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

Energy-Aware Matrix Computations on Multithreaded Architectures

Castelló de la Plana, March 2014

Ph.D. Thesis

Presented by: Manuel Francisco Dolz Zaragozá
Supervised by: Enrique S. Quintana Ortí
Pedro Alonso Jordá

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

Energy-Aware Matrix Computations on Multithreaded Architectures

Manuel Francisco Dolz Zaragozá

Contents

List of Figures

List of Tables

The ultimate "computer," our own brain, uses only ten watts of power – one-tenth the energy consumed by a hundred-watt bulb.

Paul Valéry

Resumen

Desde hace décadas, la computación de altas prestaciones ha concentrado sus esfuerzos en la optimización de algoritmos aplicados a la resolución de problemas complejos que aparecen en un amplio abanico de aplicaciones de casi todas las áreas científicas y tecnológicas. En este sentido el uso de herramientas y técnicas, tales como la computación paralela y distribuida, han impulsado la mejora de las prestaciones en este tipo de algoritmos y aplicaciones. Hoy en día, el término optimización hace referencia a la reducción del tiempo de ejecución, aunque también a la energía necesaria para su cómputo.

Concretamente, en el camino hacia los sistemas Exaescala (capaces de alcanzar la barrera de los ExaFLOPS: 10^{18} instrucciones en coma flotante por segundo), el consumo de energía es un aspecto fundamental al que esta rama científica debe hacer frente [38, 55]. Algunos estudios realizados sobre el consumo de energía revelan que una plataforma Exaescala construida con la tecnología hardware actual disiparía 220 MW, haciéndola económicamente inviable [50, 54, 58, 65, 90]. En otras palabras, incluso con la tasa de mejora en la eficiencia energética con la que cuentan las supercomputadoras en los últimos años [3], el consumo de energía de 20 MW que se pretende que genere una plataforma Exaescala a finales de esta década, se multiplicaría por un factor muy significativo, convirtiéndose en inviable. Por tanto, si se tiene que superar la barrera de la energía, se necesita un enfoque global de potencia y energía, en particular uno que tenga como objetivo el desarrollo de hardware más eficiente en cuanto al consumo, y utilice aplicaciones que sean conscientes de la energía disipada, tanto en el sistema operativo como en las bibliotecas de cómputo, comunicación y aplicaciones paralelas. La búsqueda de soluciones verdes o fuentes de energía alternativas que permitan reducir las emisiones de CO_2 a la atmósfera demuestran la creciente preocupación por el medio ambiente. En el ámbito de las tecnologías de la información y, más concretamente, en la computación de altas prestaciones, la comunidad cientifico-técnica muestra especial interés en el desarrollo de componentes, herramientas y técnicas que permitan minimizar el consumo energético.

Este trabajo de investigación aborda las posibilidades de ahorro energético que pueden conseguirse en arquitecturas multinúcleo e híbridas (CPU–GPU). Las técnicas de ahorro definidas son aplicadas sobre el contexto de la computación de altas prestaciones, concretamente sobre aplicaciones que aprovechan el paralelismo a nivel de tareas, en un ámplio abanico de problemas de álgebra lineal densa y dispersa. Este estudio se completa con resultados experimentales obtenidos mediante medidores de energía, que validan las ganancias conseguidas; al mismo tiempo, el uso de estas estrategias mantiene un constante compromiso entre prestaciones y ahorro de energía.

Objetivos

Los objetivos de esta tesis doctoral están orientados al estudio, análisis y aprovechamiento de las técnicas de ahorro disponibles en las arquitecturas de computadores actuales con el fin de mejorar el rendimiento energético en aplicaciones que requieren la resolución de problemas de álgebra lineal densa y dispersa.

En concreto, los objetivos del proyecto pueden resumirse de la forma siguiente:

- Documentación de la literatura existente sobre el ahorro de energía en procesadores multinúcleo y plataformas híbridas GPU. Análisis de los mecanismos de ahorro de energía disponibles a nivel hardware y software.
- Desarrollo de un entorno de experimentación y validación de pruebas: definición de las plataformas y arquitecturas, herramientas de perfilado, traceado y visualización, uso de dispositivos de medición de energía y desarrollo de una biblioteca para interactuar con los mismos.
- Análisis y desarrollo de un modelo de consumo para algoritmos que explotan el paralelismo de tareas, basado en parámetros de consumo de la arquitectura y propios de la aplicación.
- Análisis de estrategias de ahorro desde el punto de vista teórico, mediante simulación y estudio de su impacto en las prestaciones y ahorros de energía.
- Validación de las técnicas de ahorro sobre aplicaciones y bibliotecas de álgebra lineal densa y dispersa, mediante un estudio completo sobre el impacto en las prestaciones y las ganancias de consumo que pueden conseguirse gracias a la utilización de estas técnicas.

El resultado de esta tesis es un conjunto de conocimientos que permiten aplicar estrategias de reducción de energía en las aplicaciones estudiadas, que además sirvan de referencia para futuros desarrollos. Uno de los puntos más importantes de esta metodología es el que se refiere a la modelización y simulación de las estrategias. El modelo de consumo propuesto en la tesis permite conocer cómo los sistemas consumen energía, facilitando así la labor de implementar de forma eficiente aplicaciones optimizadas en tiempo de ejecución y consumo energético.

Metodología y desarrollo

En el desarrollo de la tesis se ha empleado la metodología clásica de desarrollo de sistemas, con las siguientes etapas:

1. Revisión del estado del arte.
2. Análisis de requerimientos.
3. Estudio de distintas estrategias de gestión/reducción de energía con mínimo impacto en el rendimiento.
4. Modelización del funcionamiento de dichas estrategias y simulación de las mismas.
5. Desarrollo y adaptación de las técnicas de reducción de energía en aplicaciones, mediante módulos e interfaces flexibles.
6. Verificación y validación de los módulos e interfaces mediante diferentes tipos de aplicaciones y sistemas.

7. Integración, verificación y documentación.

La tesis doctoral se ha desarrollado bajo las etapas propuestas. Asimismo, incluye información detallada sobre la documentación, planificación, desarrollo de los entornos empleados, resultados y conclusiones. En general, se pretende dar una visión global sobre los técnicas y mecanismos a seguir para promover y programar aplicaciones eficientes, tanto por tiempo de ejecución como por la energía requerida para ejecutarse.

Contribuciones

Los resultados, métodos y técnicas descritos en esta tesis han sido publicados en *workshops*, congresos y revistas de carácter internacional. Las contribuciones de esta tesis se pueden agrupar en cuatro líneas principales.

Medición de potencia en computadores

La primera de las contribuciones es un entorno completo para el análisis de la potencia y prestaciones de las aplicaciones científicas de altas prestaciones. Este marco de trabajo ha sido evaluado con diversos algoritmos clave en el campo de la álgebra lineal densa, demonstrando así sus beneficios en la detección de cuellos de botella y puntos de baja eficiencia energética. El marco presentado ofrece información de la potencia y el rendimiento para diferentes tipos de aplicaciones paralelas, desde códigos paralelos MPI que se ejecutan en clusters de escala moderada, a aplicaciones multihilo que corren en plataformas multinúcleo e híbridas CPU–GPU. Además, se ha desarrollado la biblioteca PMLib, que permite recopilar datos de potencia y facilita la interacción con los dispositivos de medición de potencia conectados a las plataformas evaluadas. El diseño modular de esta herramienta permite la sencilla integración de nuevos módulos para la recolección de nuevas medidas.

Modelos de potencia y energía

Como contribución principal de esta parte, se presenta un modelo de potencia/energía capaz de estimar el consumo de energía para aplicaciones que explotan el paralelismo a nivel de tarea. Al mismo tiempo, se demuestra que es posible modelar sistemáticamente la potencia y la energía consumida en arquiecturas paralelas multinúcleo. Dos propiedades del modelo propuesto son su portabilidad y generalidad, ya que no requiere el acceso a contadores hardware de bajo nivel, dependientes de la plataforma. Esta aproximación es válida, en general, para estimar la potencia y la energía de aplicaciones que explotan el paralelismo a nivel de tarea y realizan operaciones intensivas de aritmética entera o en coma flotante y no está necesariamente restringida al álgebra lineal densa.

