIC) 18 microcontroller from *Microchip*, to perform AD conversion. Each channel consisted of the aforementioned HXS 20-NP transducer and a 10-bit resolution AD channel in the microcontroller. All the channels shared a reference voltage of 2.5 V generated by the transducers. Data was sent to the host computer over an asynchronous RS232 port, and the sampling rate was therefore limited by the speed of the communications link (115,200 bauds in the selected microcontroller).

2.2.2 The PMLib library

The PMLib software package is developed and maintained by the *HPC&A* research group at the *Universitat Jaume I* to investigate the power usage of HPC applications. The current implementation of this package provides an interface to utilize all the afore-mentioned wattmeters and several tracing tools. We next portray the interface of PMLib using a practical example.

Power measurement is controlled from the application using a collection of routines to query information on the power measurement units, create counters associated with a device where power

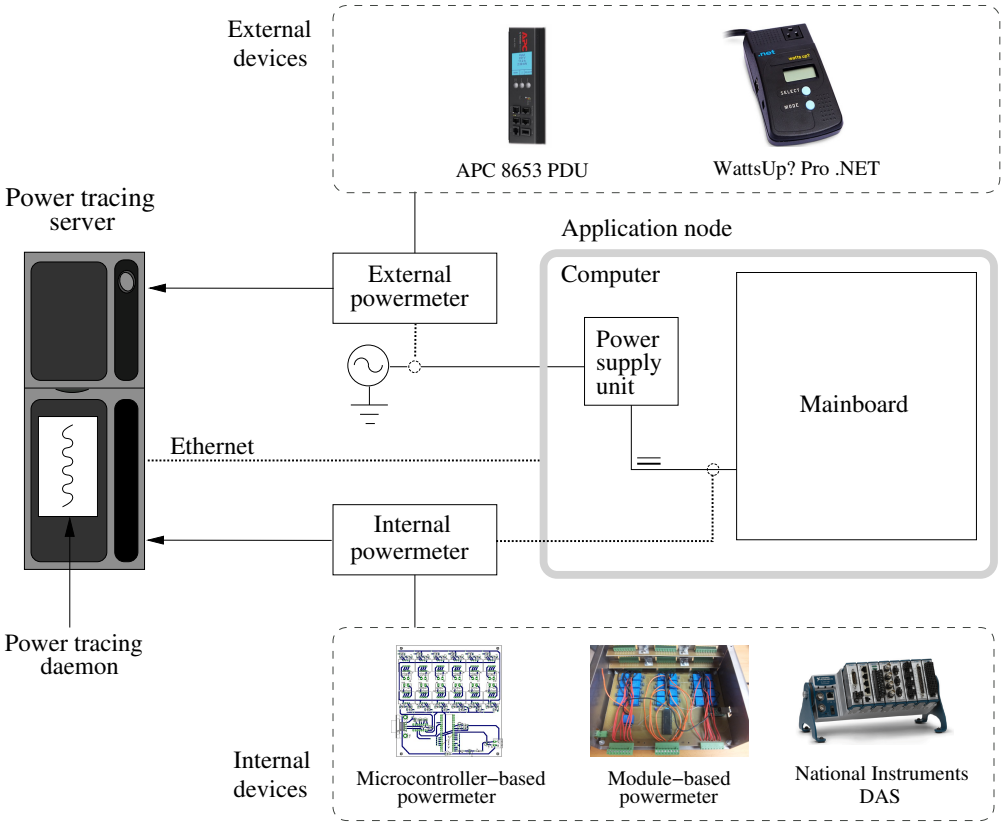


Figure 2.5: Single-node application system and sampling points for external and internal wattmeters.

data is stored, start/interrupt/continue/terminate power sampling, etc. All this information is managed by the PMLib server, which is in charge of obtaining these data from the devices and return the appropriate answers, via the interface of the PMLib routines, to the invoking application (client). This client-server interaction is exposed in Figure 2.6.

Figure 2.7 displays a detailed example illustrating the use of PMLib. The code first declares the most important variables. Next, two server structures are initialized with their respective Internet Protocol (IP) addresses and the port that will be used for the communication with both servers. Here, the first server measures power samples, and it is located in a separated machine to avoid interfering with the parallel application. Moreover, the C-states are recorded using the second server, which is placed in the same machine where the parallel application runs, so that it can collect the requested data from local registers. The invocation to the function `pm_get_devices` establishes a communication with the server in order to obtain a list with the names of wattmeters connected to it. With this information, the call to `pm_get_device_info`, with one of the detected wattmeters, returns specific information on this device.

The next two calls to `pm_set_lines` select the lines to measure (distinct wattmeters may have different numbers of lines) depending on the hardware whose power consumption will be analyzed. Next, the function `pm_create_counter` is called twice, to create one counter associated with the `DCMeter1` wattmeter and a second one that is bound to the C-states. The measurement is initiated and terminated from the application via routines `pm_start_counter` and `pm_stop_counter`, respectively. In this case, we measure the power and record the C-states during the execution of routine `dgemm`. Then, the sampling process is momentarily interrupted, by invoking `pm_stop_counter`, and continued later, by using `pm_continue_counter`, to measure the power for routine `dsyrk`.

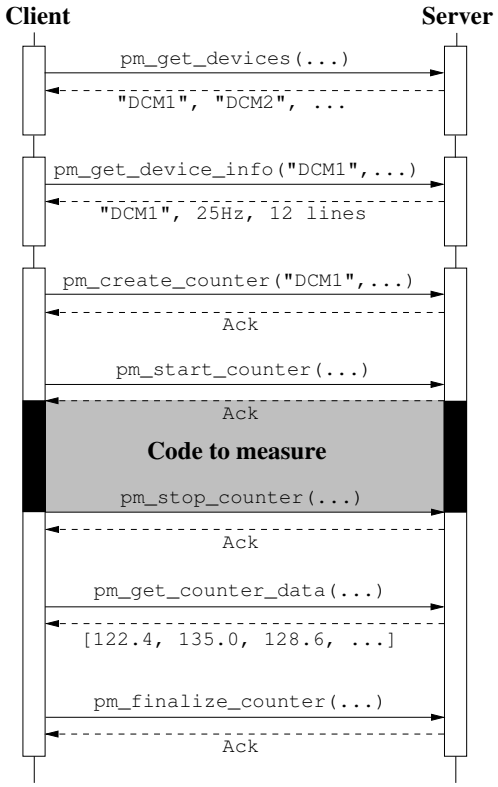


Figure 2.6: Diagram of the communication between client (running a scientific application) and the (PMLIB) server.

After that, `pm_get_counter_data` saves the collected data onto the corresponding counter structure; this information is printed in one of the available formats (in the example, `Paraver` format, by using `pm_print_data_paraver` routine); and, finally, the counters are destroyed using `pm_finalize_counter`.

2.2.3 Module to detect power-related states

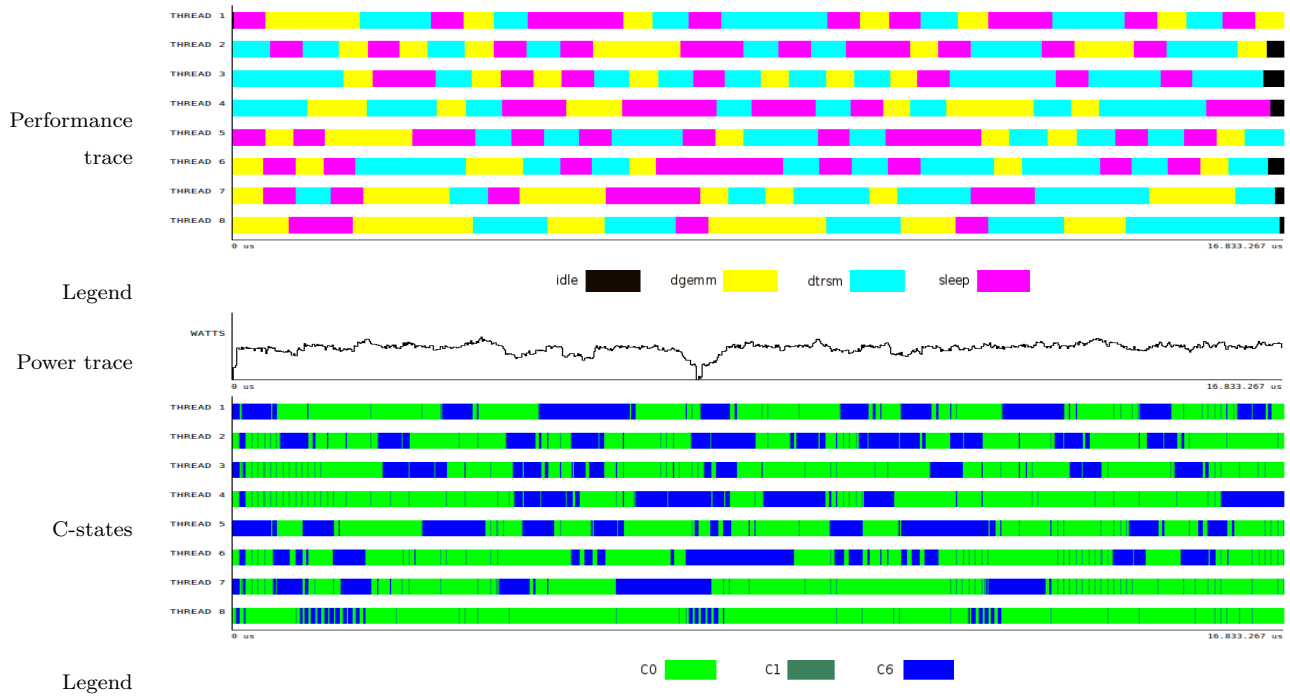
Our power framework obtains a trace of the C-states of each core. In order to gather information on the C-states, a daemon integrated into the power framework accesses the appropriate MSR register for each core and state, with a user-configured frequency. The daemon reads values corresponding to the total time (in microseconds) spent in a certain state. This value is then subtracted from the previous read, normalized, and stored together with a timestamp in a file with a user-selected format.

The state-recording daemon has to run on the same platform as the application and, thus, it introduces a certain overhead (in terms of execution time as well as power consumption) that, depending on the software that is being monitored, can become nonnegligible. To avoid this effect, the user should experimentally adjust the sampling frequency of this daemon with care.

Figure 2.8 offers a graphical example of the information that can be collected with the power-tracing framework, when combined with the performance tracing tool `Extrae` and the visualization tool `Paraver`. The views correspond to the execution of a synthetic parallel benchmark that randomly issues three types of computational kernels: `dgemm` (matrix-matrix product), `dtrsm` (triangular system solve), and `sleep`. The test was run using 8 threads on a platform equipped with two Intel Xeon E5504 cores at 2.00 GHz, and 24 Gbytes of RAM. The performance trace in the

```
int main (int argc, char *argv[]) {
    server_t  server1,  server2;
    counter_t counter1, counter2;
    line_t    lines1,   lines2;
    device_t  disp; char      **list;
    int       i, num_devices, freq1=0, freq2=0, aggr1=1, aggr2=1;
    // ... Some other variables...

    // Initializes the servers' structures
    pm_set_server("150.128.82.30", 6526, &server1);
    pm_set_server("127.0.0.1",     6526, &server2);

    // Query on #devices connected to server1, and obtain handles.
    // Then, output information, e.g., for device[0]
    pm_get_devices(server1, &list, &num_devices);
    pm_get_device_info(server1, list[0], &disp);
    printf("Name: %s \t Max freq: %d \t Number of lines: %d \n",
            disp.name, disp.max_frecuency, disp.n_lines);

    // Select the lines to measure
    pm_set_lines("0-11", &lines1);
    pm_set_lines("0-31", &lines2);

    // Create a counter for powermeter DCMeter1
    pm_create_counter(list[0], lines1, !aggr1, freq1, server1, &counter1);

    // Create a counter for C-states
    pm_create_counter("Cstates", lines2, !aggr2, freq2, server2, &counter2);

    // Start to collect samples: power, C-states
    pm_start_counter(&counter1);
    pm_start_counter(&counter2);

    // Sampled application code fragment
    dgemm( &transa, &transb, &m, &n, &k, &alpha,  &A[k*lda+i], &lda,
           &B[j*ldb+k], &ldb, &beta,   &C[j*ldc+i], &ldc );

    // Stop collecting samples
    pm_stop_counter(&counter2);
    pm_stop_counter(&counter1);

    // ...  Some other nonsampled application code fragment ...

    // Continue to collect samples: only power
    pm_continue_counter(&counter1);

    // Sampled application code fragment
    dsyrk(&transa, &transb, &m, &n, &alpha,  &A[k*lda+i], &lda, &beta,   &C[i*ldc+i], &ldc
    );

    //Stop collecting samples
    pm_stop_counter(&counter1);

    // Dump collected data onto memory
    pm_get_counter_data(&counter2);
    pm_get_counter_data(&counter1);

    // Print power data in Paraver format
    pm_print_data_paraver("out.prv", counter1, lines1, 0, "us");

    // Print C-states data in Paraver format
    pm_print_data_paraver_cstates("cstates.prv", counter2, lines2, 0, "us");

    //Finalize the counters
    pm_finalize_counter(&counter2);
    pm_finalize_counter(&counter1);
    return 0; }
```

Figure 2.7: Example of use of PMLib.



	idle	dgemm	dtrsm	sleep
THREAD 1	0,14 %	26,41 %	37,77 %	35,69 %
THREAD 2	1,78 %	28,32 %	34,22 %	35,69 %
THREAD 3	2,03 %	19,62 %	54,56 %	23,79 %
THREAD 4	1,41 %	28,02 %	40,83 %	29,74 %
THREAD 5	0,10 %	22,32 %	38,87 %	38,70 %
THREAD 6	1,66 %	19,54 %	46,09 %	32,71 %
THREAD 7	0,98 %	36,23 %	38,00 %	23,79 %
THREAD 8	0,57 %	45,19 %	42,34 %	11,89 %
Total	8,68 %	225,65 %	333,68 %	231,00 %
Average	1,08 %	28,21 %	41,71 %	28,00 %
Maximum	2,03 %	45,19 %	54,56 %	38,70 %
Minimum	0,10 %	19,54 %	34,22 %	11,89 %
StDev	0,70 %	8,21 %	5,84 %	8,25 %
Avg/Max	0,53	0,62	0,76	0,75

	C0	C1	C6
THREAD 1	64,28 %	0,17 %	35,55 %
THREAD 2	64,47 %	0,15 %	35,38 %
THREAD 3	76,92 %	0,16 %	22,92 %
THREAD 4	70,54 %	0,12 %	29,34 %
THREAD 5	63,19 %	0,14 %	36,67 %
THREAD 6	71,83 %	0,13 %	28,04 %
THREAD 7	75,56 %	0,21 %	24,23 %
THREAD 8	92,92 %	0,04 %	7,04 %
Total	579,71 %	1,12 %	219,17 %
Average	72,46 %	0,14 %	27,40 %
Maximum	92,92 %	0,21 %	36,67 %
Minimum	63,19 %	0,04 %	7,04 %
StDev	9,14 %	0,04 %	9,12 %
Avg/Max	0,78	0,67	0,75

Figure 2.8: Example of performance and power traces captured by `Extrae` and PMLIB framework, visualized with `Paraver`.

top plot displays the task activity per core; the second plot corresponds to the aggregated power dissipated by the mainboard of the machine, captured with the NI wattmeter operating at 1 KHz; the C-states trace in the third plot represents the variations of the cores between processor states C0, C1, C3 and C6 (with a sampling frequency of 10 Hz). The table at the bottom reports the information contained in the performance and C-states traces in numerical format.

The PMLIB framework also obtains a trace of the P-states of each core. Following the procedure described previously for the C-states, this tool reads the corresponding MSR registers to obtain the required information, and processes them to provide a visual trace which shows the different P-states of a core during the execution of an application.

2.3 Enrichment of PMLib

The PMLib framework has been extended, as part of this dissertation, to accommodate recent processor technology. These enhancements consisted in designing and implementing three new modules which run in the internal-PMLib daemon and interact with the PMLib library to obtain power/energy measurements from the RAPL sensors of an Intel CPU, from the NVML library in NVIDIA GPU devices, and from the MIC management library in Intel Xeon Phi coprocessors.

2.3.1 Running Average Power Limit (RAPL)

Many recent architectures (e.g., Intel Xeon "Sandy-bridge", AMD Opteron "Bulldozer", NVIDIA K20 "Kepler", and IBM Power 7) expose power and/or temperature sensors/models to the programmer. In particular, Intel introduced the RAPL [113, 114] interface with the Sandy Bridge microarchitecture. RAPL is available in newer versions of the Xeon server-level CPUs and provides sensors to measure the power and energy consumption of the CPU-level components listed below:

- RAPL PKG: Complete CPU package.
- RAPL PP0: Processor cores only.
- RAPL PP1: A specific device in the uncore.
- RAPL DRAM: Memory controller.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each socket. Moreover, RAPL sensors can be configured and examined by reading MSRs via RDMSR instructions, which refer to specific register addresses. RAPL divides the platforms into domains, which are physically meaningful for power management. The specific RAPL domains available in a platform vary across product segments. Platforms targeting the client segment support the following RAPL domain hierarchy:

- Package.
- Two power planes: PP0 and PP1.

On the other hand, platforms targeting the server segment support the following RAPL domain hierarchy:

- Package.
- Power plane: PP0.
- Dynamic Random Access Memory (DRAM).

Additionally, each RAPL domain supports the following set of capabilities, with some of them being 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 the duration is domain-specific.

Domain	Power Limit	Energy Status	Policy	Perf Status	Power info
PKG	MSR_PKG_POWER_LIMIT	MSR_PKG_ENERGY_STATUS	RESERVED	MSR_PKG_PERF_STATUS	MSR_PKG_POWER_INFO
DRAM	MSR_DRAM_POWER_LIMIT	MSR_DRAM_ENERGY_STATUS	RESERVED	MSR_DRAM_PERF_STATUS	MSR_DRAM_POWER_INFO
PP0	MSR_PP0_POWER_LIMIT	MSR_PP0_ENERGY_STATUS	MSR_PP0_POLICY	MSR_PP0_PERF_STATUS	RESERVED
PP1	MSR_PP1_POWER_LIMIT	MSR_PP1_ENERGY_STATUS	MSR_PP1_POLICY	RESERVED	RESERVED

Table 2.3: RAPL MSR interfaces and RAPL domains.

- 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 one of the previous capabilities requires specific units in order to describe them. Power is expressed in Watts, Time in seconds, and Energy in Joules. Scaling factors are supplied to each unit to make the information presented meaningfully in a finite number of bits. Units for power, energy, and time are exposed in the read-only `MSR_RAPL_POWER_UNIT` MSR. Each level of the RAPL hierarchy provides a respective set of RAPL interface MSRs. Table 2.3 lists the RAPL MSR interfaces available for each RAPL domain.

Starting from this configuration, it was possible to modify the internal-PMLib daemon to periodically record power information. However, on the Intel architecture these readings are only possible in privileged kernel mode. Hence, we require kernel-level support for energy readings.

In the implementation of the PMLib server, each wattmeter/interface has a specific module to read power. Thus, we created a new module for the RAPL readings with the code shown in Figure 2.9 (for simplicity we only show the code of the module for platforms targeting the server segment. For platforms targeting the client segment we only have to change dram by pp1). The PMLib server is implemented in Python, and each module is a Python class which inherits some attributes of a general class `Device`. Specifically, this class is `RAPLDevice` and we implemented the appropriate functions. The most important one is the `read` function. In this function, at first, we open the MSRs of CPU 0, because we only have one socket (we need to open the MSRs of a CPU of each socket to read). Next, we establish the correct power, energy and time units, reading the `MSR_RAPL_POWER_UNIT` register and applying the corresponding shift. Then, we read the MSR registers (`MSR_PKG_ENERGY_STATUS, MSR_PP0_ENERGY_STATUS, MSR_DRAM_ENERGY_STATUS`) to obtain the energy of all the components (socket, cores and DRAM), and use these values to calculate the uncore energy. Given that these MSRs are updated every millisecond, a wait of the user-configured period (higher than 1 ms) is introduced before reading these values again and subtracting the previous ones to obtain the energy of the interval. We note that the PMLib framework returns a list of power samples recorded at a given frequency. Thus, at the end of the `read` function, we transform the energy samples into power samples. All this process is repeated while the server is running.

The implementation of the RAPL module in PMLib is useful to recover instant power samples and to draw the power trace of an application. However, if we only want to calculate the total energy of an application, we do not need to use PMLib. Instead, we only need to subtract the energy measurements at the beginning and at the end of the application. We developed a specific code for this purpose that can be directly added to an application (see Figure 2.10). To use this code we should take into account that the MSR Energy Status registers have a wraparound time of around 60 seconds. Therefore, if the code takes longer to run, we should read the RAPL counters several times, because otherwise, the total energy calculated will not be correct.

```
# Define the MSR register addresses
MSR_PKG_ENERGY_STATUS    = 0x611
MSR_PP0_ENERGY_STATUS    = 0x639
MSR_DRAM_ENERGY_STATUS   = 0x619
MSR_RAPL_POWER_UNIT      = 0x606

class RAPLDevice(Device.AttachedDevice):

  def read_msr(self, fd, which):
    os.lseek(fd, which, 0)
    return struct.unpack("=Q", os.read(fd, 8))[0]

  def read(self):
    #Create the vectors which store the measurements
    energy_before    = [0] * self.n_lines
    energy_after     = [0] * self.n_lines
    power            = [0] * self.n_lines

    frequency    = self.max_frequency ** -1              # Reading frequency
    fd= os.open("/dev/cpu/%d/msr" % 0, os.O_RDONLY)      # Open MSR registers

    # Read the MSR_RAPL_POWER_UNIT register and select the corresponding bits of it
    result= self.read_msr(fd, MSR_RAPL_POWER_UNIT);
    power_units = 0.5 ** float(result&0xf)
    energy_units= 0.5 ** float((result>>8)&0x1f)
    time_units  = 0.5 ** float((result>>16)&0xf)

    t1=time.time()

    # First energy readings
    energy_before[0]= self.read_msr(fd, MSR_PKG_ENERGY_STATUS) * energy_units # socket
    energy_before[1]= self.read_msr(fd, MSR_PP0_ENERGY_STATUS) * energy_units   # cores
    energy_before[2]= self.read_msr(fd, MSR_DRAM_ENERGY_STATUS) * energy_units # dram
    energy_before[3]= energy_before[0] - (energy_before[1] + energy_before[2])  # uncore

    # Wait frequency seconds
    w= frequency - (time.time()-t1)
    if (w > 0): time.sleep(w)

    # This loop is repeated until we stop the server
    while self.running:
      t1=time.time()

      # Energy readings
      energy_after[0]= self.read_msr(fd, MSR_PKG_ENERGY_STATUS) * energy_units
      energy_after[1]= self.read_msr(fd, MSR_PP0_ENERGY_STATUS) * energy_units
      energy_after[2]= self.read_msr(fd, MSR_DRAM_ENERGY_STATUS) * energy_units
      energy_after[3]= energy_after[0] - (energy_after[1] + energy_after[2])

      # Calculate the power
      power[0] = (energy_after[0] - energy_before[0])/w
      power[1] = (energy_after[1] - energy_before[1])/w
      power[2] = (energy_after[2] - energy_before[2])/w
      power[3] = (energy_after[3] - energy_before[3])/w

      # Update the values for the next iteration
      energy_before[0] = energy_after[0]
      energy_before[1] = energy_after[1]
      energy_before[2] = energy_after[2]
      energy_before[3] = energy_after[3]

      yield power                # Return the power

      # Wait frequency seconds
      w= frecuency - (time.time()-t1)
      if (w > 0): time.sleep(w)

    os.close(fd)                 # Close the MSRs
```

Figure 2.9: RAPL module implemented in the PMLib server.

```
int main ( int argc , char * argv []) {
  struct timeval start,end;
  long int time;
  long long result;
  uint64_t data;
  double package_before, dram_before, cores_before, power_units,
         energy_units, package_after, dram_after, cores_after;
  double energy, cores, dram, package;
  int fd;

  // Start the measurements
  fd = open("/dev/cpu/0/msr", O_RDONLY);                  // Measurements in socket 0
  pread(fd, &data, sizeof data, MSR_RAPL_POWER_UNIT);
  result=(long long) data;
  energy_units=pow(0.5,(double)((result>>8)&0x1f));       // Establish the energy units

  pread(fd, &data, sizeof data, MSR_PKG_ENERGY_STATUS);   // Socket
  result=(long long) data;
  package_before=(double)result*energy_units;

  pread(fd, &data, sizeof data, MSR_DRAM_ENERGY_STATUS); // DRAM
  result=( long long)data;
  dram_before=(double)result*energy_units;

  pread(fd, &data, sizeof data, MSR_PP0_ENERGY_STATUS);   // Cores
  result=( long long)data;
  cores_before=(double)result*energy_units;

  // OPERATIONS TO MEASURE

  // Stop the measurements
  pread(fd, &data, sizeof data, MSR_PKG_ENERGY_STATUS);
  result=(long long) data;
  package_after=(double)result*energy_units;

  pread(fd, &data, sizeof data, MSR_DRAM_ENERGY_STATUS);
  result=( long long)data;
  dram_after=(double)result*energy_units;

  pread(fd, &data, sizeof data, MSR_PP0_ENERGY_STATUS);
  result=( long long)data;
  cores_after=(double)result*energy_units;

  // Calculate the total energy
  energy = (package_after-package_before)+
           (dram_after-dram_before);                      // Total energy

  cores = cores_after-cores_before;                       // Total energy of cores

  dram = dram_after - dram_before;                        // Total DRAM energy

  package = package_after - package_before;               // Total package energy

  close(fd);                                              // Close the MSRs
}
```

Figure 2.10: Example of use of RAPL directly from the code.

2.3.2 NVIDIA Management Library (NVML)

NVML [147] is a C-based API for monitoring and managing various states of NVIDIA GPU devices. It provides direct access to the queries and commands exposed via `nvidia-smi`. Concretely, the next information can be detected using NVML:

- ECC status information (error counts).
- GPU load (memory & processor).
- Temperature and fan speed.
- Active computational processes.
- GPU clock rates.
- Power management.

In our case, we use the functions provided by the *pyNVML* library [156] (Python interface to GPU management and monitoring functions) to create a module of PMLib that provides power information on the GPUs (see Figure 2.11). Specifically, in the `read` function of the class `NVMLDevice`, we employ the `nvmlDeviceGetPowerUsage` function [145] to retrieve the power usage reading for the device, in milliWatts. With an accuracy of +/- 5 Watts, this is the power draw for the entire board, including GPU, memory, etc. In a previous step, we have to initiate the library and acquire the handle for a particular device, based on its index, calling the `nvmlDeviceGetHandleByIndex` routine. In Figure 2.11, for simplicity we assume that there is only one GPU attached to the machine. Thus, in the call to `nvmlDeviceGetHandleByIndex` we use 0 as index parameter. If there are more devices, we will have to initiate a handle per device using its corresponding index. After the initialization, we call routine `nvmlDeviceGetPowerUsage` repeatedly every *frequency* seconds, until the server is stopped. When the server completes its execution, we shut down the NVML library.

2.3.3 MIC Management Library (libmicmgmt)

`Libmicmgmt` [112] is a C/C++ library that exposes a set of APIs to applications in order to monitor and configure several metrics of the Intel Xeon Phi coprocessor platform. It also allows communication with other agents, such as the System Management Controller. Following a successful boot of the Intel Xeon Phi coprocessor card(s), the primary responsibility of `libmicmgmt` is to establish connections with the host driver and the coprocessor micro Operating System (OS), and subsequently allow software to monitor/configure Intel Xeon Phi coprocessor parameters. There are three main communication channels. The first channel is established with the coprocessor directly via the SCIF library. The second and third are established with the host driver via `ioctls`, and either the `sysfs` interface on Linux platforms or the WMI interface on Windows platforms, as indicated in Figure 2.12. The list of APIs included in the management library can be classified in several broad categories.

We use `libmicmgmt` to implement a PMLib module which reads the power information of the Intel Xeon Phi coprocessor. For this purpose, we employ functions of the *Power utilization API*, taking advantage of the Python implementation of this library in order to implement the PMLib module. In Figure 2.13 we illustrate the code of the `read` function of the `MICDevice` class. The behaviour of this function is similar to that explained earlier for the other modules. First, we initialize a vector to store the samples of the different devices and we establish the

```
from pynvml import *

class NVMLDevice(Device.AttachedDevice):

def read(self):
  #Create the vector which store the measurements
   power       = [0] * self.n_lines

   frequency    = self.max_frequency ** -1

  # Initiate NVML and the handle
   nvmlInit()
   handle = nvmlDeviceGetHandleByIndex(0)              # There is only one GPU,
                                                       # so we use the index 0
   t1=time.time()

  # Power reading
   power[0]=nvmlDeviceGetPowerUsage(handle)/1000.0     # The value is in milliwatts,
                                                       # and we convert it to Watts
  # Wait frequency seconds
   w= frequency - (time.time()-t1)
   if (w > 0): time.sleep(w)

   while self.running:

     t1=time.time()

     # Power reading
     power[0]=nvmlDeviceGetPowerUsage(handle)/1000.0   # The value is in milliwatts,
                                                       # and we convert it to Watts
     # Return power
     yield power

     # Wait frequency seconds
     w= frecuency - (time.time()-t1)
     if (w > 0): time.sleep(w)

  # Stop NVML
   nvmlShutdown()
```

Figure 2.11: NVML module implemented in the PMLIB server.

reading frequency. Then, we call routine `micmgmt.mic_get_ndevices` to query the number of MIC devices connected to the host and we iterate over those devices (we use the instruction `mic = micmgmt.MicDevice(device)` to access a specific device) to obtain the instant power samples of each one calling the `mic.mic_get_inst_power_readings()` routine. This process is repeated every *frequency* seconds until the server is stopped.

2.3.4 Comparison of power sampling interfaces

As an additional contribution of our development, we evaluated two power sampling/modeling approaches, one which uses RAPL+MSR to obtain the power information and another one that leverages a DC wattmeter using a DAS from NI (NI9205 module+NIcDAQ-9178 chassis). This device is connected directly to the 12V wires from the power supply unit to the system motherboard, as these lines feed the processor and main memory of the system [69]. The external-PMLIB daemon reads the data captured by the DAS from a USB 2.0 port in the chassis. Therefore the execution of the external-PMLIB does not introduce any overhead in the performance nor power traces.

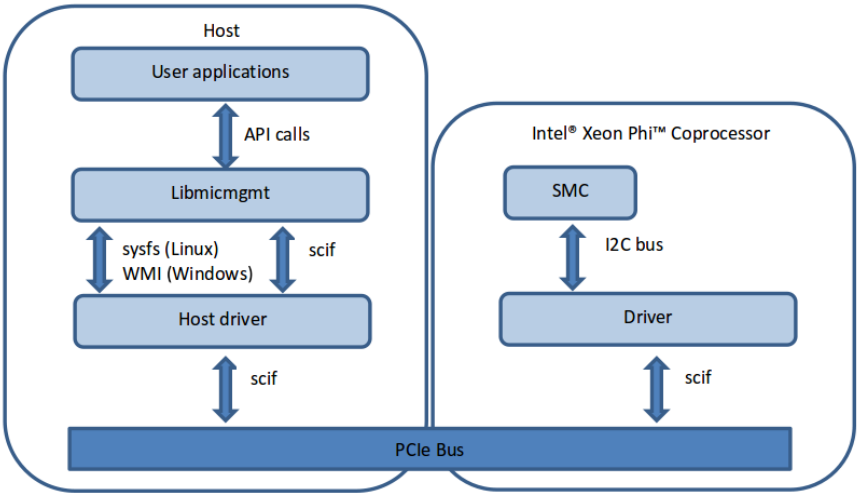


Figure 2.12: Intel Xeon Phi Coprocessor Management Library Architecture for SCIF, `sysfs` and WMI Communication Channels.

However, the readings of the RAPL counters using MSR registers are carried out by the internal-PMLib daemon, and this can introduce a nonnegligible overhead during the execution.

While we can expect that RAPL+MSR provides accurate power reads, the goal of this study is to assess whether the periodic access to the MSRs from PMLib results in a significant overhead that blurs part of the advantages of this method. In particular, we performed this study using a synthetic test composed of a sequence of runs of the `cpuburn` benchmark [136] followed by the `sleep` Linux system call, for periods of 30 seconds each. The power profiles (for approximately 180 secs.) obtained from running this test on a single core of the platform, captured by both RAPL+MSR and NI wattmeter, are displayed in Figure 2.14. The results show the synchronization of the measures taken by both interfaces, validating the RAPL+MSR interface to sample the power consumption. As could be expected, the rates obtained from the NI wattmeter (up to 79 Watts) are higher than those reported for RAPL+MSR (up to 36 Watts), since the former device measures the power draw not only for the CPU socket (which includes the core/uncore components, the memory controller, the QPI interconnect, etc.), but also for the Dual In-line Memory Modules (DIMMs) and other off-chip components of the mainboard.

Table 2.4 details the overhead introduced by the access to the MSR from PMLib. There we report the maximum and average power observed while running the test on 1 and 4 cores of the platform, with measures taken from the internal-PMLib daemon (i.e., from RAPL+MSR) at 1, 10 and 100 samples/sec.; and from the external-PMLib daemon (i.e., the wattmeter) at 1,000 samples/sec. To avoid noise introduced by outliers, we consider the power data corresponding to the intervals [125,145] sec. (third execution of `sleep`) and [155,175] sec. (third execution of `cpuburn`) only.

The measurements obtained from RAPL+MSR in the Table 2.4 show a small overhead introduced by the internal-PMLib, which depends on the sampling rate: about 0.5 Watts when it is increased from 1 to 100, and basically independent of the test (idle or `cpuburn`). To analyze the global power dissipation, the same tests were repeated using next the external-PMLib interface only, with the results given in Table 2.5. Comparing Tables 2.4 and 2.5, we can conclude that the overhead of the internal-PMLib is negligible.

```
import micmgmt

class MICDevice(Device.AttachedDevice):

def read(self):
  #Create the vector which store the measurements
   power       = [0] * self.n_lines        # self.n_lines is the number of MIC devices

   frequency   = self.max_frequency ** -1

  # Query number of devices:
   device_count = micmgmt.mic_get_ndevices()

   t1=time.time()

  # Iterate over cards in system
   for device in range(device_count):
     mic = micmgmt.MicDevice(device)
     print "Found KNC device: %s" % mic.mic_get_device_name()

     try:
       # Power reading
       power[device] = float(mic.mic_get_inst_power_readings())/1000000

     except micmgmt.MicException as e:
       print "Failed to get power readings: %s: %s" % (mic.mic_get_inst_power_readings(), e)

  # Wait frequency seconds
   w= frequency - (time.time()-t1)
   if (w > 0): time.sleep(w)

   while self.running:

     t1=time.time()

     for device in range(device_count):
       mic = micmgmt.MicDevice(device)

       try:
         # Power reading
         power[device] = float(mic.mic_get_inst_power_readings())/1000000

       except micmgmt.MicException as e:
         print "Failed to get power readings: %s: %s"%(mic.mic_get_inst_power_readings(), e)

     yield power

    # Wait frequency seconds
     w= frecuency - (time.time()-t1)
     if (w > 0): time.sleep(w)
```

Figure 2.13: MIC module implemented in the PMLIB server.

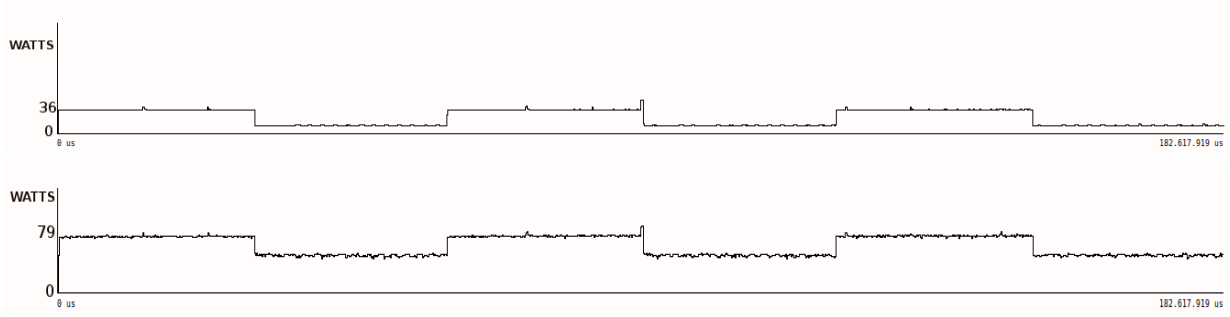


Figure 2.14: Power profiles of the synthetic test consisting of interleaved calls to `sleep` and `cpuburn`, obtained from RAPL+MSR at 100 samples/sec. (top), and the NI wattmeter at 1,000 samples/sec. (bottom).

Source	RAPL freq.	idle		1 × cpuburn		4 × cpuburn	
		max	avg	max	avg	max	avg
RAPL+MSR	1	9.0	8.7	24.8	24.0	49.6	49.2
	10	10.8	8.9	27.6	24.4	50.3	49.2
	100	21.0	9.2	34.0	24.5	55.3	49.6
NI DAS	1	69.1	43.7	77.6	62.3	102.9	92.6
	10	69.7	43.6	79.7	62.4	103.3	92.3
	100	68.8	43.2	78.1	62.7	102.8	92.5

Table 2.4: Power measurements obtained from RAPL+MSR and the NI module using internal-PMLib and external-PMLib respectively, with both daemons in simultaneous operation. Column "RAPL freq." indicates the sampling/rate of the internal-PMLib daemon, while the rate for the external one was 1,000 samples/sec.

Source	idle		1 × cpuburn		4 × cpuburn	
	max	avg	max	avg	max	avg
NI	70.6	43.7	77.5	63.0	103.3	92.2

Table 2.5: Power measurements obtained from the NI module using external-PMLib with only that daemon in operation at a rate of 1,000 samples/sec.

2.4 Automatic Detection of Power Sinks

The use of PMLib can reveal an energy-inefficient hardware configuration or expose a waste of energy incurred by an application [24], runtime [26], library or governor [27] related with the parallel execution of linear algebra codes. In all these cases, the power bottlenecks are detected as disagreements between the application activity and the system power consumption and, in general, the source can be tracked down to the use of power-hungry busy-waits (i.e., polling loops) with little benefits on the execution time. However, during these studies, side-by-side visual inspection of performance and power traces is a cumbersome task at best, while also being an error prone process.

To tackle this problem, as part of this dissertation, we introduce a parallel, a posteriori analyzer to detect power bottlenecks with several major advantages over the previous manual approach:

- The tool automates and accelerates the inspection process.
- The process is more reliable, as the detection of power sinks is based on a comparison between the application performance trace and the C-state traces per core (instead of the performance trace vs the power trace for the complete socket/platform that we exploited earlier).
- The analyzer is flexible: the task types that correspond to "useful" work can be defined by the user; and the length of the analysis interval and the divergence (discrepancy) threshold are parameters that can be adjusted to the desired levels.
- The inspection introduces a low overhead during the collection of the samples, with little side effects on the application performance and power traces.

The interaction of the inspection tool with the rest of the framework is illustrated in Figure 2.15.

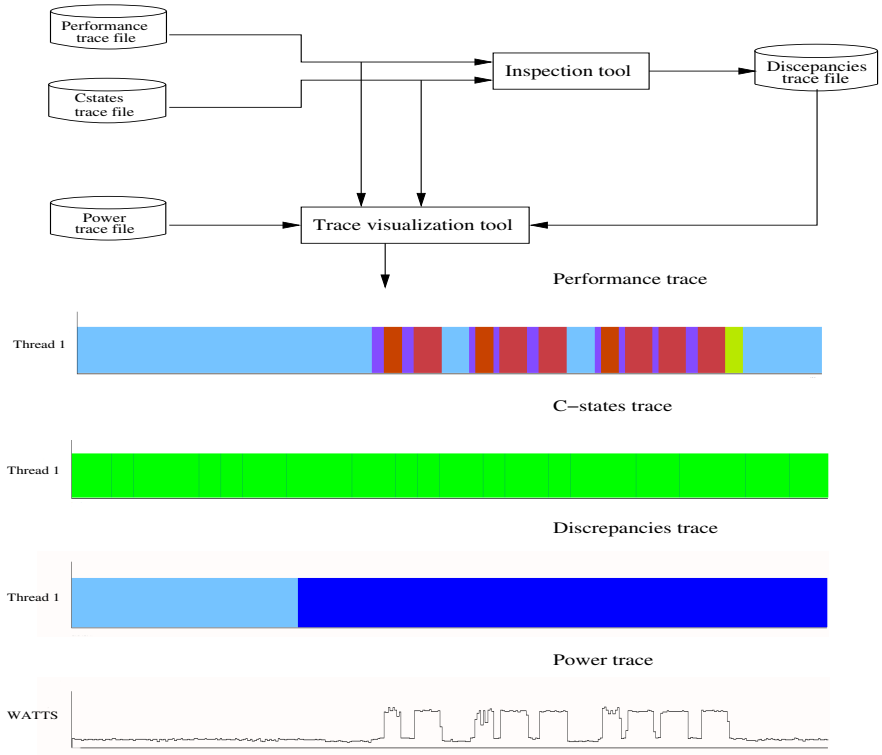


Figure 2.15: Operation of the inspection tool to detect and report power sinks.

2.4.1 Operation and implementation

We next describe the analyzer and its properties in more detail. The inspection tool is developed in Python and, after the execution of the application, can be applied to analyze a performance trace, in principle produced by `Extrae`, and a compatible C-state trace per core produced by the internal-PMLib daemon.

The use of `Extrae`, combined with the visualization environment `Paraver`, is convenient because it allows to interactively analyze the behaviour of concurrent scientific applications. However, this decision does not prevent the integration of other performance suites in our framework, as the inspection tool only processes the performance traces, and does not interact in any other manner with the specific tool that produced them.

On the other hand, the C-state trace is obtained with the internal-PMLib daemon, which monitors the state of the processor cores at runtime. Besides, the read frequency of the daemon is also configurable, allowing the user to adjust the level of noise introduced in the performance and power traces.

In order to process the information contained in the traces, the inspection tool splits them into intervals of a certain length t (configurable by the user), starting at time instants 0 and $0.5t$ of the traces. The reason for the double division of the traces, starting at 0 and $0.5t$, is to avoid that power bottlenecks pass the analysis undetected if they are split between two consecutive intervals. All the intervals are then inserted (enqueued) into a task pool for their analysis by the multi-threaded analyzer, employing the Python `Pool` class [148]. For each CPU core and interval, the analyzer compares the ratio of time that the application was inactive (i.e., performing no useful computation from the application point of view) while the core remained in state C0 (active), detecting a potential power sink whenever the difference between these two values is above a given threshold (configurable by the user). While the version of the inspection tool that we just described performs a linear analysis of the traces, we have also developed an alternative implementation that performs a binary search of power bottlenecks and, in general, is faster when the number of discrepancies is small.

The result of the bottleneck search is twofold: analytical and graphical. On one hand, it produces a simple text output with a tuple (c, t_i, t_f, $\%divergence$) per detected power sink, where c is the core identifier, $[t_i, t_f]$ is the time interval where the bottleneck occurs in the trace timeline, and the last parameter quantifies the divergence between the application inactivity and the core C-state ratios in that period. On the other hand, the analyzer also produces a new trace that allows a rapid identification of the sources of discrepancies using a visualization tool, in our case, `Paraver`.

2.4.2 Examples

In this section we illustrate the possibilities of the inspection tool to detect power bottlenecks using two complex applications on multicore technology platforms. The next experiments were carried out using IEEE double-precision arithmetic in the two platforms:

WT_INT is a shared-memory multiprocessor composed of two quad-core Intel Xeon 5504 processors (total of 8 cores) running at 2.00 GHz, with 32 Gbytes of DDR3 Random Access Memory (RAM) memory under Linux Ubuntu (kernel 2.6.32-220.4.1.el6.x86 64).

WT_IVY is a hybrid platform equipped with an Intel Xeon i7-3770 processor (total of 6 cores) running at 3.5 GHz, with 16 Gbytes of DDR3 RAM, and an NVIDIA Tesla C2050 ("Fermi"). The operating system is Linux Ubuntu (kernel 2.6.32-220.4.1.el6.x86 64).

The software employed in the evaluation includes Intel MKL (composer_xe_2011_sp1.9.293) and CUDA (v6.5). The tracing and visualization facilities were `Extrae` (v2.2.0) and `Paraver` (v4.1.0).

ILUPACK

The first example of the inspection tool corresponds to the concurrent solution of sparse linear systems using ILUPACK[1], a package that computes and applies multi-level preconditioners to solve linear systems. The parallelization of this solver for Symmetric Positive Definite matrices (SPD) on multicore platforms in [21] relies on a task partitioning of the sparsity graph related to the

[1] http://ilupack.tu-bs.de

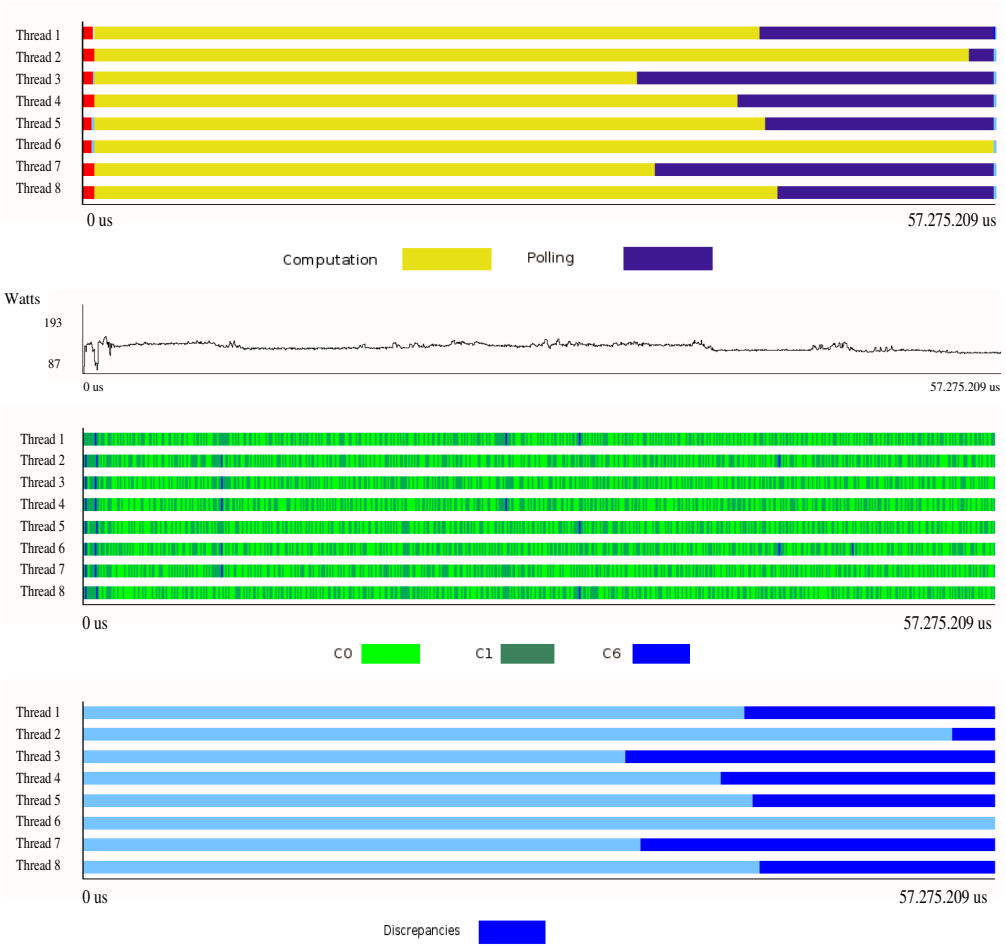


Figure 2.16: Performance (top), power (top-middle), C-states (bottom-middle) and discrepancies (bottom) traces, visualized with `Paraver`, for the concurrent execution of ILUPACK.

coefficient matrix of the system, that yields a Task Acyclic Graph (TAG) with the structure of a binary tree capturing the dependencies between tasks. At runtime, tasks from the TAG that are ready for execution (i.e., those with their dependencies with all other tasks fulfilled) are mapped to threads on-demand, using a dynamic scheduler that manages a centralized queue on where the ready tasks are pushed, combined with certain data locality heuristics. Upon completion of a task, the thread in charge of its execution checks whether new tasks have became ready for execution, inserting them into the ready queue. In the multi-threaded variant of ILUPACK when a thread encounters no work to execute, it simply polls the centralized queue till a ready task becomes available.