Además de proporcionar estimaciones precisas del consumo de energía, nuestro modelo tiene como objetivo orientar y ayudar a reducir el tiempo de ejecución de aplicaciones paralelas ejecutadas en entornos que explotan el paralelismo a nivel de tarea. Por ejemplo, en situaciones en las que existen diferentes asignaciones de tareas a los recursos computacionales que pueden tener un rendimiento casi equivalente en las plataformas híbridas con aceleradores hardware, nuestro modelo puede ser utilizado para estimar automáticamente la energía de diferentes configuraciones de hardware y seleccionar la más eficiente sin que el rendimiento se vea afectado.

Técnicas de ahorro de energía

Una de las principales aportaciones de esta parte son dos técnicas clave para el ahorro de energía basadas en el escalado del voltaje y de la frecuencia de los procesadores. El algoritmo de reducción de holguras, o *Slack Reduction Algorithm*, aprovecha las holguras existentes a causa de las dependencias de tareas y, en la medida de lo posible, asigna una frencuencia de ejecución menor a las tareas no críticas. Al mismo tiempo, también se ofrece una estrategia adicional, el algoritmo *Race-to-Idle*, que persigue un objetivo opuesto al algoritmo anterior, asignando siempre la máxima frecuencia para la ejecución de las tareas de las aplicaciones con el objetivo de reducir su tiempo de ejecución y generando así mayores tiempos de inactividad en los que poder ahorrar energía.

Además, se ha desarrollado un simulador capaz de aplicar y evaluar el impacto de estas técnicas en algoritmos de álgebra lineal densa. En el estudio se ofrece un análisis completo de la eficiencia energética generada por las dos técnicas. Los resultados demuestran que, en condiciones reales, es posible una reducción en el consumo de energía bajo determinadas condiciones.

Entornos de ejecución consicientes del consumo en álgebra lineal densa y dispersa

La principal aportación de esta parte es el diseño e implementación de técnicas de ahorro de energía que, incorporadas en *runtimes* de bibliotecas que explotan el paralelismo a nivel de tarea, son capaces de generar ahorros energéticos durante la ejecución de aplicaciones y de algoritmos paralelos. Estas técnicas reemplazan las esperas activas realizadas por hilos inactivos, por esperas pasivas, reduciendo así el consumo energético durante estos periodos de tiempo. Las técnicas se han integrado en el *runtime* SuperMatrix de la biblioteca de álgebra lineal densa `libflame` en procesadores multinúcleo y plataformas híbridas CPU–GPU. Una contribución crucial es la genericidad de estas técnicas, pues pueden ser aplicadas sobre otros *runtimes* que presenten comportamientos ineficientes desde el punto de vista energético. Un estudio completo de estas técnicas, empleando la factorización de Cholesky y la factorización LU demuestra que se puede ahorrar hasta un 15 % de la energía consumida sin comprometer las prestaciones.

Como contribución adicional en esta parte de la tesis, las técnicas de ahorro también se han aplicado en el campo del álgebra lineal dispersa, en particular en el cálculo del precondicionador y la resolución iterativa de ecuaciones lineales en ILUPACK. Al mismo tiempo, se ha aprovechado el modelo de energía proporcionado para caracterizar el consumo de la resolución completa de este algoritmo.

Una conclusión general de este estudio es que en las operaciones limitadas por el acceso a memoria, como el ejemplo de ILUPACK considerado, la reducción del voltage y de la frecuencia es, en ocasiones, beneficioso y no compromete el tiempo de ejecución. Este aspecto no se observa en operaciones limitadas por la velocidad del procesador, en las que la reducción de la frecuencia tiene un impacto negativo en las prestaciones y el consumo. Por lo tanto, los esfuerzos de reducir el consumo energético en operaciones limitadas por el ancho de banda de memoria deben aprovechar cuidadosamente la reducción de la frecuencia de los procesadores con el fin de conservar las prestaciones y reducir la potencia. Para el caso de ILUPACK se presenta un estudio completo donde se demuestra el uso de esta estrategia.

Finalmente, el estudio realizado empleando las técnicas que reemplazan las esperas activas por pasivas en combinación con la reducción de la frecuencia revela un ahorro en el consumo de energía entre un 7 y un 13 % en ILUPACK, sin afectar prácticamente al rendimiento.

Líneas abiertas de investigación

La preocupación por la eficiencia energética en el campo de la computación de altas prestaciones es una disciplina relativamente nueva, por lo que se derivan muchas líneas de investigación como conclusión de esta tesis. Algunos de los posibles trabajos se detallan a continuación.

Uno de los objetivos a corto plazo es incorporar las técnicas de energía en *runtimes* de entornos de trabajo que explotan el paralelismo a nivel de tarea y bibliotecas de álgebra lineal, como por ejemplo SMPSs, OmpSs, MAGMA y PLASMA [118, 102, 5]. Como objetivo a medio plazo, se pretenden diseñar y desarrollar nuevas técnicas y heurísticas de mapeado de tareas conscientes del consumo energético en arquitecturas heterogéneas, donde pueda ser beneficioso ejecutar, por ejemplo, operaciones que requieren menor carga computacional en procesadores menos potentes, generando así ahorros energéticos. Finalmente, como objetivo a largo plazo se propone la generación de un modelo de potencia para bibliotecas paralelas de alto rendimiento. Esta operación sobre plataforma multinúcleo es un primer paso hacia un objetivo más ambicioso: hacer viable el modelo para una gran colección de códigos numéricos basados en paso de mensajes para clusters de gran escala equipados con procesadores multinúcleo.

Agradecimientos

Motivación, dedicación, perseverancia, esfuerzo y sacrificio son palabras clave que se derivan del trabajo de esta tesis doctoral, aunque también existen de otras: satisfacción, recompensa, nuevos conocimientos, proyectos y metas. En general, términos que determinan el resultado de una intensa etapa de mi vida, que me ha permitido afrontar experiencias que nunca antes hubiera imaginado y superarme en cosas de las que no me creía capaz, tanto a nivel personal como profesional. Sin embargo, nada de esto hubiese sido posible sin la ayuda de las personas que me han brindado la oportunidad de trabajar con ellas, y las que, desinteresadamente, me han acompañado en cada momento de esta aventura. A todas ellas, me gustaría expresar mi más sincero agradecimiento.

A mis directores, Enrique S. Quintana Ortí y Pedro Alonso Jordá. A Enrique, por su apoyo, por ser una fuente infinita de conocimientos y trabajador incansable. A Pedro, por haber confiado en mí desde el primer momento, por su dedicación y su ánimo en mis malos momentos.

A Rafael Mayo Gual, por su desinteresada colaboración en este trabajo, sus originales ideas y su forma de afrontar los problemas.

A la Universitat Jaume I, Generalitat Valenciana, Comisión Europea y Ministerio de Ciencia y Educación, por su soporte económico durante estos cuatro años y sin el cual este trabajo no hubiese sido posible.

A las personas del grupo HPC&A de la Universitat Jaume I y a las que en algún momento han estado en él, José, José Manuel, Sergio B., Asun, Maribel, Juan Carlos, Greg, Germán, Merche, Alfredo, Toni, Fran, Ruymán, Maria, Sandra, Sergio I., Héctor, Adrián, Sisco y Sonia.

Mi más sincero agradecimiento al profesor Thomas Ludwig, por ofrecerme la oportunidad de integrarme en su entorno de trabajo y hacerme sentir uno más. También a mis compañeros del grupo WR de la Universidad de Hamburgo, Michael, Raúl, Konstantinos, Fabian, Timo, Julian, Petra, Uli, Hermann, Nathanael, Marc, Michaela, Anna, Sandra y Michaele.

A mis amigos, familia y muy especialmente a mis padres, Manolo y Maribel, por su comprensión y soporte, por estar ahí, tanto en lo bueno como en lo malo y, sobre todo, por ofrecerme toda la ayuda posible.

– ¡Gracias! · Gràcies! · Thanks! · Danke! –

Hamburgo, marzo de 2014.

CHAPTER 1

Energy-Aware High Performance Computing

Energy consumption is already recognized as the crucial factor that will limit the performance of future microprocessors [50, 54, 58, 65, 90], leading to the design and adoption of heterogeneous architectures and dark silicon [38, 55]. As we target the ExaFLOP barrier, this issue is becoming a major concern. Reaching such an impressive performance rate using as-of-today most energy-efficient technology will require more than 220 MW [3], which amounts for a significant part of the energy produced by a modern nuclear plant to feed just a single supercomputer. At a large scale, directors of data processing centers and supercomputing facilities are painfully aware of the cost of the energy consumed by the computational resources and auxiliary systems deployed in these sites. In response to this, electrical engineers and computer architects are actively pursuing the design and development of power-friendly hardware and, consequently, current CPUs, memories, disks and networking devices now feature low-power modes that enable a trade-off between performance and energy. On the other hand, heterogeneous architectures that combine general-purpose multicore technology with hardware accelerators also seem to provide the answer to the continued pressure to further reduce power consumption [3, 38], historically exerted by the mobile and embedded appliances, but now also in place for High Performance Computing (HPC) facilities and data centers [50, 54].

The HPC community is currently well aware that reducing the energy drawn from compute-intensive applications is a concern almost in equal terms with the conventional quest for high performance. While these two factors seem in principle orthogonal to each other, our insights shown throughout this dissertation will reveal mutual implications that addressed together may yield relevant synergies.

1.1 The Road Towards Exascale Computing. The Power Wall

As we progress on the road to Exascale systems, the economic cost of energy consumption and the pressure exerted by power dissipation on cooling equipment are rapidly becoming major hurdles to the deployment of new HPC facilities. As-of-today, the most energy-efficient HPC supercomputers, equipped with NVIDIA graphics processors (GPUs), delivers close to 4.5 GFLOPS/W[1] [3]. Simple arithmetic shows that building an ExaFLOP system based on this scalable technology would

[1] 1 GFLOPS = 10^9 floating-point arithmetic operations, or FLOPS, per second.

require about 220 MW, yielding this approach economically unfeasible. Even if we can maintain the considerable improvements experienced by the most energy efficient systems of the Green500 list during the last 5 years, the goal of building a 20 MW Exascale system will still be largely exceeded. This is the so-called *power wall*. Therefore, if we want to continue enjoying the significant advances enabled by scientific computing and supercomputers during these past decades, a holistic investigation is needed to improve energy-efficiency of HPC hardware and software. In this sense, the HPC community is responding with a rising awareness of energy issues.

Significant challenges essential to developing Exascale computing need to be considered. The summary report of the Advisory Committee Advanced Scientific Computing 2010 [25] of the U.S. Department of Energy, identifies the following subjects as examples of complexity challenges that can extremely transformed through Exascale computing: aerospace, astrophysics, biological and medical systems, weather and climate, combustion, materials science and fusion energy. Exascale computing is, therefore, an emerging technology with a special and strategic value. On the other hand, it is clear that Exascale machines will not become real unless the *power wall* problem is adequately addressed. A large potential for energy reduction can be obtained by re-engineering facilities and hardware. Actually, computer manufacturers are already aware and focus hardware development under these restrictions. However, there exist a large collection of software in which considerable energy savings could be achieved via the transformation or redesign of the underlying methods. The difficulty, and probably the reason why these improvements are not yet available, is related to the fact that this type of strategy requires a highly interdisciplinary synergy of different science disciplines.

The ExaFLOP challenge is not just the achievement of that scale of computing performance, but compromises also the goal of attaining it with a reasonable power budget. Thus, the issue concerns both performance and energy efficiency. In this sense, the Top500 [4] and the Green500 [3] ranks account for machines with the best performance and energy efficiency, respectively. The problem with these metrics is that they are mutually independent. For instance, in the top 20 positions of the Green500 there are systems that are near the bottom of the performance heap; and in the upper echelons of the Top500 there are many energy-inefficient systems. In order to visualize performance and energy efficiency trends of both lists, a new way to synthesize these information has been designed in [112]. The *exascalar graph* in Figure 1.1 represents platforms of the Top500/Green500 lists from 2011 to 2013 organized on a double-logarithmic scale of efficiency (MFLOPS/W) and performance (MFLOPS). This representation displays the distances between the different computers considering the Exascale goal of 10^{12} MFLOPS (or 1 EFLOPS) in a 20 MW envelope (or 50,000 MFLOPS/W). Note also the iso-power lines (red diagonal dashed) and iso-exascalar curves (green curves) in the graph. For instance, machines in the iso-exascalar curve $\varepsilon = -2$, are a factor of 100 times away from the Exascale goal; in the same way, the 20 MW iso-power line represents the so-called *power wall*. As can be observed, the power trends of some supercomputers in the figure have almost reached the *power wall* being 100 times away of the Exascale goal.

The growth in energy consumption in modern HPC supercomputers has been stimulated by the increasing density of the hardware components and the decrease of the price for hardware, as exemplified by the Top500 and the Green500 lists. However, the reduction of total energy consumption of an HPC system develops with a slow pace. As a consequence, we notice steadily increasing costs for energy in HPC systems due to constantly increasing power consumption. These issues increase total costs of ownership (TCO) and change the relation between the TCO and the acquisition costs, so that, currently a significant fraction of the TCO is due to the high energy consumption of these installations.

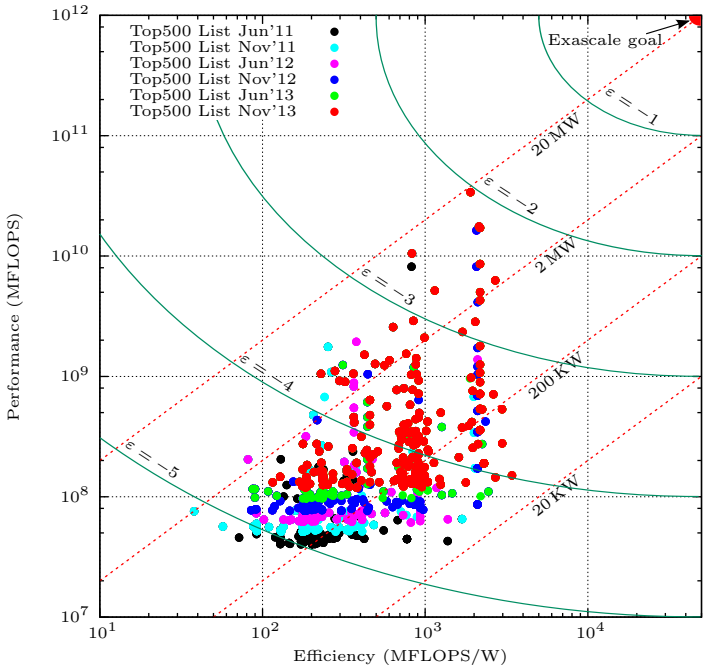


Figure 1.1: Performance-Efficiency scalar graph for the Top500 supercomputers from 2011 to 2013.

The energy consumption generated by all these supercomputing platforms results in electricity costs of several millions of euros. This implies an energy consumption of approximately 1 GW per ExaFLOP with modern regular hardware technology – a value that yields unacceptable electricity and carbon footprints. One approach to reduce this value lies in the development of low energy consumption hardware. Potential candidates are special accelerators hardware like GPUs or the Intel Xeon Phi, DSPs (digital signal processors) or FPGAs (field programmable gate arrays). However, these mechanisms are not easy to be used efficiently and most of these specific hardware solutions need a special adaptation of the application software [50, 54, 58, 65, 90]. While there already exist attempts to improve the energy consumption, most of them focus on acceleration of the code or development of low-consuming devices. This is, indeed, beneficial, but not sufficient to face the challenge of an energy-efficient Exascale computing.