Figure 2.16 shows fragments of the performance, power and C-states traces obtained with `Extrae` and PMLib for the parallel execution of ILUPACK on WT_INT platform. This particular period corresponds to the computation of the preconditioner, which takes the first 57.27 secs. of the total execution, and is followed by a longer iterative solution process (not included in the figure). The TAG for this initial stage is organized as a binary tree with bottom-up dependencies between tasks, and the bulk of the work is concentrated in the leaf tasks. This is visible in the performance trace of the figure (top plot), which shows that initially all threads are occupied with computation (except for a brief configuration period) but, towards the end, an increasing number

of threads become idle. As discussed in the previous paragraph, these "idle" threads then perform a busy-wait, polling for more work (in the form of a ready task) that will not become available till the solution stage (not included in the figure). Interestingly, the power and C-state traces (middle plots) do not show any significant variation due to threads entering the polling phase. This is captured in the power-sink (discrepancies) trace (bottom plot), which reveals a period where the application was performing no useful computation, but the cores basically remained in the C0 state.

As an alternative to the previous power-hungry strategy, there exists a power-aware version of the runtime underlying ILUPACK, which applies an "idle-wait"(blocking) whenever a thread does not encounter a task ready for execution and, thus, becomes inactive. (Note that setting the necessary conditions for the operating system to promote the cores into a power-saving C-state is as much as we can do, since we cannot explicitly enforce the transition from the application code.) As in the original version of the runtime, upon completing the execution of a task, a thread updates the corresponding dependencies identifying those tasks that have become ready for execution. However, in the power-aware runtime, the algorithm also ensures that the number of active threads (non-blocked threads) is, at most, equal to the number of ready tasks, releasing blocked threads if necessary. The number of threads is limited by the number of threads of the system. The effect of idle-wait on the power trace and use of the C-states is illustrated in Figure 2.17. Compared with the previous implementation, the new runtime effectively allows inactive cores to enter a power-saving C-state, thus yielding the sought-after power reduction. As expected, the inspection tool does not detect any discrepancy in this new implementation (see bottom plot of Figure 2.17), because the C-states trace shows a variation to C6 state when the threads entering the blocking phase (i.e., they are performing an idle-wait).

LU Factorization

The second example of the inspection tool is obtained during the computation of the LU factorization (with partial pivoting) of a dense matrix, using the `FLA_LU` routine of the `libflame` library, parallelized with the SuperMatrix runtime [159]. The hybrid CPU-GPU version of this runtime [158] interleaves the execution of certain computations of the factorization on the CPU (factorization of panels) with other tasks on the GPU (triangular system solves and updates of the trailing submatrix), and controls the necessary data transfers between the main memory and the device. The computations on the CPU are performed via calls to Intel MKL [111], while the GPU computations and the transfers rely on explicit calls to NVIDIA CUBLAS and CUDA [146], respectively.

Figure 2.18 reports the complete performance and C-state traces, obtained with `Extrae` and PMLib, for the execution of the LU factorization on WT_IVY. In principle, one could expect that, during the execution of a GPU kernel (`Trsm_gpu` and `Gemm_gpu`), the operating system promoted the idle CPU to a power-saving sleep C-state (C1 or deeper). However, the traces in the top and middle plots show that this is not the case, and this is clearly detected in the discrepancy trace in the bottom plot of the figure. In [28] it is showed how to avoid this power-costly behaviour of the CUDA runtime, by appropriately invoking routine `cudaSetDeviceFlags`.

Impact of power sinks

The previous two examples demonstrate the potential of the inspection tool to easily detect discrepancies between the performance and C-state traces that identify potential power-sinks. Furthermore, they also illustrate the interface of the inspection tool, by showing how this information is reported in the form of a trace that can visualized using `Paraver`.

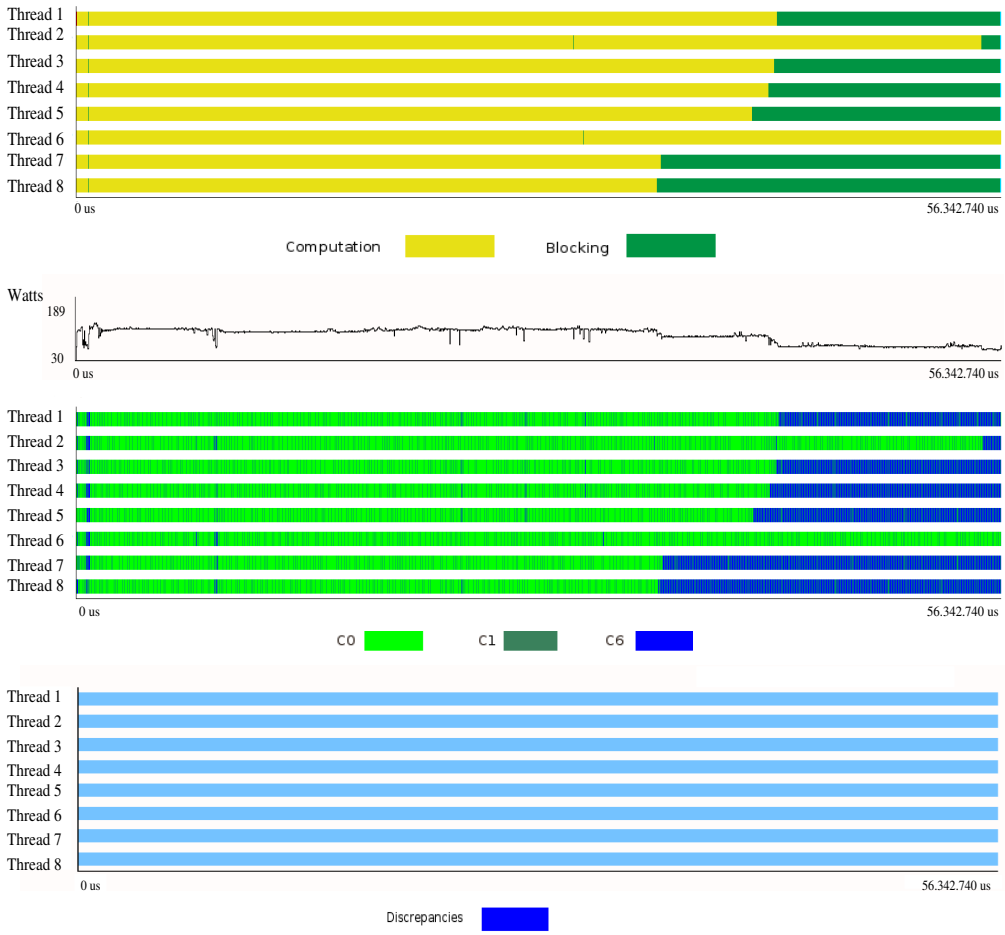


Figure 2.17: Performance (top), power (top-middle), C-states (bottom-middle) and discrepancies (bottom) traces, visualized with `Paraver`, for the power-aware concurrent execution of ILUPACK.

	Computation	Polling	C0	C1	C6	Discrepancies
THREAD 1	72.00%	25.56%	99.33%	0.29%	0.39%	27.49%
THREAD 2	96.45%	2.50%	99.25%	0.26%	0.50%	4.77%
THREAD 3	59.90%	39.14%	99.53%	0.10%	0.37%	40.59%
THREAD 4	70.81%	28.13%	99.48%	0.10%	0.42%	30.11%
THREAD 5	74.00%	25.14%	99.29%	0.90%	0.61%	26.61%
THREAD 6	99.18%	0.00%	99.34%	0.22%	0.45%	0.00%
THREAD 7	61.52%	37.17%	99.53%	0.12%	0.35%	38.84%
THREAD 8	75.03%	23.69%	99.27%	0.10%	0.64%	25.74%

Table 2.6: Example of analytical summary of the performance, C-state and discrepancy traces reported by the inspection tool.

In addition, the analyzer also elaborates some simple statistical information, in terms of an analytical report for the global execution or a certain period of the execution; see Table 2.6. This type of information can be next processed to obtain a rough estimation of the costs of the

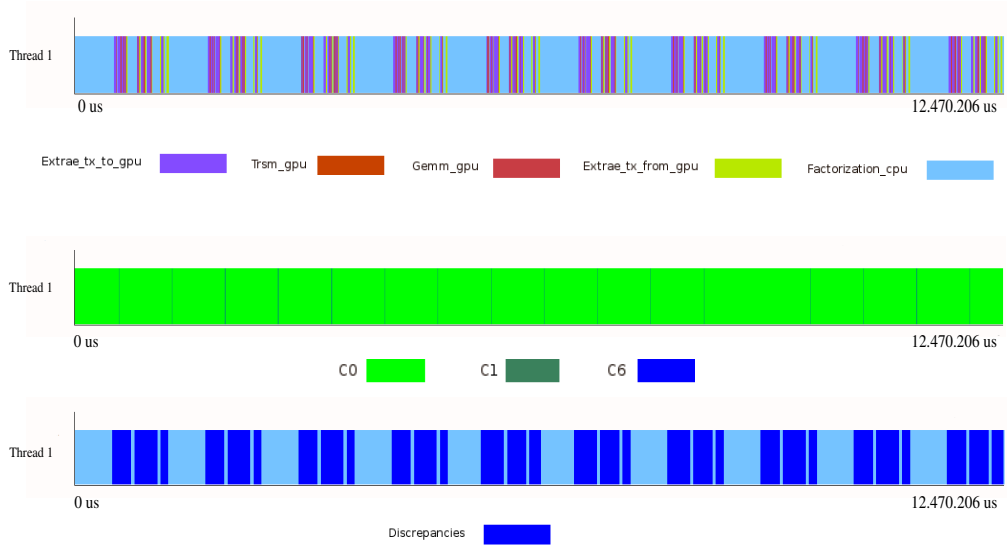


Figure 2.18: Performance (top), C-states (middle) and discrepancies (bottom) trace, visualized with `Paraver`, for the concurrent execution of the LU factorization in `libflame`.

power sinks. Assume for simplicity that there is only one type of sink during the execution of the inspected application. The analytical report from the analyzer then offers an estimate of how long certain threads/cores spent doing "useless" work and, therefore, wasting power (e.g., up to 40.59% for thread 3 in Table 2.6). Of course, in order to obtain a measure of the overall energy cost, we need to take into account what is the power rate of a core that remains in a power-saving sleep state. That estimation is easy to obtain as part of an independent and simple experiment. The hard part, though, is to calibrate the opposite situation, i.e., how much of the total power consumption that is in the power trace corresponds to the "power-wasting" core(s). Unfortunately, our current sampling means only offers either the total power consumption (using a wattmeter) or, alternatively, an approximation to the power consumption per CPU socket (using the RAPL power model), while we would like to have accurate estimates of power consumption per core instead. To overcome this difficulty, we can still design a special test that mimics the power sink that was detected in a particular experiment, and which could give us the sought-after estimate. In ILUPACK, for example, this would be a busy-wait on a condition variable, performed by a single core, which would allow us to evaluate the power cost of this process. For the dense linear algebra factorization, on the other hand, the test should consider a sequence of long CUDA invocations, and the appropriate configuration of the CUDA runtime to reproduce the power sink observed for this application.

Armed with these estimates, by simply calculating the differences between the power rate due to these power-hungry behaviour/configuration and that of the power-saving state, as well as the total duration of the power sinks, we can obtain a rough approximation of the energy-costs due to hotspots, and the potential savings that a more power-friendly implementation and/or configuration of the software could yield. Note that all this elaboration only needs a reliable report on the duration of the power bottlenecks and the data from the calibration experiments.

2.5 Concluding Remarks

In this chapter we have presented a power-tracing framework composed of internal/external wattmeters, a power tracing modular package, power-related modules, etc., that is easily integrable with standard performance tracing and visualization tools. The framework offers useful information on power usage of scientific workloads running on a variety of parallel platforms, from MPI applications operating on a moderate-scale cluster to multi-threaded codes that run on a multicore+GPU platform. We have also described our contributions, which expand the framework to adapt it to new processor technologies. Concretely, these additions obtain power and energy measurements reading the RAPL counters, and also with the NVML and MIC Management libraries.

As an additional contribution, we have investigated the use of power/temperature estimates offered by recent processor technology vs the exploitation of a professional data acquisition system. While the former approach is extremely appealing in that it comes for free with the processor, the fact that recording the power information can only be done from the same platform, using the same resources that are involved in the application execution, necessarily results in an overhead that may impair the accuracy of the results. In both cases, we miss the possibility of accessing traces of the power dissipation per core. These two drawbacks seem to be solved by alternative processor technology like that present in the IBM Power7, where power information is available per core, and it is recorded by a separate device, in principle without any overhead.

Moreover, we have introduced an extension of the power tracing tool PMLib that can be leveraged to automatically detect power bottlenecks during the execution of an application by comparing the performance and the core C-state traces. The analysis is performed in parallel, using Python `Pool` class, and the user can configure the process, for instance, to adjust the part of the trace to inspect, the length of the analysis interval, and the detection threshold, among others. We have used this framework to develop more energy-efficient HPC linear algebra libraries, which leverage idle periods during the execution using dynamic frequency-voltage scaling and avoiding busy-waits.

Finally, we have exemplified the potential of the framework using two linear algebra codes. These studies reveal the possibilities that the framework offers to analyze the behaviour of the applications and detect the power bottlenecks as well as the impact of these power sinks.

CHAPTER 3

Solution of Large Sparse Linear Systems and ILUPACK

Large sparse linear systems arise in many application areas such as partial differential equations, quantum physics, or problems from circuit and device simulation. They all share the same central task that consists in efficiently solving large sparse systems of equations. For a large class of application problems, sparse direct solvers have proven to be extremely efficient. However, the enormous size of the underlying applications arising in 3-D PDEs or the large number of devices in integrated circuits currently requires fast and efficient iterative solution techniques, and this need will be exacerbated as the dimension of these systems grows. This in turn demands for alternative approaches such as, approximate factorization techniques, combined with iterative methods based on Krylov subspaces, which have become an attractive alternative for these kinds of application problems. Specifically, in this chapter, we revisit the different solution methods, focusing on CG and PCG, and we describe various preconditioning techniques. Finally, we introduce the ILUPACK library, that contains highly efficient multi-level incomplete LU factorization solvers, based on Krylov subspace methods, for large-scale sparse application problems.

The chapter is divided as follows. Section 3.1 reviews the solution methods and the CG. Section 3.2 describes the PCG and analyzes the ILU Preconditioning Techniques. Section 3.3 offers an introduction to ILUPACK, the numerical package for solving large systems of equations, which has been parallelized in this thesis to improve its performance and energy efficiency.

3.1 Solving Sparse Linear Systems

A sparse matrix is defined as a matrix which has very few non-zero elements, such that special techniques can be utilized to take advantage of this property. These sparse matrix techniques exploit the idea that the zero elements do not need to be stored and, therefore, one of the key issues is to define data structures for these matrices that are well suited for the efficient implementation of standard solution methods, which can yield huge computational savings. On the one hand, a formal characterization, commonly used, defines that a matrix $A \in \mathbb{R}^{n \times n}$ is sparse if

$$nnz(A) = O(n)$$

where $nnz(A)$ is the number of non-zero entries of $A = (a_{ij})$. On the other hand, there is another principle which characterizes A as a sparse matrix: there is a number $p \ll n$, such that every row

and column of A has at most p non-zero entries. The *density* of a matrix is a term closely related to the sparse notion, which is defined as the quotient $nnz(A)/n^2$. Another term is the *pattern* of non-zero elements of a matrix A, which is defined as the set $\mathcal{A} = \{(i,j) : a_{i,j} \neq 0\} \subseteq \{1,...,n\} \times \{1,...,n\}$, and, thus, $nnz(A)$ coincides with the cardinality, $|A|$, of the set $\mathcal{A}$.

The solution of sparse linear systems of large dimension consists in finding the solution $x \in \mathbb{R}^n$ of the linear equation system

$$Ax = b, \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix of large dimension and arbitrary structure, and $b \in \mathbb{R}^n$ is the vector of independent terms.

In this section we classify the methods for the solution of sparse linear systems, explaining their advantages, disadvantages and utility. Moreover, we explain in more detail the CG method, which is the object of this dissertation.

3.1.1 Classification of the solution methods

Traditionally, the methods for the solution of linear systems have been divided into two groups: *direct* and *iterative*. The main difference between them is that, in the first, the cost of the algorithm is known beforehand because they reduce the matrix to a canonical form by using elementary transformations, while in the second, the number of iterations and the final cost of the method can change depending on the properties of the matrix and the desired accuracy of the solution.

In the 60s, 70s and 80s, direct methods [63] were the frequent choice for the solution of large systems, due to their robustness and predictable behaviour. However, in the last decades iterative methods [166] have increased their popularity thanks to several factors. First, the size and complexity of the new generation of systems arising from the common applications have increased significantly. Second, the robustness, reliability, and efficiency of the iterative methods have improved remarkably thanks to the advances in the *preconditioning* techniques, being the most critic element to improve the performance of these kind of methods. Last, it is easier to develop parallel iterative methods than parallel versions of direct methods, because of the features of the main operations of each one.

Direct methods

Direct methods for the solution of linear systems transform the problem (3.1) into a simpler one, whose solution is less difficult. Many of these methods are based in the LU factorization or Gaussian elimination [88, 183]. This process decomposes the coefficient matrix of the system (3.1) A into the product of two triangular matrices, $L \in \mathbb{R}^{n \times n}$ unit lower triangular and $U \in \mathbb{R}^{n \times n}$ upper triangular, such that $A = LU$. Afterwards, the solution of (3.1) can be obtained by solving two linear systems with triangular structure, with the triangular factors as coefficient matrices; particularly, first $Ly = b$, and then $Ux = y$.

The fact that both the transformation and the subsequent resolution of the reduced system are calculated in a number of steps known a priori, combined with the numeric robustness of these methods, makes them very useful for solving systems where the coefficient matrix is *dense*. When the coefficient matrix is large and *sparse*, the use of these methods usually introduces new non-zero elements in the transformation process, which increase the cost of the process. Therefore, the implementation of direct methods for sparse matrices aims to reduce the computational and storage cost of Gaussian elimination, avoiding operations with the zero entries and minimizing the *fill-in* produced during the factorization process [63, 72]. The application of these methods is based on the use of re-orderings, to reduce the *fill-in*, and the partitioning of the system, to identify dense

blocks in the factors. The block structure of the factors enables the use of basic linear algebra *subprograms* (BLAS3 [71]), which is essential to obtain high performance in modern superscalar processors. However, the success of the resultant methods depends on their capacity to efficiently manage the *fill-in*. Although direct methods are capable of solving reasonably large systems of equations, their spatial and temporal costs scale poorly with the problem dimension, especially for matrices arising from the discretization of tridimensional PDEs, due to the massive *fill-in* produced in the factorization. In summary, these methods are extremely reliable and numerically robust, and besides, it is often possible to predict the computational and storage resources they require. For these reasons, they are usually chosen to solve systems of equations in industrial environments, where the key is reliability, but the fill-in is not too high.

Iterative methods

For linear systems of equations with extremely large dimension and/or in which the level of fill-in of direct methods is very high, iterative methods are the only feasible alternative. These methods solve the system through a sequence of operations which are applied to an initial solution, so that the final solution is progressively approximated. These operations are grouped into phases, or iterations, and the results corresponding to the steps of one iteration are the entry for the next iteration. This property makes these methods especially efficient for the solution of large linear systems of equations, because the process can be stopped when the desired precision is achieved. Although iterative methods require less memory, and usually execute less arithmetic operations than direct methods, they are also less reliable. Depending on the application, iterative methods sometimes fail, and usually, it is necessary the application of *preconditioning* techniques, which modify the numerical properties of the system, to accelerate the convergence. The *preconditioner* cannot be explicitly applied to the coefficient matrix, because it would modify its non-zero pattern. Therefore, a new step is introduced in the iterative process which applies the *preconditioner* during the resolution process. The advantages of iterative methods, with respect to direct methods, are their high scalability when the dimension of the problem increases and the ease to design and implement parallel versions.

There exists a huge variety of iterative methods, each one with specific features, which can be classified as follows:

- Stationary iterative methods: The major research efforts on these methods were made between the 50s and the 70s, yielding popular methods such as Jacobi, Gauss-Seidel, SOR and SSOR [166]. These methods were applied to solve linear systems arising from the discretization of eliptic PDEs. The stationary iterative methods are defined by means of the following mathematical recursion:

$$x^{(k+1)} \leftarrow Tx^{(k)} + c, \text{ with } k = 0,1,2,\ldots \tag{3.2}$$

  where $x^{(k)}$ denotes the approximation to the solution of $Ax = b$ in the k-th iteration, T is a predefined iteration matrix (fixed) which characterizes the particular method, c is a predefined vector (fixed) and $x^{(0)}$ is the initial approximation. Despite its mathematical refinement, stationary methods present serious limitations, for example, the lack of applicability or the parameter's dependence, because it is difficult to find an appropriate value for them without having specific information of the eigenvalue's spectrum of A. Therefore, these methods can converge very slowly, or even can not converge. In [98], Hageman and Young explain several procedures to adaptively estimate the value of these parameters, as well as the use of

acceleration techniques based on the generation of Krylov subspaces. That reference marks the transition between the first and the second periods in the history of iterative methods.

- PROJECTION METHODS: The research activity in these methods started in the mid-70s. The majority of these methods are based on the generation of Krylov subspaces, and the associated variables contain information that varies in each iteration. The most relevant methods in this group are CG, BICG, BICGSTAB and GMRES [167].

  The starting point of any projection method is the linear system $Ax = b$, and a vectorial subspace $\mathcal{K}$ of dimension k, such that $k < n$, generated by a base of vectors $V = \{v_1, \ldots, v_k\}$. The main objective is to find an approximation to the solution $\hat{x}$ inside the subspace $\mathcal{K}$, for which $\hat{x} = Vy$, with $y \in \mathbb{R}^k$. One of the most accepted methods to calculate y is to force the restriction that the residual vector, $r = b - A\hat{x}$, is orthogonal to another subspace $\mathcal{L}$ generated by a base of vectors $W = \{w_1, \ldots, w_k\}$, such that

$$W^T \cdot (b - AVy) = 0. \tag{3.3}$$

  Finding y from (3.3), we obtain a $k \times k$ linear system, $y = (W^T AV)^{-1} \cdot W^T b$, where $W^T AV$ must be invertible. Algorithm 3.1 shows a basic scheme of a projection method.

Algorithm 3.1 Projection method prototype

1: **while** not convergence **do**
2: Select the subspaces $\mathcal{K}$ and $\mathcal{L}$
3: Choose the bases $V = \{v_1, \ldots, v_k\}$ and $W = \{w_1, \ldots, w_k\}$
4: $r := b - Ax$
5: $y := (W^T AV)^{-1} W^T r$
6: $x := x + Vy$
7: **end while**

The relation between the subspaces $\mathcal{K}$ (search subspace) and $\mathcal{L}$ (constraints subspace) defines the type of projection that is employed. If $\mathcal{K} = \mathcal{L}$ the projections are *orthogonal*, while in the case that $\mathcal{K} \neq \mathcal{L}$ the projections are *oblique*. It is worth to mention that in the search of an approximation to the solution, the dimension of the subspaces is incremented by one in each step.

Next, we show how to introduce the Krylov subspace methods as an alternative inside projection methods. Mathematically, all the Krylov subspace methods calculate, in each iteration $k = 1,2,\ldots$, an approximation $x^{(k)}$ to the solution of $Ax = b$ inside the subspace

$$x^{(k)} = x^{(0)} + \mathcal{K}_k(A, r^{(0)}) = x^{(0)} + span\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, \ldots, A^{k-1} r^{(0)}\}, \tag{3.4}$$

where $x^{(0)}$ is a starting solution, $\mathcal{K}_k(A, r^{(0)})$ is the Krylov subspace of dimension k, generated by the matrix A and the initial residual $r^{(0)} = b - Ax^{(0)}$, and $span\{r^{(0)}, Ar^{(0)}, \ldots, A^{k-1} r^{(0)}\}$ is the subspace generated by the corresponding vectors $r^{(0)}, Ar^{(0)}, \ldots, A^{k-1} r^{(0)}$. The distinct methods are defined by the restrictions they impose to extract $x^{(k)}$ from this subspace. The following four approaches are the most common:

1. The *Ritz-Galerkin* approximation chooses $x^{(k)}$ so that the residual $r^{(k)} = b - Ax^{(k)}$ is orthogonal to the subspace $\mathcal{K}_k(A, r^{(0)})$. This approximation is the basis of popular methods such as CG, Lanczos, FOM and its variants IOM and DIOM.

2. The *minimum residual* approximation calculates $x^{(k)}$ in order that $r^{(k)} = b - Ax^{(k)}$ is orthogonal to the subspace $A \cdot \mathcal{K}_k(A, r^{(0)})$ or, similarly, in such a way that $\| b - Ax^{(k)} \|$ is minimum between all the vectors of the subspace $\mathcal{K}_k(A, r^{(0)})$. GMRES, MINRES and ORTHODIR are based on this approximation.

3. The *Petrov-Galerkin* approximation computes $x^{(k)}$ so that $r^{(k)} = b - Ax^{(k)}$ is orthogonal to another appropriate subspace of k dimension, $\mathcal{L}_k$. Bi-CG and QMR are obtained when $\mathcal{L}_k = \mathcal{K}_k(A^T, r^{(0)})$.

4. The *minimum error* approximation determines $x^{(k)}$ forcing that $\| x - x^{(k)} \|_2$ is minimum between the vectors of the subspace $A^T \cdot \mathcal{K}_k(A^T, r^{(0)})$. SYMMLQ and GMERR belong to this class.

It is important to note that there are other methods like CGS, Bi-CGSTAB and FMGRES which combine ideas from these approximations. Their implementation details, mathematical basis, and numerical analysis can be found in [32, 40, 56, 166].

- MULTILEVEL METHODS: These methods appeared at mid-70s to improve the efficiency of the stationary iterative methods [55, 57, 97], and were designed to rapidly and efficiently solve linear systems arising from the discretization of PDEs. Multilevel methods, which exploit discretizations with a hierarchy levels of a given problem to obtain optimal convergence, can be applied as iterative methods, or as preconditioners of CG or other methods based on the generation of Krylov subspaces. In this family, we can find the *multigrid* methods, which use meshes instead of levels, and revolve around a stationary method such as Jacobi or Gauss-Seidel. Stationary methods reduce very slowly the magnitude of the low frequency components of the error, but they are really fast to reduce the high frequency components. Multigrid methods exploit the possibility of building a coarser system, and transform the low frequency components of the error of the original system into high frequency components of the coarser system, which can be reduced using a stationary method. Applying this strategy recursively through different mesh sizes, we can obtain a method with a computational cost lineally dependent on the problem dimension. This multigrid method property is called algorithmic scalability and it is generalized for multi-level methods.

  The algorithmic scalability of multi-level methods is crucial to develop parallel algorithms which can maintain their efficiency when the problem size increases in the same proportion as the number of processors. Furthermore, the intrinsic hierarchy of the multi-level methods allows their adaptation to the memory hierarchy, as well as to the different kind of parallelism available in high performance computing platforms. This adaptation is properly produced by combining the number of levels and the size of the problems to be solved in each level. These properties have favoured the use of the *AMG methods* in the actual software packages. The intrinsics of the parallelization of AMG can be found in [186].

- DOMAIN DECOMPOSITION METHODS: This family of methods became popular at 80s, mainly due to the emergence of parallel computing platforms [157, 175], but nowadays, they (and their multi-level variants) are only used as preconditioners. Mathematically, we can consider these methods as an extension of block preconditioners like Jacobi or Gauss-Seidel [166]. They can also be considered as a general paradigm to decompose a problem into smaller problems, with the purpose of applying parallel processing. More information about these methods can be found in [133, 175].

3.1.2 The Conjugate Gradient method

In this thesis we focus on the *Conjugate Gradient (CG) method*, because this is the solver integrated in the ILUPACK library to tackle sparse SPD linear systems. The CG algorithm was proposed in December of 1952 by Hestenes and Stiefel [103], and it is one of the best known iterative techniques for solving sparse SPD linear systems. The method is a realization of an orthogonal projection technique onto the Krylov subspace $\mathcal{K}_k(A, r_0)$, where r_0 is the initial residual, and the residuals follow the *Ritz-Galerkin* approximation. Because of A is symmetric, some simplifications can be applied to derive the method.

There are different alternatives to obtain CG; one of them assumes that **k steps of the Lanczos algorithm** have been completed. As a result of the iterative process, we obtain a tridiagonal matrix T_k, applying the orthonormal basis V_k of the Krylov subspace $\mathcal{K}_k(A, r_0)$ onto the matrix A [166],

$$\mathcal{K}_k(A, r_0) = span\{V_k\} \ , \ V_k^T V_k = I_k \Rightarrow V_k^T A V_k = T_k \ , \ T_k \text{ is tridiagonal.}$$

The derivation of the method starts from the LU factorization of the tridiagonal matrix, $T_k = L_k U_k \equiv$

$$\begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & . & . & . & \\ & & \beta_{k-1} & \alpha_{k-1} & \beta_k \\ & & & \beta_k & \alpha_k \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ \lambda_2 & 1 & & & \\ & . & . & & \\ & & \lambda_{k-1} & 1 & \\ & & & \lambda_k & 1 \end{pmatrix} \times \begin{pmatrix} \eta_1 & \beta_2 & & & \\ & \eta_2 & \beta_3 & & \\ & & . & . & \\ & & & \eta_{k-1} & \beta_k \\ & & & & \eta_k \end{pmatrix}. \quad (3.5)$$

Applying the expression included in the Algorithm 3.1, the approximation to the solution can be rewritten as,

$$x_k = x_0 + V_k(V_k^T A V_k)^{-1} V_k^T r = x_0 + V_k T_k^{-1}(V_k^T r) = x_0 + V_k U_k^{-1} L_k^{-1}(\beta_1 e_1),$$

and reorganizing the operands conveniently, the previous expression is modified as follows,

$$P_k = V_k U_k^{-1}, \quad z_k = L_k^{-1}(\beta_1 e_1) \ \Rightarrow \ x_k = x_0 + P_k z_k.$$

The definition of P_k allows to describe a recurrence which relates the vectors of V_k and P_k,

$$p_k = \eta_k^{-1}[v_k - \beta_k p_{k-1}],$$

in which the scalars are directly obtained from (3.5),

$$\lambda_k = \frac{\beta_k}{\eta_{k-1}}, \quad \eta_k = \alpha_k - \lambda_k \beta_k. \quad (3.6)$$

On the other hand, the definition of z_k and its decomposition,

$$x_k = \begin{bmatrix} z_{k-1} \\ \zeta_k \end{bmatrix}, \quad \zeta_k = -\lambda_k \zeta_{k-1},$$

allow to obtain another recurrence to update the solution of the system,

$$x_k = x_{x-1} + \zeta_k p_k.$$

These expressions are the basis to define CG as a variant of the Lanczos method for the solution of linear systems.

We note that this method has been obtained by applying Gaussian elimination without pivoting on a triangular system, so it can fail if the chosen pivot is zero. There are several alternatives to solve this problem such as, for example, the use of QR or LQ factorizations, the incorporation of pivoting,... [166].

Another alternative to derive CG exploits **the numerical properties of the vectors** involved in the method. It is known that the residual vectors are orthogonal, given that $r_k = \sigma_k v_{k+1}$, but we do not know about the orthogonality of the auxiliary vectors, p_k. If we apply the definition of P_k on the product $P_k^T A P_k$,

$$P_k^T A P_k = (V_k U_k^{-1})^T A V_k U_k^{-1} = U_k^{-T} V_k^T A V_k U_k^{-1} = U_k^{-T} T_k U_k^{-1} = U_k^{-T} L_k,$$

we can observe that the product, which result should be a symmetric matrix, is also equal to the product of two lower triangular matrices, which is lower triangular. The only possibility to achieve both conditions is that the product is a diagonal matrix, and therefore the auxiliary vectors are A-orthogonal, or conjugate.

These properties can be used to derive CG, starting from the recursions relating the auxiliary vectors with the solution vectors and the residual vectors,

$$x_{k+1} = x_k + \alpha_k p_k, \quad r_{k+1} = r_k - \alpha_k A p_k. \tag{3.7}$$

Exploiting the residuals orthogonality, we can calculate the value of α_k ,

$$r_k^T r_{k+1} = r_k^T (r_k - \alpha_k A p_k) = 0 \;\Rightarrow\; \alpha_k = \frac{r_k^T r_k}{r_k^T A p_k}. \tag{3.8}$$

The auxiliary vectors, p_{k+1}, are defined as a lineal combination of r_{k+1} and p_k,

$$p_{k+1} = r_{k+1} + \beta_k p_k, \tag{3.9}$$

from which it is possible to derive an expression that modifies the definition of α_k,

$$r_k^T A p_k = (p_k - \beta_{k-1} p_{k-1})^T A p_k = p_k^T A p_k \;\Rightarrow \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}. \tag{3.10}$$

Moreover, taking advantage of the A-orthogonality of the auxiliary vectors, we can multiply (3.9) by Ap_k to calculate the value of β_k,

$$\beta_k = -\frac{r_{k+1}^T A p_k}{p_k^T A p_k}, \tag{3.11}$$

on which we can apply the definition of the residual vectors that appears in (3.7), obtaining

$$A p_k = \frac{1}{\alpha k}(r_{k+1} - r_k) \;\Rightarrow \beta_k = -\frac{1}{\alpha k}\frac{r_{k+1}^T (r_{k+1} - r_k)}{(p_k^T A p_k)} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}. \tag{3.12}$$

Figure 3.1 displays the final algorithm, in which the scalar α_k does not coincide with the elements of T_k.

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	**Iterative CG solve**
O1. $v_j := Ap_j$	O1. SpMV
O2. $\alpha_j := \sigma_j / p_j^T v_j$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $\sigma_{j+1} := r_{j+1}^T r_{j+1}$	O5. DOT
O6. $\beta_j := \sigma_{j+1} / \sigma_j$	O6. SCALAR OPERATION
O7. $p_{j+1} := r_{j+1} + \beta_j p_j$	O7. XPAY (similar to AXPY)
O8. $\tau_{j+1} :=\| r_{j+1} \|_2= \sqrt{\sigma_{j+1}}$	O8. 2-NORM
$j := j + 1$	
end while	

Figure 3.1: Algorithmic formulation of CG. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

3.2 Preconditioned CG

The methods presented in previous section are well founded theoretically, but they usually suffer from slow convergence for problems which arise from typical applications such as fluid dynamics or electronic device simulation. In all these cases, preconditioning is a key ingredient for the success of Krylov subspace methods. This section reviews the concept of preconditioning and details how to apply preconditioning to CG in order to improve its convergence. Later, we focus on the ILU preconditioning techniques because they are applied in ILUPACK.

3.2.1 Introductory concepts of preconditioning

Lack of robustness is a widely recognized weakness of iterative solvers. In general, these methods quickly converge if the matrix of the system is a good approximation to the identity matrix I_n, but this condition rarely fulfills. This drawback hampers the acceptance of iterative methods in industrial applications despite their intrinsic appeal for the solution of very large linear systems. Both the efficiency and robustness of iterative techniques can be improved by integrating some sort of preconditioning, which is simply a means of transforming the original linear system into one which has the same solution, but which is usually easier to solve with an iterative solver. In general, the reliability of iterative techniques, when dealing with various applications, depends more on the quality of the preconditioner than on the particular Krylov subspace method used.

Let us consider the options available for preconditioning a system. The first step is to find a preconditioning matrix M, which can be defined in many different ways as long as if satisfies a few minimal requirements. From a practical point of view, the most important requirement for M is that the linear system $My = z$ shoult be inexpensive to solve, because the preconditioned algorithms require a linear system solution with the matrix M at each step. Moreover, from a numerical point of view, M should be clearly nonsingular, and it should be related to A in some sense, so that the new matrix $(M^{-1}A)$ is almost the identity.

Once a preconditioning matrix M is available, there are three known ways of applying the preconditioner [166]. The precondiotioner can be applied from the left, leading to the preconditioned system:

$$M^{-1}Ax = M^{-1}b \; . \tag{3.13}$$

Alternatively, it can also be applied from the right:

$$AM^{-1}\hat{x} = b,\ x \equiv M^{-1}\hat{x}. \tag{3.14}$$

Note that the above formulation amounts to making the change of variables $\hat{x} = Mx$, and solving the system with respect to the unknown $\hat{x}$.

Finally, a common situation is when the preconditioner is available in the factored form $M = M_L M_R$ where, typically, M_L and M_R are triangular matrices. In this situation, the preconditioning can be split as

$$M_L^{-1} A M_R^{-1}\hat{x} = M_L^{-1} b,\ x \equiv MR^{-1}\hat{x}. \tag{3.15}$$

When the original matrix is symmetric it becomes imperative to preserve symmetry, and the split preconditioner seems mandatory. However, there are other ways of preserving symmetry, or rather to take advantage of symmetry, even if M is not available in a factored form.

Finding a good preconditioner to solve a given sparse linear system is often viewed as a combination of art and science. Theoretical results are rare and some methods work surprisingly well, often despite expectations. Usually, a preconditioner can be defined as any subsidiary approximate solver which is combined with an outer iteration technique, typically one of the Krylov subspace iterations. For example, scaling all rows of a linear system to make the diagonal elements equal to one is an explicit form of preconditioning. The resulting system can be solved by a Krylov subspace method and may require fewer steps to converge than with the original system (although this is not guaranteed). As another example, solving the linear system

$$M^{-1}Ax = M^{-1}b,$$

where M^{-1} is some complicated mapping that may involve Fast Fourier Transform (FFT) transforms, integral calculations, and subsidiary linear system solutions, may be another form of preconditioning. Here, it is unlikely that the matrices M and $M^{-1}A$ can be computed explicitly. Instead, the iterative processes operate with A and with M^{-1} whenever needed. In practice, the preconditioning operation M^{-1} should be inexpensive to apply to an arbitrary vector.

In general, the preconditioning techniques can be classified into two groups with the following characteristics:

- SPECIFIC-PURPOSE TECHNIQUES: These techniques are designed for a specific set of applications, such as those managed by PDEs of a concrete type. In many cases, it is possible to detect and use information derived from the physics of the problem to construct very efficient preconditioners. These techniques require a complete knowledge of the subjacent application, and minor changes in the application can sometimes jeopardize the efficiency of the method. In this category we can find the multigrid [134, 181] and domain decomposition [157, 175] techniques.

- GENERAL-PURPOSE TECHNIQUES: These techniques are strictly algebraic, and calculate M from the matrix of the original system, without any information on the application. Although they do not obtain the optimal solution, they offer more than acceptable performance for a wide range of applications. The algebraic preconditioners are easier to compute and use, and it is usually possible to improve their efficiency with specific adjustements for each problem. Inside this group we can encounter preconditioners designed for a specific class of matrices, as for example, preconditioners for SPD matrices or M-matrices [166]. The preconditioning techniques based on the approximate factorization of A, or its inverse A^{-1}, such as ILU [166] (Incomplete LU) preconditioners or SPAINV [166] (SParse Approximate Inverse) preconditioners respectively, are examples of this class of preconditioning techniques. We dive deeper into the ILU preconditioners in Section 3.2.3.

3.2.2 Definition of PCG

Consider a SPD matrix A and suppose that a preconditioner M is available. We can then assume that M is also SPD. Then, one can precondition the system in any of the three manners shown in the previous section, i.e., as in (3.13), (3.14), or (3.15). Note that, in general, the first two systems are no longer symmetric. In this section we consider strategies for preserving symmetry when introducing the preconditioner in the CG [166].

When M is available in the form of an incomplete Cholesky factorization,

$$M = LL^T ,$$

then a simple way to preserve symmetry is to use the "split" preconditioning option (3.15) which yields the SPD matrix,

$$L^{-1}AL^{-T}\hat{x} = L^{-1}b, \quad x = L^{-T}\hat{x}. \tag{3.16}$$

However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that $M^{-1}A$ is self-adjoint for the M-inner product,

$$(x, y)_M \equiv (Mx, y) = (x, My)$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M.$$

Therefore, an alternative is to replace the usual Euclidean inner product in the CG algorithm by the M-inner product.

If the CG algorithm is rewritten with this new inner product, and $r_j = b - Ax_j$ denotes the original residual and $z_j = M^{-1}r_j$ the residual for the preconditioned system, the scalars calculated in CG can be obtained as follows

$$\alpha_k = \frac{(z_k,z_k)_M}{(M^{-1}Ap_k,p_k)_M} = \frac{(r_k,z_k)}{(Ap_k,p_k)}, \quad \beta_k = \frac{(z_{k+1}, z_{k+1})_M}{(z_k, z_k)_M} = \frac{(r_{k+1}, z_{k+1})}{(r_k, z_k)}.$$

These expressions are the basis to define the preconditioned version of CG, corresponding to the algorithm in Figure 3.2.

3.2.3 ILU Preconditioning Techniques

The triangular factors obtained from the LU factorization of a sparse matrix A are usually denser than A, due to the fill-in produced during the factorization process. Despite the development of very efficient reordering heuristic techniques to reduce the fill-in [63, 72], direct methods are not feasible to solve extremely large systems of equations, due to the computational and storage resources they require. A simple and intuitive solution to tackle this problem consists in rejecting some elements of the matrices whose value changes from zero to non-zero during the factorization process, obtaining, thus, two triangular factors, $\tilde{L} \in \mathbb{R}^{n\times n}$ lower triangular unit and $\tilde{U} \in \mathbb{R}^{n\times n}$ upper triangular unit, so that $A \approx \tilde{L}\tilde{U}$. The process that obtains this approximation is called ILU factorization of A, and $\tilde{L}$ and $\tilde{U}$ are the incomplete factors of A. Subsequently, $M = \tilde{L}\tilde{U}$ can be applied as preconditioner of (3.1) in the hope that $M^{-1}A \approx I$.

In this section, we introduce the basic methods and techniques that can be applied to derive preconditioners from ILU factorizations of A [131]. We first describe a general ILU preconditioner, and then we discuss the ILU(0) factorization, the simplest form of the ILU preconditioners. Finally, we show how to obtain more accurate factorizations, some of which are used in ILUPACK.

O0. $A \rightarrow M$	O0. PRECONDITIONER COMPUTATION
Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	**Iterative CG solve**
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j / p_j^T v_j$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $z_{j+1} := M^{-1} r_{j+1}$	O5. PRECONDITIONER APPLICATION
O6. $\sigma_{j+1} := r_{j+1}^T z_{j+1}$	O6. DOT
O7. $\beta_j := \sigma_{j+1} / \sigma_j$	O7. SCALAR OPERATION
O8. $p_{j+1} := z_{j+1} + \beta_j p_j$	O8. XPAY (similar to AXPY)
O9. $\tau_{j+1} :=\| r_{j+1} \|_2$	O9. 2-NORM
$j := j + 1$	
end while	

Figure 3.2: Algorithmic formulation of PCG. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

Gaussian Elimination and its approximate variant

A general algorithm for building *ILU* factorizations can be derived by performing *Gaussian elimination* and dropping some elements in predetermined non-diagonal positions [88, 183]. Starting with $A_0 = A$, the process applies, at each iteration $k = 1,2,\ldots,n-1$, a *Gaussian transform* to $A_{k-1} \in \mathbb{R}^{n \times n}$, which is the resultant matrix of applying the $k-1$ previous transforms to A_0. This transform annihilates the elements of the k-th column of A_{k-1} which are below the main diagonal. If we denote as $A_k = (a_{ij}^{(k)})$ the elements of the resultant matrix after applying k Gaussian transformations, the effect of the transformation in the k-th iteration can be expressed as the following matrix product

$$A_k = L_k^{-1} A_{k-1}, \tag{3.17}$$

with

$$L_k^{-1} = I - \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0_k \\ a_{*k}^{(k-1)} \end{pmatrix} e_k^T, \tag{3.18}$$

and the vector $a_{*k}^{(k-1)} \in \mathbb{R}^{n-k,1}$ is made by the elements below the main diagonal of the k-th column of A_{k-1}:

$$a_{*k}^{(k-1)} = (a_{k+1k}^{(k-1)} \cdots a_{nk}^{(k-1)})^T. \tag{3.19}$$

It is easy to demonstrate that with L_k^{-1} defined as in (3.18) and $k = 1,2,\ldots,n-1$, the application of $n-1$ Gaussian transforms to $A_0 = A$ reduce this matrix to an upper triangular matrix:

$$L_{n-1}^{-1} \ldots L_1^{-1} A = U \rightarrow A = (L_1 \ldots L_{n-1}) U = LU. \tag{3.20}$$

Furthermore, given that

$$L_k = I + \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0_k \\ a_{*k}^{k-1} \end{pmatrix} e_k^T \tag{3.21}$$

it is easy to verify that $L = L_1 \ldots L_{m-1}$ is lower triangular unit.

Working with the expressions (3.17)–(3.21), the derivation of the *right-looking* algorithmic variant of the LU factorization (in the Algorithm 3.2) is immediate. The elements of L and U are denoted, respectively, as l_{ij} and u_{ij}. In each iteration of the external loop, the strictly lower triangular part of the k-th column of L (line 3) satisfies (3.21), and also the k-th row of U (line 4), which coincides with the k-th row of A_{k-1}. Afterwards, the two nested internal loops are responsible of applying the Gaussian transform corresponding to the k-th iteration, that is calculating A_k according to (3.17).

At this point of the dissertation, we can analyze how the *fill-in* is produced during the LU factorization of a *sparse* matrix A. Consider that $\mathcal{A}$, $\mathcal{A}_k$, $\mathcal{L}$ and $\mathcal{U}$ respectively denote the patterns of non-zero elements of A, A_k, L and U, where the last two matrices are calculated by the Algorithm 3.2. The update in line 7 of the Algorithm 3.2 produces a new element (i,j) in the $\mathcal{A}_k$ set (and, therefore, in $\mathcal{L} \cup \mathcal{U}$) if $(i,k) \in \mathcal{L}$, $(k,j) \in \mathcal{U}$ and $(i,j) \notin \mathcal{A}_{k-1}$. This new element is a *fill-in*, though this term is also used to refer to the $(\mathcal{L} \cup \mathcal{U}) - \mathcal{A}$ set, i.e, to refer the positions corresponding to non-zero elements in the factors associated to zero elements in A.

LU factorization algorithms for sparse matrices exploit two observations with the objective of reducing computational and storage cost:

1. It is not necessary to carry out the update if $(i,k) \notin \mathcal{L}$ or $(k,j) \notin \mathcal{U}$.
2. It is not necessary to calculate neither store element l_{ij} if $(i,j) \notin \mathcal{A}$, and $(i,k) \notin \mathcal{L}$ or $(k,j) \notin \mathcal{U}$, for $k = 1,2,\ldots,i-1$.

Algorithm 3.2 *Right-looking* (or KIJ) algorithmic variant of the LU factorization

1: $A_0 \leftarrow A$
2: **for** $k = 1 : n-1$ **do**
3: $\quad l_{ik} \leftarrow a_{ik}^{(k-1)} / a_{kk}^{(k-1)}$ **with** $i = k+1 : n$
4: $\quad u_{kj} \leftarrow a_{kj}^{(k-1)}$ **with** $j = k : n$
5: $\quad$ **for** $i = k+1 : n$ **do**
6: $\quad\quad$ **for** $j = k+1 : n$ **do**
7: $\quad\quad\quad a_{ij}^{(k)} \leftarrow a_{ij}^{(k-1)} - l_{ik} u_{kj}$
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end for**

The LU factorization becomes an ILU factorization when some elements of the matrices involved in the elimination process are rejected. The different basic techniques to derive ILU preconditioners are distinguished by the strategy they employ to reject those elements. For example, we can reject the elements situated in those positions which do not belong to a static pattern $\mathcal{P}$. The pattern $\mathcal{P}$ must include the positions corresponding to the elements in the main diagonal ((1,1),(2,2),...,(n,n)), since, otherwise, the elimination process fails due to a division by zero.