1.2 Motivation and Objectives

We have argued earlier that a few abstraction layers in HPC systems (mostly in hardware) are already subject to energy efficiency considerations. However, there is a layer in the field of software development that is to a great extent energy-oblivious. The novelty, in this sense, is the incorporation of power-saving mechanisms to improve energy usage. This goal comprises a wide range of applications: from mathematical kernels to libraries and application programs.

The research of this dissertation triggers a new approach, where energy consumption is regarded as a top priority. The HPC community has to assume that energy consumption is a major economic and ecological concern nowadays. New HPC-based knowledge has to perform a paradigm shift from high-sustained performance to low energy-to-solution. The motivation of this dissertation is to work with this new paradigm in order to pave the way to efficient software. Specifically, we instantiate this paradigm shift in the field of the matrix computations on multicore and many-core processors.

In this dissertation we investigate energy consumption for the specific domain of dense and sparse linear algebra field. Hardware features that leverage low-power states are used to tune and minimize energy-to-solution of key algorithms. We also develop tools to analyze the power consumption of scientific applications. Power models are also designed to predict energy consumption of key algorithms to reduce total energy dissipation. The ultimate goal of this dissertation is to increase awareness of the *power wall* in high performance computing as well as present new techniques, methodologies and guidelines to easily design energy-efficient scientific applications.

The objectives of this work can be summarized as follows:

- Analysis of energy-saving mechanisms, methodologies and tools available in hardware and software.
- Development of a power-performance profiling and analysis environment for parallel scientific applications.
- Analysis and design of power and energy models for linear algebra operations.
- Design of basic energy-saving techniques for applications that exploit task-level parallelism.
- Validation of the techniques on multicore and hybrid CPU–GPU platforms.

We subsequently describe the state-of-the-art in the topics addressed in this dissertation, and detail the advance that our research brings about.

1.3 State-of-the-Art

HPC centres are substantial consumers of energy, necessary to feed the computational resources and auxiliary systems that have enabled the breakthrough scientific advances achieved during the past few decades. The recent hardware developments of computer architectures, especially in multi/manycore and accelerator technologies, have allowed considerable performance gains in computing and the continuation of historical trends [4]. In consequence, this technology has been rapidly adopted in HPC facilities. Nevertheless, further performance improvements, attained from a substantial increase in the number of cores, is constrained by the aggregated energy budget necessary to feed these large-scale HPC systems. In particular, power consumption has a direct impact on the operation and maintenance costs of these centres, compromising their existence and impairing the installation of new ones. As-of-today, the electricity costs for many HPC centres will exceed the hardware acquisition costs in just a few years. Furthermore, energy consumption results in carbon dioxide emission, a hazard for the environment and public health; and heat, which reduces the reliability and lifetime of hardware components. Therefore, the concerns about the rise of an energy crisis, climate change and fault-tolerance in large-scale systems lead to a very well justified call for energy efficiency in HPC.

At the other end of the spectrum, mobile computing devices are equipped with low-power hardware components to maximize their battery life. This constraint from the embedded and mobile market segments is forcing hardware manufacturers to improve their designs for better energy efficiency. Processor, memory and hard disks nowadays feature low-power modes that allow a trade-off between performance and power by applying energy-friendly techniques such as dynamic voltage and frequency scaling (DVFS) [86] and idle states (e.g., spin down idle disk platters). These tools have to be used carefully, though, since an increase in the application run time may outweigh the power decrease such that the total energy is increased.

In recent years, these energy-saving mechanisms from the mobile market have found their way into server architectures and, thus, the systems installed at large HPC centres are now equipped with power-aware hardware. However, the system software, communication libraries, and application codes in these systems are most often insensitive to power consumption. Therefore, although the ExaFLOP challenge will undoubtedly lead to new ground-breaking scientific discoveries, it is also certain that it calls for greener and more efficient system software, middleware and application algorithms than those in use today.

1.3.1 Energy-efficiency analysis and profile of applications

The development of Exascale systems made it clear that the use of current technologies, algorithmic practices and performance metrics are not adequate. Existing tools in HPC mainly focus on the monitoring and evaluation of performance metrics. In this sense, hardware has a wide range of sensors and measurement devices related to power consumption with varying granularity and informative value. Recently, new frameworks have been developed/adapted from other research areas (e.g. mobile computing) in order to analyze power consumption. Most of these packages focus only on power consumption but are oblivious to performance aspects. Therefore, it is crucial to identify adequate sensors, hardware counters and measurement devices to gain detailed insights about both power consumption and performance.

Optimizing the power consumption of scientific applications requires enhanced monitoring and profiling capabilities. Monitoring power and performance metrics such as node or device power consumption and performance counters is important but the correlation of these measurements with the application itself is necessary in order to understand the mutual interactions. By analyzing the measurements, optimization keystones can be detected and implementation alternatives can be generated (see Figure 1.2). Therefore, it is necessary to create an adequate measurement environment which supports fine-grain measurements of power consumption at component level. Further, visualization tools have to integrate these new metrics in order to correlate them with the application timeline. This allows to characterize different kinds of parallel applications (MPI/OpenMP and P-Thread) with the power consumption during execution, as well as to identify power sinks in the software application.

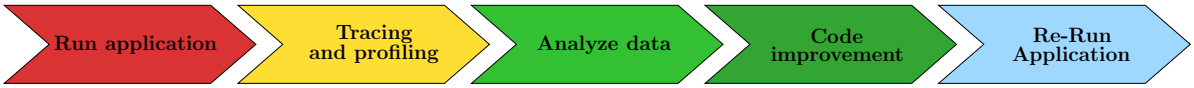


Figure 1.2: Power-performance cycle optimization of scientific applications.

Several works address the aforementioned topic and an excellent survey on hardware, software, and hybrid tools for power profiling is given in [123]. Concretely, hardware solutions for measuring power consumption include: PowerMon2 [33], an internal wattmeter (coupled between the computer's power supply unit and mainboard) for fine-grain measuring of computers samples the power running through the DC lines, offering a basic software interface. PowerPack [61] employs a commercial DC wattmeter from National Instruments connected to the lines coming out from the power supply unit (PSU). This software also performs a number of tests with the purpose of identifying which lines feed different components such as disks, memory, network interface controllers (NICs), processors, etc. This information can be exploited by the user to gain insights on where and how applications consume power. PowerPack exhibits a user-friendly interface, and targets applications running on single-node platforms, though PowerPack's information can be "manually" aggregated for parallel Message Passing Interface (MPI) applications. The HDTrace [83] package,

on the other hand, offers a tracing and simulation environment for the power-performance footprint of MPI programs on a cluster. This software supports MPICH2 and the parallel file system Parallel Virtual File System (PVFS).

Identification of consumption per software component in HPC applications facilitates the optimization of the power consumption, since interferences can be analyzed in detail. Furthermore, the availability of information on software power consumption enables detailed decisions about software behavior, which were not possible before. Using this information, applications, libraries and the operating system can adjust their power consumption to meet performance goals (quality-of-service) or power budgets employing already known interfaces.

1.3.2 Energy-aware metrics and power models

As infrastructure or user demands often impose restrictions on the power draft, new metrics accounting for the key factors of runtime, power draft, and total energy consumption of a system/application combination are essential for optimization. While the GFLOPS/W is the *de facto* standard to measure the energy efficiency of a computing system [3, 4], recent work [34] provides strong evidence that this metric tends to promote power-hungry algorithms that deliver high sustained performance, when in reality the objective should be the reduction of the total energy, preferably along with the minimization of the time-to-solution (TTS).

To avoid this effect, one can rely on the energy-delay product (EDP) [85], which combines both factors, TTS and energy, into a single figure of merit, thus promoting algorithms that are more energy-friendly. The metric $f(\text{TTS}) \cdot energy$ (FTTSE) proposed in [34], progresses further along this line by considering the product between the energy and a function f of TTS. Different cost functions for f thus lead to distinct cases: $f(t) = 1$, with $t = \text{TTS}$, renders FTTSE as simply equivalent to the energy; while FTTSE is analogous to EDP when $f(t) = t$. More interestingly, the linear model $f(t) = \alpha \cdot t$, with α a scalar, allows to weight (penalize) for time increases; and the exponential case $f(t) = e^{\alpha(t-\tau)}$ exerts a stronger pressure when the TTS exceeds a certain threshold τ. Hence, a progress in the current state-of-the-art in energy-aware HPC is needed to shift from using a metric that, at best, offers a limited view of the true energy usage, to a truly energy-aware approach that is at the same time modular and thus easily repeatable and expandable.

In this sense, HPC manufacturers are also pursuing energy models to estimate power consumption in real time. For example, the Intel Sandy Bridge architecture includes the new Running Average Power Limit (RAPL) interface [77] to account for the power consumed by the chip at different levels under a specific design. In this architecture, internal circuitry is able to estimate current power based on a model driven by hardware counters, temperature and leakage data. Recent NVIDIA GPUs are also able to report power usage via the NVIDIA Management Library (NVML) [100].

Many studies for new power and energy models exist. Previous work to model power consumption of benchmarks as well as more general applications leverage hardware counters reflecting processor and memory activity [80, 35, 62]. Other researchers have used processor performance events [37], architectural parameters and parameters drawn from application's characteristics [87]. Among these, the authors in [117] derive an analytic, workload-independent piece-wise linear power model that maps performance counters and temperature to energy usage.

1.3.3 Energy-aware linear algebra

The development of energy-aware implementations of basic numerical linear algebra kernels is a key problem as a significant fraction of the numerical applications running in the HPC centres in

the Top500 list can be decomposed into a few linear algebra operations/problems like, e.g., linear systems of equations or eigenvalue computations. The solution to these problems is currently addressed with existing energy-oblivious libraries developed in the mid 90s, when supercomputers with a single processor per node were the mainstream.

Current HPC libraries for linear algebra exploit the hardware concurrency of multicore processors at the BLAS-level, employing multithreaded implementations of a few basic linear algebra kernels (e.g., the matrix-matrix product or the triangular system solve). For many years, this approach has been successfully exploited by the scientific community, as it provided a clean interface that allowed the development of complex numerical solvers, independent of the underlying target architecture, with performance portable to different architectures. However, with the increase of the number of cores in the systems (e.g., Intel Xeon Phi), this solution has rapidly become suboptimal as exploiting concurrency at the BLAS-level forces an excessive number of thread synchronizations, introducing a non-negligible overhead.

In the recent years, several projects have demonstrated the benefits of extracting parallelism at a higher level, both for dense [26, 40, 110] and sparse [7] linear algebra computations, via runtimes that decompose the operations into fine-grained tasks, and perform an out-of-order dependency-aware scheduling of the tasks. Successful examples of this solution have been provided in projects `libflame` (SuperMatrix) [128], PLASMA (Quark) [106], SMPSs [118], StarPU [119], etc., following ideas/techniques that can be traced back to Cilk [43]. In all these projects, algorithms are statically or dynamically decomposed into a collection of tasks (or kernel operations), identifying the data dependencies among them. The result is a Directed Acyclic Graph (DAG) that captures the dependency information implicit to the algorithm, and which is then passed to a scheduler in charge of issuing tasks to the computational resources. As a result, tasks are executed in the order dictated by data dependencies (data-flow parallelism) instead of the order they appear in the code (control-flow parallelism), which unleashes a richer degree of concurrency.

Unfortunately, as-of-today, these execution environments pursue raw performance as the ultimate value for the end-user but they are completely oblivious to the energy that is consumed to deliver these results. Initial research has investigated the possibility of being more energy-friendly while maintaining the iso-efficiency/iso-scalability of a parallel solver, and the benefits that such an approach can yield. This can be achieved, e.g., by scheduling non critical tasks to slower, low-power cores (in a heterogeneous environment) or by applying DVFS, avoiding busy-waits in the communications, and aggressively promoting idle cores to low-consuming states.

In this context, there exist a number of related investigations to our work. In [56], the authors model a scheduler for clusters that can map tasks and adjust node frequencies, depending on the number of pending jobs. In [126, 91] the authors discuss scheduling of independent tasks (jobs) in a DVFS-enabled processor, while in [66] this technology is used to schedule tasks with dependencies in a multiprocessor setup. The authors of [92] introduce several real-time, energy-aware schedulers for tasks with dependencies. The work in [129] describes a platform that combines real-time mapping with DVFS to reduce energy usage of dependent tasks. The algorithm LPHM in [32] dynamically adjusts the execution time of non critical tasks using DVFS. In [88] new heuristics are proposed for an energy-aware task scheduler in a heterogeneous cluster. In [81] a strategy is employed to stretch or reduce the execution time of non critical jobs. In [115], the authors perform a similar investigation, but frequency is statically tuned at the beginning of the algorithm, and fixed for its complete duration. The authors of [82] also follow the same strategy, with stretch/compress stages, that are iteratively applied until the consumption of power is below a certain threshold. The algorithm LPHEFT is presented in [32] as a means to reduce energy consumption, based on scheduling of idle time-slots (or gaps). In general, there exist a number of works which have analyzed the trade-off between energy and performance enabled by DVFS; see, e.g., [60]. Some

of these tackle the execution of a DAG representing tasks and data dependencies under certain conditions, in most cases reporting the theoretical gains which can be expected from this; see [82].

With the introduction of the CUDA [101] and OpenCL [103] programming standards, GPUs have been increasingly adopted in HPC systems for their affordable price, favorable energy-performance balance and, due to their vast amount of hardware concurrency, the excellent acceleration factors demonstrated for many compute-intensive applications with ample data-parallelism [3, 4]. However, the adoption of these hardware accelerators as a path to reduce the energy-to-solution (ETS) for certain applications has introduced an additional burden on the shoulders of these platforms' programmers.

In particular, this type of hardware accelerators has to be attached to a conventional (multicore) processor, and efficiently programming a heterogeneous platform consisting of one to several multicore processors and multiple GPUs is still a considerable challenge. Furthermore, heterogeneous HPC clusters, with nodes consisting of several general-purpose multicore processors plus one or more GPUs, and a high-performance node interconnect, now include more than one memory address space per node. Inter-node communication in a parallel platform has always been regarded as pure overhead, as it diminishes performance and increases energy consumption. The situation is thus worse for heterogeneous CPU–GPU clusters, as they feature an additional communication channel, a slow PCI-e bus, between the data in the main memory and the hardware accelerator. Therefore, careful mapping/scheduling of the computations is specially needed to reduce the sources of overheads in these platforms. The reason is that, when dealing with these parallel systems, in addition to facing the programming difficulties intrinsic to concurrency, the developer has to cope with multiple issues as the existence of multiple memory address spaces, the different programming models, and now, the energy efficiency.

1.4 Structure of the Document

The manuscript is structured in seven chapters. Chapter 1 reviews the energy challenge in the Exascale HPC era. In addition, it describes the motivation, main goals, and structure of the document.

Chapter 2 describes the basic concepts underlying the BLAS and LAPACK specifications for dense linear algebra problems; reviews two basic execution frameworks, `libflame` and SMPSs, and the algorithms for the Cholesky, LU and QR factorizations. The chapter also offers basic introduction to ILUPACK.

Chapter 3 describes the power-performance analysis framework developed within this dissertation. This basically includes the profiling and tracing tools, the PMLib library, and the integration of the suite in a general framework. To demonstrate its usage, a detailed power and performance analysis of a representative dense linear algebra operation is offered. The chapter also provides a brief overview of the hardware energy-saving mechanisms and describes the different platforms and power measurement devices used in this work.

Chapter 4 introduces a series of power and energy models for linear algebra operations on multithreaded architectures. The main contribution of the chapter comprises the power and energy models, in conjunction with the methodology to gather and assemble the necessary data. Two variants of the general model which follow a simple and a contention-aware approaches are described. These models are completed with an additional case that handles the P-states, and two new specific instances for multi-socket and hybrid CPU–GPU platforms.