We next review the procedure to incorporate the discard of elements into the Gaussian elimination process, and we derive the relationship between the factors that are obtained when this process is completed and the coefficient matrix of the system. For that, we assume that the discarding of elements is done following a preestablished pattern $\mathcal{P}$, though the same conclusions can be extracted using other criteria.

In the k-th iteration, the elimination process starts from an approximation to A_{k-1}, denoted by $\hat{A}_{k-1}$. Afterwards, a Gaussian transform is applied to $\hat{A}_{k-1}$: $\tilde{A}_k = \tilde{L}_k^{-1} \hat{A}_{k-1}$, and, in order to limit the number of non-zero elements of $\tilde{A}_k$, we reject certain elements of $\tilde{A}_k$ as, for example, those

situated in the positions that do not belong to $\mathcal{P}$. If $\hat{A}_k$ is the matrix resulting from the rejections in $\tilde{A}_k$, then

$$\hat{A}_k = \tilde{A}_k - R_k, \tag{3.22}$$

with $R_k = (r_{ij}^{(k)})$ defined as follows

$$r_{ij}^{(k)} = \begin{cases} 0 & \text{if } 1 \leq i \leq k \text{ or } 1 \leq j \leq k, \\ 0 & \text{if } k < i \leq n,\ k < j \leq n \text{ and } (i,j) \in \mathcal{P}, \\ \tilde{a}_{ij}^{(k)} & \text{if } k < i \leq n,\ k < j \leq n \text{ and } (i,j) \notin \mathcal{P}. \end{cases} \tag{3.23}$$

The first k rows and columns of R_k are zero because the Gaussian transform $\tilde{A}_k = \tilde{L}_k^{-1}\hat{A}_{k-1}$ only modifies (updates) elements in the resultant block of the intersection of the $n-k$ last rows and columns. Therefore, the elements of R_k corresponding to this block are zero or equal to $\hat{a}_{ij}^{(k)}$, depending on whether (i,j) belongs to $\mathcal{P}$ or not.

Taking into account the previous considerations, it is possible to derive the relationship between $\tilde{A}_0 = A_0 = A$ and $\hat{A}_{n-1}$, through the recurrence

$$\hat{A}_j = \tilde{L}_j^{-1}\hat{A}_{j-1} - R_j, \quad j = 1,2,\ldots,n-1. \tag{3.24}$$

Thus, combining the corresponding terms, we have that

$$\tilde{U} = \hat{A}_{n-1} = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_1^{-1}\hat{A}_0 - \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_2^{-1}R_1 - \cdots - \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{k+1}^{-1}R_k - \cdots - R_{n-1} \tag{3.25}$$

is an upper triangular matrix. For $s = 1,2,\ldots,n-1$, the first s rows and columns of R_s are zero, and hence it is true that $\tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{s+1}^{-1}R_s = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{s+1}^{-1}\tilde{L}_s^{-1}\cdots\tilde{L}_1^{-1}R_s$. Taking also into account that $\hat{A}_0 = \tilde{A}_0 - R_0$, then (3.25) can be simplified as

$$\tilde{U} = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_1^{-1}(A - (R_0 + \cdots + R_{n-1})) = \tilde{L}^{-1}(A - R). \tag{3.26}$$

Multiplying both parts of the equivalence by $\tilde{L}$ and solving for A, we have

$$A = \tilde{L}\tilde{U} + R, \tag{3.27}$$

where $\tilde{L} = \tilde{L}_1\cdots\tilde{L}_{n-1}$ is a lower triangular unit matrix, $\tilde{U} = \hat{A}_{n-1}$ is an upper triangular matrix, and $R = \sum_{k=0}^{n-1} R_k$ is the residual matrix. If we denote the elements of the residual matrix as $R = (r_{ij})$, then $r_{ij} = 0$ if $(i,j) \in \mathcal{P}$, and $r_{ij} = \sum_{k=0}^{min(i,j)-1} \tilde{a}_{ij}^{(k)}$ if $(i,j) \notin \mathcal{P}$. The expression (3.27) is independent of the strategy employed to reject elements, because this rule only influences the characterization of the residual matrix R. Thus, we can remark that (3.27) is the general expression of the ILU factorization.

ILU$_\mathcal{P}$ Factorization

The ILU factorization which rejects the elements that do not belong to a preestablished pattern of positions is called ILU$_\mathcal{P}$ factorization. In Algorithm 3.3 we show the *right-looking* algorithmic variant of this factorization. The first line of the algorithm initializes $\hat{A}_0$ with the elements of A that belong to $\mathcal{P}$. Each iteration of the external loop obtains the strictly lower triangular part of the k-th column of $\tilde{L}$ (line 3) and the upper part of the k-th row of $\tilde{U}$ (line 4). The two internal loops calculate $\hat{A}_k$ accordingly to (3.24). The loop in line 5 only iterates through the elements of the strictly lower triangular part of the k-th column that belong to $\mathcal{P}$; similarly, the loop in line 6 only goes through those of the upper triangular part of the k-th row that belong to $\mathcal{P}$. Therefore,

both loops do not process those combinations of i and j for which it is guaranteed that $l_{ik}u_{kj} = 0$, avoiding the calculation of unnecessary operations to obtain $\tilde{L}_k^{-1}\hat{A}_{k-1}$. Moreover, the algorithm saves additional operations because it avoids the updates corresponding to the (i,j) positions which do not belong to $\mathcal{P}$ (line 7), and it also saves memory, because only the elements belonging to $\mathcal{P}$ are stored.

Algorithm 3.3 *Right-looking* (or KIJ) algorithmic variant of the ILU$_\mathcal{P}$ factorization

1: $\hat{a}_{ij}^{(0)} \leftarrow a_{ij}$ **with** $(i,j) \in \mathcal{P}$
2: **for** $k = 1 : n-1$ **do**
3: $\quad \tilde{l}_{ik} \leftarrow \hat{a}_{ik}^{(k-1)} / \hat{a}_{kk}^{(k-1)}$ **with** $i = k+1 : n, (i,k) \in \mathcal{P}$
4: $\quad \tilde{u}_{kj} \leftarrow \hat{a}_{kj}^{(k-1)}$ **with** $j = k : n, (k,j) \in \mathcal{P}$
5: $\quad$ **for** $i = k+1 : n, (i,k) \in \mathcal{P}$ **do**
6: $\quad\quad$ **for** $j = k+1 : n, (k,j) \in \mathcal{P}$ **do**
7: $\quad\quad\quad \hat{a}_{ij}^{(k)} \leftarrow \hat{a}_{ij}^{(k-1)} - \tilde{l}_{ik}\tilde{u}_{kj}$ **if** $(i,j) \in \mathcal{P}$
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end for**

The *right-looking* variant of the ILU factorization is of theoretical interest because it can be derived immediately from the Gaussian elimination process. However, from the practical point of view, the efficient implementation of the update corresponding to the two internal loops is complex, because the storage format of the sparse matrices is usually Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC). In the ILU$_\mathcal{P}$ factorization, the downside is less serious, because the non-zero pattern of the factors is known beforehand and, therefore, if the data structures employed in the storage are created in a previous step, it is not necessary to dynamically accommodate new elements during the process. However, when the structure of the triangular factors is not known a priori, the usual solution to this problem requires a change to the data structures and/or the algorithmic variant of the ILU factorization. To accommodate the CSR (or CSC) storage format, an algorithmic variant of the factorization different to the *right-looking* is often used. Specifically, in practice we use *delayed-update LU* algorithmic variants, as for example, the *row-lazy* (IKJ) [93], the *column-lazy* [93], or the *row-column lazy* [93] (the Crout variant), on which all the updates of one iteration of the external loop are respectively applied over a row, column, or row and column. Thus, once they have been computed, it is possible to easily incorporate them into the data structures employed for the factors.

In Algorithm 3.4 we illustrate the IKJ algorithmic variant of the ILU$_\mathcal{P}$ factorization. The i-th iteration of the external loop obtains the elements of the i-th row located in the strictly lower triangular part of $\tilde{L}$ (line 5) and those of the i-th row located in the upper triangular part of $\tilde{U}$ (line 10). For this reason, CSR is the most convinient storage format for the data structures corresponding to $\tilde{L}$ and $\tilde{U}$. For each value of k, the loop of line 4 removes a new element $\hat{a}_{ik}^{(k-1)}$ of the i-th row of $\hat{A}_{k-1}$ if $(i,k) \notin \mathcal{P}$. In order to do this, the loop of line 6 subtracts to the i-th row of $\hat{A}_{k-1}$ the result of multiplying the elements of the k-th row located in the upper triangular part $\hat{A}_{k-1}$ (i.e. the upper triangular part of $\tilde{U}$) by $\tilde{l}_{ik}$. As a result of this removal, we obtain the i-th row of $\hat{A}_k$. At the end of the loop of line 4, the i-th row of $\hat{A}_{i-1}$, i.e. of $\tilde{U}$, is obtained. The efficient implementations of the IKJ algorithmic variant of the ILU$_\mathcal{P}$ factorization do not explicitly generate the $\hat{A}_k$ matrices, with $k = 0,1,\ldots,n-1$. Instead, the updates corresponding to the k loop are carried out in a vector w, which also stores the elements of the i-th row of $\tilde{L}$ in the positions

corresponding to the already dropped elements. In Algorithm 3.5, we present the IKJ variant when the w vector is employed.

Algorithm 3.4 *Row-lazy* (or IKJ) algorithmic variant of the ILU$_\mathcal{P}$ factorization

1: $\hat{a}_{ij}^{(0)} \leftarrow a_{ij}$ **with** $(i,j) \in \mathcal{P}$
2: $\tilde{u}_{1j} \leftarrow \hat{a}_{1j}^{(0)}$ **with** $j = 1:n, (1,j) \in \mathcal{P}$
3: **for** $i = 2:n$ **do**
4: **for** $k = 1:i-1, (i,k) \in \mathcal{P}$ **do**
5: $\tilde{l}_{ik} \leftarrow \hat{a}_{ik}^{(k-1)} / \hat{a}_{kk}^{(k-1)}$
6: **for** $j = k+1:n, (k,j) \in \mathcal{P}$ **do**
7: $\hat{a}_{ij}^{(k)} \leftarrow \hat{a}_{ij}^{(k-1)} - \tilde{l}_{ik}\tilde{u}_{kj}$ **if** $(i,j) \in \mathcal{P}$
8: **end for**
9: **end for**
10: $\tilde{u}_{ij} \leftarrow \hat{a}_{ij}^{(i-1)}$ **with** $j = i:n, (i,j) \in \mathcal{P}$
11: **end for**

The IKJ variant and the *right-looking variant* of the ILU$_\mathcal{P}$ factorization produce equivalent incomplete factors for the same $\mathcal{P}$, if this pattern is preestablished and does not suffer changes during the factorization [166]. On the contrary, if $\mathcal{P}$ is dynamically generated during the elimination process, then we do not have any guarantee that both variants will produce the same result.

Algorithm 3.5 *Row-lazy* (or IKJ) algorithmic variant of the ILU$_\mathcal{P}$ factorization

1: **for** $i = 1:n$ **do**
2: $w_j \leftarrow a_{ij}$ **with** $j = 1:n, (i,j) \in \mathcal{P}$
3: **for** $k = 1:i-1, (i,k) \in \mathcal{P}$ **do**
4: $w_k \leftarrow w_k / \tilde{u}_{kk}$
5: **for** $j = k+1:n, (k,j) \in \mathcal{P}$ **do**
6: $w_j \leftarrow w_j - w_k\tilde{u}_{kj}$ **if** $(i,j) \in \mathcal{P}$
7: **end for**
8: **end for**
9: $\tilde{l}_{ij} \leftarrow w_j$ **and** $w_j \leftarrow 0$ **with** $j = 1:i-1, (i,j) \in \mathcal{P}$
10: $\tilde{u}_{ij} \leftarrow w_j$ **and** $w_j \leftarrow 0$ **with** $j = i:n, (i,j) \in \mathcal{P}$
11: **end for**

ILU(0) Factorization and its generalization to ILU(l)

The ILU factorization with no fill-in, denoted by ILU(0), takes $\mathcal{P}$ to be precisely the pattern of $\mathcal{A}$. In this case, the non-zero element patterns of $\tilde{L}$ and $\tilde{U}$ coincide with the non-zero element pattern of the lower and upper triangular parts of A, respectively. Thus, $(\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}) - \mathcal{A} = \emptyset$. The same concept can be applied for the Cholesky factorization $A = LL^T$ in the case of SPD matrices, obtaining in this case an *Incomplete Cholesky (IC)* factorization with no fill-in, or IC(0).

The factorizations with no fill-in are easy to implement, their cost is relatively small, and they are also quite efficient as preconditioners on significant problems, like in systems which arise from the numeric resolution of elliptic PDEs using the finite difference method [166]. However, for more complex problems, the ILU factorizations with no fill-in usually obtain inaccurate approximations

of A, i.e. with $\|R\|$ "large" in (3.27), because they discard many elements of large size in the factorization process.

In order to increase the precision of the approximate factorization, we can include positions in $\mathcal{P}$ which are not present in the non-zero element pattern of A. In the family of *level of fill-in* preconditioners or ILU(l), these positions are selected in the set $\mathcal{L} \cup \mathcal{U}$, i.e., in the set of positions in which the non-zero elements of $L + U$ are located. Therefore $\mathcal{P} = \tilde{\mathcal{L}} \cup \tilde{\mathcal{U}} \subseteq \mathcal{L} \cup \mathcal{U}$. The l parameter is a preestablished integer number which determines the positions of the set $\mathcal{L} \cup \mathcal{U}$ that will form the pattern $\mathcal{P}$. This pattern is formulated in a prior step to the numeric factorization, known as symbolic ILU(l) factorization, which is gathered in Algorithm 3.6. In this process we only take into account the positions of the elements involved in the elimination, but not their numeric values. The algorithm assigns an integer number $\tilde{p}_{ij}$, named *level of fill-in*, to each "accepted" position (i,j), i.e., to each position selected as member of $\mathcal{P}$. In the i-th iteration, all the (i,j) positions of the i-th row which belong to the non-zero element pattern of A are accepted, and the algorithm assigns them a level of fill-in $\tilde{p}_{ij} = 0$ (line 3). Afterwards, the loop in line 4 proceeds with the elimination of the (i,k) positions, with $k < i$, which belong to $\mathcal{P}$ in *ascending order*[1]. When a specific (i,h) position is eliminated, any of the (h,j) positions crossed by the loop in line 5 may cause fill-in in the (i,j) position. In these cases, the level of fill-in w corresponding to the (i,j) position is obtained as the addition of the level of fill-in of the originating entries plus 1 (line 6). If the value of w is equal or lower than the prefixed constant l, then the (i,j) position is accepted in line 11. The fill-in of an (i,j) position may be caused by some pairs of entries $(i,k),(k,j) \in \mathcal{P}$, with $1 \leq k < min(i,j)$. Therefore, the algorithm assigns the smallest level of fill-in possible to the positions of $\mathcal{P}$ (line 9). We have to take into account that if $l = 0$, then the algorithm obtains $\mathcal{P} = \mathcal{A}$, i.e., the pattern corresponding to the ILU(0) preconditioner; whereas if $l = \infty$, then $\mathcal{P} = \mathcal{L} \cup \mathcal{U}$, i.e., that corresponding to the LU factorization of A. In Figure 3.3 we show the $\mathcal{P}$ pattern calculated by the ILU(l) symbolic factorization for an example of a sparse matrix and four different values of l.

An alternative way to interpret the above definition of fill-in level can be drawn from the graph model of Gaussian elimination, which is a standard tool used in sparse direct solvers. Algorithm 3.6 is supported by the *incomplete fill-path theorem* and its relation with the *fill-path theorem*. Consider the adjacency graph $G(A) = (V, E)$ of the matrix A. These results typify the presence of an (i,j) position in $\mathcal{P}$ and in $\mathcal{L} \cup \mathcal{U}$, respectively, according to the *fill-paths* of the adjacency graph $G(A)$. A fill-path in $G(A)$ is a path between two vertices i and j, such that all the vertices in the path, except the end points i and j, are numbered less than i and j. The fill-path theorem establishes that there is a fill-in entry (i,j) $\in \mathcal{L} \cup \mathcal{U}$ at the completion of the Gaussian elimination process if and only if there exists a fill-path between i and j in $G(A)$. The incomplete fill-path theorem establishes that the position (i,j) belongs to $\mathcal{P}$ with fill-in level $\tilde{p}_{ij} = k$ if and only if there exists a fill-path of length $k + 1$ between i and j. Therefore, $\mathcal{P} \subseteq (\mathcal{L} \cup \mathcal{U})$. Furthermore, the set $(\mathcal{L} \cup \mathcal{U}) - \mathcal{P}$ is composed by the (i,j) positions for which all the fill-paths from i to j have a length longer than $l + 1$. Hence, if we measure the distance between two vertex in $G(A)$ in terms of fill-path length, the ILU(l) preconditioner accepts those (i,j) positions for which the i, j vertex are "nearby" in $G(A)$.

In most cases, the efficiency of the ILU(l) factorization as preconditioner, in terms of its capacity to accelerate the convergence of the iterative method, significantly improves by considering addi-

[1] In Algorithms 3.4 and 3.5 it was also necessary to eliminate the elements in ascending order. However, satisfying this requirement in Algorithm 3.6 can significantly increase the cost because the pattern $\mathcal{P}$ can be different for two different iterations of the loop in line 4 and, therefore, it is necessary to maintain the order in the data structure used for $\mathcal{P}$. In general, it is necessary to maintain this order for all the factorization algorithms based on the IKJ variant which dynamically generates the structure of each row of the factors.

Algorithm 3.6 ILU(l) symbolic factorization

```
1: P ← ∅
2: for i = 1 : n do
3:    insert (P, (i,j)) and p̃_ij ← 0 with (i,j) ∈ A
4:    for k = 1 : i − 1 in ascending order, (i,k) ∈ P do
5:       for j = k + 1 : n, (k,j) ∈ P do
6:          w ← p̃_ik + p̃_kj + 1
7:          if w ≤ l then
8:             if (i,j) ∈ P then
9:                p̃_ij ← min(p̃_ij, w)
10:            else
11:               insert (P, (i,j))
12:               p̃_ij ← w
13:            end if
14:         end if
15:      end for
16:   end for
17: end for
```

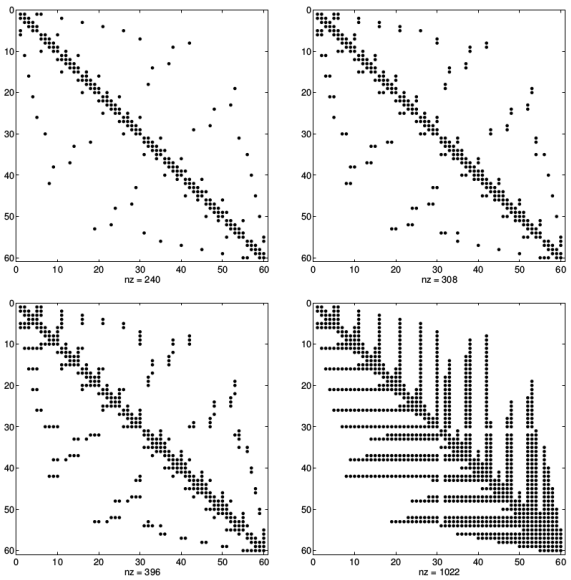


Figure 3.3: $\mathcal{P}$ pattern computed by the ILU(l) symbolic factorization for an example of a sparse matrix and four different values of l. From left to right and from top to bottom: $l = 0,1,2$ and ∞. The non-zero element pattern of A is equal to the ILU(0) factorization pattern.

tional fill-in levels, i.e., greater values of l. However, when $l \geq 1$ it is possible that $\mathcal{P}$ may get closer to $\mathcal{L} \cup \mathcal{U}$, even with relatively small values of l. Even though in Figure 3.3 we do not observe this behaviour for $l \leq 2$, a considerable number of examples of the Harwell-Boeing collection of sparse matrices have a "maximum" number of fill-in levels that is relatively small. (The maximum level of fill-in is the smallest level of fill-in l which accomplishes that the ILU(l) factorization coincides with the LU). When in those cases we need to use values of $l \geq 1$ to obtain efficient preconditioners, the solver composed by the ILU(l) preconditioner plus an iterative method is no longer efficient compared with direct methods. Another problem is that, except in concrete cases, it is not possible to predict the computational and storage costs required by the ILU(l) factorizations. However, the main disadvantage of level of fill-in preconditioners is that, for many matrices, the level of fill-in may not be a good indicator of the magnitude of the rejected elements. Thus, the algorithm can reject many elements of great magnitude, obtaining an imprecise ILU factorization, i.e., with "large" $\|R\|$. Although it does not exist a direct relation between $\|R\|$ and the efficiency of the precondictioner, the experience reveals that, usually, this factor causes an increase of the number of iterations required by the iterative method to solve the system.

Modified ILU (MILU) Factorization

In all the techniques presented thus far, the elements that were dropped during the incomplete elimination process are simply discarded. There are also techniques which attempt to reduce the effect of dropping by compensating for the discarded entries. For example, a popular strategy is to add up all the elements that have been dropped at the completion of the k-loop of Algorithm 3.4, and then accumulate this sum on the diagonal entry in U. This diagonal compensation strategy gives rise to the Modified ILU (MILU) factorization.

This strategy guarantees that the row sums of A are equal to those of LU. For PDEs, the vector of all ones represents the discretization of a constant function, and this additional constraint forces the ILU factorization to be exact for constant functions in some sense. Therefore, it is not surprising that the algorithm often does well for such problems. For other problems, or problems with discontinuous coefficients, MILU algorithms are usually not better than their ILU counterparts.

This generic idea of lumping together all the elements dropped in the elimination process and adding them to the diagonal of U can be used for any form of ILU factorization. In addition, there are variants of diagonal compensation in which only a fraction of the dropped elements are added to the diagonal.

Dropping elements according to their magnitude: ILUT(τ, p) Factorization

Incomplete factorizations which rely on the levels of fill-in are blind to numerical values because elements are dropped depending only on the structure of A. This can cause some difficulties for real problems that arise in many applications. A few alternative methods are available which are based on dropping elements in the Gaussian elimination process according to their magnitude rather than their locations. With these techniques, the non-zero pattern $\mathcal{P}$ is determined dynamically. The simplest way to obtain an incomplete factorization of this type is to take a sparse direct solver and modify it by adding lines of code which will ignore "small" elements. However, most direct solvers have a complex implementation involving several layers of data structures that may turn this approach ineffective. It is desirable to develop a strategy which is more akin to the ILU(l) approach.

A generic ILU algorithm with threshold can be derived from the IKJ version of Gaussian elimination by including a set of rules for dropping small elements. In the following, *applying a*

dropping rule to an element only means replacing the element by zero if it satisfies a set of criteria. A dropping rule can be enforced to a whole row by applying the same rule to all the elements of the row.

Algorithm 3.7 ILU(τ,p) Factorization

1: **for** $i = 1 : n$ **do**
2: $\tau_i \leftarrow \tau \|e_i^T A\|$
3: $w_j \leftarrow a_{ij}$ **with** $j = 1 : n, a_{ij} \neq 0$
4: **for** $k = 1 : i - 1$, $w_k \neq 0$ **do**
5: $w_k \leftarrow w_k / \tilde{u}_{kk}$
6: **if** $|w_k| \leq \tau_i$ **then**
7: $w_k \leftarrow 0$
8: **else**
9: **for** $j = k : n, \tilde{u}_{kj} \neq 0$ **do**
10: $w_j \leftarrow w_j - w_k \tilde{u}_{kj}$
11: **end for**
12: **end if**
13: **end for**
14: $w_j \leftarrow 0$ **with** $j = i : n, |w_j| \leq \tau_i$
15: Cut on the bias the strict lower triangular part of w to p elements
16: Cut on the bias the strict upper triangular part of w to p elements
17: $\tilde{l}_{ij} \leftarrow w_j$ **with** $j = 1 : i - 1, w_j \neq 0$
18: $\tilde{u}_{ij} \leftarrow w_j$ **with** $j = i : n, w_j \neq 0$
19: $w \leftarrow 0$
20: **end for**

The ILU(τ, p), based on the IKJ variant of the ILU factorization, is shown in Algorithm 3.7. The τ parameter is a positive real number, called *discard tolerance*, which controls the magnitude of the elements accepted in $\tilde{L}$ and in $\tilde{U}$; and the p parameter is a positive integer which controls the maximum number of elements accepted in each row of $\tilde{L}$ and $\tilde{U}$. The loop in line 4 only rejects the elements w_k ($\hat{a}_{ik}^{(k-1)}$) of the i-th row located in the strictly lower triangular part which are non-zero. An element $w_k \neq 0$ is rejected (line 7) if the magnitude of $w_k/\tilde{u}_{kk}$ is less than or to the threshold τ_i, saving thus the updates concerning its elimination. In the opposite case, w_k is eliminated according to Algorithms 3.4–3.5, although only the elements w_j for which $\tilde{u}_{kj} \neq 0$ are updated, and the fill-in caused by the updates in line 10 is accepted *temporarily*. After finalizing the loop in line 4, the elements of the upper triangular part of w of magnitude less than τ_i are rejected in line 14. Afterwards, in line 15 (and 16) the elements of magnitude less than the p elements of large magnitude of w in the strict lower (and upper) triangular parts are rejected. In this way, $|\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}| \leq 2np$. Finally, in line 17 (18) the i-th row is added to the data structure which stores $\tilde{L}(\tilde{U})$.

Algorithm 3.7 employs a discard tolerance *relative* to the magnitude of the i-th row of A to reject the elements corresponding to the i-th row of $\tilde{L}$ (line 7) and $\tilde{U}$ (line 14). This relative tolerance, τ_i, is obtained in line 2 as the product of the *absolute* discard tolerance, τ, and the norm of the i-th row of A, $\|e_i^T A\|$. The discarding criteria based on relative tolerance are, in general, more reliable than the criteria based exclusively on the absolute tolerance, and they usually yield a good solution. A common disadvantage for these criteria is the selection of a satisfactory value for τ. For this selection, it is common to start from a representative subset of the linear systems to be solved in the context of a specific application and, through experimentation, we can determine a

good value for the systems which arise from the same application. In most cases, selecting values of τ in the interval [10^{-4},10^{-2}] yield good results, though the optimum value depends on the concrete problem [165].

In the ILUT factorization process, the elements of w_k which are rejected in line 7 *are not removed*, saving in this way the updates of the loop in line 9. Note the difference between removing and rejecting: in order to remove an element w_k, it is necessary to complete the updates corresponding to the loop in line 9, and afterwards, w_k stores the value $w_k/\tilde{u}_{kk}$, i.e., the entry corresponding to $\tilde{L}$. However, when an element w_k is rejected, this element is "artificially" substituted by the zero value. A possible modification of Algorithm 3.7 may remove all the non-zero elements of the strictly lower triangular part, and then reject the elements of $\tilde{L}$ which have a magnitude lower than τ_i in line 14, and analogously to $\tilde{U}$. Unfortunately, it is possible to demonstrate that this modification of the ILUT factorization not only increments the computational cost, but also decreases the accuracy of the approximation [61]. Thence, there exists a mathematical justification to maintain the rejected elements. So, from the mathematical point of view, cutting on the bias the rows of $\tilde{L}$ is not considered a good practice, even though it supposes not having a priori a threshold for $|\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}|$. Again, the best combination for the p and τ values depends on the specific problem, though we can usually obtain good results by adjusting only the value of τ and fixing p to a "large" value. For example, the default value for p in ILUPACK is $n + 1$ [52].

In the efficient implementations of Algorithm 3.7, w is a data structure such as, for example, a heap or a binary search tree, which maintains the order between the identifiers of the column of the non-zero elements when the fill-in is produced in w. Operations of the form "$w_k \leftarrow 0$" are implemented removing the w_k element of the data structure; those of the form "$w_j \leftarrow w_j - w_k\tilde{u}_{kj}$", updating or inserting a new element if the operation produces fill-in; whereas queries of the form "$w_k \neq 0$" are implicitly done by extracting the following element of w. When the fill-in admitted by the factorization is relatively small, the cost required to maintain this structure does not suppose a significant part of the global cost of the process. However, when more fill-in is progressively admitted (reducing the value of τ and/or increasing that of p), this cost may dictate the total time of the factorization [166].

The Crout ILU Approach: ILUC Factorization

A notable disadvantage of the standard delayed-update IKJ factorization is that it requires access to the entries in the k-th row of $\tilde{L}$ in sorted order of columns. This is further complicated by the fact that the working row (denoted by w) is dynamically modified by fill-in as the elimination proceeds. Searching for the leftmost entry in the k-th row of $\tilde{L}$ is usually not a problem when the fill-in allowed is small. Otherwise, when an accurate factorization is sought, it can become a significant burden and may ultimately even dominate the cost of the factorization. Sparse direct solution methods that are based on the IKJ form of Gaussian elimination obviate this difficulty by a technique known as the Gilbert-Peierls method [87], but because of dropping, this technique cannot be directly used on ILU factorizations. An alternative is to reduce the cost of the searches through the use of clever data structures and algorithms, such as binary search trees or heaps [62]. The Crout formulation provides the most elegant solution to the problem. Moreover, the Crout version of Gaussian elimination has additional advantages which make it one of the most appealing ways of implementing incomplete LU factorizations.

Particularly, the Crout approach [166] is a *delayed update* of the factorization, and it does not require any order in the elimination of the non-zero elements of the i-th row, i.e., the elements $\hat{a}_{ik}^{(k-1)} \neq 0$ for $k = 1, \ldots, i-1$. In the k iteration of the external loop of this variant, the updates corresponding to the elimination of the k-th column are *delayed*, i.e., are applied in subsequent

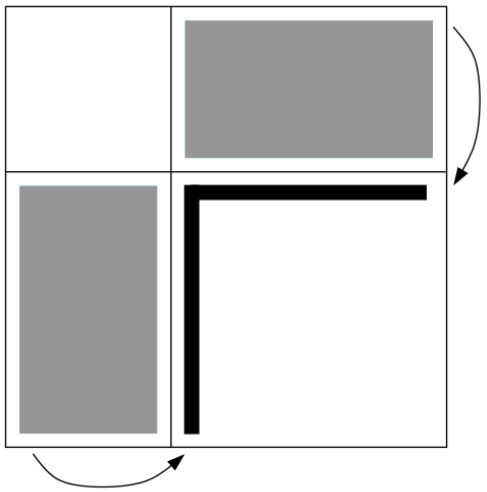


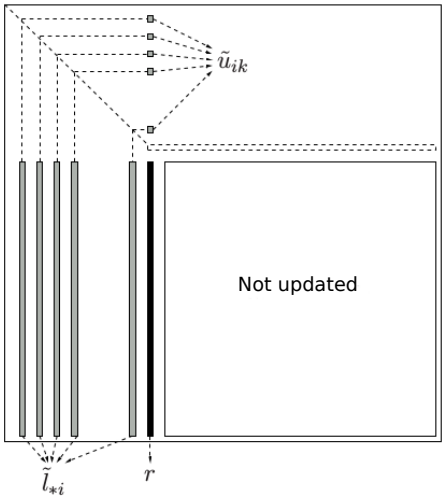
Figure 3.4: Computational pattern of the Crout algorithm.

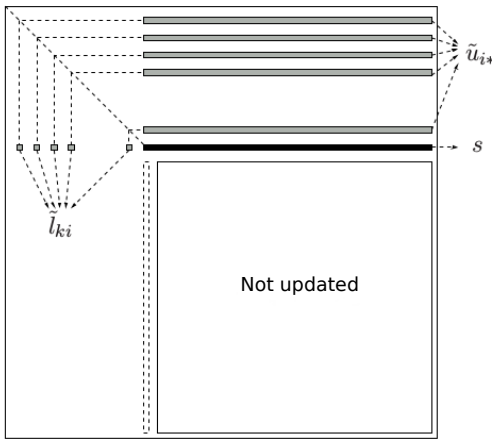
iterations. Besides, at the beginning of the k-th iteration, the updates corresponding to the elimination of the $k-1$ first rows *have not been applied yet* to the block formed by the intersection of the last $n-k+1$ rows and columns. So, the k-th iteration of the Crout approach applies the delayed updates to the $k-1$ previous iterations in the first row and column of this block, i.e., to the elements of the k-th column located in the strict lower triangular part of A, and the elements in the k-th row located in the upper triangular part of A. In Figure 3.4 the parts of the factors being computed at the k-th step are shown in black and those being accessed are in the gray areas. Furthermore, in the top part of Figure 3.5 we show the elements involved in the updates of the k-th iteration of the Crout factorization and, in the bottom part, the pseudo-codes in charge of their execution. On the left part of Figure 3.5 we can recognize the update of the k-th column, and on the right part, the update of the k-th row. The rows and columns coloured in gray contain elements of $\tilde{L}$ and $\tilde{U}$ accessed in read mode; the row and column in black contain the elements which are updated in the k-th iteration; r and s are work vectors, and

$$\tilde{l}_{*i} = (\tilde{l}_{k+1,i} \cdots \tilde{l}_{n,i})^T \ , \ \tilde{u}_{i*} = (\tilde{u}_{i,k} \cdots \tilde{u}_{i,n}) \ , \ i = 1, \ldots, k-1. \tag{3.28}$$

In the pseudo-codes of Figure 3.5, the updates of r and s can be applied *in any order*, which represents a significant advantage with respect to the IKJ variant from the point of view of the factorization cost and its implementation. Actually, in the efficient implementation of the Crout ILU factorization, r and s are, substantially, two dense work vectors [124]. The sparse structure of the problem is exploited saving updates if $\tilde{u}_{ik} = 0$ or $\tilde{l}_{ji} = 0$ for the pseudo-code on the left part of Figure 3.5, or if $\tilde{l}_{ki} = 0$ or $\tilde{u}_{ij} = 0$, for that on the right part. Moreover, the operations made by the pseudo-code in the left part of the Figure are *independent* from those of the pseudo-code in the right part and, therefore, both codes can be executed in parallel. Although this kind of parallelism can be exploited to reduce the cost of the factorization in multithread processors, the scalable parallel algorithms to compute ILU factorizations exploit parallelism in which the degree of concurrency is significantly higher.

The pseudo-codes in Figure 3.5 are completed applying discard rules to the work vectors r and s, scaling the elements of r by $(u_{kk})^{-1}$, putting the non-zero elements of r on the k-th column of $\tilde{L}$ and the non-zero elements of s on the k-th row of $\tilde{u}$, and iterating for $k = 1, 2, \ldots, n$. The *Incomplete LU Crout (ILUC) factorization* is shown in Algorithm 3.8. The more appropriate storage formats for $\tilde{L}$ and $\tilde{U}$ are, respectively, CSC and CSR, since we store $\tilde{L}$ by columns and $\tilde{U}$ by rows.





$r \leftarrow 0$
$r_i \leftarrow a_{ik}$ **with** $i = k+1:n, a_{ik} \neq 0$
for $i = 1:k-1, \tilde{u}_{ik} \neq 0$ **do**
  for $j = k+1:n, \tilde{l}_{ji} \neq 0$ **do**
    $r_j \leftarrow r_j - \tilde{l}_{ji}\tilde{u}_{ik}$
  end for
end for

$s \leftarrow 0$
$s_j \leftarrow a_{kj}$ **with** $j = k:n, a_{kj} \neq 0$
for $i = 1:k-1, \tilde{l}_{ki} \neq 0$ **do**
  for $j = k:n, \tilde{u}_{ij} \neq 0$ **do**
    $s_j \leftarrow s_j - \tilde{l}_{ki}\tilde{u}_{ij}$
  end for
end for

Figure 3.5: Updates carried out in the k-th column (left) and row (right) of A during the k-th iteration of the Crout variant of the ILU factorization.

Algorithm 3.8 ILUC Factorization

1: **for** $k = 1:n$ **do**
2: $r \leftarrow 0$
3: $r_i \leftarrow a_{ik}$ **with** $i = k+1:n, a_{ik} \neq 0$
4: **for** $i = 1:k-1, \tilde{u}_{ik} \neq 0$ **do**
5: **for** $j = k+1:n, \tilde{l}_{ji} \neq 0$ **do**
6: $r_j \leftarrow r_j - \tilde{u}_{ik}\tilde{l}_{ji}$
7: **end for**
8: **end for**
9: $s \leftarrow 0$
10: $s_j \leftarrow a_{kj}$ **with** $j = k:n, a_{kj} \neq 0$
11: **for** $i = 1:k-1, \tilde{l}_{ki} \neq 0$ **do**
12: **for** $j = k:n, \tilde{u}_{ij} \neq 0$ **do**
13: $s_j \leftarrow s_j - \tilde{l}_{ki}\tilde{u}_{ij}$
14: **end for**
15: **end for**
16: Apply discard to r
17: Apply discard to s
18: $\tilde{u}_{kj} \leftarrow s_j$ **with** $j = k:n, s_j \neq 0$
19: $\tilde{l}_{jk} \leftarrow r_j/\tilde{u}_{kk}$ **with** $j = k+1:n, r_j \neq 0$
20: **end for**

From the point of view of an efficient implementation of Algorithm 3.8, two potential sources of difficulty will require a careful and somewhat complex implementation. First, looking at lines 6 and 13, only the section $(k+1:n)$ of the i-th column of $\tilde{L}$ is required, and similarly, only the section $(k:n)$ of the i-th row of $\tilde{U}$ is needed. Second, line 4 has to access to the k-th column of $\tilde{U}$, which is stored by rows, while line 11 accesses the k-th row of $\tilde{L}$, stored by columns.

The first issue can be easily handled by keeping pointers that indicate where the relevant part of each row of $\tilde{U}$ (respectively column of $\tilde{L}$) starts. An array `Ufirst` of size n can be used to store, for each row i of $\tilde{U}$ the index of the first column that will be used next. If k is the current step number, this means that `Ufirst(i)` would hold the first column $index \geq k$ of all non-zero entries in the i-th row of $\tilde{U}$; in the case of $\tilde{L}$ we would use an analogous vector, `Lfirst`. These pointers are easily updated after each elimination step, assuming that column indices (respectively column indices for $\tilde{L}$) are in increasing order.

For the second issue, consider the situation for the $\tilde{U}$ factor, in which the problem is that the k-th column of $\tilde{U}$ is required for the update of $\tilde{L}$, but $\tilde{U}$ is stored row-wise. An elegant solution to this problem is known since the pioneering days of sparse direct methods [76, 86], but before discussing this idea, consider the simpler solution of including a *linked list* for each column of $\tilde{U}$. These linked lists, implemented using a work vector `Ulist`, would be easy to update because the rows of $\tilde{U}$ are computed one at a time. Each time a new row is computed, the non-zero entries of this row are queued to the linked lists of their corresponding columns. However, this scheme would entail non-negligible additional storage. A clever alternative would be to exploit the array `Ufirst` mentioned above to form incomplete linked lists of each column [124]. With this implementation, `Ulist(`k`)` stores the row identifier of the first non-zero element of the column k of $\tilde{U}$, `Ulist(Ulist(`k`))` contains the row identifier of the second, and so on, until the last element of the list. If the last element of the list belongs to the row j, then `Ulist(`j`)` has the value 0. The interesting point is that, though the columns structures constructed in this manner are incomplete, they become complete as soon as they are needed. A similar technique is used for the rows of the $\tilde{L}$ factor.

The solution described in the previous paragraph was used in the first packages of direct methods, although it felt into disuse when the problem dimensions increased. Current software packages of direct methods employ techniques to efficiently manage the sparse structure of the problem [63]. However, in [124] was showed that the use of linked lists for the ILUC factorization is an efficient solution because the accepted fill-in to obtain an effective preconditioner is, in general, significantly lower. In fact, linked lists are currently employed in leading preconditioned software packages, such as ILUPACK.

In addition, to avoiding searches, the Crout version of ILU has another important advantage. The straightforward dropping rules used in ILUT can be easily adapted for ILUC, and it also enables some new dropping strategies which may be viewed as more rigorous than the standard ones presented so far. Thus, the data structure of ILUC allows options which are based on estimating the norms of the inverses of $\tilde{L}$ and $\tilde{U}$. Moreover, it is important to emphasize that the ILUT and ILUC factorizations do not generally obtain the same approximation of A, even if they employ the same parameters to the discard rules. Accordingly, in the ILU factorizations with discard that are based on the magnitude of elements, the algorithmic variant is also important in the quality of the approximation.

ILDU and AINV Factorizations

In this section we present two factorizations within the preconditioning techniques based on the approximate factorizations of A and A^{-1}. Firstly, we consider the LDU factorization of A and the

corresponding *Incomplete LDU (ILDU)* factorization, and later the foundations of the AINV are introduced.

The factorization process of A decomposes the coefficient matrix of the system into the product $A = LDU$, where $L \in \mathbb{R}^{n\times n}$ is unit lower triangular, $D \in \mathbb{R}^{n\times n}$ is diagonal, and $U \in \mathbb{R}^{n\times n}$ is unit upper triangular. When A is symmetric, we use the expression "LDL^T factorization", as $U = L^T$. Moreover, if it is positive definite, all the elements of the main diagonal of D are positive. Consider the $A = \hat{L}\hat{U}$ factorization, with $diag(\hat{U}) = (\hat{u}_{11}\hat{u}_{22}\cdots\hat{u}_{nn})$, then $L = \hat{L}, DU = \hat{U}$ and, therefore,

$$D = \begin{pmatrix} \hat{u}_{11} & 0 & 0 & \cdots & 0 \\ 0 & \hat{u}_{22} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \hat{u}_{nn} \end{pmatrix} \tag{3.29}$$

If we exploit the relation between L, D and U factors and $\hat{L}$ and $\hat{U}$, the derivation of the KIJ algorithmic variant of this factorization is immediate from Algorithm 3.2. Similar changes must be applied to derive the Crout and the IKJ approaches of the ILDU factorization. It is easy to check that, if we partition the coefficient matrix as

$$A = \begin{pmatrix} \beta & d^T \\ c & E \end{pmatrix}, \tag{3.30}$$

with $\beta \in \mathbb{R}$, and c, d^T and E of size according to β, then

$$A = \begin{pmatrix} \beta & d^T \\ c & E \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ p & I \end{pmatrix} \begin{pmatrix} \delta & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} 1 & q^T \\ 0 & I \end{pmatrix}, \tag{3.31}$$

with $\delta = d_{11}, p = (l_{21}\cdots l_{n1})^T, q^T = (u_{12}\cdots u_{1n})$ and $S = (a_{ij}^{(1)})_{i,j=2\cdots n}$ formed from the elements obtained after the execution of the first iteration of the KIJ variant of the LDU factorization. Matrix $E - p\delta q^T$ is called *Schur complement of* β in A and corresponds to the matrix resulting of the application on A of the updates corresponding to the elimination of c. Equation (3.31) is called *partial* LDU factorization, because the $n-1$ remaining columns have not been removed. This factorization is completed recursively by applying (3.30) and (3.31) to S.

The partial factorization (3.31) is converted into an incomplete partial ILDU factorization if we reject elements of p and q^T, usually those of lower magnitude than a threshold τ. If we denote the result of rejecting some elements of p and q^T, respectively, as $\tilde{p}$ and $\tilde{q}^T$ then the *approximate* Schur complement is defined as

$$\tilde{S} = E - \tilde{p}\delta\tilde{q}^T \approx S. \tag{3.32}$$

The partial ILDU factorization is completed by applying recursively the same process to $\tilde{S}$.

The *AINV* factorizations [41, 42] directly obtain the incomplete factors of A^{-1}. When we do not discard elements during the process, these factorizations compute two unit upper triangular matrices, $W \in \mathbb{R}^{n\times n}$ and $Z \in \mathbb{R}^{n\times n}$, and a diagonal matrix, $D \in \mathbb{R}^{n\times n}$, so that $W^TAZ = D$; i.e., $A^{-1} = ZD^{-1}W^T$. If A admits an LDU factorization, then $W = L^{-T}$ and $Z = U^{-1}$. Therefore,

$$W^TA = DU \text{ and } Z^TA^T = DL^T. \tag{3.33}$$

The corresponding algorithm appears in [166].

Inverse-based dropping rules: ILDU_INV factorization

The main objective of *Inverse-based dropping rules* is to obtain ILU preconditioners less sensitive to the errors produced during the dropping of the elements of the factors. In [45, 46] these rules were evaluated in combination with pivoting techniques, whereas in [124] they were efficiently implemented taking advantage of the ILUC factorization, but without pivoting. In this section, for simplicity, we only explain the dropping rules, and omit the use of pivoting.

The inverse-based dropping rules are derived from the relation between the ILU and AINV factorizations. In [50] it was demonstrated that it is possible to calculate two incomplete factors $\tilde{L}$ and $\tilde{U}$ such that $L^{-T} \approx \tilde{W}$ and $\tilde{U}^{-1} \approx \tilde{Z}$, with $\tilde{W}$ and $\tilde{Z}$ the factors computed by the AINV factorization, [50, 131]. Concretely, the proximity of these factors suggests that, in the dropping rules, $\|e_k^T \tilde{L}^{-1}\|$ and $\|\tilde{U}^{-1} e_k\|$ can replace $\|\tilde{W} e_k\|$ and $\|\tilde{Z} e_k\|$, respectively. After applying this substitution, and taking the infinite norms $\|e_k^T \tilde{L}^{-1}\|$ and $\|\tilde{U}^{-1} e_k\|$, the *inverse-based rules* drop an element $\tilde{l}_{ik}$, with $i > k$ when

$$|\tilde{l}_{ik}| \cdot \|e_k^T \tilde{L}^{-1}\|_\infty \leq \tau; \tag{3.34}$$

and an element $\tilde{u}_{ki}$ when

$$|\tilde{u}_{ki}| \cdot \|\tilde{U}^{-1} e_k\|_\infty \leq \tau. \tag{3.35}$$

To reject those elements $\tilde{l}_{ik}(\tilde{u}_{ki})$ which satisfy (3.34) ((3.35)) in the k-th iteration of the ILDU factorization, we need estimations of the infinite norm of the k-th row (column) of $\tilde{L}^{-1}(\tilde{U}^{-1})$, i.e., estimations of $\|e_k^T \tilde{L}^{-1}\|_\infty$ ($\|\tilde{U}^{-1} e_k\|_\infty$).

Assume that $L \in \mathbb{R}^{n \times n}$ is a sparse matrix with unit lower triangular structure, and consider that we want to obtain estimations of $\|e_k^T L^{-1}\|_\infty$, for $k = 1, \ldots, n$. Taking into account that the infinite matrix norm [88] of $e_k^T L^{-1} \in \mathbb{R}^{1,n}$ is defined as

$$\|e_k^T L^{-1}\|_\infty = \max_{\|b\|_\infty = 1} \|e_k^T L^{-1} b\|_\infty = \max_{\|b\|_\infty = 1} |e_k^T L^{-1} b|, \tag{3.36}$$

then, for each vector $\hat{b}$ with $\|\hat{b}\|_\infty = 1$, $\|e_k^T L^{-1}\|_\infty \geq |e_k^T L^{-1} \hat{b}|$. If we get a vector $\hat{b}$ for which the magnitude of $|e_k^T L^{-1} \hat{b}|$ is near its upper bound $\|e_k^T L^{-1}\|_\infty$, with $k = 1, \ldots, n$ then the solution components of the system $Lx = \hat{b}$ can be employed as estimators of $\|e_k^T L^{-1}\|_\infty$; i.e., $|x_k| \approx \|e_k^T L^{-1}\|_\infty$, for $k = 1, \cdots, n$.

In Algorithm 3.9 we illustrate the steps to compute estimators $t_k^L \approx \|e_k^T L^{-1}\|_\infty$ for $k = 1, \cdots, n$. In fact, the factorization in ILUPACK [52] includes by default this calculation combined with the ILUC factorization. However, we should emphasize that the method employed in ILUPACK computes, in each iteration k of Algorithm 3.9, a second round in which it attempts to improve the quality of the estimator [46]. Furthermore, the ILU factorization in ILUPACK incorporates pivoting to control the magnitude of $\|e_k^T \tilde{L}^{-1}\|_\infty$ and $\|\tilde{U}^{-1} e_k\|_\infty$ during the factorization process with the objective of improving the quality of the preconditioner [46, 51].

Multigrid methods

The convergence of preconditioned Krylov subspace methods for solving linear systems arising from discretized PDEs tends to slow down considerably as these systems become larger. This degradation in the convergence rate, compounded with the increased operation count per step due to the problem size, results in a severe loss of efficiency. In contrast, the class of methods in this section, *Multigrid methods*, are capable of achieving convergence rates which are, in theory, independent of the mesh size. One significant difference with the preconditioned Krylov subspace approach is that Multigrid methods were initially designed specifically for the solution of discretized elliptic

Algorithm 3.9 Estimator of $\|e_k^T L_\infty^{-1}\|$, with $k = 1 : n$, used in ILUPACK.

$v^{(0)} = (v_1^{(0)}, \cdots, v_n^{(0)})^T = (0, \cdots, 0)^T$
for $k = 1 : n$ **do**
  $x^+ \leftarrow 1 - v_k^{(k-1)}$
  $x^- \leftarrow -1 - v_k^{(k-1)}$
  $\mathcal{P} \leftarrow \{m : k+1 \le m \le n, l_{mk} \neq 0\}$
  $v^+ \leftarrow \|(v_j^{(k-1)} + l_{jk}x^+)_{j\in\mathcal{P}}\|_1$
  $v^- \leftarrow \|(v_j^{(k-1)} + l_{jk}x^-)_{j\in\mathcal{P}}\|_1$
  if $v^+ > v^-$ **then**
    $x_k \leftarrow x^+$
  else
    $x_k \leftarrow x^-$
  end if
  $v_j^{(k)} = v_j^{(k-1)}$ **with** $j = k+1 : n, l_{jk} = 0$
  for $j \in \mathcal{P}$ **do**
    $v_j^{(k)} \leftarrow v_j^{(k-1)} + l_{jk}x_k$
  end for
  $t_k^L \leftarrow \max(|x^+|, |x^-|)$
end for

PDEs. The methods were later extended (AMG) in different ways to handle other PDE problems, including nonlinear ones, as well as problems not modelled by PDEs. Because these methods exploit more information on the problem than standard preconditioned Krylov subspace methods, their performance can be vastly superior. On the other hand, they may require implementations at hand that are specific to the physical problem, in contrast with preconditioned Krylov subspace methods which attempt to be "general-purpose".

The Multigrid paradigm was incorporated to the Krylov methods to obtain competitive solvers for large dimension systems. In particular, a new family of preconditioners derived from *multigrid variants of the ILU factorization* improve the scalability of the solvers based on Krylov subspaces imitating the AMG methods. The multi-level ILU factorizations incorporate a mechanism to split the unknown set of the original system into two subsets: often the independent set is called the *fine set* and the complementary one is the *coarse set*. This mechanism is called *coarse-grid selection*. For the following discussion, we assume that the fine and the coarse sets have, respectively, k and $n - k$ unknowns. Then, we can obtain a permuted system

$$Ax = b \rightarrow P^T APP^T x = P^T b \rightarrow P^T AP\hat{x} = \hat{b}, \tag{3.37}$$

which can be partitioned as follows

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix}, \hat{x} = \begin{pmatrix} \hat{x}_B \\ \hat{x}_C \end{pmatrix}, \hat{b} = \begin{pmatrix} \hat{b}_B \\ \hat{b}_C \end{pmatrix}, \tag{3.38}$$

where $P \in \mathbb{R}^{n\times n}$ is the permutation matrix, $B \in \mathbb{R}^{k\times k}$ contains the coefficients corresponding to the unknowns of the *fine set*, F and E those of the unknowns of the *fine-coarse set* and *coarse-fine set*, respectively, and $C \in R^{(n-k)\times(n-k)}$ the coefficients of the unknowns of the *coarse set*. A block LU factorization will help establish the link with AMG-type methods,

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ EB^{-1} & I \end{pmatrix} \begin{pmatrix} B & 0 \\ 0 & S_C \end{pmatrix} \begin{pmatrix} I & B^{-1} \\ 0 & I \end{pmatrix}, \tag{3.39}$$

where S is the Schur complement of C, $S = C - EB^{-1}F$. We can calculate a LU factorization of the block B,

$$B = L_B U_B. \tag{3.40}$$

To compute this factorization we can use any of the methods previously described in this section. Afterwards, the factors L_B and U_B are used to "invert" B^{-1}, and to obtain the next approximation of the Schur complement

$$S_C \approx \tilde{S}_C = C - (EU_B^{-1})(L_B^{-1}F). \tag{3.41}$$

Taking into account the previous steps, we obtain the following block factorization

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ EU_B^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \tilde{S}_C \end{pmatrix} \begin{pmatrix} U_B & L_B^{-1}F \\ 0 & I \end{pmatrix}. \tag{3.42}$$

The block factorization (3.42) can be employed as preconditioner of the system (3.38) but, for that, it is necessary to calculate approximations of B and S_C which can economically be "inverted". In this sense, we can compute an ILU factorization, instead of a LU factorization, approximating B as,

$$B = \tilde{L}_B \tilde{U}_B + R_B. \tag{3.43}$$

The resultant incomplete block factorization is the following

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ E\tilde{U}_B^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \tilde{S}_C \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{L}_B^{-1}F \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & 0 \\ 0 & 0 \end{pmatrix}. \tag{3.44}$$

Moreover, an efficient implementation of the algorithm which computes the factorization (3.44) approximates the matrices $E\tilde{U}_B^{-1}$ and $\tilde{L}_B^{-1}F$, extending the ILU factorization (3.43) to the blocks E and F. For this purpose, we can apply an ILUC factorization. As a result of this process, we obtain the following incomplete block factorization

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & R_F \\ R_E & 0 \end{pmatrix}, \tag{3.45}$$

with

$$\tilde{L} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix}, \tilde{U} = \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix}, R = \begin{pmatrix} R_B & R_F \\ R_E & 0 \end{pmatrix}. \tag{3.46}$$

When some elements of $\tilde{L}$ and $\tilde{U}$ located in positions corresponding to non-diagonal blocks are rejected, then $\tilde{L}_E = E\tilde{U}_B^{-1} + R_E$ and $\tilde{U}_F = \tilde{L}_B^{-1}F + R_F$, where R_E and R_F contain "small" elements rejected during the process. Besides, the Schur complement is defined now as follows

$$S_C \approx \hat{S}_C = C - \tilde{L}_E \tilde{U}_F - R_C, \tag{3.47}$$

where R_C is the additional dropping. From a practical point of view, in order to save memory, an efficient implementation discards the factors $\tilde{L}_E$ and $\tilde{U}_F$ once each level of the preconditioner is calculated, keeping only two sparse rectangular matrices E and F, frequently much sparser than $\tilde{L}_E$ and $\tilde{U}_F$. Consequently, in this efficient implementation, the matrices R_E and R_F containing the discarded elements of these factors are also eliminated. This approach is adopted in the software packages for the computation of a multi-level ILU factorization, such as ILUPACK.

3.3 ILUPACK

ILUPACK [47, 52] is the abbreviation for **I**ncomplete **LU** factorization **PACK**age, and it is a software library for the iterative solution of large sparse linear systems, fully written in FORTRAN 77 and C, and available at http://ilupack.tu-bs.de. The package implements a multi-level incomplete factorization approach (multi-level ILU) based on a special permutation strategy called "inverse-based pivoting" combined with Krylov subspace iteration methods. Its main use consists of application problems such as linear systems arising from PDEs and it supports single and double precision arithmetic for real and complex numbers. Among the structured matrix classes that are supported by individual drivers are symmetric and/or Hermitian matrices that may or may not be positive definite as well as general square matrices. The main drivers can be called from C, C++, and FORTRAN, but an interface to MATLAB is also available.

ILUPACK is mainly based on incomplete factorization methods (ILUs) applied to the system matrix in conjunction with Krylov subspace methods. The ILUPACK hallmark is the so-called inverse-based approach combined with the Crout variant of the ILDU factorization. It was initially developed to connect the ILUs and their approximate inverse factors [51]. These relations are important since, in order to solve linear systems, the inverse triangular factors resulting from the factorization are applied rather than the original incomplete factors themselves. The information extracted from the inverse factors will in turn help to improve the robustness for the incomplete factorization process. While this idea has been successfully used to improve robustness, its downside was initially that the norm of the inverse factors could become large such that small entries could hardly be dropped during Gaussian elimination. To overcome this shortcoming, a multi-level strategy was developed to limit the growth of the inverse factors. This led to the inverse-based approach, and hence the incomplete factorization process, that was eventually implemented in ILUPACK. This solution benefits from the information of bounded inverse factors while being efficient at the same time [51].

In this section, we first define the computation of the preconditioner in ILUPACK, which is based on the previous concepts of this chapter, and then, we explain how the computed preconditioner is applied in this package.

3.3.1 Computation of the preconditioner

Preconditioning in ILUPACK relies on the so-called *inverse-based approach*, which improves the robustness of classical ILDU factorizations by bounding the growth of the entries in the inverses of the triangular factors. To justify this, consider the ILDU factorization

$$A = \tilde{L}\tilde{D}\tilde{U} + R \ , \tag{3.48}$$

where $\tilde{L}$, $\tilde{U}^T$ are unit lower triangular matrices, $\tilde{D}$ is diagonal, and R is the error matrix which collects those entries that were dropped during the factorization. Applying the preconditioner $M = \tilde{L}\tilde{D}\tilde{U}$ on the original matrix, we obtain the preconditioned matrix

$$\tilde{L}^{-1}A\tilde{U}^{-1} = \tilde{D} + \tilde{L}^{-1}R\tilde{U}^{-1}. \tag{3.49}$$

Although dropping typically results in some "relatively small" error matrix R, both $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ may exhibit very large norms, so that the application of the preconditioning can significantly amplify the size of the entries in R. This may directly impact the convergence rate of the preconditioned iterative solver.

The inverse-based preconditioning approach relies on approximate factorizations with "bounded" inverse triangular factors; i.e., factorizations with $\|L^{-1}\| \leq \kappa$ and $\|U^{-1}\| \leq \kappa$, for some prescribed

small threshold $\kappa > 1$. In practical applications, an ILDU factorization of the system at hand does not typically satisfy this requirement, so that pivoting is necessary to bound the inverse triangular factors during the computation. Pivoting is accommodated in a multi-level framework in order to construct a hierarchy of partial inverse-based approximations, as sketched in the following multi-level algorithm:

1. **Preprocessing step**. Matrix A is scaled by diagonal matrices D_l and D_r and reordered by permutation matrices P_l and P_r:

$$A \to D_l A D_r \to P_l^T D_l A D_r P_r = \hat{A}.$$

   These operations can be considered as a preprocessing prior to the numerical factorization. They typically include scaling strategies to equilibrate the system, scaling and permuting based on maximum weight matchings, and finally, fill-reducing orderings such as nested dissection, (approximate) minimum degree, etc.

2. **Factorization step**. At each step of the Crout variant of the ILU factorization, the method is interlaced with a pivoting strategy which yields a nonexpensive estimation of the norm of a new row/column of the inverse factors. If the estimation exceeds the threshold k, the current pivot is rejected and the corresponding row/column are moved to the bottom/right-end of the matrix. Otherwise, the pivot is accepted and dropping is applied to the current row/column before they are incorporated to the factors (see Figure 3.6). Collecting the permutations due to the inverse-based pivoting on P, we obtain the following partial ILDU factorization of a permuted matrix:

$$P^T \hat{A} P = \begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{S}_C \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} + \begin{bmatrix} R_B & R_F \\ R_E & 0 \end{bmatrix}. \tag{3.50}$$

   In practice, for an efficient implementation, as stated previously, we can discard $\tilde{L}_E$ and $\tilde{U}_F$, and therefore, the matrices R_E and R_F are zero. The method applies additional dropping to the approximate Schur complements $\tilde{S}_C$, so that we actually compute

$$\hat{S}_C = \tilde{S}_C + R_C = C - (\tilde{L}_E \tilde{D}_B \tilde{U}_F) + R_C. \tag{3.51}$$

3. **Restarting step**. Steps 1 and 2 are repeatedly applied to $A = \hat{S}_C$ until $\hat{S}_C$ is void or "sufficiently dense" to be efficiently factorized by a level 3 BLAS-based direct factorization kernel.

When the multi-level method is applied over multiple levels, a cascade of factors are usually obtained, as shown in Figure 3.7. Moreover, the computed multi-level factorization is adapted to the structure of the underlying system. Hence, in this case, the multi-level preconditioner can be expressed recursively, at a given level l, as

$$M_l \approx \tilde{D}^{-1} \tilde{P} P, \quad \begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} P^T \tilde{P}^T \tilde{D}^{-1}, \tag{3.52}$$

where $\tilde{L}_B$, $\tilde{L}_F$, $\tilde{D}_B$, $\tilde{U}_F$ and $\tilde{U}_B$ are blocks of the factors of the multi-level $\tilde{L}\tilde{D}\tilde{U}$ preconditioner (with $\tilde{L}_B$ unit lower triangular, $\tilde{U}_B$ unit upper triangular and $\tilde{D}_B$ diagonal); and M_{l+1} stands for the preconditioner computed at level $l+1$.

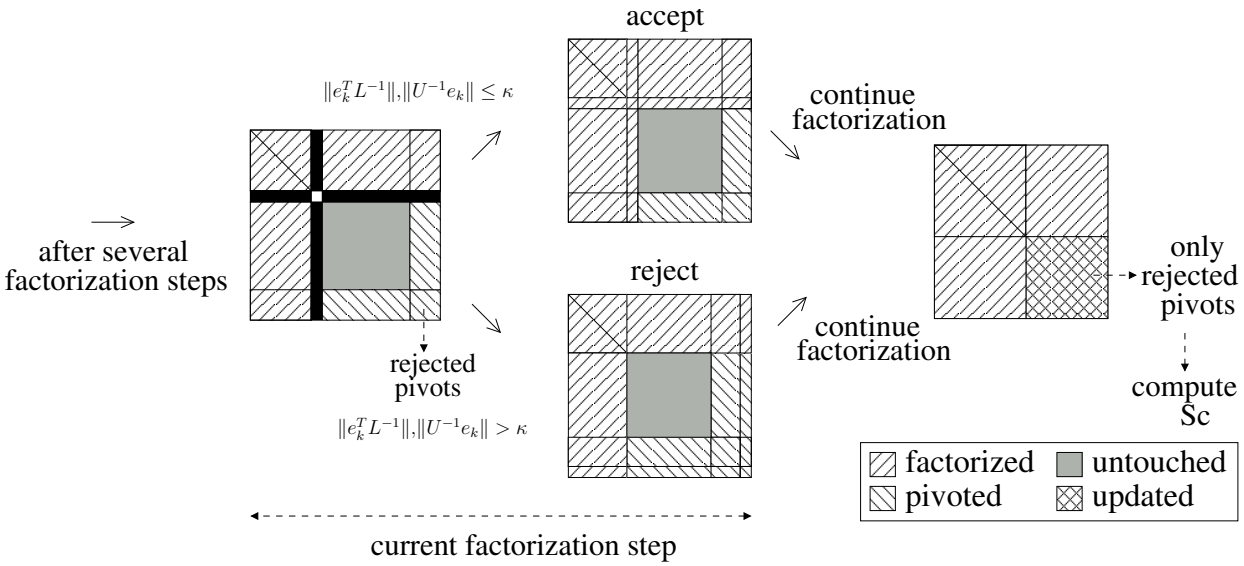


Figure 3.6: A step of the Crout variant of the preconditioner computation in ILUPACK.

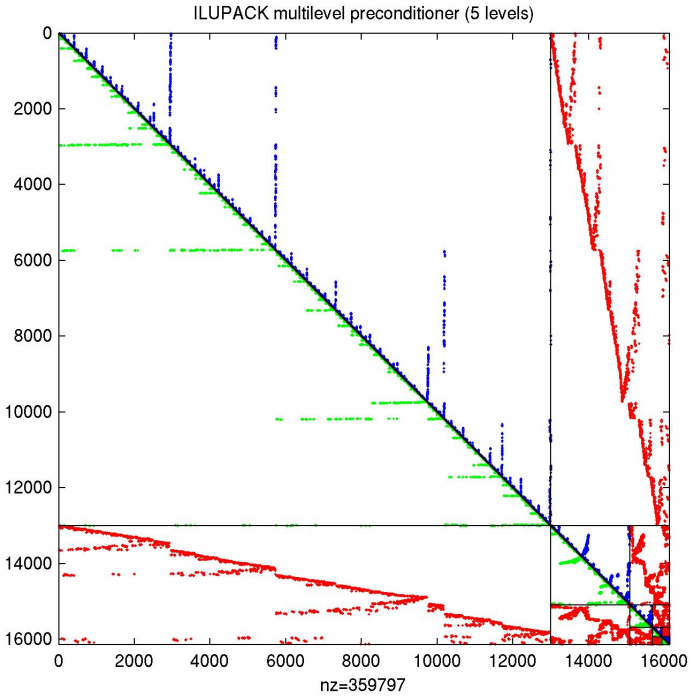


Figure 3.7: ILUPACK multi-level factorization of five-point matrix arising from Laplace PDE discretization.

3.3.2 Application of the preconditioner

Figure 3.2 offers an algorithmic description of the PCG method. The computation of the preconditioner M, explained in the previous section, is the first step of the solver (O0). The subsequent iteration involves a sparse matrix-vector product (SpMV) (O1), the application of the preconditioner (O5), and several vector operations (DOT products, AXPY-like updates, 2-norm; in O2–O4 and O6–O9). In the remainder of this section, we focus on the application of the preconditioner.

For simplicity, let us next remove the subscripts in the corresponding operation (O5) of Figure 3.2: $z := M^{-1}r$. Applying the preconditioner in level l (i.e., computing $z := M_l^{-1}r$), then requires solving the system of linear equations:

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} P^T \tilde{P}^T \tilde{D}^{-1} z = P^T \tilde{P}^T \tilde{D}^{-1} r. \tag{3.53}$$

Breaking down (3.53), we first recognize two transformations to the residual vector r. First, $r' := Dr$ applies the diagonal scaling to this vector; then the ordering step is applied to compute $\hat{r} := P^T \tilde{P}^T r'$. Once these transformations are completed, the system

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} w = \hat{r} \tag{3.54}$$

is solved for $w(= P^T \tilde{P}^T \tilde{D}^{-1} z)$ in three steps. Initially, by

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} y = \hat{r} \tag{3.55}$$

for y; then solving recursively

$$\begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} x = y \tag{3.56}$$

for x; and finally solving for w

$$\begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} w = x. \tag{3.57}$$

In turn, the expressions in (3.55) and (3.57) also need to be solved in two steps. Assuming vectors y and $\hat{r}$ are split conformally with the blocks of the factors, for (3.55) we have

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} y_B \\ y_C \end{bmatrix} = \begin{bmatrix} \hat{r}_B \\ \hat{r}_C \end{bmatrix}. \tag{3.58}$$

This system is then tackled by initially solving the unit lower triangular system

$$\tilde{L}_B y_B = \hat{r}_B \tag{3.59}$$

for y_B, and then computing

$$y_C := \hat{r}_C - \tilde{L}_E y_B. \tag{3.60}$$

Splitting the vectors like before, Equation (3.56) involves the diagonal-matrix multiplication

$$x_B := D_B^{-1} y_B, \tag{3.61}$$

and the recursive step

$$x_C := M_{l+1}^{-1} y_C. \tag{3.62}$$

In the base step of the recursion, the size of M_{l+1} is equal to zero and then only x_B has to be computed. Finally, after an analogous partitioning, equation (3.57) can be reformulated as

$$w_C := x_C \tag{3.63}$$

and

$$\tilde{U}_B w_B = x_B - \tilde{U}_F w_C \ , \tag{3.64}$$

such that z is simply obtained form $z := D(\hat{P}^T(P^T w))$.

Remember that, with the purpose of saving memory, ILUPACK discards the factors $\tilde{L}_E$ and $\tilde{U}_F$ once each level of the preconditioner is calculated, keeping only two sparse rectangular matrices E and F, frequently much sparser than $\tilde{L}_E$ and $\tilde{U}_F$, such that $\tilde{L}_E = E\tilde{U}_B^{-1}\tilde{D}_B^{-1}$, and $\tilde{U}_F = \tilde{D}_B^{-1}\tilde{L}_B^{-1}F$. This improvement changes (3.60) to

$$y_C := \hat{r}_C - E\tilde{U}_B^{-1}\tilde{D}_B^{-1}y_B, \tag{3.65}$$

which, in combination with (3.59), yields

$$y_C := \hat{r}_C - E\tilde{U}_B^{-1}\tilde{D}_B^{-1}\tilde{L}_B^{-1}\hat{r}_B. \tag{3.66}$$

Furthermore, (3.64) also changes to

$$\tilde{U}_B w_B = \tilde{D}_B^{-1}y_B - \tilde{D}_B^{-1}\tilde{L}_B^{-1}Fw_C. \tag{3.67}$$

Now, (3.66) can be tackled by first solving

$$\tilde{L}_B\tilde{D}_B\tilde{U}_B s_B = \hat{r}_B \tag{3.68}$$

for s_B, and then obtaining

$$y_C := \hat{r}_C - Es_B, \tag{3.69}$$

while (3.67) can be tackled by solving

$$\tilde{L}_B\tilde{D}_B\tilde{U}_B\hat{s}_B = Fw_C \tag{3.70}$$

for $\hat{s}_B$, and then performing

$$w_B := s_B - \hat{s}_B. \tag{3.71}$$

To summarize, at each level the procedure implemented by ILUPACK performs two sparse matrix-vector multiplications and solves two linear systems of the form $\tilde{L}\tilde{D}\tilde{U}x = b$. In addition, three other types of operations are distinguished: diagonal scaling, vector permutation, and vector updates of the form $x := a - b$.

CHAPTER 4

Exploiting Task-Parallelism in ILUPACK

The increment of thread-level hardware parallelism in multicore architectures, leading to processors that nowadays support between dozens and hundreds of threads (e.g., 64 threads in the IBM PowerPC A2 processor and 240 in the Intel Xeon Phi processor), has guided the development of several *data-flow* programming models in the past few years with the purpose of decoupling the description of an algorithm from the "mechanics" of its parallel execution, hence reducing the coding effort and improving source code portability.

Data-flow programming models assume that data dependencies characterize a number of "correct" concurrent schedules by defining a partial execution order on the operations (tasks) that compose the algorithm. In general, modern data-flow programming models are assisted by a specialized runtime which analyzes data dependencies, and orchestrates the parallel execution in order to optimize performance. Emblematic examples of these type of programming models include, among others, DAGuE/ParSEC [54], Harmony [67], Mentat [89], Qilin [130], StarPU [177], Uintah [44], XKaapi [84], and the target of our work, OmpSs [74, 2]. For the particular domain of dense linear algebra, the application of these models and/or similar approaches has resulted in a collection of high performance libraries (DPLASMA, `libflame`, MAGMA, PLASMA, etc.). However, the application of data-flow programming paradigms to the parallel solution of sparse systems of linear equations is still unripe, mostly due to sparse linear algebra operations being much more challenging than their dense counterparts. In particular, data-flow runtime-assisted linear system solvers using supernodal direct and ILU-type iterative methods have been proposed only recently, for example in [106, 119, 122] and [22], respectively.

In this chapter we analyze the PCG method in ILUPACK with the goal of exposing task parallelism. The parallelization scheme can be leveraged to implement different versions of the solver using OmpSs, MPI and a combination of both, achieving significant performance gains. Moreover, this scheme can be also applied to easily parallelize other ILU-type iterative solvers.

The chapter is structured as follows. Section 4.1 describes how to extract the task concurrency in the PCG method. Section 4.2 reviews the parallel programming models relevant for this dissertation. Section 4.3 introduces the target platforms and the test cases employed in the evaluation experiments of this chapter. Next, Sections 4.4 and 4.5 present two task-parallel implementations of ILUPACK solver for multicore processors with OmpSs and for clusters of multicore processors using MPI+OmpSs. Section 4.6 provides optimized implementations with OmpSs and MPI for

NUMA platforms and manycore hardware co-processors based on the Intel Xeon Phi. Finally, some concluding remarks are included in Section 4.7.

4.1 Task-Level Concurrency in the PCG Method

We first present the strategy which is followed to extract task concurrency in the PCG method in ILUPACK. In Subsection 4.1.1 we define the main concepts to partition a matrix into an adjacency graph. Then, in Subsections 4.1.2 and 4.1.3 we explain how to use the adjacency graph to extract task parallelism in the preconditioner computation, its application, and the remaining operations in the PCG method.

4.1.1 Nested dissection

For the computation of approximate factorizations, concurrency can be exposed by means of graph-based algorithms, such as graph coloring or graph partitioning techniques. Among these classes of algorithms, *nested dissection* orderings enhance parallelism in the approximate factorization of A by partitioning its associated adjacency graph $G(A)$ into a hierarchy of vertex separators and independent subgraphs [47]. For example, in Figure 4.1, $G(A)$ is partitioned after two levels of recursion into four independent subraphs, $G_{(3,1)}$, $G_{(3,2)}$, $G_{(3,3)}$, and $G_{(3,4)}$, first by separator $S_{(1,1)}$ and then by separators $S_{(2,1)}$ and $S_{(2,2)}$ This hierarchy is constructed so that the size of the vertex separators is minimized while simultaneously balancing the size of the independent subgraphs. Therefore, relabeling the nodes of $G(A)$ according to the levels in the hierarchy leads to a reordered matrix, $A \leftarrow P^T AP$, with a structure amenable to efficient parallelization. In particular, the leading diagonal blocks of $P^T AP$ associated with the independent subgraphs can be first processed independently; after that, $S_{(2,1)}$ and $S_{(2,2)}$ can be computed in parallel, and finally, the separator $S_{(1,1)}$ is processed. This type of concurrency can be expressed as a binary task dependency tree (see Figure 4.1), where the nodes represent concurrent tasks and the edges dependencies among them.

State-of-the-art reordering software packages, such as Metis [135] or Scotch [174], provide fast and efficient multi-level variants of nested dissection orderings [116]. There also exist parallel versions of these packages (ParMetis [117], mt-metis [123] and PT-Scotch [60]) that exploit several types of concurrency during the computation of the permutation. Specifically, in ILUPACK we use parallel versions of the Metis package, depending on whether the implementation is for shared-memory (mt-Metis) or distributed-memory (ParMetis).

ParMetis

ParMetis [117] is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs and meshes, as well as to obtain fill-reducing orderings of sparse matrices. ParMetis extends the functionality provided by Metis and includes routines that are especially suited for parallel Adaptive Mesh Refinement (AMR) computations and large-scale numerical simulations. In particular, ParMetis provides the following functionality [118]:

- Partition unstructured graphs and meshes.
- Repartition graphs that correspond to adaptively refined meshes.
- Partition graphs for multi-phase and multi-physics simulations.
- Improve the quality of existing partitionings.

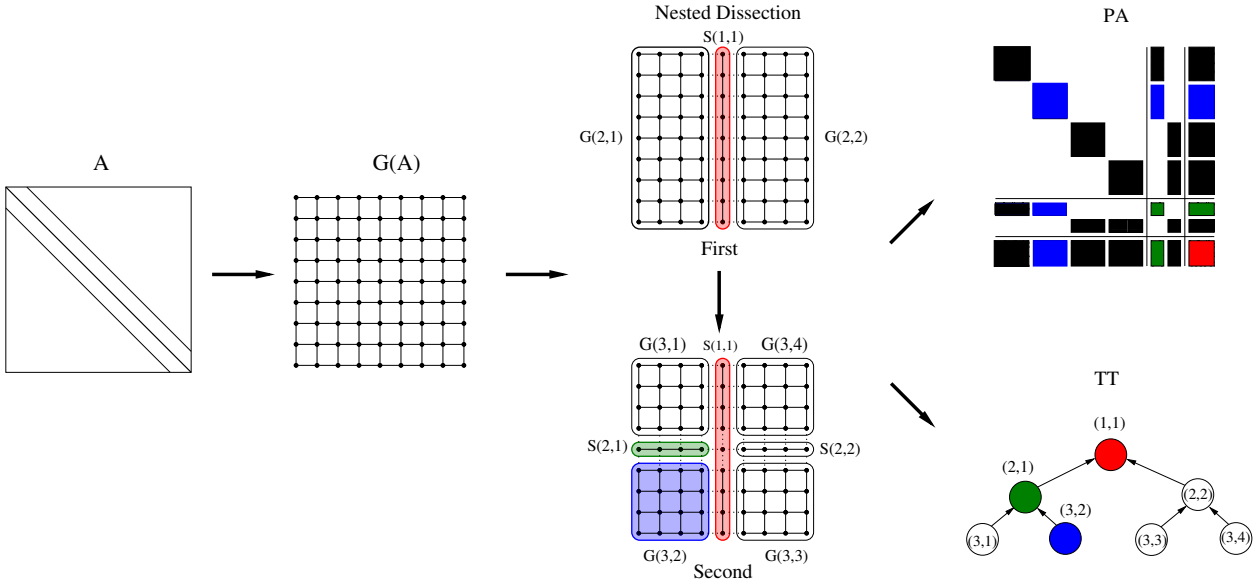


Figure 4.1: Nested dissection reordering. In this example $G(A)$ is partitioned into four independent subgraphs. Colors are used to illustrate the correspondence between the blocks of the permutation to be factorized, and the tasks in charge of their factorization (nodes of the tree).

- Compute fill-reducing orderings for sparse direct factorization.
- Construct the dual graphs of meshes.

Concretely, in ILUPACK we use the routine `ParMETIS_V3_NodeND` to compute a fill-reducing ordering of a sparse matrix using multi-level Nested Disection (ND) [118]. This routine returns an array with the result of the ordering (permutation), together with the number of nodes for each sub-domain and each separator. With this information, the application builds their own structures representing the adjacency graph corresponding with the sparse matrix.

mt-Metis

Previous implementations of ILUPACK for shared-memory leveraged a modified version of serial Metis. This tuned version, parallelized based on Metis, attained a speed-up factor of 2× on 16 cores. More recently, in [123], the Metis' developers presented an efficient version of the library for multicore architectures.

In [123], the authors presented shared-memory parallel algorithms for generating vertex separators, using those vertex separators to generate a fill-reducing ordering via ND in parallel. They also introduced task scheduling to maximize cache efficiency for the ND problem, achieving a speed-up of up to 10× on 16 cores, while producing orderings with only 1.0% more fill-in and requiring only 0.7% more operations than the original ND routines included in Metis. The proposed implementation, called mt-Metis, is 1.5× faster, producing 3.7% less fill-in, and requiring 14.0% fewer operations than ParMetis [116].

Although the mt-Metis implementation offers an efficient parallelization of the Metis library, it had some drawbacks for our purpose. First, mt-Metis only generates the permutation associated to the partition, but not the tree structure, which is required by ILUPACK. Hence, we modified

the implementation to return a vector with the partition of the left and right nodes, as well as the size of the separator, for each level of the tree. From this vector, we created the tree structure for ILUPACK. Second, mt-Metis partitions the graph with a number of leaves equal to the number of cores executing the code. We adjusted the original library to obtain a tree with more leaves than cores. Finally, the recursion in mt-Metis is stopped when the graph is smaller than a certain size, and the Multiple Minimum Degree Ordering (MMDO) [127] is applied to the remaining graph. In order to accelerate the execution, we introduced two additional options to finish the recursion. Next, we resume the three options to finish the ND computation in mt-metis execution:

- Execute recursively ND and, when the graph is smaller than a certain size, apply MMDO to the remaining graph. This is the option employed by the original library.
- Execute recursively ND until the tree has the desired number of leaves, and then apply MMDO to the remaining graph. The difference with respect to the previous option is that here we do not have to wait until the graph is smaller than a specified size. We can stop when we have a graph with the leaves indicated in the execution.
- Execute recursively ND until the tree has the desired number of leaves, and then finish the execution avoiding the ordering step.

Note that the main objective of the new alternatives is to reduce the execution time of the algorithm.

4.1.2 Computation of the preconditioner

In order to design a task-parallel version of ILUPACK, we need to decompose the computation of the preconditioner into tasks, identifying the dependencies among them, and mapping the tasks to the execution nodes. For that purpose, the task-parallel version exploits the connection between sparse matrices and adjacency graphs [166], extracting parallelism via ND, as explained in Subsection 4.1.1. Consider for example a graph-based reordering, defined by a permutation $\bar{P} \in \mathbb{R}^{n \times n}$, such that

$$\bar{P}^T A \bar{P} = \left[\begin{array}{cc|c} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right]. \tag{4.1}$$

Computing partial ILU factorizations of the two leading blocks, A_{00} and A_{11}, yields the following partial approximation of $\bar{P}^T A \bar{P}$

$$\left[\begin{array}{cc|c} L_{00} & 0 & 0 \\ 0 & L_{11} & 0 \\ \hline L_{20} & L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{cc|c} D_{00} & 0 & 0 \\ 0 & D_{11} & 0 \\ \hline 0 & 0 & S_{22} \end{array}\right] \left[\begin{array}{cc|c} U_{00} & 0 & U_{20} \\ 0 & U_{11} & U_{21} \\ \hline 0 & 0 & \mathrm{I} \end{array}\right] + E_{01},$$

where

$$S_{22} = A_{22} - (L_{20} D_{00} U_{20}) - (L_{21} D_{11} U_{21}) + E_2 \tag{4.2}$$

is the approximate Schur complement. By recursively proceeding in the same manner with S_{22}, the ILU factorization of $\bar{P}^T A \bar{P}$ is eventually completed. At this point we note that for SPD matrices, instead of applying an ILU factorization, we apply an IC factorization, so that $U = L^T$.

The block structure in (4.1) exposes a coarse-grain concurrency during these computations. Concretely, the permuted matrix there can be decoupled into two submatrices, so that the ILU

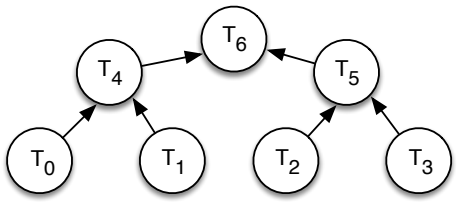


Figure 4.2: Dependency tree of the diagonal blocks. Task T_j is associated with block A_{jj}.

factorizations of the leading block of both submatrices can be concurrently obtained:

$$A_{22} = A_{22}^0 + A_{22}^1 \, , \quad \begin{cases} \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] = \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{20} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & S_{22}^0 \end{array}\right] \left[\begin{array}{c|c} U_{00} & U_{20} \\ \hline 0 & \mathrm{I} \end{array}\right] + E_0, \\ \\ \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] = \left[\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & S_{22}^1 \end{array}\right] \left[\begin{array}{c|c} U_{11} & U_{21} \\ \hline 0 & \mathrm{I} \end{array}\right] + E_1. \end{cases} \tag{4.3}$$

Then, we can also compute in parallel the Schur complements corresponding to both partial approximations:

$$S_{22}^0 = A_{22}^0 - (L_{20} D_{00} U_{20}) + E_2^0; \quad S_{22}^1 = A_{22}^1 - (L_{21} D_{11} U_{21}) + E_2^1 \, .$$

However, the construction of (4.2) involves a synchronization step before the addition of these two blocks can be computed

$$S_{22} \approx S_{22}^0 + S_{22}^1 \, , \quad E_2 \approx E_2^0 + E_2^1 \, . \tag{4.4}$$

To unveil increasing amounts of task parallelism, we can identify a larger number of independent diagonal blocks, by applying permutations analogous to $\bar{P}$ on the two leading blocks. For example, by reordering and renaming the blocks properly, a block structure similar to (4.1) is obtained:

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{00} & 0 & \hat{A}_{02} & 0 & 0 & 0 & * \\ 0 & \hat{A}_{11} & \hat{A}_{12} & 0 & 0 & 0 & * \\ \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \hat{A}_{00} & 0 & \hat{A}_{02} & * \\ 0 & 0 & 0 & 0 & \hat{A}_{11} & \hat{A}_{12} & * \\ 0 & 0 & 0 & \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} & * \\ \hline * & * & * & * & * & * & * \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{00} & 0 & 0 & 0 & A_{04} & 0 & A_{06} \\ 0 & A_{11} & 0 & 0 & A_{14} & 0 & A_{16} \\ 0 & 0 & A_{22} & 0 & 0 & A_{25} & A_{26} \\ 0 & 0 & 0 & A_{33} & 0 & A_{35} & A_{36} \\ \hline A_{40} & A_{41} & 0 & 0 & A_{44} & 0 & A_{46} \\ 0 & 0 & A_{52} & A_{53} & 0 & A_{55} & A_{56} \\ \hline A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} \end{array}\right] . \tag{4.5}$$

Figure 4.2 illustrates the dependency tree for the factorization of the diagonal blocks in (4.5). The nodes that lie in the same level of the tree can be factorized in parallel and the edges of the preconditioner Directed Acyclic Graph (DAG) define the dependencies between the diagonal blocks (tasks), i.e., the order in which these blocks of the matrix have to be processed. We identify three classes of nodes in the figure:

- LEAF NODES. These nodes are responsible for the factorization of the four leading diagonal blocks in parallel.
- INTERMEDIATE NODES. They factorize in parallel the next two intermediate diagonal blocks, A_{44} and A_{55}. These blocks cannot be factorized unless the leading diagonal blocks corresponding to its children have already been factorized, i.e, A_{00} - A_{11} and A_{22} - A_{33} respectively.

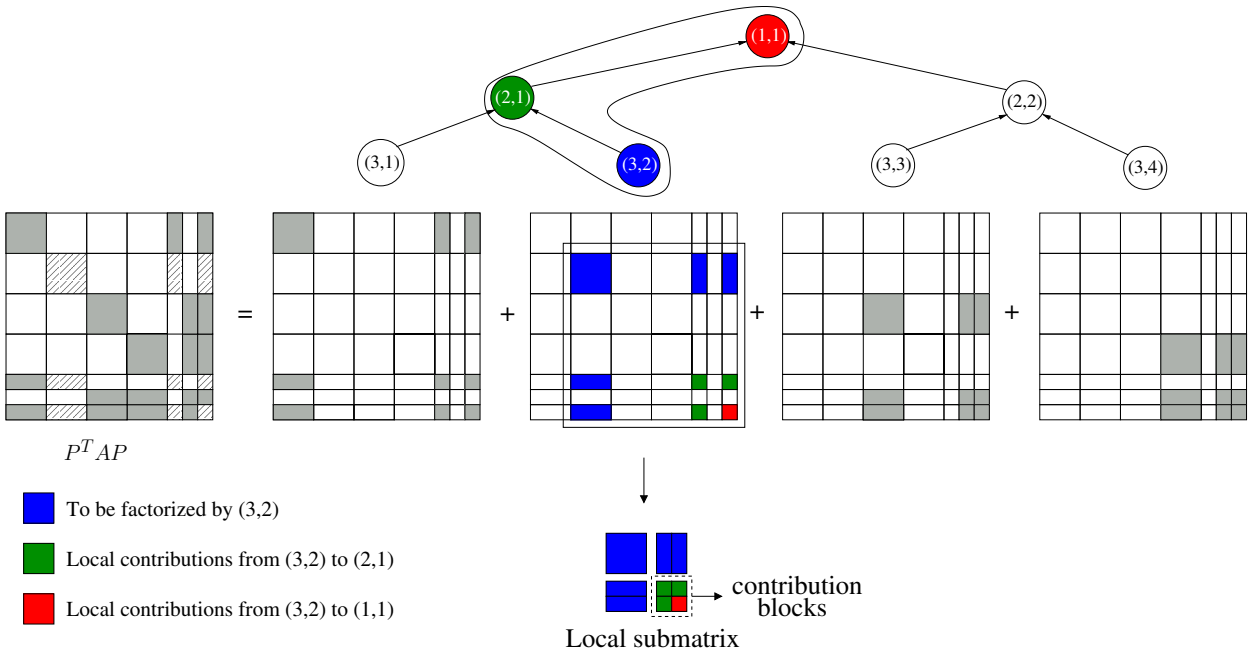


Figure 4.3: Matrix decomposition and local submatrix associated to a single node of the task tree.

- ROOT NODE. This node sequentially factorizes the last diagonal block, A_{66}. This approximation can be only computed when all the preceding diagonal blocks have been processed.

The parallel computation of the preconditioner starts by disassembling A, with one submatrix for each leaf node, as shown in Figure 4.3. For instance, the submatrices in (4.5) are decomposed as

$$\left[\begin{array}{c|cc} A_{00} & A_{04} & A_{06} \\ \hline A_{40} & A^0_{44} & A^0_{46} \\ A_{60} & A^0_{64} & A^0_{66} \end{array}\right], \left[\begin{array}{c|cc} A_{11} & A_{14} & A_{16} \\ \hline A_{41} & A^1_{44} & A^1_{46} \\ A_{61} & A^1_{64} & A^1_{66} \end{array}\right], \left[\begin{array}{c|cc} A_{22} & A_{25} & A_{26} \\ \hline A_{52} & A^2_{55} & A^2_{56} \\ A_{62} & A^2_{65} & A^2_{66} \end{array}\right], \left[\begin{array}{c|cc} A_{33} & A_{35} & A_{36} \\ \hline A_{53} & A^3_{55} & A^3_{56} \\ A_{63} & A^3_{65} & A^3_{66} \end{array}\right], \tag{4.6}$$

where

$$A_{44} = A^0_{44} + A^1_{44}\ ,\ A_{55} = A^2_{55} + A^3_{55}\ ,\ A_{66} = A^0_{66} + A^1_{66} + A^2_{66} + A^3_{66}\ . \tag{4.7}$$

Thus, the partial ILU factorization of these submatrices can be computed concurrently. For example, computing the ILU of A_{00}, we obtain the following partial approximation:

$$\left[\begin{array}{c|cc} \tilde{L}_{00} & 0 & 0 \\ \hline \tilde{L}_{40} & I & 0 \\ \tilde{L}_{60} & 0 & I \end{array}\right] \left[\begin{array}{c|cc} \tilde{D}_{00} & 0 & 0 \\ \hline 0 & \tilde{S}^0_{44} & \tilde{S}^0_{46} \\ 0 & \tilde{S}^0_{64} & \tilde{S}^0_{66} \end{array}\right] \left[\begin{array}{c|cc} \tilde{U}_{00} & \tilde{U}_{04} & \tilde{U}_{06} \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right] + E_{00}\ . \tag{4.8}$$

When the partial factorizations of all nodes are completed, the processes in charge of these tasks send the local Schur complement to the corresponding intermediate node, which then accumulates them to continue the process,

$$\left[\begin{array}{c|c} \hat{S}^0_{44} & \hat{S}^0_{46} \\ \hline \hat{S}^0_{64} & \hat{S}^0_{66} \end{array}\right] + \left[\begin{array}{c|c} \hat{S}^1_{44} & \hat{S}^1_{46} \\ \hline \hat{S}^1_{64} & \hat{S}^1_{66} \end{array}\right] = \left[\begin{array}{c|c} \hat{S}_{44} & \hat{S}_{46} \\ \hline \hat{S}_{64} & \hat{S}^{01}_{66} \end{array}\right],$$

$$\left[\begin{array}{c|c} \hat{S}^2_{55} & \hat{S}^2_{56} \\ \hline \hat{S}^2_{65} & \hat{S}^2_{66} \end{array}\right] + \left[\begin{array}{c|c} \hat{S}^3_{55} & \hat{S}^3_{56} \\ \hline \hat{S}^3_{65} & \hat{S}^3_{66} \end{array}\right] = \left[\begin{array}{c|c} \hat{S}_{55} & \hat{S}_{56} \\ \hline \hat{S}_{65} & \hat{S}^{23}_{66} \end{array}\right].$$

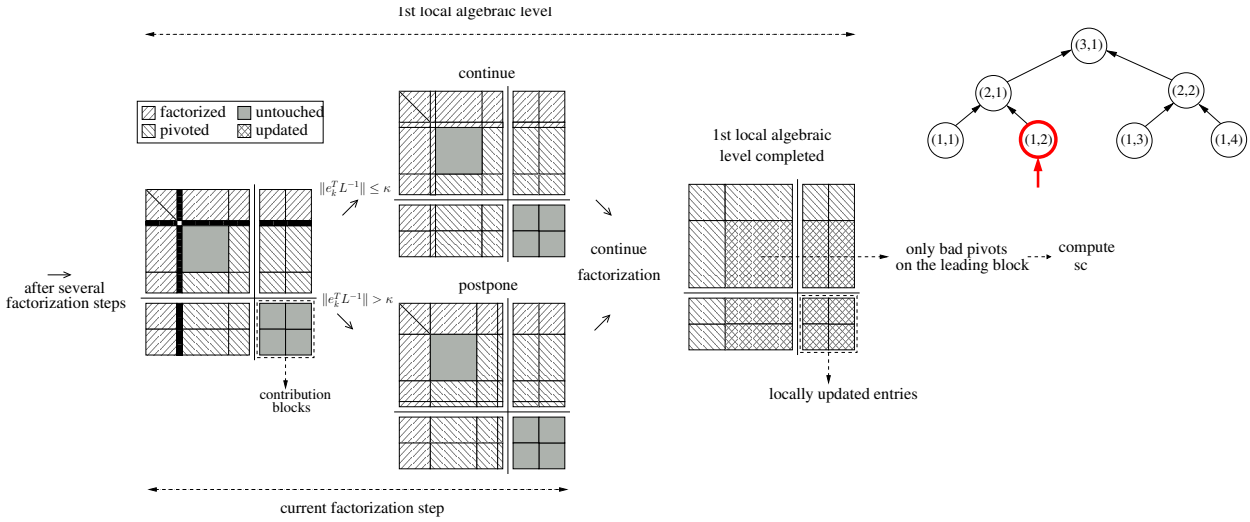


Figure 4.4: A step of the Crout variant of the parallel preconditioner computations.

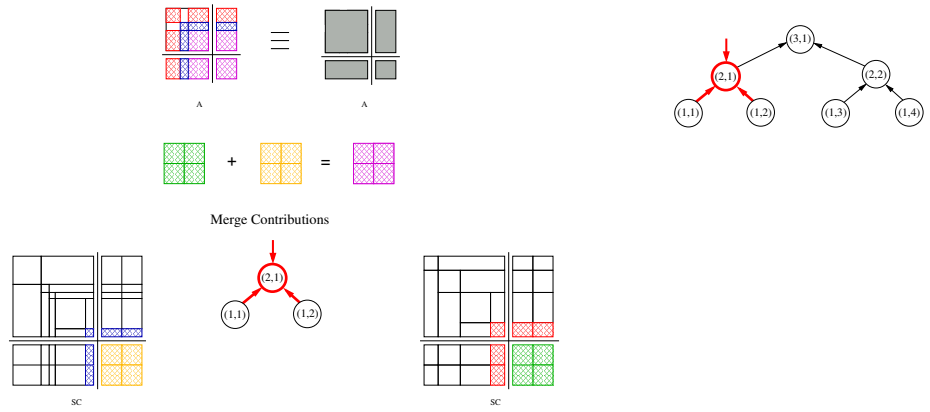


Figure 4.5: Task (2, 1) computes its own matrix from the Schur complements resulting from the local computations of its children nodes ((1,1) and (1,2)).

The matrix resulting by assembling these two submatrices presents the same structure as that defined in (4.1):

$$\left[\begin{array}{c|c} \hat{S}_{44} & \hat{S}_{46} \\ \hline \hat{S}^2_{64} & \hat{S}^{01}_{66} \end{array}\right] \oplus \left[\begin{array}{c|c} \hat{S}_{55} & \hat{S}_{56} \\ \hline \hat{S}^2_{65} & \hat{S}^{23}_{66} \end{array}\right] = \left[\begin{array}{cc|c} S_{44} & 0 & S_{46} \\ 0 & S_{55} & S_{56} \\ \hline S_{64} & S_{65} & S_{66} \end{array}\right], \; S_{66} = S^{01}_{66} + S^{23}_{66} \; . \tag{4.9}$$

This process continues traversing the dependency tree, until the root task factorizes its local submatrix.

The main change of this parallel version with respect to the sequential case is that the computation is restricted to the leading block, and therefore the rejected pivots are moved to the bottom-right corner of the leading block; see Figure 4.4, which illustrates the factorization step of a leaf node in the first level of the DAG. Figure 4.5 shows the multi-level factorization computed in each internal node, and how the merge step is performed to obtain the matrix related to the internal node (2,1).