Chapter 5 describes our theoretical DVFS-based approaches that allow to reduce energy-consumption of linear algebra operations that exploit parallelism at the task-level. The Slack Reduction

Algorithm and the Race-to-Idle Algorithm are reviewed and simulated using key dense linear algebra algorithms on different hardware architectures in that chapter.

Chapter 6 introduces our energy-aware techniques and describe how to accommodate them into task-parallel runtimes. It analyzes the potential savings of dense linear algebra operations using our energy-aware SuperMatrix runtime, evaluating its practical performance and energy consumption over a representative set of dense linear algebra operations. Furthermore, it presents results of our energy-aware ILUPACK runtime for the solution of sparse linear algebra operations, evaluating theoretical and experimental results and analyzing the impact of the P-/C-states on the time-power-energy balance.

Chapter 7 presents the main conclusions from this research. In addition, it reports the major contributions of the thesis and the publications that have been generated. A few open research lines related to the work are discussed at the end of this chapter.

CHAPTER 2

Linear Algebra Algorithms

The development of complex dense linear applications frequently relies on fundamental building blocks as, for example, the *Basic Linear Algebra Subprograms* (BLAS [97]) and the *Linear Algebra PACKage* (LAPACK [99]). In response to the evolution towards higher degrees of hardware concurrency, the dense linear algebra methods in these libraries need to be parallelized to efficiently exploit SMP (*Symmetric Multi-Processing*) processors and multicore architectures. Current multithreaded libraries operate with tasks that are executed *out-of-order* with the aim of keeping the computation resources fully occupied. Specifically, we introduce two basic libraries and frameworks that use these techniques at a high level: `libflame` [128] and SMPSs [118]. We also describe three canonical matrix decompositions: the Cholesky factorization, the LU factorization with *partial* and *incremental* pivoting, and the *traditional* and *incremental* QR decompositions. All these factorizations are implemented in `libflame` and SMPSs. In addition, we also review the task-parallel implementation of ILUPACK (*Incomplete LU factorization PACKage* [76]), a library for the numerical solution of large sparse linear systems via iterative Krylov-based methods.

The chapter is divided as follows. Section 2.1 describes the basic concepts and nomenclature behind the BLAS and LAPACK specifications for dense linear algebra problems. Section 2.2 reviews two basic execution and framework environments: `libflame` and SMPSs. Sections 2.3, 2.4, and 2.5 describe the algorithms for the Cholesky, LU and QR factorizations, respectively. A basic introduction to ILUPACK is offered in Section 2.6.

2.1 Dense Linear Algebra

2.1.1 BLAS: Basic Linear Algebra Subprograms

Several key dense algebra problems, such as the solution of systems of linear equations or eigenvalue problems, arise in a wide variety of scientific and engineering applications. Chemical simulations, automatic control or integrated circuit design are just three examples in which dense algebra operations usually conform the computationally most expensive part.

Furthermore, a collection of basic operations frequently appear during the solution of these problems, such as, e.g., the scalar product of two vectors, the solution of a triangular linear system, or the product of two matrices. These basic linear algebra routines are grouped under the name

BLAS and they were identified in a joint effort leaded by experts from diverse areas, so that the final specification of BLAS covered the basic requirements that appear in many fields of science.

The BLAS specification was originally a trade-off between functionality and simplicity. The number of routines and their parameters were designed to be reasonable. Simultaneously, the functionality was intended to be as rich as possible, covering those routines that often appear in complex problems. An illustrative example of the flexibility of the specification is the representation of a vector: the elements of the vector do not have to be stored contiguously in memory; instead, the corresponding routines provide a parameter to define the physical separation between two logically consecutive elements of the vector.

Since the first definition of the specification, BLAS has been of great relevance in the solution of linear algebra problems. The reliability, flexibility, and efficiency of the existing implementations allowed the emergence of other libraries that made internal usage of the BLAS. In addition, there are other advantages that yield the use of BLAS so appealing:

- Code legibility: the names of the routines reflect their internal functionality. This standardization makes code simpler and more readable.
- Portability: by providing a well-defined specification, the migration of codes built upon BLAS to other platforms is straight forward. Given a tuned implementation for the target machine, the ported implementations will stay optimized and the resulting codes will remain highly efficient.
- Documentation: there is a rich documentation available for each BLAS routine.

There is a generic implementation of BLAS available since its original definition [97]. This reference (or legacy) implementation offers the full functionality of the specification, but without optimizations specific to a particular hardware. However, the real value of BLAS lies on the tuned implementations developed for different hardware architectures. Since the publication of the specification, the development of implementations adapted to each hardware architecture has been a task for either processor manufacturers or the programming community. Today, vendor-specific implementations are usually employed to demonstrate the full potential of a specific processor. There exist proprietary implementations for general-purpose multicore processors from AMD (ACML [22]), Intel (MKL [79]) and IBM (ESSL [73]), as well as for specific-purpose architectures, such as Nvidia Cublas for the Nvidia graphics processors. In general, the code for each routine in those implementations is designed to make optimal use of the resources of the underlying architecture. Independent third-party implementations, such as GotoBLAS [64] or ATLAS [125], also provide optimized implementations for general-purpose processors.

The BLAS were initially implemented using Fortran thought there also exists an implementation in C. In many cases, the use of assembly language allows fine-grained optimizations that extract the full potential of the architecture.

BLAS levels

The development of BLAS is closely bound to the evolution of hardware. In the early 1970s the most common HPC platforms were equipped with vector processors. With those architectures in mind, BLAS was designed as a group of basic operations on vectors: the Level-1 BLAS, or simply BLAS-1. The ultimate goal of the specification of BLAS was to motivate processor engineers to develop fully optimized versions of their implementations following the standard specification.

In 1987, the BLAS specification was improved with a set of routines for matrix-vector operations, usually known as Level-2 BLAS or BLAS-2. Both the number of floating-point arithmetic operations

and the amount of data involved in these routines are of quadratic order, and a generic BLAS-2 implementation was published after the specification. One of the most remarkable observations of these early implementations was the adoption of column-wise storage for data in matrices.

The increasing gap between processor and main memory speeds [122] resulted in the appearance of architectures with multiple levels of cache memory, yielding a hierarchical organization of the system memory. With the popularization of those architectures, it was accepted that libraries built on top of BLAS-1 and BLAS-2 routines would never attain high performance when ported to the new architectures, since BLAS-1 and BLAS-2 are by definition memory-bound operations. Thus, the performance of the BLAS-1 and BLAS-2 routines is limited by the speed at which the memory subsystem can provide data to the execution path.

The third level of BLAS (Level-3 BLAS, or BLAS-3) was defined in 1989 in response to the aforementioned problems. The specification proposed a set of operations featuring a cubic number of floating-point arithmetic operations on a quadratic amount of data. This difference between the number of calculations and memory accesses allows a better exploitation of the principle of *locality* in architectures with a hierarchical memory via carefully designed "blocked" algorithms. In practice, these algorithms hide the memory latency and offer a performance close to the theoretical peak of the processor. Blocked algorithms partition the matrix into sub-matrices (or blocks), grouping memory accesses, and increasing the locality of reference and data reuse. By exploiting this scheme, the possibility of finding data in a closer (and faster) level of the memory hierarchy is greater and the memory access penalty is reduced. For this reason blocked algorithms are also known as *algorithms-by-blocks*.

To sum up, BLAS is divided into three levels:

- Level 1: The number of operations and the amount of data increase linearly with the size of the problem.
- Level 2: The number of operations and the amount of data increase quadratically with the size of the problem.
- Level 3: The number of operations increases cubically with the size of the problem, while the amount of data increases only quadratically.

From the perspective of performance, the main reason for this classification is the ratio between the amount of data and the number of operations performed on them. This ratio is critical in architectures with a hierarchical memory system, as those that are mainstream today.

With the emergence of modern shared-memory multicore and many-core architectures, the development of parallel implementations of BLAS has received further attention, and significant efforts have been undertaken to adapt BLAS to these architectures. The advantage of BLAS-3 is still more dramatic in these scenarios, where the memory bandwidth becomes a more challenging bottleneck since it is shared by multiple processors.

From the performance viewpoint of the parallel implementations, we can conclude that:

- The performance of BLAS-1 and BLAS-2 is dramatically limited by the pace at which memory can feed the data to the processors.
- BLAS-3 is more efficient since more calculations can be performed per memory access, attaining performances near the peak of the processor and near-optimal speedups for many operations. For this reason, the routines of BLAS-3 usually offer higher degrees of parallel performance.

Routine	Operation	Comments	FLOPS
xGEMM	$C := \alpha\ op(A)\ op(B) + \beta C$	$op(X) = X, X^T, X^H, C$ is $m \times n$	$2mnk$
xSYMM	$C := \alpha AB + \beta C$ $C := \alpha BA + \beta C$	C is $m \times n$, $A = A^T$	$2m^2n$ $2mn^2$
xSYRK	$C := \alpha AA^T + \beta C$ $C := \alpha A^TA + \beta C$	$C = C^T$ is $n \times n$	n^2k
xSYR2K	$C := \alpha AB^T + \alpha BA^T + \beta C$ $C := \alpha A^TB + \alpha B^TA + \beta C$	$C = C^T$ is $n \times n$	$2n^2k$
xTRMM	$C := \alpha\ op(A)C$ $C := \alpha\ Cop(A)$	$op(A) = A, A^T, A^H, C$ is $m \times n$	nm^2 mn^2
xTRSM	$C := \alpha\ op(A^{-1})C$ $C := \alpha\ Cop(A^{-1})$	$op(A) = A, A^T, A^H, C$ is $m \times n$	nm^2 mn^2

Table 2.1: Functionality and number of floating-point operations of the studied BLAS routines.

Overview of the BLAS-3 operations

The BLAS-3 basically targets matrix-matrix operations and its functionality was designed to be limited. For example, no routines for matrix factorizations are included, as they are supported by higher-level libraries, such as LAPACK, which implements blocked algorithms for that purpose making use of routines from BLAS-3 whenever possible. Instead, the BLAS-3 are intended to be a set of *basic* matrix algebra operations from which the developer is capable of implementing more complex routines. Specifically, BLAS-3 can operate with matrices, matrix blocks and their transposes.

Table 2.1 summarizes the BLAS-3 routines used in this dissertation operating in real arithmetic. The names of the routines follow the conventions of the rest of the BLAS specification, with the first character denoting the data type of the matrix (e.g., `S` for single precision, `D` for double precision), the second and third characters denoting the type of matrix involved (e.g., `GE`, `SY` or `TR` for general, symmetric or triangular matrices, respectively) and the rest of the characters denoting the type of operation (`MM` for matrix-matrix product; `RK` and `R2K` for rank-k and rank-$2k$ updates of a symmetric matrix, respectively; and `SM` for the solution of a triangular system of linear equations with multiple right-hand sides).

2.1.2 LAPACK: Linear Algebra PACKage

The optimization of the routines in BLAS is justified as they are basic building blocks to construct libraries that perform more complex linear algebra operations. One of the most used libraries in this sense is LAPACK [99].

LAPACK includes routines to solve fundamental linear algebra problems and represents the state-of-the-art in numerical methods on dense matrices. Alike BLAS, LAPACK offers support for both dense and band matrices but, while BLAS is focused on the solution of basic linear algebra operations, LAPACK tackles more complex problems as, for example, linear systems, linear least-squares problems or eigenvalue and singular value problems.

LAPACK was the result of a project born at the end of the 80s. The main goal was to obtain a library with the same functionality as LINPACK [52] and EISPACK [98] but with improved performance. Those libraries, designed for vector processors, do not offer reasonable performance on current high performance processors, with segmented pipelines and complex memory hierarchies. This inefficiency is mainly due to the use of BLAS-1, which does not exploit the locality of reference, resulting in a sub-optimal usage of the memory subsystem. As a consequence, the routines in LINPACK and EISPACK spend most of the time in data movements from/to memory, with the subsequent penalty on the performance [39].

The performance boost attained by the routines in LAPACK is mainly due to two main reasons:

- The integration into the library of new algorithms that did not previously exist.
- The redesign of existing and development of new algorithms to make an efficient use of BLAS.

The legacy implementation of LAPACK is public domain [99], and includes drivers to test and time the routines. As with BLAS, some hardware manufacturers have implemented specific versions of LAPACK tuned for their architectures; however, the improvements introduced in these implementations are usually not as important as those integrated in BLAS. As the performance of many implementations relies on the performance of the underlying BLAS implementation, high-level modifications, such as the selection of the block size, are the most common tweaks in the proprietary versions of LAPACK.

For multicore architectures, LAPACK extracts the parallelism by invoking a parallel (multi-threaded) version of BLAS. In other words, the routines in LAPACK do not include any kind of explicit parallelism in their codes, but rely on a multithreaded implementation of BLAS for this purpose.

LAPACK and BLAS

The possibility of using BLAS-3 routines led to a redesign of many algorithms implemented by LAPACK to work with blocks or sub-matrices [53, 51]. By leveraging these algorithms-by-blocks, many operations can be cast in terms of the most efficient routines in BLAS, and the possibilities of parallelization are improved in two ways: the internal parallelization of individual block operations and the possibility of processing several blocks in parallel [44]. LAPACK contains routines based on BLAS-3, but also scalar or unblocked codes based on BLAS-1 and BLAS-2 for some matrix operations. For example, the Cholesky factorization of a dense symmetric positive definite matrix is implemented both as a blocked routine (`potrf`) and as an unblocked routine (`potf2`).

The internal use of BLAS offers the advantages of portability, code legibility and concurrency, already discussed in Section 2.1.1.

In summary, the efficiency of LAPACK depends mainly on two factors: first, the efficiency of the underlying BLAS implementations; and second, the amount of FLOPS performed in terms of BLAS-3 routines.

2.2 Dense Linear Libraries and Tools

In our work we have also used two basic software packages that exploit task-level parallelism intrinsic to several dense linear algebra operations. First, we review `libflame` and its runtime SuperMatrix, used to demonstrate the capabilities of our proposed energy-aware techniques presented in Chapter 6. Next, we revisit the SMPSs framework and runtime, that was used to explore the energy and power models proposed in Chapter 4.

2.2.1 The `libflame` library

The FLAME (*Formal Linear Algebra Methods Environment*) project is a collaborative effort between The University of Texas at Austin and the Universitat Jaume I de Castellón that offers a unique methodology, notation, tools, and a set of application programming interfaces (APIs) to easily derive dense linear algebra algorithms and transform these into code. The result of this project is the `libflame` library.

The primary purpose of `libflame` is to provide the scientific and numerical computing communities with a modern, high-performance dense linear algebra library that is extensible, easy to use, and is available under an open source license. Seasoned users within scientific and numerical computing circles will quickly recognize the general set of functionality targeted by `libflame`. It provides a framework for developing dense linear algebra solutions, but also a ready-made library that is, by almost any metric, easier to use and offers competitive (and in many cases superior) real-world performance when compared to the more traditional BLAS and LAPACK libraries.

Sequential performance via optimized BLAS. Traditional libraries like LAPACK are layered upon the BLAS for portable performance. In its simplest mode, `libflame` is merely an alternative way of programming families of algorithms for operations, such as the LU, QR, and Cholesky factorizations, still in terms of traditional BLAS operations and linked to traditional BLAS libraries.

Traditional parallelism via multithreaded kernels. With the advent of parallel computers like symmetric multiprocessors (SMPs) and multicore architectures, the simplest approach to parallelize libraries such as LAPACK and `libflame` is to link to multithreaded BLAS kernels so that the parallel implementations remain identical to the original sequential code.