Another change is that preconditioning introduces additional levels in the recursive definition of ILU preconditioners. Thus, the parallel ILU preconditioner contains numerical and structural levels in its recursive definition. Therefore, different preconditioner DAGs involve distinct recursion steps yielding distinct preconditioners, which nonetheless exhibit close numerical properties to that obtained with the sequential ILUPACK [22].

4.1.3 The iterative PCG solve

The data dependencies in the iterative PCG solve (see the `while` loop in Figure 3.2) define a partial order for the operations that compose the loop's body. Specifically, the order $v_j \to \alpha_j \to r_{j+1} \to z_{j+1} \to \sigma_{j+1} \to \beta_j \to p_{j+1}$ must be preserved, but x_{j+1} and τ_{j+1} can be computed any time once α_j and r_{j+1} are respectively available: $\alpha_j \to x_{j+1}$, $r_{j+1} \to \tau_{j+1}$. (For simplicity, we do not consider here the dependencies between operations from different iterations.) However, further concurrency can be exposed by dividing some of these operations into subtasks.

Our parallelization of the PCG splits the data structures associated with these operations (as in (4.7)) to distribute the data into the different cores/processors. In order to reduce the synchronization steps, we consider two schemes to distribute the data for the structures involved in the operations of the PCG:

INCONSISTENT DISTRIBUTION: The values of some data are distributed as,

$$B^i = B_0^i + B_1^i + B_2^i + B_3^i \ ,$$

where the entries of the contribution blocks store partial contributions to the "global" entries of the data structure. The final value is obtained adding the partial entries of the contribution blocks.

CONSISTENT DISTRIBUTION: Redundant copies of the elements are stored as

$$B^c = B_0^c = B_1^c = B_2^c = B_3^c,$$

where the entries of the contribution blocks store redundant copies of the "global" entries of the data structure; i.e, the same value is repeated in all the blocks.

In order to reduce the communication steps, the operations in the parallel implementation of PCG need to adopt an specific distribution. Next, we define the best distribution for each PCG operation:

FULLY INDEPENDENT OPERATIONS: The vectors involved in PCG are partitioned conformally to matrix A, see e.g. (4.5), so that many operations only access the data associated with the leaves of the preconditioner DAG. The distribution that is chosen to store the operands determines the proper working of the operation:

SPMV. The distributed matrix-vector product is $B^i x^c = y^i$, where the matrix is *inconsistent*, the vector is *consistent*, and the result is *inconsistent*.

AXPY. The consistency of the vectors involved in the AXPY operations has to be the same. For example, $x^c + y^c = b^c$ or $x^i + y^i = b^i$.

DOT. This product involves different consistencies in the vectors, with one *inconsistent* and the other *consistent* ($(x^i)^T y^c = \alpha^i$ or $(x^c)^T y^i = \alpha^i$).

2-NORM. This operation computes a DOT and a square root. Therefore, it requires different type of vectors, like the DOT product.

Note that the DOT operation obtains an inconsistent scalar, and therefore a global synchronization/communication is required to obtain the corresponding consistent value.

PRECONDITIONER APPLICATION: As the definition of the recursion is maintained, the operations to apply the preconditioner, explained in the previous chapter, remain valid. However, to complete the recursion step in the task-parallel case, the preconditioner DAG has to be crossed twice per solve $z_{j+1} := M^{-1}r_{j+1}$ at each iteration of the PCG: once from bottom to top, and a second time from top to bottom. Each transition requires a synchronization/ communication step, adding vectors when the DAG is traversed upward, and copying vectors in the reverse case. Moreover, the concurrency increases/decreases as we move towards/away from the leaves. In this operation, the preconditioner (M) is *consistent*, the operand vector is *inconsistent*, and the result vector will be *consistent*.

TRANSFORMS: It is possible to change the distribution of each operand as follows. For example, if we need a data structure with a specific type of consistency, and this is the result from a previous operation, but with an inappropriate consistency type, we can convert the structure. We have two types of transforms:

- $x^i \rightarrow x^c$: To change from *inconsistent* to *consistent*, we proceed as in the preconditioner application, but with $M = I$. Thus, no changes are made inside each node, and only the additions and copies related to each DAG transition are applied.
- $x^c \rightarrow x^i$: The transformation of a data structure from *consistent* to *inconsistent* only requires a local adjustment.

In Figure 4.6 we illustrate the PCG algorithm with the combination of different types of distribution (*consistent* and *inconsistent*) for each operation, and the additional reductions required to complete the DOTs. Moreover, we have changed the order of some operations to merge the two operations that cross the DAG (O06 + O10).

Note that the number of leaves in the DAG grows exponentially with the number of nested dissection steps, so that the degree of concurrency can be easily increased by expanding additional levels. However, there exists a balance between the number of levels, that determines the number of independent tasks, and the convergence rate of the procedure. Concretely, each dissection step introduces additional numerical levels in the computation, yielding both a different DAG and a distinct preconditioner. While the numerical properties of all these preconditioners are similar, in many cases the number of iterations of the PCG solver grows significantly after a few levels (8 and more) are expanded.

4.2 Parallel Programming Models

The past few decades have witnessed a steady advancement in computer performance. This improvement is due, among other factors, to the introduction of parallel architectures. Microprocessors drove performance increments and cost reductions in computer applications for more than two decades. However, around 2003, the performance increases stopped because heat dissipation and energy consumption issues limited the CPU clock frequencies and the number of tasks that can be performed within each clock period [68]. The solution adopted was to switch to a model where the microprocessor has multiple processing units, known as cores [108].

Nowadays, there are *multicore* architectures integrating a few cores into a single microprocessor, and *manycore* architectures consisting of a large number of cores. Task parallelism is among the best alternatives to take benefit of the additional hardware concurrency in these new architectures.

O0. $A \rightarrow M$	O0. Preconditioner computation
Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j^c > \tau_{\max})$	**Iterative CG solve**
O01. $v_j^i := A^i p_j^c$	O01. SPMV
O02. $\delta_j^i := (p_j^c)^T v_j^i$	O02. DOT
Reduction (δ_j^i)	
O03. $\alpha_j^c := \sigma_j^c / \delta_j^c$	O03. SCALAR OPERATION
O04. $x_{j+1}^c := x_j^c + \alpha_j p_j^c$	O04. AXPY
O05. $r_{j+1}^i := r_j^i - \alpha_j v_j^i$	O05. AXPY
O06. $z_{j+1}^c := (M^c)^{-1} r_{j+1}^i$	O06. PRECONDITIONER APPLICATION
$r_{j+1}^c \leftarrow r_{j+1}^i$	TRANSFORMATION
O07. $\sigma_{j+1}^i := (r_{j+1}^i)^T z_{j+1}^c$	O07. DOT
O10. $\tau_{j+1}^i := (r^c)_{j+1}^T r_{j+1}^i$	O10. 2-NORM
Double Reduction (σ_j^i, τ_j^i)	
O08. $\beta_j^c := \sigma_{j+1}^c / \sigma_j^c$	O08. SCALAR OPERATION
O09. $p_{j+1}^c := z_{j+1}^c + \beta_j^c p_j^c$	O09. XPAY (similar to AXPY)
$j := j + 1$	
end while	

Figure 4.6: Algorithmic formulation of the PCG method taking into account the *consistency* of the data structures.

Therefore, the applications have to be rewritten by the developer, partitioning the workload into tasks, and mapping these tasks to the workers (cores).

Traditionally, parallel systems have been divided into two broad categories: shared memory and distributed memory [176]. The first type provides a single memory address space accessible to all the processors. In the second type, instead of having a global address space, each processor has its own memory. More recently, hybrid shared-distributed memory systems have been built, combining the features of both architectures.

The conventional parallel programming models include a pure shared memory model [68, 176] called OpenMP [150, 58, 59], and a pure message-passing model [68, 176] named MPI [90, 91, 152]. However, today it is common to mix both shared and distributed memory models to improve the performance on current hybrid architectures. In addition, the availability of General Purpose computation on GPUs, and other manycore accelerators, has lead to the Heterogeneous Parallel Programming (HPP) model, which takes advantage of the capabilities of multicore CPUs and manycore GPUs.

The objective of this dissertation is the parallelization of ILUPACK on multicore and manycore processors/accelerators. For that purpose, we first developed a data-flow version of this code with considerable levels of thread-concurrency using OmpSs, and then we implemented a hybrid version of ILUPACK using MPI+OmpSs to exploit the benefits of each programming model.

In this section we revisit the parallel programming models which are the target of our work: OpenMP, OmpSs, and MPI.

4.2.1 OpenMP

OpenMP [58, 59] is a shared-memory API that provides a portable and scalable model to facilitate shared-memory parallel programming. OpenMP is implemented as a combination of a set of compiler directives (`#pragmas`), and a runtime providing both management of the thread pool and a set of library routines. These directives can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among the threads that will execute on different processors or cores and to orchestrate accesses to shared data as needed. Therefore, OpenMP requires specialized compiler support to understand and process these directives.

The use of threads is highly structured in OpenMP because it was designed specifically for parallel applications. In particular, the switch between sequential and parallel sections of code follows the fork/join model. Thus, when a parallel region is reached, a single thread of control splits into a number of independent threads (fork), and the sequential execution is resumed when all the threads have completed the execution of their tasks (join). OpenMP is specially suited to exploit loop parallelism, and from version 3.0, it can also address the task parallelism.

The success of OpenMP can be attributed to a number of factors. One is its strong emphasis on structured parallel programming. Another is that OpenMP is comparatively simple to use, since the burden of working out the details of the parallel program is up to the compiler. It has the major advantage of being widely adopted, so that an OpenMP application will run on many different platforms from small to large Symmetric Multi-Processing (SMP) architectures and other multi-threading hardware. Nowadays, the recent versions of GNU and Intel compilers give support to the OpenMP standard.

4.2.2 OmpSs

OmpSs [2, 74] is a task-based programming model developed at Barcelona Supercomputing Center (BSC). The model supports automatic detection of data dependencies between tasks, determined at execution time by hints given in the form of directionality clauses embedded into compiler directives. Armed with this information, a task graph is generated during the execution that is then scheduled by the OmpSs runtime to the cores, exploiting the inherent task parallelism. This feature is now also integrated in the OpenMP 4.0 standard.

The use of data-dependencies between the different tasks of the program in OmpSs enables asynchronous parallelism. When a function is annotated with the task construct, each invocation of that function becomes a task creation point. The task construct allows to express data-dependencies using `in` (standing for input), `out` (standing for output) and `inout` (standing for input/output) clauses to this end. Each time a new task is created, its `in` and `out` dependencies are matched against those of existing tasks. If a Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read (WaR) dependency is found, the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime, so that tasks are scheduled for execution as soon as all their predecessors in the graph have finished, or at creation, if they have no predecessor.

OmpSs can be combined with MPI, providing a highly asynchronous programming model where communication tasks are overlapped with computation tasks avoiding the need for global synchronization. Another interesting property of OmpSs is the support for heterogeneous platforms, as it can combine CUDA or OpenCL in regular C/C++ or Fortran codes, relieving the programmer from performing data transfers, allocating memory or even compiling the kernels, which are all automatically performed by the OmpSs runtime. OmpSs has been successfully applied to appli-

cations of different nature, and especially in complex scientific codes targeted in projects such as Montblanc [138], DEEP [64] and INTERTWinE [115].

4.2.3 MPI

MPI [90, 91, 152] is a standard library interface for writing parallel programs. Its specification uses message-passing operations where communication between processes is done by ex-changing messages. Although it is oriented to distributed systems and manycore architectures, it can also be employed in shared-memory architectures and multicore processors, expanding its area of use. The goal of MPI is to establish a portable, efficient, and flexible standard for message-passing that will be widely adopted. In fact, it has become the "industry standard" for writing message-passing programs on HPC platforms. MPI favors the Single Program Multiple Data (SPMD) and the Master/Worker program structure patterns.

In the message-passing model, the processes executed in parallel have separate memory address spaces, and the programmer has to handle the tasks which are computed by each process. Hence, communication occurs when part of the address space of one process is copied into the address space of another process. This operation is done cooperatively when the first process executes a send operation and the second executes a receive operation. Communication modes in MPI comprise point-to-point, collective, one-sided (since MPI-2), and parallel I/O operations.

In conclusion, MPI is well suited for applications where portability, both in space and in time, is important. MPI is also an excellent option for task-parallel computations and for applications where the data structures are dynamic. Today, there exist several public implementations of the MPI standard, among which MPICH [140], OpenMPI [151] and MVAPICH2 [142] stand out.

4.3 Setup and Test Cases

In the following sections we describe how to exploit task parallelism in ILUPACK via solutions based on different programming models, and we also optimize the ILUPACK PCG solver for NUMA architectures and manycore accelerators. Previous to this presentation, in this section we describe the target platforms and the test cases employed in the evaluation experiments of this dissertation.

Setup

All experiments in the following sections were performed using IEEE 754 double-precision arithmetic on the following five platforms:

INT_SANDY is a server equipped with two Intel Xeon E5-2670 (8-core) processors at 2.1 GHz and 32 Gbytes of DDR3 RAM. (Section 4.4).

AMD is a server with an AMD Opteron 6276 (16-core) processor at 2.1 GHz and 64 Gbytes of DDR3 RAM. (Section 4.4).

MARENOSTRUM is a large-scale computing infrastructure at BSC, that connects 3,056 compute nodes via an Infiniband Mellanox FDR10 network. Each node contains two Intel Xeon E5-2670 processors for a total of 16 cores per server (2.6 GHz). The nodes employed in our experiments were also equipped with 64 Gbytes of DDR3 RAM. (Section 4.5).

XEON PHI is a board with an Intel Xeon Phi 5110P co-processor attached to a server through a PCI-e Gen3 slot. (The tests on this board were ran in native mode and, therefore, the

specifications of the server are irrelevant.) The accelerator board comprises 60 x86 cores running at 1,053 MHz and 8 Gbytes of GDDR5 RAM. (Section 4.6).

OPTERON is a server with four AMD Opteron 6276 (16-core) processors at 2.1 GHz and 64 Gbytes of DDR3 RAM. (Section 4.6)

Test cases

For the analysis, we mainly employed a large-scale linear system corresponding to the Laplacian equation $-\Delta u = f$ in a $3D$ unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions, $u = g$ on $\partial\Omega$, and a discretization that resulted in a SPD system, with instances of different size; see Table 4.1.

	Matrix	**Dimension n**	**#non-zeros n_z**	**Density (%)**
Laplace	A100	1,000,000	3,970,000	3.97E-6
	A126	2,000,376	7,953,876	1.99E-6
	A159	4,019,679	16,002,873	9.90E-7
	A171	5,000,211	19,913,121	7.96E-7
	A182	6,028,568	24,014,900	6.61E-7
	A191	6,967,781	27,762,041	5.72E-7
	A200	8,000,000	31,880,000	4.98E-7
	A252	16,003,008	63,821,520	2.49E-7
	A318	32,157,432	128,326,356	1.24E-7
	A400	64,000,000	255,520,000	6.23E-8

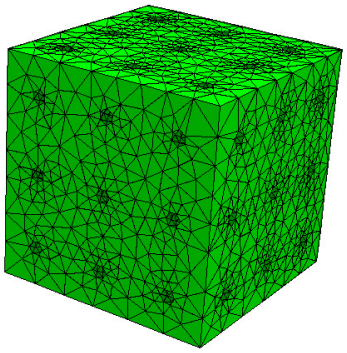
Table 4.1: Matrices employed in the experimental evaluation, where n_z only accounts for the non-zeros in the upper triangular part.

In addition, in Section 4.5 we also use an irregular 3D problem $-div(Agradu) = f$; in a 3D domain (see the left part of Figure 4.7), where $A(x, y, z)$ is chosen with positive random coefficients. Hereafter, we refer to this problem as *mygeo3*, using linear finite elements for the discretization. The size and number of non-zero elements of the resulting sparse SPD linear systems depend on the initial mesh refinement level and the number of additional mesh refinements. Based on the initial mesh shown in the left part of Figure 4.7, the mesh refinement tool NETGEN [144] refines the mesh up to two times based on the meshing levels (very coarse, coarse, moderate, fine, very fine) as provided by the software [131].

In the experiments, all entries of the right-hand side vector b were initialized to 1, and the PCG was started with the initial guess $x_0 \equiv 0$. For the tests, the parameters that control the fill-in and convergence of the iterative process in ILUPACK were set as `droptool` = 1.0E-2, `condest` = 5, `elbow` = 10, and `restol` = 1.0E-6.

4.4 Leveraging Task-Parallelism with OmpSs

In this section we describe how to exploit task parallelism for the efficient solution of sparse linear systems on multi-threaded processors via ILUPACK multi-level PCG method [13]. An initial data-flow version of this code using an *ad-hoc* runtime based on OpenMP [150] was developed in [22]. The adoption of OpenMP was enforced by the lack of a general-purpose tool that combined efficiency, stability and portability. As a result, this solution featured the undesired property of strongly coupling the numerical algorithm with the ad-hoc runtime. To address this problem, we investigate the parallelization of ILUPACK PCG on multicore processors with considerable levels of thread-concurrency using OmpSs.



Code	Initial Mesh	# refs	n	(nnz_A)	nnz_A/n
VC	very coarse	0	1,709	16,669	9.75
C	coarse	0	9,583	112,563	11.75
M	moderate	0	32,429	412,251	12.71
F	fine	0	101,296	1,368,594	13.51
VC2	very coarse	2	271,272	3,686,594	13.59
M1	moderate	1	297,927	4,134,255	13.88
VF	very fine	0	658,609	9,294,721	14.11
F1	fine	1	882,824	12,562,880	14.23
C2	coarse	2	906,882	12,854,824	14.17
VC3	very coarse	3	2,382,864	34,128,924	14.32
M2	moderate	2	2,539,954	36,768,808	14.48
VF1	very fine	1	5,413,520	78,935,174	14.58

Figure 4.7: Computational domain in 3D for *mygeo3* problem (left) and benchmark matrices resulting from several discretizations of the computational domain (right). The table (right) presents, for each benchmark, the code, the initial mesh refinement level, the number of additional refinements, the number of unknowns, the number of nonzero elements in A, and the average number of nonzero elements in each row.

Next, we describe in detail the process followed to identify and capture data dependencies with OmpSs, so as to expose and leverage the task parallelism intrinsic to the PCG method in ILUPACK, while making minimal changes to the legacy codes of this package. Moreover, we explain the optimization strategies adopted, and illustrate that this parallel data-flow implementation based on the OmpSs runtime system reports notable performance on highly concurrent platforms from Intel and AMD. Overall, we provide practical evidence that the OmpSs framework, with its embedded data-dependency analyzer and dependency-aware scheduling mechanism, provides a seamless and efficient tool to tackle a complex scientific code like the PCG solver in ILUPACK.

4.4.1 Task-parallel implementation using OmpSs

First, we introduce how to exploit the concurrency by means of OmpSs, resulting in the performance, convergence rate, and numerical accuracy that is illustrated in Section 4.4.2. Using a pair of data structures, we capture the task dependencies that appear in the two most challenging operations in the method, namely the calculation of the preconditioner and its application, passing this information to the OmpSs runtime which can then implement a correct and efficient schedule of the entire solver.

Task-parallel PCG method

The operations that compose the computation of the iterative PCG solve exhibit a clear set of dependencies which dictate almost a strict order for their computation (see subsection 4.1.3). These dependencies can be easily controlled using the OmpSs `#pragma omp task` directive. For example, the RaW dependency $\alpha_j \rightarrow x_{j+1}$ is simply enforced by declaring the headers for the routines that compute $\alpha_j := \sigma_j / p_j^T v_j$ (DOT product) and $x_{j+1} := x_j + \alpha_j p_j$ (AXPY) as follows:

```
// alpha := sigma / (p^T * v)
#pragma omp task input (n, sigma, p[0:n-1], v[0:n-1]) output(alpha)
void DOT(int *n, double *sigma, double p[], double v[], double *alpha);
```

```
// x := x + alpha * p
#pragma omp task input(n, alpha, p[0:n-1]) inout(x[0:n-1])
void AXPY(int *n, double *alpha, double p[], double x[]);
```

In practice, OmpSs identifies the data dependencies between tasks —i.e., program functions— that dictate the data-flow execution by tracking the order in which functions are invoked in a serial execution, checking the directionality (input, output or inout) of each operand in the argument's list, and matching the operands' memory addresses at runtime with those of other tasks in execution. In the example, the data dependency between the two functions is detected at execution time, when they are invoked with the same variable α (actually, the same memory address), in the first case as an output operand and in the second as an input operand.

The real opportunities to exploit concurrency in the entire PCG method lie within the computations that involve the preconditioner, the sparse matrix-vector product ($v_j := Ap_j$, SpMV), and the DOT product ($p_j^T v_j$), as described next.

Task-parallel preconditioner with OmpSs

ILUPACK is quite an involved code, which allocates/releases memory dynamically for complex data structures, turning the process of capturing the dependencies via OmpSs pragmas which are directly based on the actual function's arguments into a delicate exercise. Furthermore, proceeding along that line would require an extensive reorganization of the package and a full rewrite of certain parts. For these reasons, we instead decided to create a "skeleton" structure that explicitly exposes the dependencies in the DAG associated with the preconditioner, analogous to the use of "representants" in [34]. The key advantage of this approach is that we limit the amount of changes that are necessary to introduce OmpSs in ILUPACK's legacy code. Besides, we can leverage this "skeleton" structure and parallelization scheme to exploit the concurrency in similar solvers that use different ILU preconditioning techniques to calculate the preconditioner. In fact, we also parallelized the ILU(0) algorithm by using this methodology with minor changes in the code.

In order to describe how we capture the data dependencies and exploit task parallelism in the preconditioner with OmpSs, we will consider the DAG/binary tree represented in Figure 4.2 as a workhorse. In any case, this approach is analogous for unbalanced and/or non-binary trees. The dependencies of this graph can be easily captured using a matrix of integers, `dag[3][ntasks]`, where each column contains, for the corresponding task, the identifiers of the left/right descendant tasks and the ancestor task; see Table 4.2.

In practice, the user explicitly determines the tree-like concurrency of the preconditioner calculation in ILUPACK, before the execution commences, by carefully manipulating a graph-based symmetric reordering tool (as, e.g., Metis or Scotch) to fix the number of levels and nodes in the preconditioner, as we explained in Subsection 4.1.1. In consequence, this skeleton structure can be automatically created and initialized before the parallel computation of the preconditioner begins.

In order to explain how the computation of the preconditioner works, we consider that all the processing within any of the DAG tasks that compose the preconditioner computation is performed by invoking the same function, `ILUPrecond`, with the following header:

```
void ILUPrecond(SparseMatrix *spMat, SparseFactor *spFact, int taskid);
```

In this argument's list, `SpMat` is an input (i.e., read-only) structure containing the sparse matrix, `SpFact` is an input-output (i.e., read-write) structure for the sparse triangular factors, and `taskid` is an input integer that identifies the task to be processed during this invocation. For simplicity,

Task	T_0	T_1	T_2	T_3	T_4	T_5	T_6
Task id. j	0	1	2	3	4	5	6
Left descendant, `dag[0][`j`]`	–	–	–	–	0	2	4
Right descendant, `dag[1][`j`]`	–	–	–	–	1	3	5
Ancestor, `dag[2][`j`]`	4	4	5	5	6	6	–

Table 4.2: Contents of the `dag` data structure representing the nodes (tasks) and dependencies of the DAG in Figure 4.2. Here, `dag[0][`j`]`, `dag[1][`j`]`, and `dag[2][`j`]`, $j = 0,1,\ldots,6$, contain, respectively, the values in the rows labeled as "left descendant id.", "right descendant id.", and "ancestor". The symbol "–" is used to indicate that the task has no left/right descendents (i.e., it is a leave) or ancestor (for the root).

we omit several other parameters that are present in the function definition but are not relevant for the following discussion.

In the parallel implementation, function `ILUPrecond` is modified to include two new parameters, corresponding to `dag` and `vector`, and its header declaration preceded with the "taskifying" OmpSs pragma:

```
#pragma omp task in (vector[dag[0][taskid]], vector[dag[1][taskid]])
                 out(vector[taskid])
void ILUPrecondPar(SparseMatrix *spMat, SparseFactor *spFact, int taskid,
                int vector[], int dag[][]);
```

Here the contents of `vector` (in this case an integer array with seven entries, one per task) are irrelevant, since this structure is only used to create references to different memory addresses, which are then passed to OmpSs in order to identify the data dependencies.

To illustrate this, consider for example the following sequence of events. When function `ILUPrecond` is eventually invoked to process task T_5 (i.e., with `taskid`=5), the runtime encounters the following call:

```
// Process task T_5
#pragma omp task in (...) out(vector[5])
void ILUPrecondPar(spMat, spFact, 5, vector, dag);
```

This identifies `vector[5]` as an output of this function/task, while the input parameters are irrelevant for the discussion. This is eventually followed by a call to the same function, this time to process task T_6 (with `dag[1][6]`=5):

```
// Process task T_6
#pragma omp task in (..., vector[5]) out(...)
void ILUPrecondPar(spMat, spFact, 6, vector, dag);
```

which is also captured by the runtime, identifying `vector[5]` as an input parameter in this case. Now, the order of the calls and the references to memory, in both cases to the same address, `&vector[5]`, first as an output and then as an input, allow the runtime to identify the RaW dependency $\mathsf{T}_5 \rightarrow \mathsf{T}_6$.

Task-parallel triangular solves with OmpSs

After the computation of the preconditioner $M = LDU$, its application $z_{j+1} := M^{-1}r_{j+1}$ at each iteration of the PCG method requires the solution of two triangular systems: First, the lower triangular system $y := \hat{L}^{-1}r_{j+1}$, with $\hat{L} = LD$; and next the upper triangular system $z_{j+1} := U^{-1}y$. The parallelization of the lower triangular system solve presents the same DAG as the preconditioner computation, and therefore, the same approach can be applied. The function that performs the processing associated with each one of the DAG tasks, annotated with the corresponding OmpSs pragma, is:

```
#pragma omp task in (vector[dag[0][taskid]], vector[dag[1][taskid]])
                 out(vector[taskid])
void ILULwSolvePar(SparseFactor *spFact, double r[], double y[], int taskid,
                int vector[], int dag[][]);
```

For the parallelization of the upper triangular system, the dependencies of the DAG are inverted, which simplifies the process since only one dependency must be considered per task. The function that processes the tasks during this solve is thus annotated as:

```
#pragma omp task in (vector[dag[2][taskid]]) out(vector[taskid])
void ILUUpSolvePar(SparseFactor *spFact, double y[], double z[], int taskid,
                int vector[], int dag[][]);
```

Here `vector[dag[2][taskid]]` identifies the corresponding ancestor in the binary tree. With this scheme, the OmpSs runtime can detect that, for example, `vector[6]` is an output for task T_6 as well as an input for task T_5 (`dag[2][5]=6`). Therefore, given that during the upper triangular solve the call to function `ILUUpSolve` with `taskid=6` is encountered by the runtime before that to the same function with `taskid=5`, by matching the memory addresses, OmpSs correctly identifies and controls the RaW dependency $\mathsf{T}_6 \rightarrow \mathsf{T}_5$ for the upper triangular solve.

In summary, the entries of `vector[]` act as representants for the tasks in the corresponding DAGs, and together with the `dag[][]` structure, they govern the dependencies during the computation of the preconditioner and the iterative solution of the subsequent triangular systems.

Task-parallel sparse-matrix vector product with OmpSs

For this operation, we exploit that, after applying the appropriate recursive graph-basic reordering, defined by P (see Section 4.1.2), matrix A is disassembled into a collection of submatrices, one per leaf task, like that in (4.3). Thus, the product $v_j := (P^T AP)p_j$ can be decomposed into a number of "independent" smaller matrix-vector products (e.g., 4 in equation (4.5)). This calculation is parallelized using OmpSs pragmas that render a concurrent parallel execution of the small matrix-vector suboperations.

Task-parallel DOT product with OmpSs

This operation calculates a single DOT product per leaf task, because each task contains a block of the operand vectors, disassembled like in the sparse-matrix vector product (Subsection 4.4.1). Afterwards, there is a transformation of the scalar from inconsistent to consistent state, implemented as a reduction of the subvectors local to each thread. This involves an atomic addition and, therefore, a synchronization/barrier at the end of this operation (`pragma_omp_taskwait`).

4.4.2 Optimization and experimental results

In this section we introduce two optimization strategies to improve the performance of the previous implementation of ILUPACK using OmpSs. Furthermore, we evaluate the optimized parallelization in high-end multicore platforms equipped with Intel and AMD processors. Our results report significant performance gains, demonstrating that OmpSs provides an efficient and close-to-seamless means to leverage the concurrency in a complex scientific code like ILUPACK.

All experiments in this section were performed on the INT_SANDY and AMD platforms, described in Section 4.3. Moreover, the software included the Mercurium C/C++ compiler (1.99.0) with support for OmpSs, Metis (4.0.3) for the graph reorderings, and ILUPACK (2.4). For the evaluation we employed the A200 matrix (see Table 4.1).

Prioritizing tasks

Taking into account that the bulk of the computational load is concentrated in the leaves, during the computation of the preconditioner and the subsequent iterative PCG solve, it is important to assign priorities so that the leaves of the task dependency trees are executed first. The primary reason for advancing the execution of these tasks is that it provides a better chance to balance the distribution of the workload among the threads.

In order to assign priorities using OmpSs, we had to distinguish the leaf tasks in the calls to `ILUPrecond`, `ILULwSolve` and `ILUUpSolve`; and include the appropriate `priority` clause as part of the taskifying directive. For example, for the first routine, we created two different routine calls:

```
#pragma omp task in(...) out(...) priority(high)
void ILUPrecondPar_LeafTask(...);

#pragma omp task in(...) out(...) priority(low)
void ILUPrecondPar_NoLeafTask(...);
```

which internally simply invoked the original routine `ILUPrecond`.

The effect of introducing priorities on the computation of the preconditioner is graphically illustrated in Figure 4.8, which clearly shows how the priority mechanism enforces that the leaves of the task dependency tree are executed first. (All execution profiles in this dissertation were obtained with `Extrae` [79] v2.5.1.)

Controlling task granularity to reduce overhead

Our initial experiments with the iterative PCG stage revealed an excessive cost of the vector operations (DOT product, AXPY update, and 2-norm), much higher than could be expected from their theoretical cost; see the top plot in Figure 4.9. Further investigation revealed that this overhead was due to the large number of tasks that were created for each vector operation. To tackle this problem, we merged certain operations of the PCG iteration in order to increase their granularity. In particular, for the sequence of operations that compose the PCG loop in Figure 4.6, we merged the computation of v_j with α_j (SPMV with DOT product); x_{j+1} and r_{j+1} (AXPYs); and ζ_{j+1} with τ_{j+1} (DOT product and vector 2-NORM).

Figure 4.9 reveals the outcome of collapsing these vector operations, showing much narrower time "bands" for the execution of the corresponding tasks in the merged version.

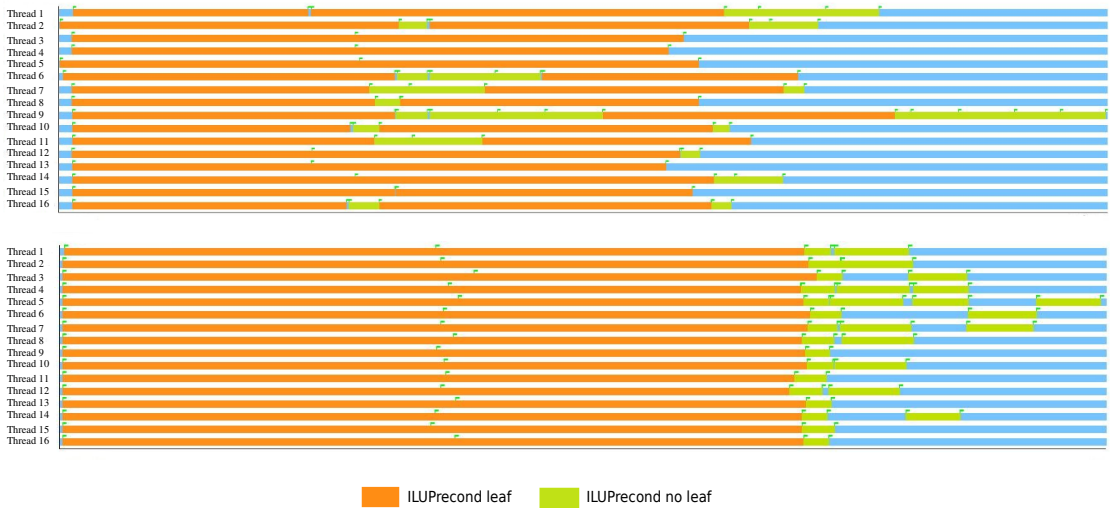


Figure 4.8: Trace of the preconditioner computation without and with priorities (top and bottom, respectively) on the Intel Xeon E5-2670, using 16 cores/threads and a decomposition of the sparse matrix into a tree with 32 leaves, for the A200 problem.

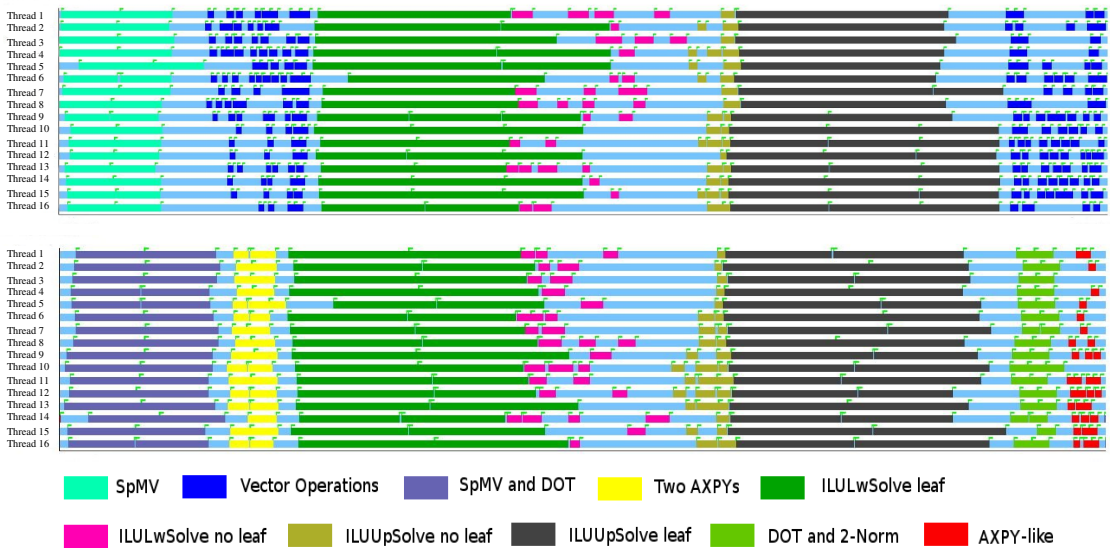


Figure 4.9: Trace of a single PCG iteration of the solve stage with unmerged and merged kernels (top and bottom, respectively) on the Intel Xeon E5-2670, using 16 cores/threads and a decomposition of the sparse matrix into a tree with 32 leaves, for the A200 problem.

DAG concurrency vs acceleration

As argued earlier, there is a trade-off between the computational cost of the preconditioner computation/iterative PCG solve and the concurrency of these two stages, which is determined by the number of levels/leaves of the task dependency trees and the cost of the individual tasks that are involved in the sparse matrix-vector product and the construction/application of the preconditioner. To illustrate this situation, we consider first the solution of the target linear system partitioned into a DAG (tree) with a single leaf/level vs one with multiple levels, using a single core of the Intel-based (INT_SANDY) server in all cases. For example, the computation of the preconditioner for the single-leaf DAG, using the sequential implementation of ILUPACK, required 137.07 seconds. This time is reduced when the multi-level DAG/preconditioner is computed using only one core and the associated DAG consists of up to 128 leaves (concretely, 123.27 and 136.33 seconds for 32 and 128 leaves, respectively), due to differences in the fill-in patterns between the single level and multi-level cases; but the difference then grows to 172.61 seconds for 256 leaves, due to the additional flops associated with the higher number of levels. For the iterative PCG solve, the multi-level partitionings incur an increased computational cost as well as a higher number of iterations (see next subsection). The outcome of these combined factors is that, on the INT_SANDY server, the PCG solve requires 193.17 seconds in the single-leaf DAG vs 200.10, 180.53 and 298.81 seconds with 2, 32 and 128 leaves, respectively, when executed on a single core. The large increase in the 128-leaf DAG is also due to the additional flops required by the superior number of levels.

Figure 4.10 reports the speed-up of the parallel (data-flow) implementations of preconditioner computation and PCG solve (per iteration) for the two platforms employed in the evaluation. The acceleration rates were always computed with respect to the sequential legacy implementation of ILUPACK, running with a a single thread/core. For the parallel implementation, in general the best results are obtained when the number of leaves equals or doubles the number of threads/cores. We emphasize that the speed-ups embed the increment in the computational cost that occurs when the number of levels in the DAG is increased. Thus, for the INT_SANDY server, the speed-ups vary between 2.09/1.56 for 2 cores and 32/256 leaves; and 12.11/9.67 for 16 cores and 32/256 leaves for the calculation of the preconditioner. (The superlinear speed-up in the execution with 2 cores/32 leaves can be due to a better utilization of the cache system or a smaller fill-in.) The values for the iterative solve stage are similar: 2.31/0.95 for 2 cores and 16/256 leaves; and 9.44/4.80 for 16 cores and 32/256 leaves. Slightly lower speed-ups were obtained for the AMD platform.

DAG concurrency vs numerical properties of the solver

In order to assess the numerical behaviour of the preconditioner/PCG solver as a function of the number of leaves/tasks (i.e., concurrency), we utilize the A-norm defined in [104], with the estimator in [178], as a measure of the numerical accuracy of the approximate solution x_j computed at the j-th iteration: $\|x - x_j\|_A$, where x stands for the correct solution. Figure 4.11 shows that, for a fixed residual A-norm, there is a slight increase in the iteration count as the number of leaves grows from 1 (sequential legacy implementation in ILUPACK) up to 256. For example, in order to achieve a residual of order 1.0e–12, the sequential code requires 68 iterations, while this value grows to 77, 79 and 85, for 2, 32 and 256 leaves, respectively. In any case, from the numerical point of view, the parallel methods can still deliver the same level of accuracy (residual A-norm) as the sequential implementation at the expense of a slight increase of the theoretical cost, which is more than compensated in the parallel execution.

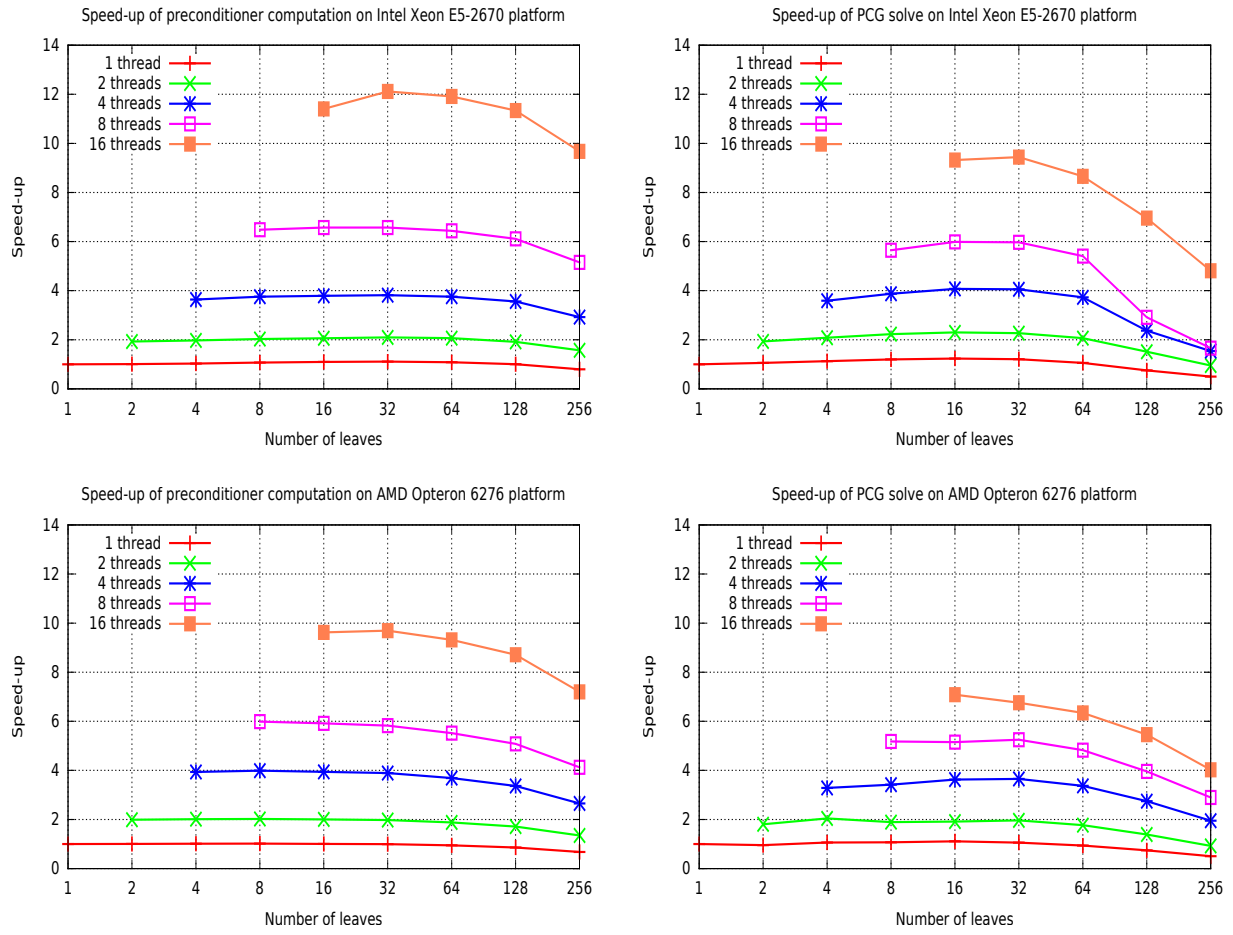


Figure 4.10: Speed-ups attained with the data-flow ILUPACK method parallelized with OmpSs, for the A200 problem. The left-hand side plots correspond to the computation of the preconditioner and the right-hand side plots to the iterative PCG solve.

4.5 Exploiting Task-Parallelism with MPI + OmpSs

In this section we introduce a parallel implementation of the preconditioned iterative solver for sparse linear systems underlying ILUPACK that explores the interoperability between the message-passing MPI programming interface and the OmpSs task-parallel programming model [14]. Our approach commences from the task dependency tree derived from a multi-level graph partitioning of the problem, and statically maps the tasks in the top levels of this tree to the cluster nodes, fixing the inter-node communication pattern. This mapping induces a conformal partitioning of the tasks in the remaining levels of the tree among the nodes, which are then processed concurrently via the OmpSs runtime system.

In Section 4.4, we exploited the task parallelism exposed by the DAG associated with the sparse matrix to develop a parallel version of ILUPACK PCG solver for shared-memory multiprocessors that relies on OmpSs [12, 13]. Moreover, a parallel version of ILUPACK for clusters using MPI was developed in previous works [12, 23]. Unfortunately, the previous MPI version of ILUPACK

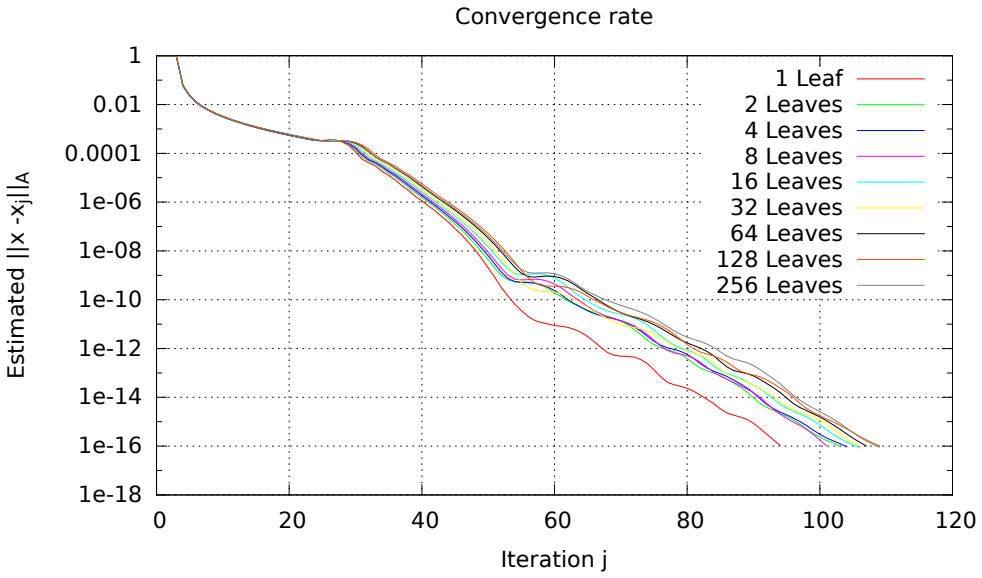


Figure 4.11: Error estimation via the A-norm and convergence rate for different number of leaves/-tasks for the A200 problem.

could only map one leaf of the DAG to each MPI rank, impeding the exploitation of other types of parallelism inside the nodes. This is a strong limitation for clusters consisting of "fat" nodes, equipped with a significant numbers of cores per node, as the static correspondence between tasks and MPI ranks may result in an unbalanced distribution of the workload and, therefore, be a source of inefficiency.

We present a new implementation of ILUPACK which merges MPI and OmpSs to exploit the benefits of each programming model, and allows the execution of the solver with more than one leaf per MPI process. In addition, we perform an experimental evaluation in order to assess the impact of the MPI+OmpSs configuration, problem dimension, and number of leaves per core on the performance of the iterative solve. Our results on the MARENOSTRUM cluster, equipped with 16 Intel Xeon cores per node, reveals that the MPI+OmpSs version consistently outperforms the initial MPI code in terms of both strong and weak scaling.

4.5.1 Task-Parallel implementation with MPI+OmpSs

In this subsection, we first briefly review how to exploit the task parallelism explicitly exposed by the DAG, using either OmpSs or MPI, to then introduce our approach that combines both parallel programming models to yield a task-parallel MPI+OmpSs solution.

Parallelization using OmpSs

The opportunities to exploit task parallelism in ILUPACK PCG method lie within the computations that involve the preconditioner (computation and application) as well as the vector operations, as we have described previously in Section 4.4.

Parallelization with MPI

The original MPI-based parallel version of ILUPACK, introduced in [23], spawns one MPI rank per leaf (task) of the DAG, with a one-to-one static mapping between leaves and ranks. This task-rank correspondence is fixed before the preconditioner computation, by the root process, which sends the information for each leaf to the appropriate MPI rank. The same mapping is then maintained during the complete execution, for all computations and iterations, including the preconditioner computation/application and vector operations.

The operations with the preconditioner potentially transform the dependencies of the DAG into communications among MPI ranks. To reduce the number of transfers, an inner task is always mapped to one of the two MPI ranks where the two "children" tasks were mapped to. For example, in a DAG consisting of 4 leaves mapped to 4 MPI ranks, R0-R3, in order to collapse the first level when the graph is traversed bottom-up during the lower triangular system solve, ranks R0, R2 send their data to R1, R3, respectively. The receivers then accumulate this information with the results from their own computations, and process the tasks in the next higher level, while the senders block till the top-down traversal of the TDG during the upper triangular system solve. Following this strategy, traversing the DAG only requires a communication between "sibling" tasks/"neighbour" MPI ranks.

Disassembling the matrix and the vectors, according to (4.6), allows all other computations of PCG to operate with the leaves, avoiding any communication, except for the DOT operations, which require an MPI reduction (`MPI_Reduce`) to accumulate the values computed in each node.

Combining MPI+OmpSs

In general, a strong motivation for mixing OmpSs with MPI is to unleash a higher level of asynchronism, for example in order to overlap communication with computation. In this particular work, the major advantage of combining both programming models is to exploit dynamic scheduling within the cluster nodes via OmpSs.

The first step to obtain an MPI+OmpSs solution is to develop a *new* MPI version of ILUPACK where a MPI rank can handle a subtree of the DAG comprising several leaves and the related inner tasks. With this version, OmpSs can then be applied to process the tasks mapped to each MPI rank, dynamically distributing the work between several OmpSs threads. For example, consider a two-level DAG composed of one root task and two leaves to be executed on a processor with two cores. If the computational cost associated with the leaves is unbalanced, this can be tackled by expanding an additional level of the DAG, yielding a three-level tree with four leaves. Now, if the parallelization is based on MPI only, an optimal mapping of the tasks to MPI ranks requires a prior knowledge of the computational costs of the tasks. Compared with this, an OmpSs parallel version with 2 threads features a dynamic mapping of tasks to threads that is more flexible and can exploit the resources more efficiently by, e.g., prioritizing the execution of the costlier tasks.

The MPI+OmpSs version still requires an initialization where the root process distributes the data corresponding to (the leaves of) the subtrees among the MPI ranks. The MPI+OmpSs version of ILUPACK is then divided into a sequence of interleaved OmpSs and MPI stages, with the former ones computing the tasks internal to the subtrees local to the MPI ranks by using OmpSs, and the latter requiring communication between MPI ranks. In particular, the computation of the preconditioner comprises only one stage of each type, but its application in the PCG has two OmpSs stages per iteration because the DAG is traversed twice. Figure 4.12 illustrates the initial distribution for a DAG with 8 leaves, together with a scheme of the execution of the two stages in the preconditioner computation. In that example, the OmpSs threads process the tasks within

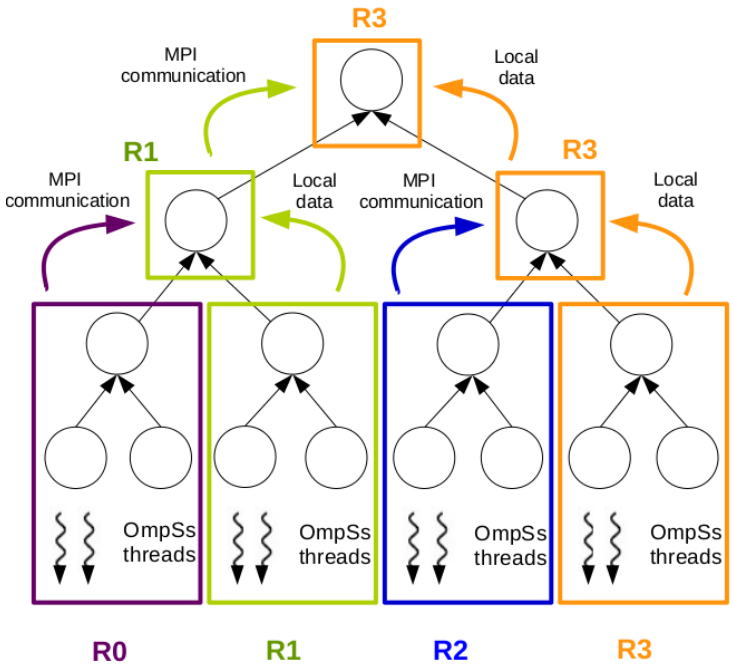


Figure 4.12: Mapping of a DAG to 4 MPI ranks (R0–R3) with 2 OmpSs threads per rank.

the bottom two levels, with no MPI communication involved. For the top two levels, the OmpSs threads remain inactive and it is the MPI ranks that are in charge of processing the tasks. The dot operations also exhibit the same two stages: On the leaves, the OmpSs threads accumulate their local subvectors, and an atomic reduction is then applied to compute the reduction inside each MPI rank. These local values are then reduced using an MPI collective primitive. The sparse matrix-vector product and the remaining vector computations of the PCG iteration operate in the bottom level only and, therefore, are computed by OmpSs threads with no MPI communication involved.

4.5.2 Experimental results

In the following we analyze the performance of two parallel versions of the PCG solver in ILUPACK: one based on MPI that can handle several leaves per MPI rank, with no intervention of OmpSs (hereafter, referred to as MPI-only); and an alternative variant that combines MPI+OmpSs, also capable of processing several leaves per MPI rank, but which does so via OmpSs threads internally to each node. The MPI+OmpSs code was compiled using Mercurium C/C++ (1.99.8), with the OpenMPI (1.8.1) flags `-showme:compile` and `-showme:link`. The MPI-only variant was compiled with the same version of OpenMPI. Other software included OmpSs (15.06), ILUPACK (2.4), and ParMetis (4.0.2) for the graph reorderings. In the executions with the MPI-only version, we spawned one MPI rank per core (i.e., 16 per node); while for MPI+OmpSs, we tested distinct combinations of MPI ranks and OmpSs threads, with the numbers of ranks multiplied by the number of threads always being equal to 16 per node.

Hereafter, we consider the behaviour of the iterative PCG solver only, without the preconditioner computation, because the computational cost of the latter is in general smaller and we observed no significant performance differences between the MPI-only and MPI+OmpSs parallel versions of

this procedure. In addition, several previous experiments revealed that the best performance was obtained when splitting the sparse matrix via nested dissection to generate a DAG with a number of leaves that equals or doubles the number of cores. Therefore, in the following we analyze only these two cases.

Analysis of configurations

In order to assess the performance of the parallel MPI+OmpSs version of ILUPACK, we first evaluate different combinations of MPI ranks and OmpSs threads per node (configurations). Given the node target architecture, with 2 sockets/8 cores per socket, we employ 1, 2, 4, 8 or 16 MPI ranks per node and the corresponding number OmpSs threads that fill all cores per node: 16, 8, 4, 2 or 1, respectively. We will denote these configurations as 1R/16T, 2R/8T, 4T/4T, 8R/2T, and 16R/1T (#Ranks/#Threads). Figure 4.13 reports the ratio of execution time of these configurations with respect to the MPI-only implementation for the A400 problem, splitting the problem to obtain one leaf per core and two leaves per core. Both graphs reveal that, for almost all cases, the best option is 2R/8T, which mimics the internal socket/core architecture of the servers. Furthermore, we also note that the extreme configurations, 1R/16T and 16R/1T, deliver the lowest performance. In the case that employs 1 rank and 16 OmpSs threads, this is due to the intersocket communications. In the alternative with 16 MPI ranks and 1 thread per rank the reason is the overhead introduced by the OmpSs runtime system. In order to avoid this, when exploiting the hardware concurrency using MPI ranks only, we will not employ the OmpSs runtime system in the following. Similar results are shown in Figure 4.14 for two benchmarks of *mygeo3* (see Figure 4.7), on which we only split the problem to obtain two leaves per core.

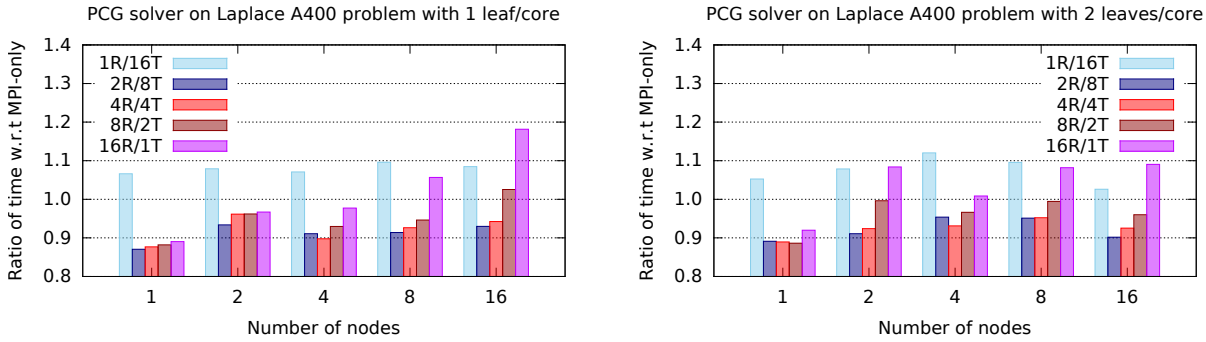


Figure 4.13: Ratio of execution time per PCG iteration with respect to the MPI-only version for the Laplace A400 problem for different configurations, using 1 leaf per core (left) and 2 leaves per core (right).

Analysis of scalability

We first evaluate the strong scalability of the parallel solvers. Figure 4.15 shows the execution time per iteration of the PCG solve for the A400 problem as the resources are increased from 16 cores/1 node to 256 cores/16 nodes. In general, as expected, there is a decrease in the iteration time as the number of cores grows. If we compare the two versions, the results demonstrate that the MPI+OmpSs variant consistently outperforms the MPI version (with no underlying OmpSs runtime system), by a margin that is around 5–10%. Moreover, there is a slight difference between

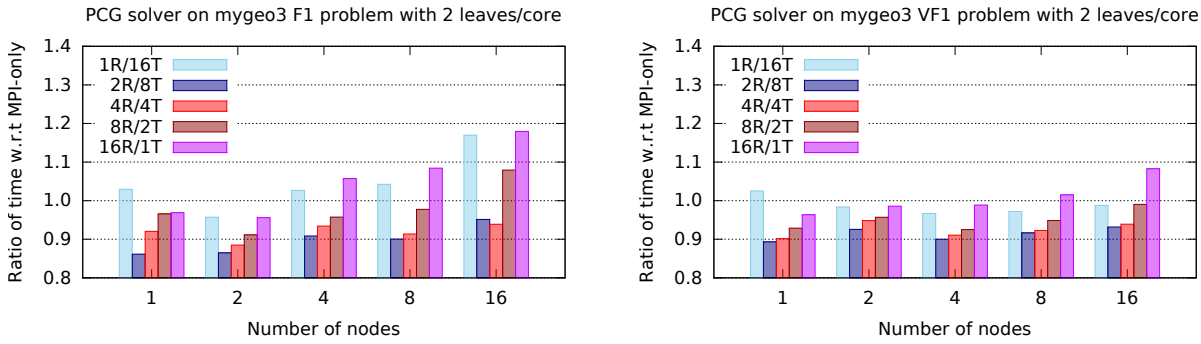


Figure 4.14: Ratio of execution time per PCG iteration with respect to the MPI-only version for two instances of *mygeo3* problem for different configurations, using 2 leaves per core.

the cases with one or two leaves per core that is enlarged with the number of cores, revealing the DAG with one leaf per core as the best choice for 32 or more cores. The reason is that, as the amount of computational resources grows, the additional concurrency explicitly exposed by further splitting the computational load (sparse matrix/adjacency graph) does not compensate the overhead that is introduced for this particular (moderate) problem dimension.

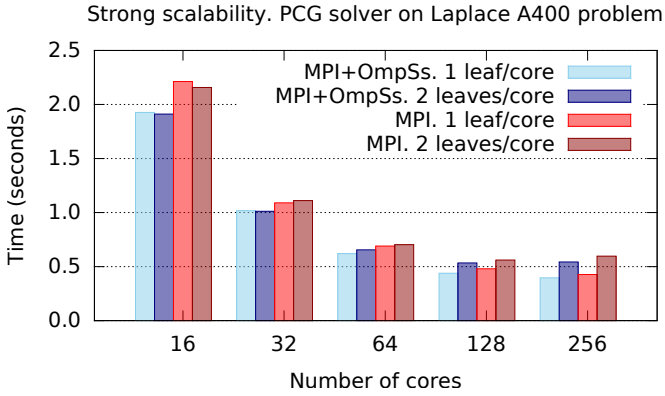


Figure 4.15: Execution time per PCG iteration for the Laplace A400 problem.

The next experiment aims to provide an evaluation of weak scaling for the parallel solvers. Unfortunately, for ILUPACK PCG solve it is not possible to generate an instance of the Laplace problem with a computational complexity that grows exactly in proportion to the number of resources. To approximate this scenario, we set the size (n) and the number of non-zeros of the sparse matrix (n_z) to be roughly proportional to the number of cores. However, we emphasize that n and n_z only offer an estimation of the computational cost, as other factors such as the fill-in/quality of the preconditioner play a relevant role. Figure 4.16 reports the performance of the parallel implementations of the PCG solve (per iteration) for different matrices in Table 4.1. These results show that the execution times grow with the number of cores/problem dimension. The reason is that the number of actual floating-point arithmetic operations per iteration increases faster than n_z. Comparing both implementations, the MPI+OmpSs version outperforms the MPI variant; and the difference between the cases with one or two leaves per core also grows with the number of cores.

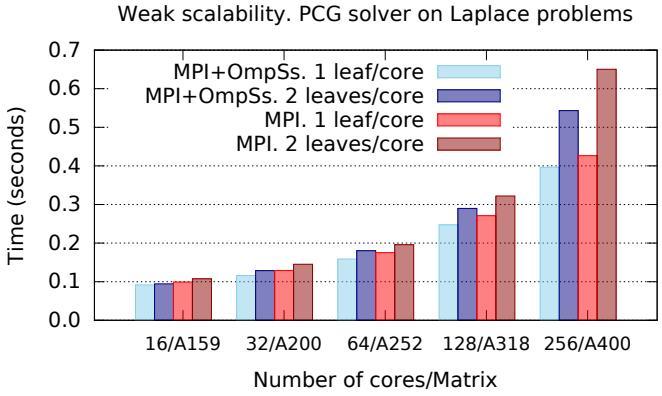


Figure 4.16: Execution time per PCG iteration for different Laplace problems.

4.6 Tuning the Task-Parallel ILUPACK on Many-core Architectures

In this section we present specialized implementations of the preconditioned iterative linear system solver in ILUPACK for NUMA platforms and manycore hardware co-processors based on the Intel Xeon Phi [12]. For the conventional x86 architectures, our approach exploits task parallelism via the OmpSs runtime as well as a message-passing implementation based on MPI, respectively yielding dynamic and static schedules of the work to the cores.

In particular, in the following we revisit our task-parallel versions of ILUPACK (described previously in Sections 4.4 and 4.5), making the following new contributions:

- Our task-parallel implementations target a pair of "conventional" x86-based architectures with large numbers of cores: an Intel Xeon Phi 60-core accelerator (XEON PHI) and a NUMA server with 4 AMD Opteron 6276 sockets totalling 64 cores (OPTERON).

- For the task-parallel version of ILUPACK based on OmpSs, we reformulate our previous implementation to exploit nested parallelism in order to tackle the ample hardware concurrency of the Intel- and AMD-based systems. In addition, we analyze the benefits of a "scattered" mapping of the threads on the Intel Xeon Phi and we enhance the solver to produce a NUMA-aware execution for the AMD server.

- Alternatively, on these two conventional platforms, we also explore the use of the MPI-based implementation of ILUPACK to extract task parallelism and transparently deal with NUMA effects. On the Intel Xeon Phi, we expose the similarities between the thread mapping strategy and the MPI rank mapping policy in this case.

- Finally, we use a common reference application to experimentally evaluate these parallelization alternatives (OmpSs or MPI combined with task parallelism), target platforms (AMD x86 manycore server and Intel Xeon Phi accelerator), and numerical semantics (sequential vs task-parallel) from the perspectives of performance, convergence rate, and numerical accuracy.

4.6.1 OmpSs implementations

We next describe the modifications implemented in the OmpSs version of ILUPACK solver to efficiently execute in NUMA architectures and manycore accelerators. These optimizations

include the exploitation of nested parallelism, the correct mapping of threads to cores, and the accommodation of a NUMA-aware execution.

Exploiting nested parallelism

The operations that appear in the iterative PCG solve (`while` loop in Figure 4.6) define a partial order which enforces an almost strict serial execution. Specifically, at the $(j+1)$-th iteration

$$\ldots \rightarrow \mathrm{O9} \rightarrow \overbrace{\mathrm{O1} \rightarrow \mathrm{O2} \rightarrow \mathrm{O3} \rightarrow \mathrm{O5} \rightarrow \mathrm{O6} \rightarrow \mathrm{O7} \rightarrow \mathrm{O8} \rightarrow \mathrm{O9}}^{(j+1)\text{-th iteration}} \rightarrow \mathrm{O1} \rightarrow \ldots$$

must be computed in that order, but O4 and O10 can be computed any time once O3 and O5 are respectively available. Further concurrency can be exposed by dividing some of these operations into subtasks, as described, for example, in subsection 4.1.3.

Handling the dependencies is easy at the task/subtask levels but rapidly becomes a burden for the OmpSs runtime when the number of cores in the target architecture is large. This scenario asks for a high number of (sub)tasks which, in ILUPACK, necessarily exhibit a small computational cost except for the operations involving the leaf nodes of the preconditioner DAG. In subsection 4.4.2 we increased the granularity of the subtasks by modifying the code to *merge* three pairs (or trios) of operations in the PCG solve into a single "group" of subtasks each [13]: O1+O2+O3, O4+O5 and O7+O10; see Figure 4.6. For example, the SpMV+dot in O1+O2+O3, applied to (4.6), are combined by merging each one of the small matrix-vector products $\bar{v}_i := \bar{A}_{ii} p_j$, $i = 0, \ldots, 3$, with the reduction of the corresponding elements of O2. Additionally, the ordered execution of the groups was controlled by inserting explicit barriers (`#pragma omp barrier`) between each group of subtasks: O1+O2+O3, 04+O5, O6, O7+O10, O8+O9.

In the new implementation, we eliminate the explicit barriers and instead rely on OmpSs to elegantly deal with the nested parallelism exhibited by the task/subtask dependencies. Concretely, the nested variant defines O1+O2+O3, O4+O5, O6, O7+O10, and O8+O9 as five coarse-grain OmpSs tasks (via `#pragma omp task`) and off-loads the complete detection and control of the dependencies to the OmpSs runtime. In addition, this version also divides these five macro-operations into fine-grain subtasks, and merges pairs of them as described above. In order to illustrate this, consider for example O1+O2, consisting of the SpMV $v_j := Ap_j$ and the dot $\alpha_j := \sigma_j / p_j^T v_j$. The code that performs this operation is annotated in Figure 4.17, where, for simplicity, we do not illustrate how to deal with the reduction on α_j (variable `alpha`).

Mapping threads to cores on the Intel Xeon Phi

The Intel Xeon Phi supports up to 4 hardware threads per physical core, while our task-parallel approach spawns one OmpSs thread per leave in the preconditioner DAG. A critical aspect in this platform is how to bind the OmpSs threads to the hardware threads/cores in order to distribute the workload. The mapping is controlled using the NANOS [143] runtime environment variable `NX_ARGS`, passing the appropriate values via arguments `--binding_stride`, `--binding_start` and `--smp_workers`. Specifically, the first argument governs how many hardware threads are to be skipped between the mapping of two consecutive OmpSs threads; the second identifies the starting point (first hardware thread) for a strided round-robin mapping; and the third argument specifies the total number of OmpSs threads. Thus, for example, by setting `--binding_stride`=1 we completely populate a core with 4 OmpSs threads before mapping threads to a new core. On the other extreme, `--binding_stride`=4 populates all cores with a single, two,... OmpSs thread(s)

```
// SpMV_DOT computes v_j := A * p_j and  alpha_j := sigma_j / (p_j^T * v_j)
// Coarse-grain task
#pragma omp task input (n, sigma, p[0:n-1]) output (v[0:n-1], alpha)
{
  SpMV_DOT(int *n, double *sigma,
           double p[], double v[], double *alpha) {
    // Initialization code ...
    for (id_task = 0; id_task < num_leaves_in_DAG; id_task++) {
      // Fine-grain (sub)task
      #pragma omp task
      {
        SpMV_DOT_LEAF(int *task_id, int *n, double *sigma,
                      double p[], double v[], double *alpha) {
          // Merged SpMV and DOT operating with leaf task_id ...
        }
      }
    }
    // Termination code ...
  }
}
```

Figure 4.17: Example of code illustrating the nested parallelism implemented in ILUPACK.

before assigning a second, third,... thread to them. Figure 4.18 illustrates these two examples of binding.

NUMA-aware execution on the AMD server

In order to attain high performance on the four-socket target AMD server, it is important to accommodate a NUMA-aware execution. This is achieved in our implementation with the NANOS environment variable `NX_ARGS` and the argument `--schedule=socket` combined with a careful modification of the ILUPACK code. Concretely, our code records in which socket each task was executed during the initial calculation of the preconditioner. This information is subsequently leveraged, during all iterations of the PCG solve, to enforce that tasks which operate on the same data that was generated/accessed during the preconditioner calculation are mapped to the same socket where they were originally executed. The fragment of code in Figure 4.19 illustrates how this is achieved in the merged code for O1+O2. This strategy ensures that, during the PCG iteration, a task is always executed on (any core of) the same socket that computed the corresponding task during the computation of the preconditioner (recorded into array `preconditioner_socket`) using NANOS routine `nanos_current_socket()`.

4.6.2 MPI implementations

For this particular work, we ported the task-parallel MPI implementation of ILUPACK described in [23] to the Intel Xeon Phi accelerator and the AMD server. The MPI implementation also exploits the task concurrency explicitly exposed by the preconditioner DAG during the calculation of the preconditioner and the subsequent PCG iteration, but employs MPI ranks (i.e., processes) instead of the threads leveraged in the OmpSs version. A second major difference is that, in the MPI implementation, the tasks are mapped to the MPI ranks *a priori*, that is, before the execution commences. The execution with the MPI implementation is thus the result of a static schedule (static mapping of tasks to MPI ranks) instead of a dynamic one as occurs with the OmpSs implementation.

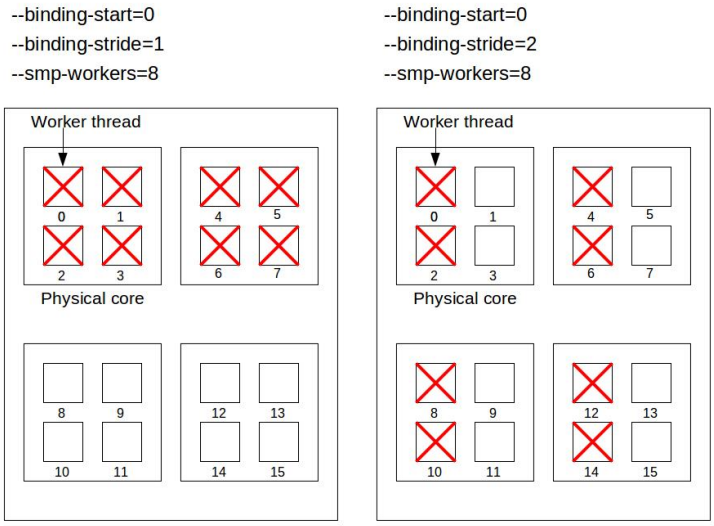


Figure 4.18: Examples of binding using different values of NANOS arguments.

```
for (id_task = 0; id_task < num_leaves_in_DAG; id_task++) {
  // Fine-grain (sub)task
  socket = preconditioner_socket[task_id];
  nanos_current_socket(socket);
  #pragma omp task
  {
    SpMV_DOT_LEAF(int *task_id, int *n, double *sigma,
                  double p[], double v[], double *alpha) {
      // Merged SpMV and DOT operating with leaf task_id ...
    }
  }
}
```

Figure 4.19: Example of code implementing the NUMA-aware execution in ILUPACK.

To distribute the MPI ranks among the processor cores of the Intel Xeon Phi, we include the options:

```
-genv I\_MPI\_PIN\_MODE=lib \
-genv I\_MPI\_PIN\_PROCESSOR\_LIST=\$mapping
```

in the `mpirun` invocation, with a list of cores in `$mapping` specifying the binding of ranks to cores.

To reduce inter-process communication, the original MPI implementation already ensures that the same MPI rank executes the operations associated with the "same" tasks of the preconditioner calculation and the PCG iteration. Note that we had to modify the OmpSs version manually to enforce a similar behaviour at the socket level in the NUMA-aware implementation for the AMD server.

4.6.3 Experimental results

We next evaluate the performance of different task-parallel implementations of ILUPACK, based on MPI and OmpSs, on XEON PHI and OPTERON (see Section 4.3). For XEON PHI the compiler and MPI implementation are part of Intel `icc` 13.1.3 20130607 (Intel MPI Library for Linux* OS, Version 4.1 Update 1 Build 20130507); and for OPTERON the compiler is Intel `icc` 11.1 20100806 and the MPI implementation is OpenMPI 1.6. Moreover, the software employed in the experiments included ILUPACK (2.4); the Mercurium C/C++ compiler/Nanox (releases 1.99.6/0.9a for XEON PHI and 1.99.1/0.8a for OPTERON) with support for OmpSs; and Metis (4.0.01) and ParMetis (4.0.2) for the graph reorderings with the OmpSs and MPI implementations respectively. For each platform, we employ the largest Laplace problem size that fits into its main memory (see Table 4.1).

The experiments report the speed-up compared with the sequential implementation of ILUPACK, running on the corresponding platform (a single core of XEON PHI or OPTERON). Therefore, the results show the execution time of the parallel solver normalized with respect to that of the sequential version. In order to expose enough task concurrency, for the task-parallel cases we partition the matrix into DAGs with one leaf per worker (either an OmpSs thread or an MPI rank), while the sequential version "solves" a DAG/matrix with a single task. We emphasize that the semantics of the task-parallel version differ with the number of leaves (hereafter l) in the preconditioner DAG, and they are also different from the sequential semantics. However, we ensure that the solvers are comparable by stopping the iteration process when the same residual, of order `restol`, is attained.

On XEON PHI, the number of physical cores that are actually used in the task-parallel executions, denoted by c, depends on the number of workers w that are spawned, between 1 and 32, and how many workers are mapped per core, w_c=1, 2 or 4 (inverse of the binding-stride factor): $c = w/w_c$; see Table 4.3. On OPTERON, the number of cores is simply given by $c = w$, as one worker is mapped at most per core; see Table 4.3. On both platforms, $w = l$.

Table 4.4 reports the speed-ups attained by the OmpSs and MPI implementations of ILUPACK in XEON PHI for the benchmark A171 (see Table 4.1). (Similar results were obtained for the smaller test cases A126 and A159.) The data comprised there reveals the following trends along different dimensions:

- *Iso-workers and Iso-DAGs (same w or column of the table).* Fixing the number of workers while we increase the level of "saturation" of the cores (i.e., raise w_c) has a clear negative effect on the OmpSs implementation and a slightly smaller one on the MPI one, for both the preconditioner computation and the PCG solve.

#Workers	w,l=	1	2	4	8	16	32	64
XEON PHI	w_c=1	1	2	4	8	16	32	--
	w_c=2	1	1	2	4	8	16	--
	w_c=4	1	1	1	2	4	8	--
OPTERON		1	2	4	8	16	32	64

Table 4.3: Number of cores (c) for the experimental evaluation on XEON PHI and OPTERON. The cases with 64 workers were not evaluated on XEON PHI due to lack of enough memory for the MPI implementations.

		OmpSs					MPI				
#Workers	w,l=	2	4	8	16	32	2	4	8	16	32
Precond.	w_c=1	1.9	3.8	7.5	12.8	22.4	2.0	3.9	7.3	13.3	23.2
	w_c=2	1.4	2.9	5.5	9.8	16.9	1.4	2.9	5.7	10.6	18.3
	w_c=4	1.4	1.7	3.3	5.9	10.5	1.4	1.6	3.3	6.3	11.6
PCG solve	w_c=1	1.9	3.9	8.0	15.5	27.7	2.0	3.9	7.6	13.4	22.6
	w_c=2	1.5	3.1	6.2	11.9	18.0	1.6	3.1	5.9	10.9	19.4
	w_c=4	1.5	1.8	3.5	5.0	4.1*	1.6	1.5	2.8	5.1	11.4

Table 4.4: Speed-ups of the task-parallel OmpSs and MPI implementations of the preconditioner computation and PCG solve in XEON PHI for matrix A171.

- *Iso-saturation (same w_c or row of the table).* Keeping constant the saturation, while increasing the number of workers w, implies a growth also in the number of physical cores (hardware resources) and, as could be expected, an increase of performance (except for one case in the OmpSs implementation of the PCG solve, marked with the superscript "$\star$").
- *Iso-cores (same c, cell color or diagonal of the table).* Fixing the number of cores to solve a problem, as the number of workers w grows, involves a proportional increase of the level of saturation of the cores. In other words, we maintain a constant volume of cores, while we increase the amount of workers and, simultaneously, the saturation of these hardware resources. This has a positive effect on both the OmpSs and MPI implementations of the preconditioner computation as well as the MPI implementations of the PCG solve. When the saturation level is w_c=4, the OmpSs implementation of the PCG solve suffers from the increase to 32 workers.
- *Overall performance of OmpSs vs MPI.* In general, when the number of cores/workers is small, appear slight performance differences in favor of the MPI implementation for the preconditioner computation and the OmpSs implementation for the PCG solve. On the other hand, as these values increase, OmpSs becomes the overall winner for the PCG solve while both implementations offer close performance for the preconditioner computation.

Table 4.5 shows the speed-ups attained by the OmpSs and MPI implementations of ILUPACK in OPTERON. The number of parameters is now more reduced, which leads to a simpler analysis. First, the NUMA-aware OmpSs implementation of the PCG solve clearly outperforms its NUMA-oblivious counterpart, with the difference rapidly growing with the number of threads (and therefore cores). Also, as expected, increasing the number of workers yields higher speed-ups, as more resources are employed in the solution of the problem. We note here that this result demonstrates that

		OmpSs						MPI					
#Workers	w,l=	2	4	8	16	32	64	2	4	8	16	32	64
Precond.		2.0	4.0	6.3	10.5	14.5	22.9	2.0	3.9	7.3	12.8	19.5	23.9
PCG solve	NO	1.9	3.9	6.1	8.6	6.9	10.2	--	--	--	--	--	--
	NA	2.2	5.1	8.2	13.4	14.7	27.2	2.3	4.1	8.2	13.1	21.3	17.2

Table 4.5: Speed-ups of the task-parallel OmpSs and MPI implementations of the preconditioner computation and PCG solve in OPTERON for matrix A318. NO and NA denote respectively the NUMA-oblivious and NUMA-aware implementations of the PCG solve.

the computational overhead intrinsic to partitioning the matrix into a DAG consisting of more tasks is compensated by a superior level of task concurrency to be exploited by a larger number of workers/cores. Finally, the OmpSs and MPI implementations deliver similar performance in the preconditioner computation when the number of cores is large, but OmpSs attains a much higher speed-up for the PCG solve when 64 workers are employed.

Comparison of the solvers

We finally evaluate the numerical behaviour of the task-parallel solvers, using a common matrix case (A171). For this purpose, we leverage the A-norm defined in [104], with the estimator in [178], as a measure of the numerical accuracy of the approximate solution x_j computed at the j-th iteration of the PCG solve: $\|x - x_j\|_A$, where x stands for the correct solution to the linear system. For the task-parallel solvers, we use 32 workers/cores of XEON PHI and 64 on OPTERON.

Figure 4.20 relates the estimated residual of the solutions computed by the parallel solvers to the execution time. These results show that the numerical behaviour of the task-parallel implementations based on OmpSs and MPI is almost identical, which could be expected as they operate on DAGs with the same number of leaves. The small differences are due to the use of different versions of the Metis graph partitioning package to decompose the problem into tasks. The time difference between XEON PHI and OPTERON is explained by the use of 64 cores at 2.1 GHz in the latter vs only half of that number of cores, at about half the frequency as well (1,053 MHz), in the former.

4.7 Concluding Remarks

In this chapter we have analyzed the parallelization of the PCG method implemented as part of the ILUPACK software for the solution of large-scale sparse linear systems. A careful reorganization of the coefficient matrix of the system, using e.g. a graph partitioning tool like Metis or Scotch, exposes enough coarse-grain task parallelism for todays' multi-threaded processors. The approach extracts task parallelism by splitting the sparse matrix into multiple levels, yielding a directed acyclic graph, with the form of a binary tree, where the nodes represent tasks, the arrows indicate data dependencies, and most computational work is performed in the leaf tasks. This graph is then traversed from bottom-up for the computation of the preconditioner and one of the triangular solves during its application, and top-down for the second triangular solve. Simultaneously, this partitioning can be leveraged to automatically implement different versions of this solver for shared-memory and distributed-memory architectures. A key advantage of the approach designed in this

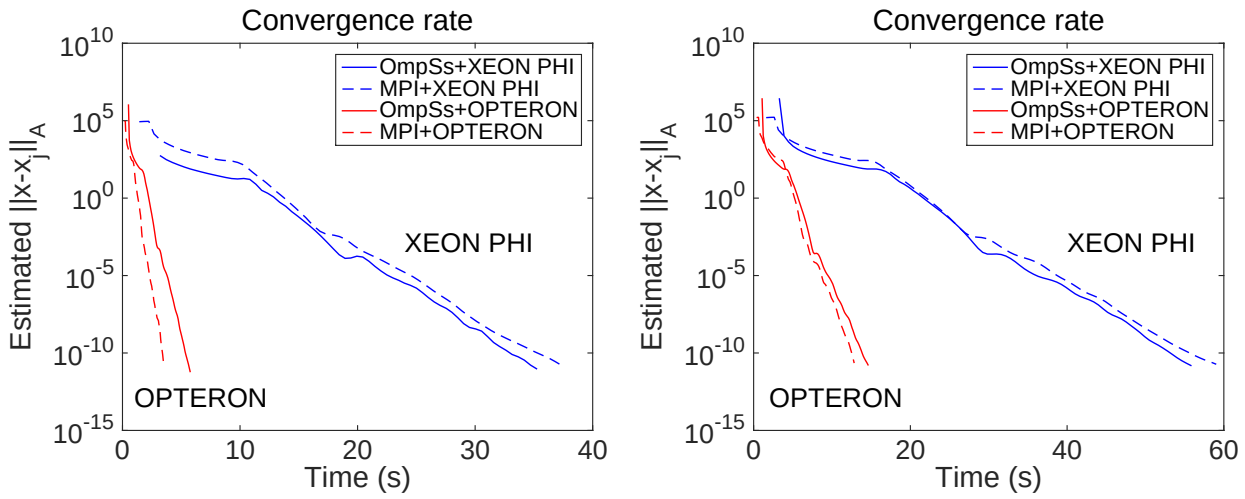


Figure 4.20: Convergence speed of the task- and data-parallel solvers for matrices A126 (left) and A171 (right).

dissertation is that the parallelization scheme can be easily applied to other ILU-type iterative solvers.

For shared-memory we employ the OmpSs programming model to facilitate the use of a "skeleton" structure that captures the tree-like dependencies of the subsequent preconditioner computation and triangular solves involved in the preconditioner application. This strategy requires minor changes to the legacy code in ILUPACK, while allowing to pass the dependency information to OmpSs which then orchestrates a concurrent execution of the complete PCG method, including the sparse matrix-vector product, vector (BLAS-1) operations and, especially, the computations involving the preconditioner. The results on two platforms equipped with state-of-the-art multicore processors, using a large-scale linear system, report notable performance for the parallel solvers. One relevant property of the new parallel data-flow solver is that the numerical method/software is decoupled from the runtime.

In addition, we have presented a new parallel implementation of ILUPACK on clusters of multicore processors, combining MPI and OmpSs models. The task parallelism in this version is extracted in the same manner, and the tree can be expanded into further levels to expose any number of tasks and, therefore, degree of concurrency. However, doing so yields different preconditioners and, from a certain depth, produces a significant overhead. In general, the best compromise is to generate up to two leaves per core, to allow the OmpSs scheduler optimize the computation. The experimental results confirm this assert for configurations with a reduced number of nodes, where the overhead is compensated by the OmpSs optimization. For unstructured matrices, the OmpSs runtime system accelerates the computation in most scenarios, due to the irregularity of the node sizes. The best solution combining MPI and OmpSs corresponds to a configuration that maps one MPI rank and eight OmpSs threads per socket, mimicking the internal architecture of the cluster nodes. With these parameters, the new MPI+OmpSs version of ILUPACK outperforms the initial implementation for clusters, which was based on MPI and could only process one leaf per rank.

Finally, we have presented two parallel implementations of ILUPACK, based on the OmpSs runtime and the MPI message-passing library to exploit task parallelism on x86 manycore architectures. Compared with our previous work using this library, our OmpSs-based implementations employ nested parallelism to tackle task dependencies (instead of explicit barriers) at execution

time, and introduce an architecture-aware implementation of the PCG solve for NUMA systems. Our experimental results on these manycore platforms (an Intel Xeon Phi accelerator and a 64-core AMD NUMA server) reveal that there exists ample task concurrency in the preconditioned solver embedded into ILUPACK, showing notable speed-ups in these architectures. The direct comparison between our parallel implementations also exposes that, while they all can achieve similar residuals in the computed solution, from the point of view of performance, the best option is to employ the MPI or OmpSs versions on the AMD server.

CHAPTER 5

Characterization of Processor Architectures with ILUPACK PCG

In the introduction to this manuscript, we motivated the ubiquity of sparse linear systems in scientific computing in general, and numerical simulations as well as data analytics applications in particular. In addition, the solution of sparse linear systems via iterative methods has been recently argued to be representative of the actual performance that is experienced by a large fraction of the scientific and engineering codes running on current supercomputers. This has led to the introduction of the High Performance Conjugate Gradient (HPCG) benchmark[1] as a complement and, eventually, potential replacement for the traditional LINPACK benchmark that ranks modern supercomputers twice per year in the Top500 and Green500 lists [7, 5]. Compared with LINPACK, the HPCG benchmark is designed to exercise computational and data access patterns that match a broad set of important applications more closely. Looking under the cover, the HPCG benchmark is nothing but the CG method in disguise, enhanced with a simple local symmetric Gauss-Seidel smoother (preconditioner).

In this chapter we investigate the efficiency of state-of-the-art multicore processors using our multi-threaded implementations of the CG method accelerated with the task-parallel version of ILUPACK described in Chapter 4. Concretely, we analyze multicore architectures with very distinct designs and market targets to compare their parallel performance and energy efficiency using our own benchmark. After this brief introduction, the rest of the chapter is structured as follows. In Section 5.1 we present the architectures included in the study. In Sections 5.2 and 5.3 we elaborate the study in detail for two of the multicore processors. Finally, in Section 5.4 we present some general observations that summarize the insights gained from this analysis.

5.1 Target Multicore Architectures

For the study, we selected three different types of multicore architectures comprising two general-purpose processors from Intel, two low-power systems from ARM, and the Intel Xeon Phi. This collection is representative of todays' multicore technology (except for GPUs, which are out-of-scope for this dissertation).

[1] `http://www.hpcg-benchmark.org/`

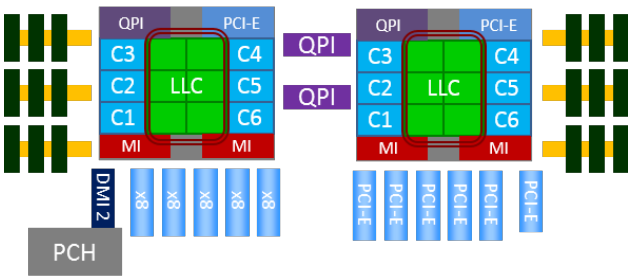


Figure 5.1: SANDY architecture. The original image is extracted from [3].

5.1.1 Intel Xeon E5-2620 (SANDY)

This server is equipped with two Intel Xeon E5-2620 (6-core) processors (from Q1'2012) and 32 Gbytes of DDR3 RAM, comprising a total of 12 cores distributed in two sockets (see Figure 5.1). The memory hierarchy is organized into 32+32 Kbytes of private L1 cache (data+instructions) per core, 2 Mbytes of private L2 cache per core, and 15 Mbytes of L3 cache shared by all cores in the same socket. The nominal frequency of the cores can be varied between 1.2 GHz and 2.0 GHz (2.5 GHz in Turbo mode), but all the cores in the same socket must operate at the same frequency.

In the SANDY platform the energy measurements were obtained by directly reading the RAPL registers from the code, as shown in Figure 2.10. RAPL defines several power planes to obtain real and estimated energy consumption values for distinct components. For SANDY, these include the cores (real consumption) and memory (modelled consumption), among others.

5.1.2 ARMv7 Cortex-A15 (A15)

The ODROID-XU3 board is furnished with a Samsung Exynos 5422 system-on-chip (SoC). This processor comprises an ARM Cortex-A15 quad-core cluster plus an ARM Cortex-A7 quad-core cluster, both implementing the ARMv7a microarchitecture. Each Cortex core has its own private 32-Kbyte L1 (data) cache. The four ARM Cortex-A15 cores share a 2-Mbyte L2 cache and the four ARM Cortex-A7 cores share a smaller 512-Kbyte L2 cache. In addition, the two clusters access a 2-Gbyte DDR3 RAM (see Figure 5.2). The frequency can be varied in the range 200 MHz–1.4 GHz for the Cortex-A7 cluster and 200 MHz–2.0 GHz for the Cortex-A15 cluster, with a 100 MHz-step in both cases. However, in order to reduce the number of experiments, we will perform our experiments with frequencies separated by 200 MHz. This then fixes the corresponding (supply) voltage as shown in the corresponding columns of Table 5.1. Note that the voltage remains constant in the frequency ranges [200,600] MHz for the Cortex-A15 [85]. All cores in the same cluster must operate at the same frequency.

For the ODROID board, the PMLIB monitoring tool [25] collects power consumption corresponding to instantaneous power readings from four independent sensors/power domains in the board (Cortex-A7 cluster, Cortex-A15 cluster, DRAM and GPU), with a sampling rate of 250 ms. When evaluating the energy of one of the clusters, we only consider the sensor corresponding to that component. Given that we do not employ the GPU in our experiments, and that the four Cortex-A15 cores and up to three Cortex-A7 cores can be disabled when idle, we can expect a negligible power consumption for these components when inactive [85]. Specifically, in the experiments in this chapter we only employ the four Cortex-A15 cores.

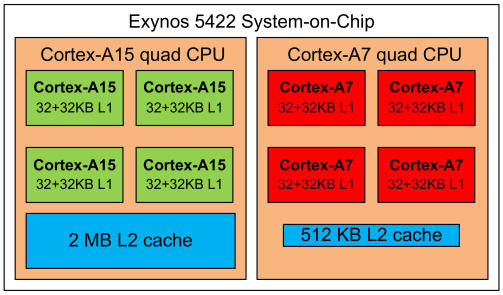


Figure 5.2: ODROID-XU3 architecture.

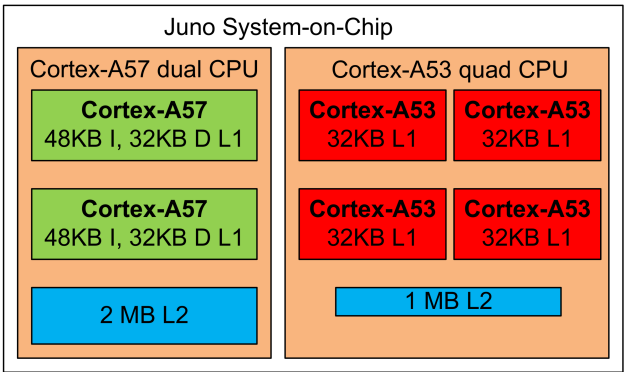


Figure 5.3: Juno architecture.

5.1.3 ARM Cortex-A57 (A57)

The JUNO development platform features an ARM Cortex-A57 dual-core cluster plus an ARM Cortex-A53 quad-core cluster, both implementing the ARMv8 microarchitecture. Each core has its own private 32-Kbyte L1 (data) cache. The two ARM Cortex-A57 cores share a 2-Mbyte L2 cache and the four ARM Cortex-A53 cores share a smaller 1-Mbyte L2 cache. Both clusters are connected to a DDR3 RAM with a capacity of 8 Gbytes (see Figure 5.3). The frequency and voltage can be varied as displayed in the corresponding columns of Table 5.1. All cores in the same cluster must operate at the same frequency.

In the JUNO board PMLIB collects power consumption data corresponding to instantaneous power readings using a data acquisition device from National Instruments connected to the internal shunt resistors available in the board (Cortex-A53 cluster, Cortex-A57 cluster, system, and GPU), with a sampling frequency of 100 Hz. When evaluating the energy of one of the clusters, we only consider the line corresponding to that component, for the same reasons exposed for ODROID. In this platform, we only use the Cortex-A57 for the experiments.

5.1.4 Intel Xeon E5-2603v3 (HASWELL)

This server is equipped with two Intel Xeon E5-2603 (6-core) processors (from Q3'2014) and 32 Gbytes of DDR3 RAM, comprising a total of 12 cores distributed in two sockets (see Figure 5.4). The memory hierarchy is organized into (192+192) Kbytes of private L1 cache (data+instructions) per core, 2 Mbytes of private L2 cache per core, and 15 Mbytes of L3 cache shared by all cores in the same socket. The nominal frequency of the cores can be varied between 1.2 GHz and 1.6 GHz.

In the HASWELL platform the energy measurements were obtained by directly reading the RAPL registers from the code, as in SANDY.

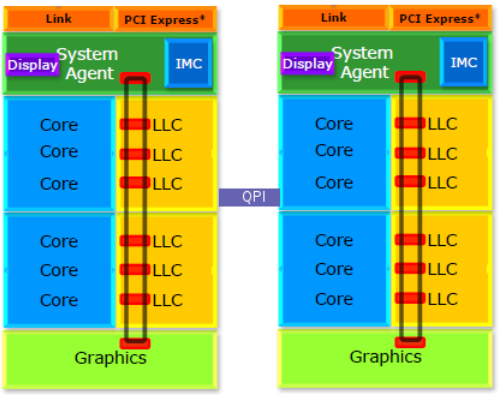


Figure 5.4: HASWELL architecture. The original image is extracted from [1].

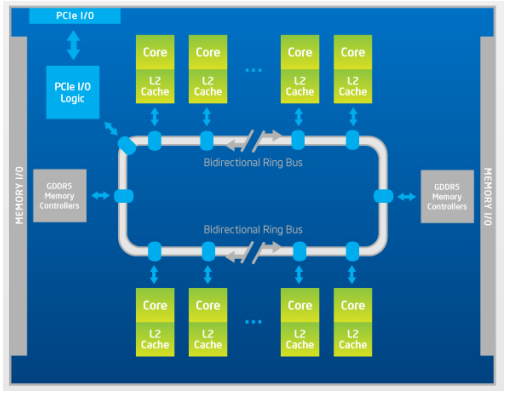


Figure 5.5: XEON PHI architecture. The original image is extracted from [4].

5.1.5 Intel Xeon Phi (XEON PHI)

This board is equipped with an Intel Xeon Phi 5110P co-processor (the tests on this board were ran in native mode and, therefore, the specifications of the server are irrelevant.) The accelerator comprises 60 x86 cores running at 1.053 GHz and 8 Gbytes of GDDR5 RAM (see Figure 5.5). In the co-processor we cannot change the frequency of the cores. The top of the memory hierarchy is formed by the coprocessor cores and the vector registers. Each core supports 4 execution contexts or threads and each thread has its own 32 vector registers. To facilitate low-latency, the L1 cache is directly integrated into the core. The L1 data and instruction caches are the second level in the coprocessor's memory hierarchy. The L1 cache holds 32 Kbytes of data and has a 3-cycle access time. The next level is the coprocessor core's local L2 cache, and the L2 caches of all the cores interconnected via the ring interconnect. Each core's local L2 cache has the capacity of 512 Kbytes, and with the processor interconnect, the total available is 30 Mbytes.

In the XEON PHI board PMLIB collects power consumption data corresponding to instantaneous power readings using the MIC module implemented reading the `libmicmgmt` library, as shown in Figure 2.13. The sampling frequency is 100 Hz.

5.1.6 General setup

Table 5.2 offers some further information about hardware in each platform. The software configurations employed in the platforms are described in Table 5.3. Moreover, to facilitate an easier understanding of the energy/power analyses in the next sections, Table 5.4 provides the power consumed when all threads are idle for each architecture and range of frequencies.

	SANDY	ODROID (A15)		JUNO (A57)		HASWELL	XEON PHI
Conf.	Freq.	Freq.	Voltage	Freq.	Voltage	Freq.	Freq.
C_1	1.200	0.200	0.912	0.450	0.810	1.200	1.053 GHz
C_2	1.300	0.400	0.912	0.625	0.850	1.300	–
C_3	1.400	0.600	0.912	0.800	0.900	1.400	–
C_4	1.500	0.800	0.925	0.950	0.950	1.500	–
C_5	1.600	1.000	0.973	1.100	1.000	1.600	–
C_6	1.700	1.200	1.023	–	–	–	–
C_7	1.800	1.400	1.062	–	–	–	–
C_8	1.900	1.600	1.115	–	–	–	–
C_9	2.000	1.800	1.191	–	–	–	–
C_{10}	–	2.000	1.318	–	–	–	–

Table 5.1: VFS configurations (voltage-frequency pairs, in V and GHz, respectively) available in the platforms.

Architecture	SANDY	ODROID(A15)	JUNO(A57)	HASWELL	XEON PHI
PROCESOR NUMBER	E5-2620	ARMv7 rev 3 (v7l)	AArch64 rev 0	E5-2603 V3	5110P
#SOCKETS	2	1	1	2	1
#CORES	12	4	2	12	60
BASE FREQUENCY	2.0 GHz	2.0 GHz	1.1 GHZ	1.6 GHz	1.053
CACHE	15 MB	2 MB	2 MB	15 MB	30 MB
TDP	95 W	15 W	30 W	85 W	225 W
VOLTAGE RANGE	0.60 V-1.35 V	0.91 V-1.32 V	0.81 V-1.00 V	0.65 V-1.30 V	–
MEMORY	32 GB	2 GB	8 GB	32 GB	8 GB
MAX. MEMORY BANDWIDTH	42.6 GB/s	14.9 GB/s	13.2 GB/s	51 GB/s	320 GB/s

Table 5.2: Hardware specifications of the platforms.

All the experiments in next sections employed IEEE754 real double-precision arithmetic and the Laplacian matrices described in Section 4.3. The problem size is the largest that fits in the main memory of each platform.

5.2 Characterization of SANDY using ILUPACK PCG

We will employ the general-purpose architecture SANDY as a workhorse to guide through the study performed for each architecture included in this chapter. The complete evaluation comprises three "dimensions", two of them architectural (number of cores/threads and operation frequency) and one corresponding to the software (number of leaves for ILUPACK's preconditioner task dependency tree). Due to the large number of tests and results that were obtained for some architectures, we organize the presentation of the performance analysis of the architecture into the following sequence of steps:

1. Evaluation of the parallel ILUPACK PCG solver for a range of representative frequencies and number of threads (thread-level parallelism), using 1, 2, 4,..., 64 leaves.

2. Selection of the optimal number of leaves for each level of thread-parallelism.

3. Evaluation of the impact of frequency on the ILUPACK PCG solver.

Architecture	SANDY	ODROID (A15)	JUNO (A57)	HASWELL	XEON PHI
GCC	4.4.6	4.8.2	4.9.1	4.4.7	5.1.0
OMPSS	16.06	16.06.1	16.06.1	16.06.1	16.06
MERCURIUM	2.0.0	2.0.0	2.0.0	2.0.0	2.0.0
NANOX	0.12a	0.10.1	0.10.1	0.10.1	0.12a
METIS	5.0.2	5.0.2	5.0.2	5.0.2	5.0.2
POWER/ENERGY MEASUREMENTS	RAPL	PMLIB	PMLIB	RAPL	PMLIB
FREQUENCY CHANGES	CPUfreq	CPUfreq	CPUfreq	CPUfreq	–

Table 5.3: Software specifications of the platforms.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
SANDY	80.92	81.64	82.14	82.73	83.29	84.04	84.63	85.31	86.00	–
ODROID	0.08	0.10	0.13	0.17	0.20	0.25	0.31	0.42	0.52	0.74
JUNO	0.052	0.061	0.062	0.074	0.075	–	–	–	–	–
HASWELL	27.04	27.09	27.13	27.31	27.42	–	–	–	–	–
XEON PHI	98.25	–	–	–	–	–	–	–	–	–

Table 5.4: Idle power (W) on the different platforms for the range of available frequency configurations (described in Table 5.1).

4. Selection of the optimal frequency for each number of threads.

5. Evaluation of the impact of the thread-level parallelism on the ILUPACK PCG solver.

This analysis is then repeated from the perspective of energy efficiency, taking as a basis the performance evaluation to justify some of the results for this second metric.

5.2.1 Performance

Number of leaves Let us commence the performance evaluation of SANDY. Following the sequencing of steps aforementioned, Table 5.5 reports the execution time for all possible combinations of tests determined by the three evaluation dimensions, using the matrix case A318 (the largest problem that fits into the 32 Gbytes available in SANDY). A careful inspection of these results reveals that a TDG consisting of 32 leaves in general offers the best performance, independently of the number of threads (rows marked as optimal, in green tick). This is an interesting insight because it provides experimental evidence that the cost overhead incurred when increasing the number of leaves in the TDG is compensated by the superior parallelism that is exposed in that case. Therefore, we will use 32 leaves in the remainder of our performance evaluation of this architecture.

Frequency To continue the performance evaluation, Figure 5.6 shows the evolution of the execution time when we tune the processor frequency for the different number of threads. To facilitate the visualization of the results in the graph, for this experiment the execution time is normalized with respect to that obtained using the lowest frequency for each number of threads. As could be expected, all cost lines show that the time decreases as the frequency is raised, because higher frequency intuitively translates into faster executions, but the decrement of the execution time is more visible when the number of threads is small. In any case, as a general rule, the best option to maximize performance in SANDY is to set the architecture to operate at the highest nominal

Threads	Leaves	Frequency (GHz) 2.0	1.9	1.8	1.7	1.6	1.5	1.4	1.3	1.2	Optimal
1	1	2,101.97	2,172.92	2,247.31	2,399.66	2,500.093	2,616.54	2,748.52	2,904.07	3089.21	
1	2	2,258.14	2,334.60	2,416.46	2,579.21	2,690.58	2,817.22	2,960.86	3,131.67	3,334.14	
1	4	2,206.33	2,282.08	2,361.39	2,521.12	2,630.13	2,754.85	2,899.63	3,064.04	3,264.76	
1	8	2,059.72	2,134.89	2,214.53	2,362.19	2,469.52	2,592.35	2,732.59	2,896.50	3,093.48	
1	16	2,032.03	2,111.61	2,195.80	2,335.55	2,448.015	2,576.99	2,723.51	2,895.42	3,099.69	
1	32	1,874.28	1,956.64	2,045.67	2,171.66	2,281.49	2,413.99	2,565.14	2,748.27	2,948.78	✓
1	64	1,984.17	2,071.40	2,167.50	2,295.86	2,418.43	2,560.25	2,720.38	2,908.628	3,129.69	
2	2	1,187.16	1,226.14	1,266.71	1,347.14	1,402.86	1,465.96	1,538.28	1,622.12	1,727.68	
2	4	1,165.28	1,202.96	1,244.12	1,324.94	1,382.15	1,444.90	1,517.23	1,602.96	1,706.14	
2	8	1,120.19	1,157.68	1,200.03	1,274.26	1,329.30	1,392.17	1,463.57	1,547.69	1,647.65	
2	16	1,107.12	1,145.65	1,188.75	1,261.74	1,318.63	1,383.81	1,459.23	1,546.18	1,653.14	
2	32	982.27	1,024.26	1,072.63	1,136.34	1,198.03	1,260.39	1,340.92	1,432.75	1,534.23	✓
2	64	1,047.51	1,091.60	1,146.10	1,212.33	1,274.25	1,345.67	1,422.87	1,521.74	1,631.68	
4	4	684.14	703.03	723.20	761.23	789.39	821.27	858.42	902.08	956.58	
4	8	648.01	668.08	690.99	725.57	752.98	786.75	823.42	868.47	920.28	
4	16	663.73	682.94	704.83	742.80	770.16	802.86	839.25	883.72	940.01	
4	32	545.73	569.35	591.83	628.01	657.54	688.49	727.68	780.62	837.15	✓
4	64	599.63	613.37	646.00	675.77	711.52	749.25	787.32	836.40	895.87	
6	8	691.88	710.49	730.62	766.18	794.15	825.38	862.83	906.51	959.29	
6	16	574.98	592.59	608.78	636.01	659.23	685.20	714.13	749.82	793.46	
6	32	438.45	457.28	476.77	502.71	529.13	549.61	578.23	626.58	659.70	✓
6	64	1631.68	499.14	514.20	535.51	551.40	584.92	611.30	644.98	691.00	
8	8	429.91	436.52	450.47	467.32	482.03	499.61	517.58	539.99	570.55	
8	16	409.90	428.64	439.63	453.47	475.24	492.00	513.11	536.69	567.34	
8	32	315.08	327.98	340.23	354.52	376.23	383.55	412.28	433.78	461.25	✓
8	64	337.87	347.93	362.25	379.06	395.51	415.95	441.87	465.72	498.70	
10	16	356.22	367.72	377.67	393.99	412.71	426.33	443.69	465.63	494.31	
10	32	283.21	296.94	301.42	309.30	321.47	339.48	354.63	373.10	398.70	✓
10	64	285.26	291.70	299.51	311.34	325.44	334.62	352.692	376.47	403.20	
12	16	306.34	312.72	322.95	336.53	348.70	364.45	383.77	407.10	436.56	
12	32	242.98	249.83	260.88	266.60	273.20	286.65	297.50	310.92	326.34	✓
12	64	253.90	255.22	267.46	272.832	281.20	291.99	297.32	313.72	337.47	

Table 5.5: Execution time (s) of the PCG solver on SANDY using different number of threads and leaves for the range of available frequencies.

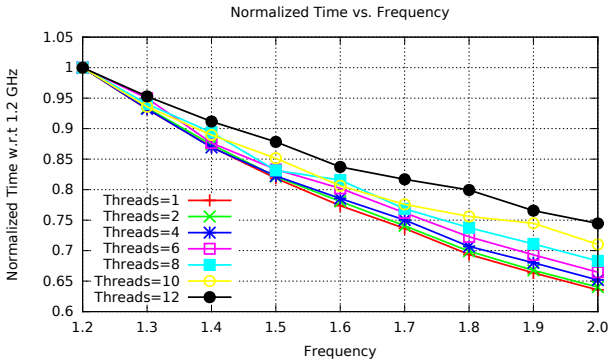


Figure 5.6: Performance vs. frequency of the PCG solver in SANDY using a TDG with 32 leaves. The execution time is normalized with respect to that obtained with the lowest frequency for each number of threads.

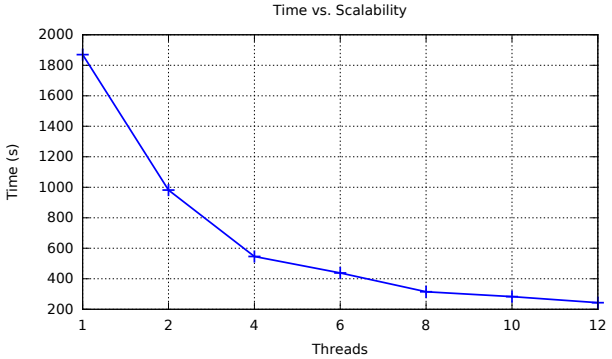


Figure 5.7: Performance vs. scalability of the PCG solver in SANDY using a TDG with 32 leaves. The experiments were run at the maximum frequency (2.0 GHz) for each number of threads.

frequency, corresponding to 2.0 GHz. This is the reference value that we will use in our third experiment with this platform. (Here we note that the highest possible frequency for this architecture is 2.5 GHz, when operating in Turbo mode. However, this frequency can only be maintained for a short period and/or a reduced number of active cores.)

Thread-level parallelism The final step of the performance study assesses the thread-level scalability of SANDY using the ILUPACK PCG solver. For this, Figure 5.7 shows the execution time as the number of threads increases from 1 to 12. The speed-up is fair till 8 threads, but stagnates as more threads from the second socket are added. This final result will be re-visited again during the assessment of the energy consumption, to justify some of the observations for that metric.

5.2.2 Energy consumption

Number of leaves We open the study concerning the energy efficiency of SANDY, using the ILUPACK PCG solver, by first reporting the consumption for all the combinations of the three dimensions: number of leaves, frequency and thread-parallelism. Table 5.6 indicates that the best number of leaves, among the five tested values (1, 2, 4,. . . , 64), corresponds to 32. This matches the optimal number of leaves from the point of view of the execution time and is not totally unexpected. The reason is that the energy consumption depends on the execution time so that, improving the latter

Threads	Leaves	Frequency (GHz) 2.0	1.9	1.8	1.7	1.6	1.5	1.4	1.3	1.2	Optimal
1	1	202.57	206.18	208.18	222.17	225.47	230.71	239.25	249.71	265.62	
1	2	216.31	219.94	224.29	237.50	242.52	249.11	256.85	269.77	281.36	
1	4	212.62	212.45	219.14	231.85	238.47	248.31	253.20	261.97	278.41	
1	8	197.10	201.08	205.16	216.95	222.48	233.31	241.02	252.27	263.55	
1	16	195.74	200.05	200.46	214.89	220.40	232.20	240.58	251.22	262.83	
1	32	180.73	185.26	188.42	200.95	205.40	217.09	226.21	239.60	251.47	✓
1	64	190.99	197.28	200.96	211.16	217.73	227.53	233.81	251.20	265.92	
2	2	122.59	124.78	126.03	131.61	134.99	138.77	144.33	149.15	156.29	
2	4	119.96	120.69	123.46	129.07	132.70	136.23	138.415	146.88	152.80	
2	8	116.03	117.71	119.1	124.19	128.25	132.05	134.24	141.73	148.29	
2	16	113.19	115.83	117.30	123.71	126.36	130.44	135.52	141.67	148.95	
2	32	102.11	103.82	107.42	111.17	115.26	119.25	124.83	131.27	137.62	✓
2	64	108.41	111.00	113.56	119.59	122.80	128.07	132.46	140.32	148.01	
4	4	76.75	78.41	78.54	81.20	82.18	84.35	85.64	89.07	92.84	
4	8	73.86	73.98	75.67	76.42	78.86	79.68	81.98	85.51	89.21	
4	16	75.89	75.89	76.68	79.22	80.05	82.40	84.23	87.19	90.35	
4	32	63.23	64.21	65.56	67.59	69.16	71.24	73.21	76.97	81.46	✓
4	64	68.92	69.58	71.51	72.97	73.25	77.32	79.67	82.50	86.03	
6	8	82.88	83.57	82.40	85.13	85.66	87.03	89.68	92.11	95.46	
6	16	69.71	70.44	70.35	70.28	72.87	74.23	75.05	76.98	79.90	
6	32	55.14	55.59	56.93	57.94	59.52	60.39	62.54	65.48	67.68	✓
6	64	59.89	60.70	60.35	62.19	62.20	64.75	64.82	67.82	71.23	
8	8	55.09	55.58	55.21	56.56	56.26	57.08	57.69	59.00	61.01	
8	16	54.32	54.67	54.88	54.30	56.51	56.60	57.31	58.58	60.60	
8	32	42.60	43.41	43.56	44.42	45.81	45.32	47.60	48.45	50.24	✓
8	64	46.07	45.80	46.55	47.19	48.30	48.03	50.89	51.76	54.34	
10	16	49.37	49.70	49.79	50.37	51.06	51.88	52.17	53.38	54.97	
10	32	40.71	41.84	41.29	41.69	42.27	42.97	43.68	43.84	46.21	✓
10	64	41.10	41.38	41.63	42.33	42.29	43.12	43.97	45.08	46.32	
12	16	44.81	43.64	44.23	44.40	45.39	45.95	46.95	48.22	49.22	
12	32	36.48	36.83	37.55	37.67	37.73	38.44	38.91	39.43	40.68	✓
12	64	38.64	37.86	39.29	39.07	39.62	39.58	39.59	40.65	42.19	

Table 5.6: Energy (KJ) consumed during the execution of the PCG solver in SANDY using different number of threads and leaves for the range of available frequencies.

metric (provided the power dissipation remains constant), translates into a linear decrease of the former. Following the sequence pattern for the evaluation, we thus fix the number of leaves in the TDG to 32 for the discussion of the remaining two dimensions with SANDY.

Frequency Figure 5.8 illustrates the fluctuations in the energy consumption when the processor frequency is varied for each number of threads. In this experiment, the energy is normalized with respect to that obtained with the lowest frequency for each number of threads. The graph reveals a decrease in the energy consumption as the frequency is raised. The same trend already appeared for the execution time, with the difference being more visible for the executions that employ a small number of threads.

In order to explain this result, we remind that the energy efficiency is the product between power dissipation and execution time. Thus, we also need to analyse the behaviour of the power with respect to the variation of frequency in SANDY. Figure 5.9 displays the variation of this metric while executing the ILUPACK PCG solver in the range of frequencies. (The power rate is normalized with respect to that obtained with the lowest frequency.) As we could expect, the power increases with the frequency, but it grows at a lower pace than the reduction experienced by the execution time, motivating the results obtained for the energy consumption. A definite source of the superior energy efficiency of an execution that proceeds at higher frequencies on SANDY is the considerable

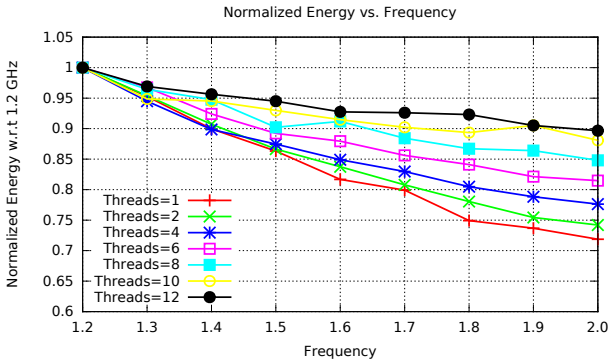


Figure 5.8: Energy consumption vs. frequency of the PCG solver in SANDY using a TDG with 32 leaves. The energy is normalized with respect to that obtained with the lowest frequency for each number of threads.

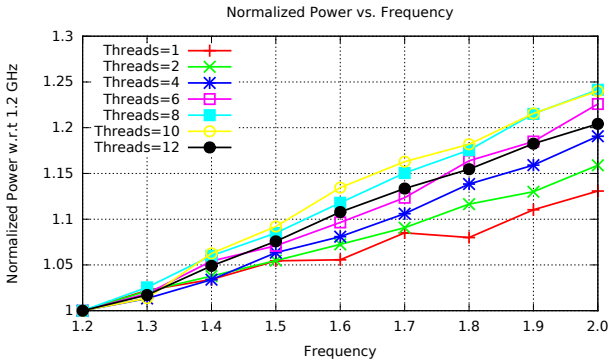


Figure 5.9: Power dissipation vs. frequency of the PCG solver in SANDY using a TDG with 32 leaves. The power is normalized with respect to that obtained with the lowest frequency for each number of threads.

idle power for this processor (see Table 5.4). In other words, it is preferrable to execute the solver at the highest possible frequency to finish early and avoid the dissipation of power.

Thread-level parallelism From the two previous experiments concerning energy consumption, we already selected the best number of leaves and optimal frequency. Once the parameters for these two dimensions are fixed, we analyze the variation of the energy efficiency as a function of the thread-level parallelism. Figure 5.10 reveals a considerable decrease in the energy consumption when we increase the number of threads up to 8, but an stagnation from that point. A rough inspection of the outcome from the analogous experiment from the point of view of performance in Figure 5.6 explains this behaviour.

5.3 Characterization of A15 using ILUPACK PCG

We next repeat the efficiency analysis for the ARM Cortex-A15 multicore processor embedded in the ODROID development board. Compared with the high-performance but power-hungry Intel Xeon socket, the A15 chip is designed for low power scenarios, yielding significantly different performance and energy consumption behaviours, as we will expose. A second major difference between the Intel Xeon server and the ODROID is the quantity of RAM available in each platform. For the latter, the

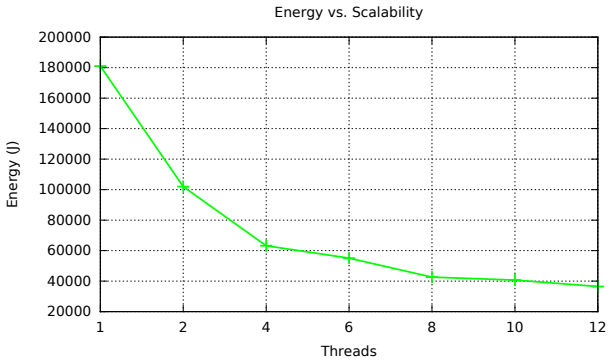


Figure 5.10: Energy consumption vs. scalability of the PCG solver in SANDY using a TDG with 32 leaves. The experiments were run at the optimal frequency (2.0 GHz) for each number of threads.

reduced amount of this resource constrains the experiments to operate with the problem instance A100. This has an impact on the degree of task-parallelism that can be exposed via nested dissection in the form of leaves in ILUPACK's TDG.

5.3.1 Performance

Number of leaves Table 5.7 reports the execution time of ILUPACK PCG for the diverse configurations resulting from the combination of the three evaluation dimensions: number of leaves per core, frequency, and number of threads. The results clearly identify a TDG with 8 leaves as the best option for the parallel executions in this platform and, therefore, we select this value as the reference for the subsequent discussions related to performance. As argued in the opening paragraph of this section, the reduced dimension of the problem that fits into the RAM of the ODROID development board can only accommodate the data for the small A100 instance. In turn, this limits the degree of concurrency of the problem, rapidly increasing the overhead of a TDG with a larger number of leaves, and justifying the superior performance of the case with only 8 leaves.

Frequency Figure 5.11 illustrates the behaviour of the execution time when we vary the frequency of the A15 cores for the different number of threads. As done earlier for this type of plot, the time is normalized with respect to that obtained by the execution at the lowest frequency. The plot reveals a considerable decrease of time from 0.2 GHz to 1.0 GHz. In contrast, for the interval comprised between 1.0 GHz and 2.0 GHz, the reduction is almost negligible. This stagnation of the execution time for the higher frequencies is due to the saturation of the memory bandwidth in this architecture, and it is often visible for memory-bound computations. Nonetheless, although the differences are small, the best option to maximize performance is to execute the code at 2.0 GHz. Therefore, we select this frequency for the last experiment concerning performance.

Thread-level parallelism To complete the performance evaluation, Figure 5.12 assesses the scalability in ODROID using the ILUPACK PCG solver with the complete socket operating at the maximum frequency. Here, we can identify a fair speed-up when the number of threads is increased up to 4 (the maximum in the A15) despite the limited dimension of the problem.

Threads	Leaves	Frequency (GHz) 2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	Optimal
1	1	90.43	97.74	106.08	117.32	131.67	152.98	184.83	238.24	346.28	692.40	✓
1	2	96.61	104.32	113.83	126.30	142.14	165.48	200.80	259.37	378.54	752.20	
1	4	93.93	101.83	111.82	124.12	140.61	163.95	199.82	259.64	380.67	760.42	
1	8	91.71	99.68	108.87	121.59	137.98	161.15	196.80	256.51	377.20	751.36	
1	16	101.62	113.93	121.71	132.96	149.79	174.84	212.60	277.48	407.18	803.71	
1	32	99.95	109.89	120.76	135.40	154.38	182.28	224.34	294.62	435.61	877.99	
1	64	117.92	128.41	140.93	157.88	180.34	212.97	262.22	344.67	511.56	1,028.46	
2	2	53.53	57.29	61.77	67.89	75.67	87.28	104.83	134.71	195.46	385.71	
2	4	52.16	56.00	60.52	66.78	74.81	86.64	104.40	134.70	196.08	388.39	
2	8	51.01	54.64	59.37	65.61	73.17	84.64	102.94	133.08	194.68	383.73	✓
2	16	57.8	61.15	65.29	71.63	79.12	91.52	110.36	143.36	208.50	409.17	
2	32	55.02	58.92	64.14	71.30	80.84	94.51	115.22	150.12	221.77	444.06	
2	64	63.51	68.56	74.27	82.46	93.31	109.47	133.73	175.10	260.13	520.05	
4	4	32.23	33.99	36.08	39.13	42.69	48.51	57.80	73.29	105.23	203.66	
4	8	32.06	33.66	35.85	38.85	41.86	47.75	56.78	71.93	103.68	200.12	✓
4	16	36.23	37.91	39.9	41.71	45.84	52.03	61.56	78.07	108.95	214.44	
4	32	34.28	36.37	38.51	42.18	46.14	52.97	62.96	80.96	117.12	233.62	
4	64	40.67	42.77	45.15	48.95	53.68	61.55	73.69	94.30	137.98	272.45	

Table 5.7: Execution time (s) of the PCG solver on ODROID using different number of threads and leaves for the range of available frequencies.

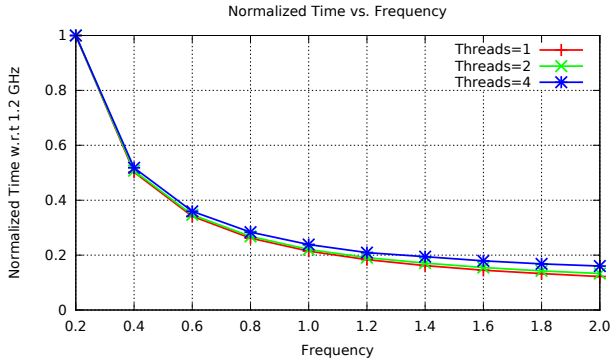


Figure 5.11: Performance vs. frequency of the PCG solver on ODROID using a TDG with 8 leaves. The execution time is normalized with respect to that obtained with the lowest frequency for each number of threads.

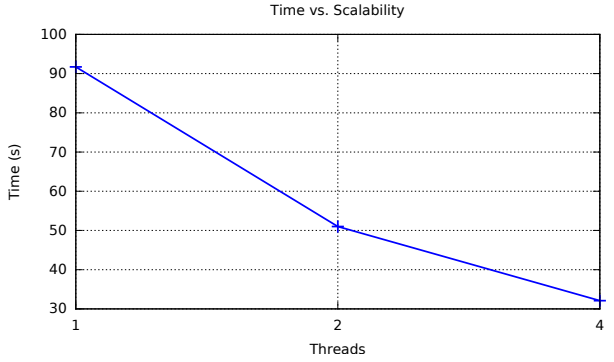


Figure 5.12: Performance vs. scalability of the PCG solver in ODROID using a TDG with 8 leaves. The experiments were run at the optimal frequency (2.0 GHz) for each number of threads.

Threads	Leaves	Frequency (GHz) 2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	Optimal
1	1	223.69	168.21	146.42	123.76	112.99	101.63	92.77	96.69	100.89	174.60	✓
1	2	242.52	181.83	157.52	133.62	122.41	111.20	100.11	103.50	110.79	157.23	
1	4	239.61	180.51	157.03	133.52	125.93	110.57	99.86	103.57	110.18	158.42	
1	8	233.04	183.39	150.48	131.90	124.67	107.91	99.55	104.17	110.47	155.52	
1	16	261.81	203.27	164.77	141.68	132.09	117.28	106.00	112.39	116.73	137.33	
1	32	256.99	194.53	165.85	148.23	134.07	124.18	113.75	116.71	126.98	151.89	
1	64	295.27	226.97	195.25	172.90	156.98	142.42	129.96	137.75	148.42	176.08	
2	2	217.92	165.53	138.67	119.76	107.79	95.55	86.21	86.04	90.68	114.29	
2	4	213.76	162.18	137.16	118.20	106.69	95.33	85.43	86.04	90.25	113.94	
2	8	209.57	158.06	134.11	116.02	104.78	93.95	84.26	84.86	90.07	113.83	✓
2	16	233.55	174.79	148.56	125.83	114.73	100.93	91.56	91.13	104.22	108.98	
2	32	229.87	176.62	149.88	128.14	119.21	106.99	96.58	97.74	103.58	118.60	
2	64	272.35	206.97	173.45	148.17	137.48	124.31	111.05	113.45	119.47	137.66	
4	4	228.48	165.98	137.08	117.10	104.41	92.42	81.26	80.89	81.89	88.08	
4	8	227.23	164.27	136.27	116.00	102.65	91.35	80.10	79.66	80.81	85.91	✓
4	16	267.83	187.81	148.54	124.94	112.23	99.80	86.78	85.58	84.55	91.51	
4	32	261.43	185.44	149.04	129.34	115.02	102.14	90.05	89.88	91.93	102.22	
4	64	308.86	217.13	173.80	151.13	133.45	118.25	105.40	105.16	107.55	117.34	

Table 5.8: Energy (J) consumed during the execution of the PCG solver in ODROID using different number of threads and leaves for the range of available frequencies.

5.3.2 Energy consumption

We complete the study of the A15 platform with the analysis of the energy efficiency of this platform via the task-parallel solver underlying ILUPACK PCG. Before we commence this evaluation, we note that the static power dissipation rate for ODROID is very small. Consequently, we can expect a different behaviour from that observed for SANDY.

Number of leaves Table 5.8 exposes that the most energy-efficient executions are again obtained using a TDG composed of 8 leaves for the parallel configurations, matching the results observed in the performance study. As stated in that case, the reason for this behaviour is the reduced size of the problem instance employed in the experiments. Concretely, the additional concurrency explicitly exposed by further splitting the computational load/TDG does not compensate the overhead that is introduced for such (small) problem dimension.

Frequency Figure 5.13 shows the relation in ODROID between energy consumption and frequency for the different number of threads. This architecture clearly exhibits a different behaviour compared with SANDY as the A15 saves energy when operating at lower frequencies. As argued, one of the reasons for this result is the negligible static power in ODROID. An additional cause is that the higher frequencies collapse the memory bandwidth, as shown Figure 5.11, yielding no increase in the performance for a (useless) higher power usage. This combination renders a frequency between 0.6 GHz and 0.8 GHz as the best option for the execution of ILUPACK PCG.

This is illustrated in more detail by Figure 5.14, which shows the evolution of the power dissipation as the frequency is modified. The power curves there expose that the increase of power is linearly proportional to that in frequency. This behaviour, together with that of time (see Figure 5.11), explains the superior energy efficiency at lower frequencies. Concretely, the reason is that the difference of time between distinct frequencies is minor compared with the gap in power. An orthogonal observation for this plot is that the power dissipation of ODROID is much lower than the power rate observed for SANDY.

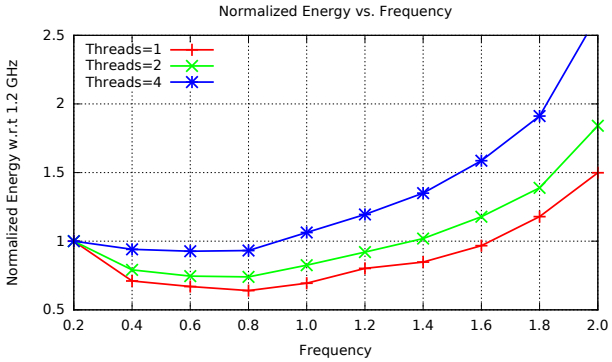


Figure 5.13: Energy consumption vs. frequency of the PCG solver in ODROID using a TDG with 8 leaves. The energy is normalized with respect to that obtained with the lowest frequency for each number of threads.

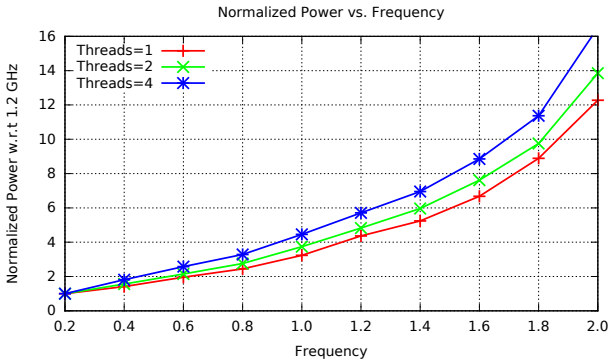


Figure 5.14: Power dissipation vs. frequency of the PCG solver in ODROID using a TDG with 8 leaves. The power is normalized with respect to that obtained with the lowest frequency for each number of threads.

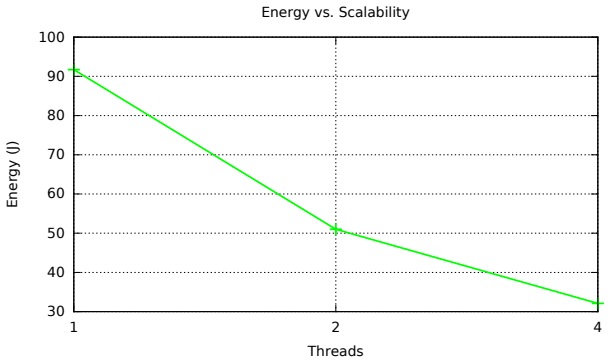


Figure 5.15: Energy consumption vs. scalability of the PCG solver in ODROID using a TDG with 8 leaves. The experiments were run at the optimal frequency (0.8 GHz) for each number of threads.

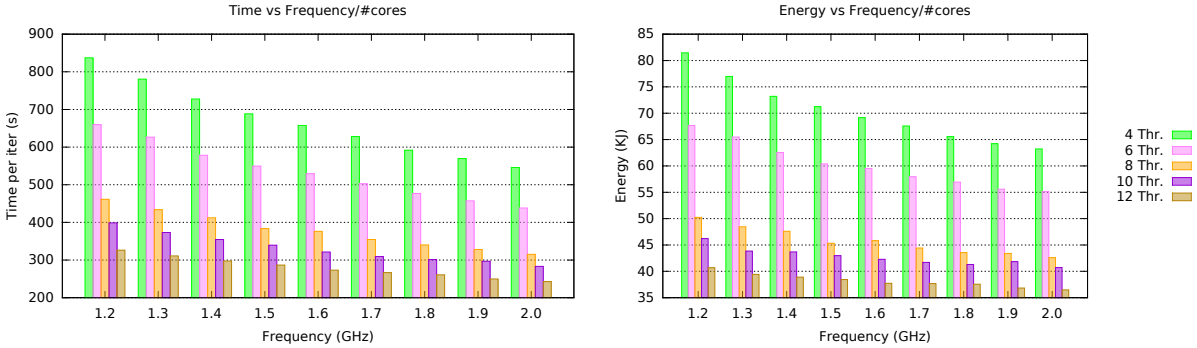


Figure 5.16: Time and energy consumption for the execution of ILUPACK PCG in SANDY.

Thread-level parallelism To close the study of this architecture, Figure 5.15 displays the scalability versus the energy efficiency in ODROID, when executing the PCG solver at 0.8 GHz, which is the optimal frequency from the perspective of energy. Here we can appreciate an important decrease in energy consumption when the number of threads raises from 1 to 4, illustrating the benefits of a parallel execution.

5.4 General Observations

As part of the dissertation, we performed a comprehensive evaluation of all the architectures described in the first section of this chapter similar to those presented in the previous two sections. In order to avoid an exhaustive list of figures and results, we next summarize them into a few plots that illustrate the interplay between frequency/thread concurrency and performance/energy consumption. In particular, Figures 5.16–5.20 report absolute values for the last two metrics against processor frequency and number of threads. Note that these are the two hardware dimensions that were targeted earlier this chapter. The third one (number of leaves per thread), depends on the software, and is set for all these experiments to the optimal. To allow an easier visualization of the differences, for those architectures with a large number of cores, we skip the results obtained with 1 and 2 cores as, in any case, they always offered worse performance and energy efficiency than other configurations with a higher level of thread concurrency.

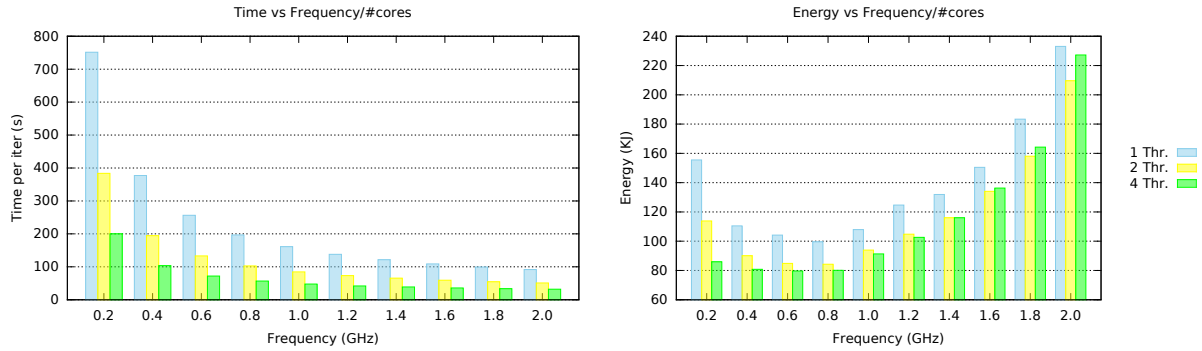


Figure 5.17: Time and energy consumption for the execution of ILUPACK PCG in ODROID.

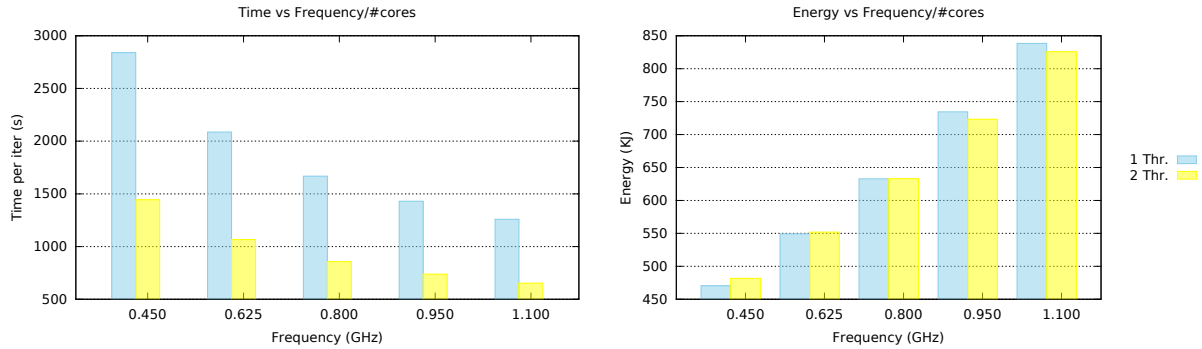


Figure 5.18: Time and energy consumption for the execution of ILUPACK PCG in JUNO.

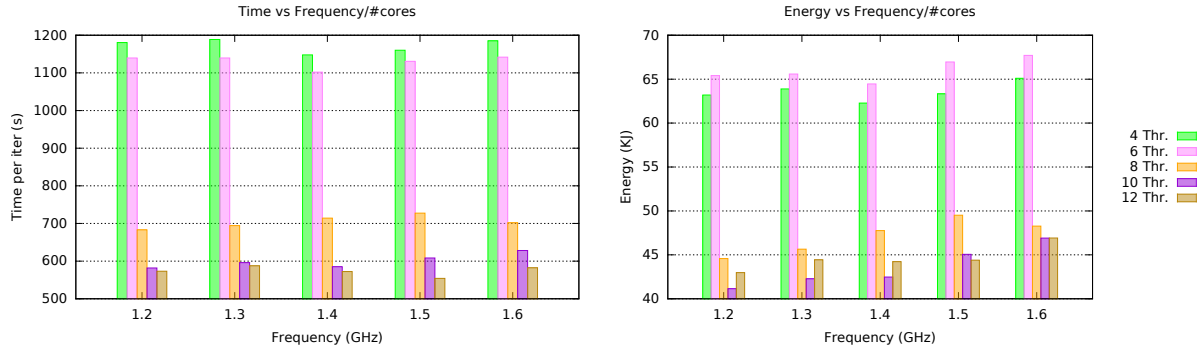


Figure 5.19: Time and energy consumption for the execution of ILUPACK PCG in HASWELL.

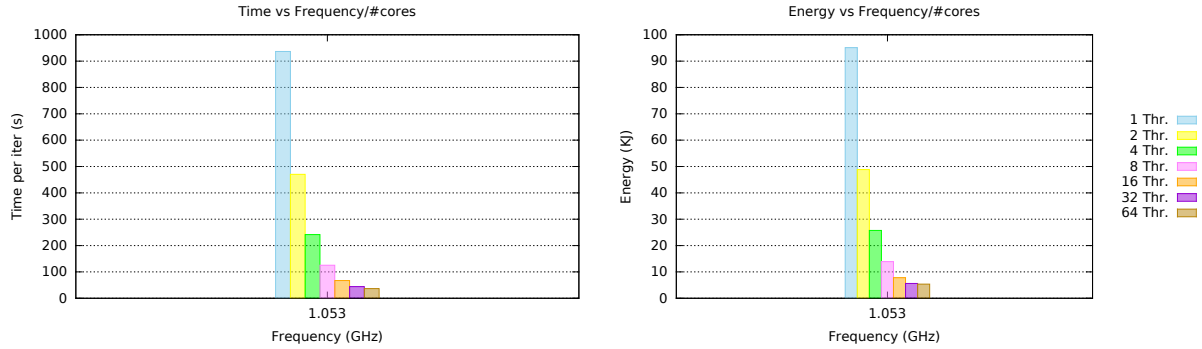


Figure 5.20: Time and energy consumption for the execution of ILUPACK PCG in XEON PHI.

A collection of general remarks can be extracted from this experimental evaluation that emphasize the differences between the performance-oriented architectures (Intel Xeon) and the low-power processors (ARM):

Point of view of performance:

1. The optimal number of leaves is mostly determined by the problem size: a larger dimension allows additional levels of task-parallelism being exposed via nested dissection without incurring into a costly overhead. In contrast, the number of leaves is basically independent of the architeture class (performance-oriented versus low-power), frequency, and number of threads. For the small and large problem instances (e.g., A100 versus A318), the optimal number of leaves is, respectively, 8 and 32.
2. The execution time in general benefits from operating at a higher frequency and/or using a larger number of cores. However, the differences may be small when the memory bandwidth is saturated as the results for the low-power architecture demonstrate.

Point of view of energy consumption:

1. The optimal numbers of leaves match those obtained when the figure-of-merit is performance. The same remarks specified for that case apply when the target metric is energy consumption.
2. The optimal frequency is the highest one for the performance-oriented architectures but, in general, a reduced level for the low-power processors. The reason for this behaviour is twofold, and can be used to further distinguish the behaviour of the two types of systems:
   (a) Performance-oriented architectures exhibit a considerable static power rate so that increasing the execution time is very costly. Contrarily, low-power processor do not suffer from this drawback.
   (b) Low-power processors tend to saturate the memory bandwidth rapidly as the frequency is raised, yielding a negligible improvement of execution time for a linear increase in the power dissipation rate. The consequence is a worse energy efficiency.
3. From the perspective of scalability, adding more cores is beneficial unless the memory bandwidth is saturated. Once that threshold is surpassed, the increase in the dissipation rate directly translates into higher energy costs.

CHAPTER 6

Conclusions

6.1 Concluding Remarks and Main Contributions

The main goal of the dissertation was *the design, implementation, and evaluation of parallel and energy-efficient iterative sparse linear system solvers for multicore processors, taking advantage of the specific hardware features present in recent manycore architectures and accelerators such as the Intel Xeon Phi.*

This objective was motivated by the importance and cost of the solution of sparse linear systems in key numerical simulations and big-data processing algorithms. In order to tackle this scenario, in this dissertation we considered the solution of large sparse systems of linear equations using preconditioned iterative methods based on Krylov subspaces. Concretely, we pursued the optimization of the solvers supported by ILUPACK for their efficient execution on multicore processors and manycore architectures. Due to the importance of energy consumption, as our HPC supercomputers progress on the road to Exascale systems, we also addressed the energy-efficient dimension in the implementation of ILUPACK solvers for sparse linear systems.

At the conclusion of this work, the main contributions of this dissertation are the following:

- The development of an automatic power-performance analysis framework to identify the power bottlenecks during the execution of concurrent applications by comparing the performance and C-state traces.
- The improvement of ILUPACK PCG method by leveraging task parallelism with OmpSs, minimizing the changes to the legacy code. The parallelization scheme can be applied to easily parallelize other ILU-type iterative solvers.
- The exploration of the interoperability between the message-passing MPI programming interface and the OmpSs task-parallel programming model, that resulted in a hybrid version of the PCG in ILUPACK for clusters of multicore processors.
- The elaboration of specialized implementations of the preconditioned iterative linear system solver in ILUPACK for NUMA platforms and the Intel Xeon Phi.
- The characterization of the energy efficiency of distinct processor designs using the parallel implementations of ILUPACK PCG.

The following sections describe these contributions and summarize the corresponding conclusions in further detail.

6.1.1 Automatic power-performance analysis framework

This dissertation developed and extended an integrated framework for power-performance analysis of parallel scientific workloads. The framework offers useful information on power and performance for different kinds of parallel applications, from MPI codes that operate on moderate-scale clusters, to multi-threaded applications that exploit the benefits of multicore+GPU platforms. In addition, we also developed complementary modules to obtain fine-grain energy measurements leveraging the power sensors and models integrated in recent architectures. Specifically, the power-performance analysis framework obtains power/energy samples from RAPL sensors (MSR registers), GPU devices (NVML library) and Xeon Phi co-processors (MIC management library).

We also analyzed the use of power estimates offered by recent processor technology versus the exploitation of a professional data acquisition system from National Instruments (NI). From this study we concluded that the measurements captured from RAPL may produce an overhead due to the fact that recording the power information can only be done from the same platform. This issue could be mildly tackled by avoiding the use of the PMLib framework and reading the RAPL sensors directly from the code. However, with these direct readings we cannot obtain power traces to analyze the power-behaviour together with the performance.

Additionally, we introduced and evaluated a key complement to the PMLib framework that consists in a powerful inspection tool that automatically identifies power sinks. The detection of power-wasting events is based on a comparison between the application performance trace and the C-state traces per core, and it is done in parallel. Moreover, the analyzer is flexible because allows the user to choose the task types that correspond to "useful" work, and adjust parameters such as the length of the analysis interval or the discrepancy threshold. This tool also offers statistical information which can help to obtain an approximation of the energy-cost due to hotspots, and the potential savings that a more power-friendly implementation can potentially yield.

As a result from this study, we provided a valuable framework to develop more energy-efficient applications. This tool helps the programmer by identifying the power bottlenecks and the impact of the power sinks. In particular, we used the framework to implement energy-aware HPC linear algebra libraries, leveraging idle periods during the execution using dynamic frequency-voltage scaling and avoiding busy-waits.

6.1.2 Task-parallel PCG method in ILUPACK

A contribution of this dissertation consisted in the extraction of task-level concurrency in the PCG method implemented in ILUPACK for the solution of large-scale sparse linear systems. Exploiting the connection between sparse matrices and adjacency graphs, nested dissection can be recursively applied to permute a sparse matrix, yielding a collection of diagonal blocks that are linked to certain subgraphs and separators. This process produces a DAG defining the dependencies between the diagonal blocks, where the subgraphs occupy the leaves and the separators correspond to the internal nodes. In this work we used Metis to apply nested dissection, and modified the parallel version of this software to generate not only the partition but also the DAG. Moreover, we adjusted the original library to obtain a DAG with more leaves than processor cores (which was a restriction of the original version of Metis).

The DAG determines the task parallelism during the computation with the preconditioner as the nodes represent tasks and the arrows indicate data dependencies. This graph is traversed from

bottom-up and top-down during the computation of the preconditioner and its application. The remaining operations on the PCG only involve the computation of the leaves, so that they can be executed in parallel. This partitioning was used in the dissertation to implement tuned versions of the PCG solver for shared-memory and distributed-memory architectures. A key advantage of the designed task-parallel scheme is that it can be easily applied to other ILU-type iterative solvers.

6.1.3 ILUPACK for multicore

Our investigation exploited the task parallelism in ILUPACK PCG to design, develop, and evaluate a new implementation of the solver for shared-memory processors using OmpSs. In this version we take advantage of the task parallelism to create a "skeleton" structure that captures the dependencies in the DAG corresponding to the most challenging operations in the method. This information is passed to the OmpSs runtime which can then enforce a correct and efficient schedule of the entire solver. This strategy offers two relevant advantages: the first is that it requires minor changes to the legacy code for ILUPACK, and the second is that the numerical method/software is decoupled from the runtime.

During the experimental evaluation we obtained performance traces and detected some execution features that could be improved. Particularly, we assigned priorities to the OmpSs tasks in order to prioritize the execution of the costlier tasks (usually the leaves of the tree). Moreover, in order to reduce the overhead due to the creation of a large amount of fine-grain tasks, we decided to merge some of them, obtaining notable benefits.

6.1.4 Hybrid ILUPACK for clusters

A hybrid version of the PCG underlying ILUPACK, combining MPI and OmpSs, was provided as a key contribution of this dissertation. This implementation explores the interoperability between both programming models to unleash an efficient execution of the solver in clusters of multicore processors. This approach leverages the task-parallel scheme to statistically map the tasks in the top levels of the binary tree to the cluster nodes, fixing the inter-node communication pattern. In addition, this mapping forces the partitioning of the tasks inside each node, completing the remaining levels of the tree. The subtree corresponding to each node is then processed concurrently via the OmpSs runtime system.

The degree of concurrency of the problem can be expanded by adding levels in the tree but, after a careful experimental evaluation, we confirmed that, from a certain depth, this approach incurs a relevant overhead. Therefore, we validated that the best compromise is to generate up to two leaves per core, to allow the OmpSs scheduler to optimize the computation.

The experimental evaluation of the hybrid implementation of ILUPACK combining MPI+OmpSs analyzed different combinations of the number of MPI ranks and OmpSs threads per node. This study revealed that the best configuration matches the internal architecture of the cluster nodes. Moreover, this analysis investigated the strong and weak scalability of the parallel solver for all the configurations. A general conclusion from this study is that the new MPI+OmpSs version, with the optimal configuration, outperforms a previous implementation for clusters, which was based on MPI and could only process one leaf per MPI rank.

6.1.5 Tuning ILUPACK on manycore architectures

In order to efficiently execute the solver in NUMA platforms and the Intel Xeon Phi, we presented two specialized implementations of ILUPACK PCG. Concretely, to develop these approaches, we introduced several improvements to tune the OmpSs version elaborated in this thesis

and an existing MPI-based implementation. The optimizations included the exploitation of nested parallelism, the correct mapping of threads to cores, and the accommodation of a NUMA-aware execution.

The experimental analysis of these implementations was carried out in an Intel Xeon Phi accelerator with 60 cores and in a 64-core NUMA server from AMD. As a conclusion from this evaluation, the experiments revealed that there exists ample task concurrency in the preconditioned solver embedded into ILUPACK, showing notable speed-ups in both architectures. Furthermore, the direct comparison between the parallel implementations exposed that they achieve similar residuals in the computed solution for both architectures. However, if we compare the performance, the best results are obtained for the AMD server.

6.1.6 Characterizing the efficiency of multicore and manycore processors

A side contribution of this dissertation was the analysis of the computational performance and energy efficiency of servers equipped with the state-of-the-art general-purpose multicore processors as well as accelerators like the Intel Xeon Phi. Following the recent introduction of the HPCG benchmark as a reference for evaluating the performance of supercomputers, we adopted ILUPACK to test performance and energy efficiency of multicore platforms. We note that HPCG and ILUPACK share the same computational and data access patterns, being representative of the performance that can be attained by todays' supercomputers. In contrast, HPCG is a generic benchmark, equipped with a serial and very simple preconditioner, compared with our task-parallel version of ILUPACK PCG.

6.2 Related Publications

The scientific contributions of this thesis have been validated with several peer-reviewed publications in international conferences and journals. Each one of these contributions is supported by, at least, one publication. The following subsections list the main publications derived from the thesis. We divide them into papers directly related to the thesis' topics and papers indirectly related to them but with a certain connection to linear algebra operations and energy efficiency. For the first group of publications, we provide a brief abstract of the main contents of the contribution.

6.2.1 Directly-related publications

Chapter 2. Automatic Power-Performance Analysis Framework

The first version of the PMLib framework for power-performance analysis was introduced in [25]. In this paper we presented a framework that measures the power consumption of commercial external AC meters and an internal DC wattmeter. Combined with the instrumentation and visualization tools `Extrae` and `Paraver`, it produces a useful environment to identify sources of power inefficiency. In [36] we extended this framework with support for new measurement devices, with the most relevant being a commercial DAS from NI. We also defined the interface of PMLib and introduced two modules to record information on processor states related to power consumption (C-states and P-states). In [38, 39] we demonstrated the use of the framework, analyzing different dense linear algebra algorithm implementations from the performance and power consumption points of view. Finally, the key contribution of this chapter, the automatic tool to detect power bottlenecks in parallel scientific applications, was presented in [37].

The following is a detailed list of the main publications related to this topic:

Alonso, P., Badia, R., Labarta, J., Barreda, M., Dolz, M., Mayo, R., Quintana-Ortí, E., and Reyes, R. Tools for power-energy modeling and analysis of parallel scientific applications. In *41st International Conference on Parallel Processing (ICPP)* (2012), pp. 420–429. Conference Proceedings [25]

> Understanding power usage in parallel workloads is crucial to develop the energy-aware software that will run in future Exascale systems. Workloads contribute towards this goal by introducing an integrated framework to profile, monitor and analyze power dissipation in parallel MPI and multi-threaded scientific applications. The framework includes an own-designed device to measure internal DC power consumption and a package offering a simple interface to interact with this design as well as commercial wattmeters. Combined with the instrumentation package `Extrae` and the graphical analysis tool `Paraver`, the result is a useful environment to identify sources of power inefficiency directly in the source application code. For task-parallel codes, we also offer a statistical software module that inspects the execution trace of the application to calculate the parameters of an accurate model for the global energy consumption, which can be then decomposed into the average power usage per task or the nodal power dissipated per core.

Barrachina, S., Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. S. An integrated framework for power-performance analysis of parallel scientific workloads. In *3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)* (2013), 114–119. Conference Proceedings [36]

> The path towards Exascale systems will require to energetically address power consumption of future high performance computing (HPC) workloads which, in turn, urges for a better understanding of power usage. We present an evolved framework to trace and analyze the power and energy consumption made by parallel scientific applications. The framework includes *i*) a flexible and extensible design that enables easy integration of different types of power measurement devices and addition of new functionality; *ii*) a new module that records information on processor states related to power consumption; and *iii*) an improved power measurement device to monitor internal direct current (DC) power consumption. This environment is thus revealed as a powerful yet easy-to-use tool to investigate and progress on the development of energy-efficient HPC applications.

Barreda, M., Dolz, M. F., Mayo, R., Quintana-Ortí, E. S., and Reyes, R. Binding performance and power of dense linear algebra operations. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2012), pp. 63–70. Conference Proceedings [39]

> We combine a powerful tracing framework with a power measurement setup to perform a visual analysis of the computational performance and the power consumption of tuned implementations for three key dense linear algebra operations: the LU factorization, the Cholesky factorization, and the reduction to tridiagonal form. Our results using 6 and 12 cores of an AMD Opteron-based platform reveal the serial/concurrent phases of the algorithms, and their connection to periods of low/high power consumption, as well as the linear dependency between execution time and energy for this class of operations.

Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Tracing the power and energy consumption of the QR factorization on multicore processors. In *12th* Conference Proceedings [38]

International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE) (2012), pp. 134–142.

> We analyze the interaction between computational performance, power dissipation and energy consumption of several high-performance implementations of the QR factorization, a crucial matrix operation for the solution of linear systems of equations and linear least squares problems. Our experimental results on a multiprocessor platform equipped with recent multicore technology from AMD show the interaction between these three factors.

Journal [37] Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Automatic detection of power bottlenecks in parallel scientific applications. In *Computer Science - Research and Development* (2013), pp. 1–9.

> In this paper we present an extension of the PMLib framework for power-performance analysis that permits a rapid and automatic detection of power sinks during the execution of concurrent scientific workloads. The extension is shaped in the form of a multi-threaded Python module that offers high reliability and flexibility, rendering an overall inspection process that introduces low overhead. Additionally, we investigate the advantages and drawbacks of the RAPL power model, introduced in the Intel Xeon "Sandy-Bridge" CPU, versus a data acquisition system from National Instruments.

Chapter 4. Exploiting Task-Parallelism in ILUPACK

The first implementation of the task-parallel PCG in ILUPACK resulting from this thesis was presented in [13]. In that paper we described how to extract the task parallelism in ILUPACK, and we introduced a parallel implementation for multicore processors with considerable levels of thread concurrency using OmpSs. Later, in [12] we tuned the previous OmpSs implementation and an existing MPI version for their efficient execution in NUMA platforms and the Intel Xeon Phi. In addition, in the second paper we also investigated the mapping of the threads to cores for both implementations on two manycore platforms. The interoperability between MPI and OmpSs was studied in [14], yielding a hybrid version of the PCG solver for clusters. This work motivated the analysis of the costs of the communications in the solver and the introduction of several communication-avoiding strategies in [20].

The following is a detailed list of the main publications related to this topic:

Conference Proceedings [13] Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., and Quintana-Ortí, E. S. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned CG method. *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)* (2014), pp. 262–269.

> In this paper we describe how to efficiently exploit task parallelism for the solution of sparse linear systems on multi-threaded processors via ILUPACK's multi-level preconditioned CG method. Using a pair of data structures, we capture the task dependencies that appear in the two most challenging operations in the method (calculation of the preconditioner and its application), passing this information to the OmpSs runtime which can then implement a correct and efficient schedule of the entire solver.
>
> Our results with high-end multicore platforms equipped with Intel and AMD processors report significant performance gains, demonstrating that OmpSs provides an efficient

and close-to-seamless means to leverage the concurrency in a complex scientific code like ILUPACK.

Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti P., and Quintana-Ortí, E. S. Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing* (2016), Vol.54, pp. 97–107. Journal [12]

We present specialized implementations of the preconditioned iterative linear system solver in ILUPACK for Non-Uniform Memory Access (NUMA) platforms and the Intel Xeon Phi and graphics accelerators. For the conventional x86 architectures, our approach exploits task parallelism via the OmpSs runtime as well as a message-passing implementation based on MPI, respectively yielding a dynamic and static schedule of the work to the cores, with different numeric semantics to those of the sequential ILUPACK. For the graphics processor we exploit data parallelism by off-loading the computationally expensive kernels to the accelerator while keeping the numeric semantics of the sequential case.

Aliaga, J. I., Barreda, M., Bollhöfer, M., and Quintana-Ortí, E. S. Exploiting task-parallelism in message-passing sparse linear system solvers using OmpSs. *22nd International European Conference on Parallel and Distributed Computing (EURO-PAR)* (2016), pp. 631–643. Conference Proceedings [14]

We introduce a parallel implementation of the preconditioned iterative solver for sparse linear systems underlying ILUPACK that explores the interoperability between the message-passing MPI parallel programming interface and the OmpSs task-parallel programming model. Our approach commences from the task dependency tree derived from a multi-level graph partitioning of the problem, and statically maps the tasks in the top levels of this tree to the cluster nodes, fixing the inter-node communication pattern. This mapping induces a conformal partitioning of the tasks in the remaining levels of the tree among the nodes, which are then processed concurrently via the OmpSs runtime system.

The experimental analysis on a cluster with high-end Intel Xeon processors explores several configurations of MPI ranks and OmpSs threads per process showing that, in general, the best option matches the internal architecture of the nodes. The results also report significant performance gains for the MPI+OmpSs version over the initial MPI code.

Aliaga, J. I., Barreda, M., Flegar, G., Bollhöfer, M., and Quintana-Ortí, E. S. Communication in task-parallel ILU-preconditioned CG solvers using MPI+OmpSs. *Concurrency and Computation: Practice and Experience* (2016). In revision. Journal [20]

We target the parallel solution of sparse linear systems via iterative Krylov subspace-based methods enhanced with ILU-type preconditioners on clusters of multicore processors. In order to tackle large-scale problems, we develop task-parallel implementations of the classical iteration for the CG method, accelerated via ILUPACK and ILU(0) preconditioners, using MPI+OmpSs. In addition, we integrate several communication-avoiding (CA) strategies into the codes, including the butterfly communication scheme and Eijkhout's formulation of the CG method. For all these implementations, we analyze the communication patterns and perform a comparative analysis of their performance and scalability on a cluster consisting of 16 nodes.

Chapter 5. Characterization of processor architectures with ILUPACK PCG

In [17] we employed a previous version of ILUPACK, which relied on an ad-hoc runtime based on OpenMP, to investigate the benefits that an energy-aware implementation of that runtime produced on the time-power-energy balance of the application. Furthermore, we proposed several simple yet accurate power models that captured the variations of average power that result from the introduction of the energy-aware strategies as well as the impact of the P-states into ILUPACK's runtime. Additionally, in [19] we analyzed the performance and energy efficiency of the OmpSs version of ILUPACK in different state-of-the-art general-purpose multicore processors and accelerators such as the Intel Xeon Phi. These analysis allowed us to characterize the efficiency of the platforms.

The following is a detailed list of the main publications related to this topic:

Journal [17] Aliaga, J. I., Barreda, M., Dolz, M. F., Martín, A. F., Mayo, R., and Quintana-Ortí, E. S. Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems. *Cluster Computing* (2013), Vol.17(4), pp. 1335-1348

> We investigate the benefits that an energy-aware implementation of the runtime in charge of the concurrent execution of ILUPACK —a sophisticated preconditioned iterative solver for sparse linear systems— produces on the time-power-energy balance of the application. Furthermore, to connect the experimental results with the theory, we propose several simple yet accurate power models that capture the variations of average power that result from the introduction of the energy-aware strategies as well as the impact of the P-states into ILUPACK's runtime, at high accuracy, on two distinct platforms based on multicore technology from AMD and Intel.

Journal [19] Aliaga, J. I., Barreda, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E. S. Characterizing the efficiency of multicore and manycore processors for the solution of sparse linear systems. *Computer Science - Research and Development* (2015), Vol.31(4), pp. 175–183.

> We analyze the efficiency of servers equipped with state-of-the-art general-purpose multicore processors as well as platforms based on accelerators such as graphics processing units (GPUs) and the Intel Xeon Phi. Following the proposal recently advocated in the High Performance Conjugate Gradient (HPCG) benchmark, we leverage for this purpose efficient implementations of ILUPACK, a preconditioned solver for sparse linear systems that comprises numerical kernels and data access patterns analogous to those of HPCG. Our study analyzes the (computational) performance and energy efficiency, with two different metrics for each: time/floating-point throughput for the former; and energy/floating-point throughput-per-Watt for the latter.

6.2.2 Indirectly-related publications

A parallel research was performed into the energy-efficiency field to explore the use of energy in some of the most relevant dense algebra routines and optimize them. In [18] we conducted a detailed analysis of the sources of power dissipation and energy consumption during the execution of current dense linear algebra kernels on multicore processors. In particular, by leveraging the RAPL, we decomposed the power-energy duo into its core, RAM, and uncore components, performing a series of illustrative experiments for a range of memory-bound to CPU-bound high performance kernels. Additionally, we investigated the energy proportionality of these three architecture components for the execution of linear algebra routines on a representative Intel Xeon socket.

In [15, 16] we analyzed the sources of power dissipation and energy consumption during the execution of high performance dense linear algebra (DLA) kernels on multicore processors. Moreover, we proposed and evaluated several strategies to adapt the concurrency throttling (CT) and the voltage-frequency setting (VFS) to obtain an energy-efficient execution of the DLA routine `dsytrd`.

Finally, in [162] we performed an experimental evaluation of the impact that voltage-frequency scaling and concurrency throttling exert on the energy consumption of the MPDATA algorithm, a key component of the multiscale fluid model EULAG.

The following is a detailed list of the main publications related to that topic:

Aliaga, J. I., Barreda, M., Dolz, M. F., and S. Quintana-Ortí, E. S. Are our dense linear algebra libraries energy-friendly? Time-Power-Energy Trade-Offs in BLAS and LAPACK. *Computer Science - Research and Development* (2015), Vol.30(2), pp. 187–196. — Journal [18]

Aliaga, J. I., Barreda M., Castaño A., Dolz M. F., and Quintana-Ortí, E. S. Strategies for adapting concurrency throttling and dynamic voltage-frequency scaling for dense eigensolvers. *15th International Conference on Computational and Mathematical Methods in Science and Engineering* (2015), Vol(1), pp. 76–80. — Conference Proceedings [16]

Aliaga, J. I., Barreda M., Castaño A., Dolz M. F., and Quintana-Ortí, E. S. Adapting concurrency throttling and voltage-frequency scaling for dense eigensolvers. *The Journal of Supercomputing* (2015), pp. 1–15. — Journal [15]

Rojek, K., Barreda, M., Quintana-Ortí, E. S., Wyrzykowski, R. Energy consumption of stencil-based MPDATA algorithms. *16th International Conference on Computational and Mathematical Methods in Science and Engineering* (2016), Vol(1), pp. 1104–1107. — Conference Proceedings [162]

6.3 Open Research Lines

This thesis fulfilled the general initial goal of designing, implementing and evaluating parallel and energy-efficient iterative sparse linear system solvers for multicore processors and manycore accelerators. During this research we identified some issues, which could become extensions of this thesis. The following list details some of the research lines which deserve further investigation:

- We have detected severe workload imbalances in some scenarios during the execution of the task-parallel versions of ILUPACK that affect the performance. A future research line is to tackle this problem, with the purpose of rendering a faster execution. For this purpose, it will be beneficial to decompose some of the computational kernels in the PCG iteration (especially the SPMV) to expose further levels of task parallelism that can be then exploited by OmpSs.

- In our task-parallel implementations, the leaf tasks present ample parallelism. However, to take advantage of that parallelism, the workload of the leaves should be well-adjusted and distributed among the nodes. Consequently, it is necessary to generate better partitionings of the solution of sparse matrices in order to balance the workload of the leaves. For this goal, it may be necessary to parameterize ParMetis (or mt-Metis) in order to improve the partitioning.

- We have realized that the concurrency that can be obtained in ILUPACK by expanding the levels of the tree is limited. As a consequence, it is necessary to add another level of concurrency to improve the scalability of the parallel solvers. Concretely, the current versions

of ILUPACK invoke, at each level a "scalar" ILU. In contrast, a future approach should rely on a "block-structured" ILU that invokes multi-threaded and cache-optimized level-3-BLAS, making possible the efficient execution of ILUPACK on a few thousand cores.

- Related to energy efficiency, a future research line should evaluate the best frequency and combination of MPI ranks/OmpSs threads, on different platforms, for the execution of each operation (kernel) in the iterative PCG. With this information it is possible to define several strategies to adapt the concurrency throttling and the voltage-frequency setting to obtain a more energy-efficient execution of the solver.

Acronyms

AC Alternating Current. 17, 19, 130

ACPI Advanced Configuration and Power Interface specification. 12, 16

AD Analog-to-Digital. 19

AINV Approximate INVerse. 6, 64, 65

AMG Algebraic MultiGrid. 6, 45, 66

AMR Adaptive Mesh Refinement. 74

API Application Programming Interface. 15, 17, 28, 83

BSC Barcelona Supercomputing Center. 12, 84

CG Conjugate Gradient. x, 4, 6, 41, 42, 44–48, 50, 51, 82, 132, 133

CPU Central Processing Unit. 16, 24, 25, 30, 34, 36, 38, 81, 82

CSC Compressed Sparse Column. 54, 61

CSR Compressed Sparse Row. 54, 61

CUDA Compute Unified Device Architecture. 14, 34, 36, 38, 83

DAG Directed Acyclic Graph. 77, 79–81, 87, 89, 92–95, 97, 98, 100, 101, 103, 105, 128, 129

DAS Data Acquisition System. 19, 29, 130

DC Direct Current. 17, 19, 29, 130, 131

DCT Dynamic Concurrency Throttling. 10

DIMM Dual In-line Memory Module. 30

DRAM Dynamic Random Access Memory. 24, 25, 110

DVFS Dynamic Voltage and Frecuency Scaling. 8, 10

FFT Fast Fourier Transform. 49

FPGA Field-Programmable Gate Array. 2

GPP general-purpose processors. xvii, 1, 2

GPU Graphics Processing Unit. 2, 11, 17, 24, 28, 36, 39, 82, 109–111

GUI Graphical User Interface. 12

HPC High Performance Computing. xvii, 2, 3, 11, 16, 19, 39, 84, 127, 128

HPCG High Performance Conjugate Gradient. 109, 130

HPP Heterogeneous Parallel Programming. 82

IC Incomplete Cholesky. 55, 76

ILDU Incomplete LDU. 64, 65, 68, 69

ILU Incomplete LU. xvii, 2, 5–7, 9, 41, 48–55, 58–61, 63, 65–69, 73, 76, 78, 80, 87, 106, 136

ILUC Incomplete LU Crout. 61, 63, 65, 67

IP Internet Protocol. 20

MIC Many Integrated Core. ix, 11, 24, 31, 112

MILU Modified ILU. 58

MMDO Multiple Minimum Degree Ordering. 76

MPI Message Passing Interface. xiii, 4–7, 9, 10, 14, 17, 39, 73, 74, 82–84, 93–99, 101, 103–107, 127–129, 132, 133, 136

MSR Model-Specific Register. ix, xiii, 21, 23–25, 29, 30, 32

MW MegaWatt. 2

ND Nested Disection. 75, 76

NI National Instruments. ix, xiii, 19, 23, 29, 30, 32, 130

NUMA Non-Uniform Memory Access. xvii, 9, 74, 84, 99–101, 103, 104, 107, 127, 129, 130, 132

NVML NVIDIA Management Library. ix, 11, 24, 28, 29, 39

OpenCL Open Computing Language. 14, 83

OpenMP Open Multi-Processing. 4, 6, 9, 14, 82, 83, 85

OS Operating System. 28

OSPM Operating System-directed configuration and Power Management. 16

PAPI Performance API. 15

PCG Preconditioned Conjugate Gradient. x, 10, 41, 51, 71, 73, 74, 80, 81, 85–87, 89, 90, 92–98, 100, 101, 103–107, 113, 114, 116, 117, 119, 121, 123, 127–130, 132, 135, 136

PDE Partial Differential Equation. 1, 2, 6, 41, 43, 45, 49, 55, 58, 65, 66, 68

PDU Power Distribution Unit. 19

PIC Peripheral Interface Controller. 19

PMAPI Performance Monitor API. 15

POSIX Portable Operating System Interface. 15

PSU Power Supply Unit. 19

RAM Random Access Memory. 34

RAPL Running Average Power Limit. ix, xiii, 11, 24–27, 29, 30, 32, 38, 39, 128

RaW Read-after-Write. 83, 86, 88, 89

SMP Symmetric Multi-Processing. 83

SNMP Simple Network Management Protocol. 19

SPD Symmetric Positive Definite. 7, 46, 49, 50, 55, 76, 85

SPMD Single Program Multiple Data. 84

TAG Task Acyclic Graph. 35

TDG Task Dependency Graph. xi, 114, 117, 119–123

USB Universal Serial Bus. 19, 29

VLSI Very-Large-Scale Integration. 1

WaR Write-after-Read. 83

WaW Write-after-Write. 83

WMI Windows Management Instrumentation. ix, 28, 30

Bibliography

[1] Haswell Architecture. Available at: http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/61451-intel-haswell-i7-4770k-i5-4670k-review-2.html.

[2] OmpSs website. http://pm.bsc.es/ompss.

[3] Sandy Bridge Architecture. Available at: http://www.qdpma.com/systemarchitecture/systemarchitecture_sandybridge.html.

[4] Xeon Phi Architecture. Available at: http://www.xicomputer.com/Solutions/Intel/xeon_phi/.

[5] The Green500 list, 2016. Available at: http://www.green500.org.

[6] *MUltifrontal Massively Parallel Solver (MUMPS 5.0.2) User's Guide*, 2016.

[7] The Top500 list, 2016. Available at: https://www.top500.org/.

[8] 40 Years of Microprocessor Trend Data. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/, 2015.

[9] A. Knüpfer, H. Brunst, *et al.* The vampir performance analysis tool-set. *Tools for High Performance Computing* (2008), 139–155.

[10] Alameldeen, A. R., Wagner, I., Chishti, Z., Wu, W., Wilkerson, C., and Lu, S.-L. Energy-efficient cache design using variable-strength error-correcting codes. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 461–472.

[11] Albers, S. Energy-efficient algorithms. *Commun. ACM 53* (May 2010), 86–96.

[12] Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing 54* (2016), 97 – 107.

[13] Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., and Quintana-Ortí, E. S. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned CG method. In *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014)* (2014), pp. 262–269.

[14] Aliaga, J. I., Barreda, M., Bollhöfer, M., and Quintana-Ortí, E. S. Exploiting task-parallelism in message-passing sparse linear system solvers using OmpSs. In *22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016)* (2016), pp. 631–643.

[15] Aliaga, J. I., Barreda, M., Castaño, A., Dolz, M. F., and Quintana-Ortí, E. S. Adapting concurrency throttling and voltage-frequency scaling for dense eigensolvers. *The Journal of Supercomputing* (2015), 1–15.

[16] Aliaga, J. I., Barreda, M., Castaño, A., Dolz, M. F., and Quintana-Ortí, E. S. Strategies for adapting concurrency throttling and dynamic voltage-frequency scaling for dense eigensolvers. In *15th International Conference on Computational and Mathematical Methods in Science and Engineering* (Rota, Spain, 2015), vol. 1, pp. 76–80.

[17] Aliaga, J. I., Barreda, M., Dolz, M. F., Martín, A. F., Mayo, R., and Quintana-Ortí, E. S. Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems. *Cluster Computing 17*, 4 (2014), 1335–1348.

[18] Aliaga, J. I., Barreda, M., Dolz, M. F., and Quintana-Ortí, E. S. Are our dense linear algebra libraries energy-friendly? Time-power-energy trade-offs in BLAS and LAPACK. *Computer Science - Research and Development 30*, 2 (2015), 187–196.

[19] Aliaga, J. I., Barreda, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. Characterizing the efficiency of multicore and manycore processors for the solution of sparse linear systems. *Computer Science - Research and Development 31*, 4 (2016), 175–183.

[20] Aliaga, J. I., Barreda, M., Flegar, G., Bollhöfer, M., and Quintana-Ortí, E. S. Communication in task-parallel ILU-preconditioned CG solvers using MPI+OmpSs. *Concurrency and Computation: Practice and Experience*. In revision.

[21] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing 37*, 3 (2011), 183–202.

[22] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Comput. 37*, 3 (2011), 183–202.

[23] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In *Applied Parallel and Scientific Computing*, K. Jónasson, Ed., vol. 7133 of *Lecture Notes in Computer Science*. 2012, pp. 162–172.

[24] Aliaga, J. I., Dolz, M. F., Martín, A. F., Mayo, R., and Quintana-Ortí, E. S. Leveraging task-parallelism in energy-efficient ILU preconditioners. In *ICT as Key Technology against Global Warming*, vol. 7453 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 55–63.

[25] Alonso, P., Badia, R. M., Labarta, J., Barreda, M., Dolz, M. F., Mayo, R., Quintana-Orti, E. S., and Reyes, R. Tools for power-energy modelling and analysis of parallel scientific applications. In *41st International Conference on Parallel Processing (ICPP)* (2012), pp. 420–429.

[26] Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2012), pp. 56–62.

[27] Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Saving energy in the LU factorization with partial pivoting on multi-core processors. In *20th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP)* (2012), pp. 353–358.

[28] Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Runtime scheduling of the LU factorization: Performance and energy. In *Energy Efficiency in Large Scale Distributed Systems*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 153–167.

[29] Anzt, H., Dongarra, J., and Quintana-Ortí, E. S. Tuning stationary iterative solvers for fault resilience. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (New York, NY, USA, 2015), ScalA '15, ACM, pp. 1:1–1:8.

[30] Ashby, S., and *et al.* The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010.

[31] Aupy, G., Benoit, A., Dufossé, F., and Robert, Y. Reclaiming the energy of a schedule: models algorithms. *Concurrency and Computation: Practice and Experience 25*, 11 (2006), 1505–1523.

[32] Axelsson, O. *Iterative Solution Methods.* Cambridge University Press, New York, NY, USA, 1994.

[33] Bacha, A., and Teodorescu, R. Dynamic reduction of voltage margins by leveraging on-chip ecc in itanium ii processors. In *ISCA* (2013).

[34] Badia, R. M., Labarta, J., Marjanovic, V., Martín, A. F., Mayo, R., Quintana-Ortí, E. S., and Reyes, R. Symmetric rank-k update on clusters of multicore processors with SMPSs. In *Applications, Tools and Techniques on the Road to Exascale Computing*, vol. 22 of *Advances in Parallel Computing*, pp. 657–664.

[35] Balay, S., Buschelman, K., Eijkhout, V.and Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., , and Zhang. *PETSc User's Manual. Tech. Rep ANL-95/11 - Revision 3.7.* Argone National Laboratory, 2016.

[36] Barrachina, S., Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. S. An integrated framework for power-performance analysis of parallel scientific workloads. *3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)* (2013), 114–119.

[37] Barreda, M., Catalán, S., Dolz, M. F., , Mayo, R., and Quintana-Ortí, E. S. Automatic detection of power bottlenecks in parallel scientific applications. *Computer Science - Research and Development* (2013), 1–9.

[38] Barreda, M., Catalán, S., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Tracing the power and energy consumption of the QR factorization on multicore processors. In *12th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)* (2012), pp. 134–142.

[39] Barreda, M., Dolz, M. F., Mayo, R., Quintana-Ortí, E. S., and Reyes, R. Binding performance and power of dense linear algebra operations. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2012), pp. 63–70.

[40] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the solution of linear systems: Building blocks for iterative methods*, 2rd ed. SIAM, Philadelphia, PA, USA, 1994.

[41] Benzi, M., Meyer, C. D., and Tuma, M. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing 17*, 5 (1996), 1135–1149.

[42] Benzi, M., and Tuma, M. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing 19*, 3 (1998), 968–994.

[43] Bergman, K., and *et al.* Exascale computing study: Technology challenges in achieving exascale systems. DARPA IPTO ExaScale Computing Study, 2008.

[44] Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C. A., and Peterson, J. R. Uintah: A scalable framework for hazard analysis. In *Proceedings of the 2010 TeraGrid Conference* (New York, NY, USA, 2010), TG '10, ACM, pp. 3:1–3:8.

[45] Bollhöfer, M. A robust ILU with pivoting baased on monitoring the growth of the inverse factors. *Linear Algebra Appl. 338*, 1-3 (2001), 201–218.

[46] Bollhöfer, M. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing 25*, 1 (2004), 86–103.

[47] Bollhöfer, M., Aliaga, J. I., Martin, A. F., and Quintana-Ortí, E. S. *Encyclopedia of parallel computing*. Springer US, Boston, MA, 2011, ch. ILUPACK, pp. 917–926.

[48] Bollhöfer, M., Grote, M., and Schenk, O. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM J. Sci. Comput. 31*, 5 (2009), 3781–3805.

[49] Bollhofer, M., Grote, M., and Schenk, O. Algebraic Multilevel Preconditioner for the Helmhotz Equation in Heterogeneous Media. *SIAM J. Sci. Comput. 31*, 5 (2009), 3781–3805.

[50] Bollhöfer, M., and Saad, Y. On the relations between ILUs and factored approximate inverses. *SIAM Journal on Scientific Computing 24*, 1 (2002), 219–237.

[51] Bollhöfer, M., and Saad, Y. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing 27*, 5 (2006), 1627–1650.

[52] Bollhöfer, M., Saad, Y., and Schenk, O. ILUPACK, vol. 2.1, preconditioning software package. http://ilupack.tu-bs.de, 2006.

[53] Borkar, S., and Chien., A. A. The future of microprocessors. *Communications of the ACM 54*, 5 (2011), 67–77.

[54] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput. 38*, 1-2 (Jan. 2012), 37–51.

[55] Brandt, A. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation 31, 138* (1997), 333 –390.

[56] Brezinski, C. *Projection methods for linear systems.* 1996.

[57] Briggs, W., Henson, V., and McCormick, S. A. *Multigrid tutorial*, 2nd ed. Society for Industrial and Applied Mathematics, 2000.

[58] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonnald, J., and Menon, R. *Parallel programming in OpenMP.* Morgan Kaufmann Publishers, 2001.

[59] Chapman, B., and van der Pas, R. *Using OpenMP. Portable shared memory parallel programming.* Morgan Kaufmann Publishers, 2007.

[60] Chevalier, C., and Pellegrini, F. PT-SCOTCH: A tool for efficient parallel graph ordering. *Parallel Comput. 34*, 6-8 (2008), 318–331.

[61] Chow, E., and Saad, Y. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics 86*, 2 (1997), 387–414.

[62] CORMEN, T. H., LEISERSON, C. E., and RIVEST, R. L. *Introduction to algorithms.* McGraw-Hill, New York, 1990.

[63] Davis, T. A. *Direct methods for sparse linear systems.* SIAM Publications, 2006.

[64] DEEP project home page. http://www.deep-project.eu/.

[65] Demmel, J., and Yelick, K. Communication avoiding (CA) and other innovative algorithms. *The Berkeley Par Lab: Progress in the Parallel Computing Landscape* (2013), 243–250.

[66] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. Design of ion-implanted MOSFET's with very small physical dimensions, volume = 9, number = 5, year = 1974. *IEEE Journal of Solid-State Circuits*, 256–268.

[67] Diamos, G. F., and Yalamanchili, S. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proc. 17th Int. Symp. High Performance Distributed Computing* (2008), HPDC '08, pp. 197–200.

[68] Diaz, J., noz Caro, C. M., and no, A. N. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems 23*, 8 (2012), 1369–1382.

[69] Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Solving some mysteries in power monitoring of servers: Take care of your wattmeters! In *Energy Efficiency in Large Scale Distributed Systems*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 3–18.

[70] Dongarra, J., et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications 25*, 1 (2011), 3–60.

[71] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16*, 1 (March 1990), 1–17.

[72] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct methods for sparse matrices.* Clarendon Press, Oxford, 1986.

[73] Duff, I. S., and Koster, J. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl., year = 2000, volume = 22, number = 4, pages = 973–996*.

[74] Duran, A., Ferrer, R., Ayguadé, E., Badia, R. M., and Labarta, J. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming 37*, 3 (2009), 292–305.

[75] Duranton, M. *et al.* The HiPEAC vision for advanced computing in horizon 2020, 2013.

[76] EISENSTAT, S. C., SCHULTZ, M. H., and SHERMAN, A. H. Algorithms and data structures for sparse symmetric gaussian elimination. *SIAM Journal on Scientific Computing 2* (1981), 225–237.

[77] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer architecture - ISCA'11* (2011), pp. 365–376.

[78] et al., M. D. The HiPEAC vision. `http://www.hipeac.net/roadmap`. [retrieved: July, 2016].

[79] Extrae: User guide manual for version 2.5.1. `http://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide`.

[80] Feng, W.-c., Feng, X., and Ge, R. Green supercomputing comes of age. *IT Professional 10*, 1 (jan.-feb. 2008), 17 –23.

[81] Fuller, S. H., and Millett, L. I. The future of computing performance: Game over or next level? In *National Research Council of the National Academies* (2011).

[82] Gaidamour, J., Hénon, P., and Saad, Y. *HIOS user's guide.* INRIA. `http://hips.gforge.inria.fr/doc.html`.

[83] Gantz, J., and Reinsel, D. Extracting value from chaos. *IDC Iview*, 1142 (2011), 9–10.

[84] Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proc. 2013 IEEE 27th Int. Symp. on Parallel and Distributed Processing* (2013), IPDPS '13, pp. 1299–1308.

[85] Gensh, R., Aalsaud, A., Rafiev, A., Xia, F., Iliasov, A., Romanovsky, A., and Yakovlev, A. Experiments with the odroid-XU3 board. Tech. Rep. CS-TR-1471, Newcastle University, May 2015.

[86] GEORGE, J. A., and LIU, J. W. *Computer solution of large sparse positive definite systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[87] GILBERT, J. R., and PEIERLS, T. Sparse partial pivoting in time proportional to artithmetic operations. *SIAM Journal on Scientific Computing 9* (1988), 862–874.

[88] Golub, G. H., and Loan, C. F. V. *Matrix computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, 1996.

[89] Grimshaw, A. S. Easy-to-use object-oriented parallel processing with Mentat. *Computer 26*, 5 (May 1993), 39–51.

[90] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable parallel programming with the message-passing interface.* The MIT Press, 1999.

[91] Gropp, W., Lusk, E., and Thakur, R. *Using MPI-2: Advanced features of the message-passing interface.* The MIT Press, 1999.

[92] Gruber, R., and Keller, V. One Joule per GFlop for BLAS2 Now! In *AIP Conference Proceedings* (2010), S. Theodore E., P. George, and T. Ch, Eds., vol. 1281, American Institute of Physics, pp. 1321–1324.

[93] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software 27*, 4 (2001), 422–455.

[94] Gupta, A. *WSMP: Watson Sparse Matrix Package Part II - direct solution of general systems. Version 16.06.* IBM T. J. Watson Research Center, 2000. Last update June 2016.

[95] Gupta, A., and Avron, H. *WSMP: Watson Sparse Matrix Package Part I - direct solution of symmetric systems. Version 16.06.* IBM T. J. Watson Research Center, 2000. Last update June 2016.

[96] Gupta, A., and Avron, H. *WSMP: Watson Sparse Matrix Package Part III - iterative solution of sparse systems. Version 16.06.* IBM T. J. Watson Research Center, 2007. Last update June 2016.

[97] Hackbusch, W. *Multi-grid methods and applications.* Springer, 2003.

[98] Hageman, L. A., and Young, D. M. *Applied iterative methods.* Academic Press, 1981.

[99] Hagemann, M., and Schenk, O. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM J. Sci. Comput. 28*, 2 (2006), 403–420.

[100] Hassan, S. M., Yalamanchili, S., and Mukhopadhyay, S. Near data processing: Impact and optimization of 3d memory system architecture on the uncore. In *2015 International Symposium on Memory Systems (Memsys 2015)* (October 2015).

[101] Hénon, P., Ramet, P., and Roman, J. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing 28*, 2 (2002), 301–321.

[102] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. An overview of the Trilinos project. *ACM Trans. Math. Software 3*, 33.

[103] Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49,6* (1952), 409–436.

[104] Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards 49* (1952), 409–435.

[105] Hoemmen., M. *Communication-avoiding Krylov subspace methods.* PhD thesis, Berkeley, 2010. UMI Order No. GAX87-23080.

[106] Hogg, J. D., Reid, J. K., and Scott, J. A. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM J. Sci. Comput. 32*, 6 (Dec. 2010), 3627–3649.

[107] HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., Toshiba Corp. *Advanced configuration and power interface specification*, 2013. Revision 5.0a.

[108] Hwu, W., Keutzer, K., and Mattson, T. G. The concurrency challenge. *IEEE Design and Test of Computers 25*, 4 (2008), 312–320.

[109] Hypre web page. `http://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/`, 2016.

[110] ILUPACK project home page. `http://ilupack.tu-bs.de`.

[111] Intel. Intel math kernel library (mkl) 11.0. `http://software.intel.com/en-us/intel-mkl`.

[112] Intel. *Intel Xeon Phi coprocessor system software developers guide*, 2014.

[113] Intel Corp. Intel Xeon processor. `http://www.intel.com/xeon`, 2012.

[114] Intel Corp. *Intel 64 and IA-32 architectures software developer manual. Volume 3B: System programming guide, Part 2*, 2015.

[115] INTERTWinE project home page. `http://www.intertwine-project.eu/`.

[116] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[117] Karypis, G., and Kumar, V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput. 48*, 1 (1998), 71–95.

[118] Karypis, G., and Schloegel, K. *ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Version 4.0.* University of Minnesota, Department of Computer Science and Engineering, Minneapolis, 2013.

[119] Kim, K., and Eijkhout, V. A parallel sparse direct solver via hierarchical dag scheduling. *ACM Trans. Math. Softw. 41*, 1 (Oct. 2014), 3:1–3:27.

[120] Kunkel, J. HDTrace - a tracing and simulation environment of application and system interaction. Tech. Rep. 2, Department of Informatics, Scientific Computing. Universität Hamburg, 2011.

[121] Kuzmin, A., Luisier, M., and Schenk, O. Fast methods for computing selected elements of the Greens function in massively parallel nanoelectronic device simulations. In *Euro-Par* (2013), S. B. Heidelberg, Ed., vol. 8097, pp. 533–544.

[122] Lacoste, X., Faverge, M., Ramet, P., Thibault, S., and Bosilca, G. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Research Report RR-8446, INRIA, Jan. 2014.

[123] LaSalle, D., and Karypis, G. Efficient nested dissection for multicore architectures. *21st International European Conference on Parallel and Distributed Computing (Euro-Par)* (2015), 467–478.

[124] Li, N., Saad, Y., and Chow, E. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing 25*, 2 (2003), 716–728.

[125] Li, X. S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Mathematical Software 31*, 3 (2005), 302–325.

[126] Li, Z., Saad, Y., and Sosonkina, M. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Lin. Alg. W. Appl. 10* (2003), 485–509.

[127] Liu, J. W. H. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Soft. (TOMS) 11*, 2 (1985), 141–153.

[128] Lively, C., Taylor, V., Wu, X., Chang, H.-C., Su, C.-Y., Cameron, K., Moore, S., and Terpstra, D. E-amom: an energy-aware modeling and optimization methodology for scientific applications. *Computer Science - Research and Development 29*, 3 (2014), 197–210.

[129] Ludwig, T. Editorial for the first international conference on energy-aware high performance computing (EnA-HPC). *Computer Science - Research and Development 25*, 3 (2010), 123–124.

[130] Luk, C.-K., Hong, S., and Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture* (2009), MICRO 42, pp. 45–55.

[131] Martín, A. F. *Utilización del paralelismo multihebra en el precondicionado y la resolución iterativa de sistemas lineales dispersos.* PhD thesis, Universitat Jaume I, Castellón, 2010.

[132] Märtin, C. Multicore processors: Challenges, opportunities, emerging trends. In *Poceedings of Embedded World Conference 2014* (Feb. 2014).

[133] Mathew, T. *Domain decomposition methods for the numerical solution of partial differential equations (Lecture notes in Computational Science and Engineering)*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[134] McCormick, S. F. *Multigrid methods.* SIAM Books, Philadelphia, PA, USA, 1987.

[135] METIS official home page. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[136] Mienik, M. CPU burn-in v1.01. http://www.cpuburnin.com/.

[137] Mittal, S. A survey of techniques for approximate computing. *ACM Comput. Surv. 48*, 4 (March 2016), 62:1–62:33.

[138] Montblanc project home page. http://www.montblanc-project.eu/.

[139] Moore, G. Cramming more components onto integrated circuits. *Electronics 38*, 18 (1965), 114–117.

[140] MPICH project home page. http://www.mpich.org/.

[141] Mucci, P. J., Browne, S., Deane, C., and Ho, G. PAPI: A portable interface to hardware performance counters. In *Department of Defense HPCMP Users Group Conference* (1999), pp. 7–10.

[142] MVAPICH project home page. http://mvapich.cse.ohio-state.edu/.

[143] NANOS project home page. http://research.ac.upc.edu/nanos.

[144] Netgen. NETGEN - automatic mesh generator. http://www.hpfem.jku.at/netgen, 2012.

[145] NVIDIA. *NVML API reference manual*, 2012.

[146] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture programming guide*, 2.3.1 ed., August 2009.

[147] Nvml: Nvidia management library. https://developer.nvidia.com/nvidia-management-library-nvml.

[148] Official Website, P. P. L. *Python*. http://www.python.org/.

[149] OmpSs project home page. http://pm.bsc.es/ompss/.

[150] The OpenMP API specification for parallel programming. http://http://openmp.org/wp/openmp-specifications/.

[151] OpenMPI project home page. https://www.open-mpi.org/.

[152] Pachecho, P. *Parallel programming with MPI*. Morgan Kaufmann Publishers, 1997.

[153] Paraver project. http://www.bsc.es/computer-sciences/performance-tools/paraver.

[154] pARMS: parallel Algebraic Recursive Multilevel Solvers. http://www-users.cs.umn.edu/~saad/software/pARMS/.

[155] PETSc project home page. http://acts.nersc.gov/petsc.

[156] pynvml: Python bindings to the nvidia management library. https://pypi.python.org/pypi/nvidia-ml-py/.

[157] Quarteroni, A. M., and Valli, A. *Domain decomposition methods for partial differential equations*. Oxford University Press, 1999.

[158] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2009), ACM, pp. 121–130.

[159] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Zee, F. G. V., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software 36*, 3 (2009), 14:1–14:26.

[160] Ramet, P. *PaStiX user's manual.* Institut National de Recherche en Informatique et Automatique (INRIA), 2013.

[161] R.D. Falgout, J. J., and Yang, U. The design and implementation of hypre, a library of parallel high performance preconditioners. *Numerical Solution of Partial Differential Equations on Parallel Computers 51* (2006), 267–294.

[162] Rojek, K., Barreda, M., Quintana-Ortí, E. S., and Wyrzykowski, R. Energy consumption of stencil-based MPDATA algorithm. In *16th International Conference on Computational and Mathematical Methods in Science and Engineering* (Rota, Spain, 2016), pp. 1104–1107.

[163] Rountree, B., Lownenthal, D. K., de Supinski, B. R., Schulz, M., Freeh, V. W., and Bletsch, T. Adagio: Making DVS practical for complex HPC applications. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 460–469.

[164] S. Shende, A. M. The TAU parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (2006).

[165] Saad, Y. ILUT: A dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications 1* (1994), 387–402.

[166] Saad, Y. *Iterative methods for sparse linear systems*, 3rd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[167] Saad, Y., and Schultz, M. H. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7, 3* (1986), 856–869.

[168] Saad, Y., and Suchomel, B. ARMS: An Algebraic Recursive Multilevel Solver for General Sparse Linear Systems. *Numer. Lin. Alg. w. Appl. 9*, 5 (2002), 359–378.

[169] Sarood, O., Miller, P., Totoni, E., and Kale, L. V. Load balancing for high performance computing data centers. *IEEE Trans. Comput. 61*, 12 (Dec. 2012), 1752–1764.

[170] Saxe, E. Power-efficient software. *ACM Queue* (2010).

[171] Schenk, O., Bollhöfer, M., and Römer, R. Awarded SIGEST paper: On large scale diagonalization techniques for the Anderson model of localization. *SIAM Review 50* (2008), 91–112.

[172] Schenk, O., and Gärtner, K. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems 20*, 3 (2004), 475–487.

[173] Schenk, O., and Gärtner, K. *Parallel Sparse Direct Solver PARDISO: User guide version 5.0.0*, 2014.

[174] SCOTCH official home page. `http://www.labri.fr/perso/pelegrin/scotch/`.

[175] Smith, B. F., Bjørstad, P. E., and Gropp, W. D. *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*, 1st ed. Cambridge University Press, New York, NY, USA, 1996.

[176] Sottile, M., Mattson, T. G., and Rasmussen, C. E. *Introduction to concurrency in programming languages*, 1st ed. Chapman & Hall/CRC, 2009.

[177] StarPU project home page. `http://runtime.bordeaux.inria.fr/StarPU/`.

[178] Strakoš, Z., and Tichý, P. On error estimation in the Conjugate Gradient method and why it works in finite precision computations. *Electronic Trans. Numer. Anal. 13* (2002), 56–80.

[179] Strzodka, R., and Göddeke, D. Pipelined mixed precision algorithms on fpgas for fast and accurate pde solvers from low precision components. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)* (2006), pp. 259–268.

[180] Tan, L., Song, S. L., Wu, P., Chen, Z., Ge, R., and Kerbyson, D. J. Investigating the interplay between energy efficiency and resilience in high performance computing. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2015), IPDPS '15, IEEE Computer Society, pp. 786–796.

[181] Trottenberg, U., Oosterlee, C., and Sch Auller, A. *Multigrid*. Academic Press, 2001.

[182] V.Pillet, J.Labarta, T.Cortes, and S.Girona. Paraver: A tool to visualize and analyze parallel code. *18th World OCCAM and Transputer User Group Technical Meeting* (1995).

[183] Watkins, D. S. *Fundamentals of matrix computations*, 2nd ed. John Wiley and Sons, inc., New York, 2002.

[184] What is C-state from DELL home page. `http://www.dell.com/support/article/us/en/04/QNA41893?c=us&l=en&s=bsd&cs=04%2Fit%2Fen`.

[185] Xu, Q., Kim, N. S., and Mytkowicz, T. Approximate computing: A survey. *IEEE Design & Test 33*, 1 (2016), 8–22.

[186] Yang, U. M. Algebraic multigrid methods – high performance preconditioners. *Lecture Notes in Computational Science and Engineering. Springer 51* (2006), 209–236.

[187] Zhu, X., Ge, R., Sun, J., and He, C. 3e: Energy-efficient elastic scheduling for independent tasks in heterogeneous computing systems. *Journal of Systems and Software 86*, 2 (2013), 302 – 314.

[188] Zlatev, Z., and Waśniewski, J. *PARASPAR: Parallel solvers for sparse linear algebraic systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, pp. 547–556.