Scheduling algorithms-by-blocks to multiple threads. The drawback of extracting parallelism only via multithreaded BLAS are numerous: *i)* each call to a BLAS operation ends with a synchronization of the threads (so-called fork-and-join parallelism); *ii)* the factorization of the current panel represents an operation in the critical path during which there is a reduced opportunity for parallelism; *iii)* scheduling the computations out-of-order to increase concurrency is difficult without greatly complicating the code. `libflame` presents some abstractions that alleviate these problems:

- The API supports storage of matrices by blocks. While traditional linear algebra libraries enforce column-major storage to map matrices to memory, the object-based FLAME/C API allows elements in a matrix themselves to be descriptors of matrices, thus providing a convenient way of expressing matrices that are hierarchically stored by blocks.
- It supports algorithms-by-blocks, which view each element of a matrix as a block, and expresses the computation to be performed in terms of operations between those blocks.
- In some cases, new algorithms had to be invented to fit the algorithms-by-blocks concept. For example, the LU factorization with partial pivoting and QR factorization via Householder transformations in their traditional incarnations operate with columns of blocks in order to identify a pivot or compute a Householder reflector. The concepts of incremental pivoting in the LU factorization and the incremental QR factorization overcome this bottleneck.
- When blocks are viewed as the units of data and operations with blocks as the units of computation, the concurrency is exposed by expressing the algorithm as a directed acyclic graph (DAG) of sub-operations. However, abstractions within the library allow a nearly identical implementation to instead build a DAG as it executes, deferring computation until the DAG is built. This provides the opportunity for sub-operations to be dispatched to threads for parallel execution even when many dependencies exist between subproblems. Thus, the `libflame` algorithms need not change as one moves from a sequential to a shared-memory parallel environment.

- It provides a run-time system, named SuperMatrix, to analyze and dynamically schedule the suboperations captured in algorithms-by-blocks. This system implements in software techniques such as out-of-order execution incorporated at the hardware level in superscalar processors. Altogether, these abstractions enable algorithms with many dependencies to be elegantly and efficiently scheduled in order to exploit thread-level parallelism.

Exploiting hardware accelerators. A recent development in high-performance computing has been the arrival of hardware accelerators like the general-purpose graphics processing units (GPUs) or the Intel Xeon Phi. As part of a prototype extension of `libflame`, the same mechanism used to target shared memory parallel architectures allows to reach, almost effortlessly, stunning performance on systems with multiple hardware accelerators [109].

2.2.2 The SuperMatrix runtime

The SuperMatrix runtime was designed from its inception for the execution of dense linear algebra operations. This runtime follows the methodology advocated in the FLAME project [59], which patronizes a separation of concerns between the derivation of new algorithms for dense linear algebra operations, and their practical coding (implementation) and high-performance execution on a platform. SuperMatrix orchestrates a seamless, task-parallel execution of the full functionality of the `libflame` dense linear algebra library [128] on a range of platforms, including multicore desktop servers [110], heterogeneous CPU–GPU systems [109], and small-scale clusters [75].

Operation. The parallel execution performed by the runtime is composed by the following phases. After the identification of tasks and dependencies, and the constitution of a logical directed acyclic graph (DAG), SuperMatrix proceeds to execute the computations represented by the DAG. For that purpose, the runtime spawns a collection of worker threads that poll a queue of tasks *ready* for execution. When SuperMatrix runs on a multicore processor, idle threads prompt the ready list for work. When a thread acquires a task, it executes the corresponding kernel in the associated processor core. Upon completion, the thread inspects the tasks in the global work queue, moving them to the ready queue in case all their dependencies are now satisfied.

Let us expose the SuperMatrix runtime operation using an example of the LU factorization with partial pivoting. Consider the LU factorization of a (nonsingular) matrix $A \in \mathbb{R}^{n \times n}$, which computes the decomposition $A = LU$, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $U \in \mathbb{R}^{n \times n}$ is upper triangular and, for simplicity, let us neglect pivoting during the presentation. Figure 2.1 (left) presents a right-looking blocked algorithm for this factorization using the FLAME notation [36].

SuperMatrix (like many other high performance runtimes for dense linear algebra) starts from a (sequential) blocked algorithm of the target matrix operation (see Figure 2.1), to obtain a task parallel data-flow execution. For this purpose, SuperMatrix first decomposes the algorithm/operation into a number of suboperations (*tasks*) of a certain granularity, while simultaneously identifying all dependencies among these. In the case of SuperMatrix and dense linear algebra operations, this can be done, e.g., taking into account only the order in which tasks appear in the algorithm as well as the operands that each task reads (inputs), writes (outputs), or reads/writes (inputs/outputs).

For example, consider an $n \times n$ matrix A composed of $s \times s = 4 \times 4$ blocks of dimension $b \times b$ each (i.e., $n = s \cdot b$). The symbolic result from the above process is the directed acyclic graph (DAG) of dependencies in Figure 2.2, where LU(k) stands for the factorization of the k-th panel (column block), an TRSM(k,j) and GEMM(k,j) refer, respectively, to the triangular system solve and the matrix-matrix update of the j-th panel with respect to the factorization of panel k (see Figure 2.1). In this case, note that the factorization of the first panel, LU(0), yields a result that

Algorithm: $A := \text{LUP\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right) := \text{LUP\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12} \quad (\text{TRSM})$$

$$A_{22} := A_{22} - A_{21} A_{12} \quad (\text{GEMM})$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 2.1: Blocked algorithm for the LU factorization.

is necessary for tasks TRSM(0,1), TRSM(0,2), TRSM(0,3); and also note how this is captured in the DAG by arcs (dependencies) between the corresponding nodes (tasks). Thus, these dependencies state that TRSM(0,1)–TRSM(0,3) cannot be executed till LU(0) is completed, but also that these three triangular solves can be performed in any order.

In summary, the DAG associated with a given algorithm dictates different "orderings" in which the tasks (suboperations) can be correctly computed, and SuperMatrix leverages this information to produce an out-of-order, data-flow schedule of the DAG and a concurrent execution of the tasks.

SuperMatrix for multithreaded platforms

The conventional approach to execute blocked algorithms in parallel is to employ a high-quality implementation of a numerical library, like LAPACK [24] or `libflame` [128], and rely on a mul-

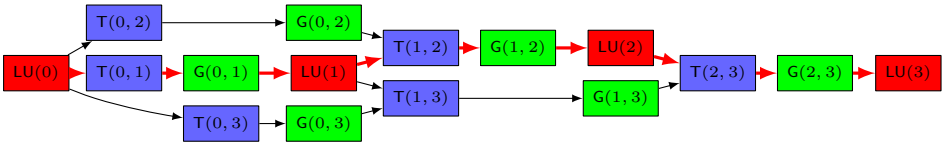


Figure 2.2: DAG with the tasks/data dependencies for the LU factorization with partial pivoting of a matrix consisting of 4×4 blocks. Red arrows identify the critical path of the algorithm.

tithreaded implementation of BLAS to exploit the hardware concurrency. Alternatively, when applied to parallelize dense linear algebra operations on current multithreaded architectures, exploiting task-parallelism with a finer granularity has been recently reported as a superior approach to exploit the hardware concurrency of current multithreaded architectures [26, 40, 110]. In order to exploit the task-parallelism intrinsic to the factorization algorithms, we leverage the SuperMatrix runtime, which conforms the layer that transparently adapts `libflame` to multithreaded architectures.

The version of SuperMatrix for heterogeneous CPU–GPU platforms [109] commits one control thread per GPU (device) of the target platform. These threads run each on a different (CPU) core of the host, and *i)* update the dependence queues; *ii)* guide the associated accelerator by carrying out the necessary data transfers and dispatching tasks for execution there; and *iii)* execute computational work that is not suited to the GPU. For example, in the LU factorization with partial pivoting, the panel factorizations are performed by the CPU cores, as this type of operations requires a fine control that renders them inappropriate for the GPU. The triangular linear system solutions (TRSM) and matrix-matrix updates (GEMM) of the remaining blocks, on the other hand, are performed in the GPUs.

It is interesting to remark that, for the particular case of parallel platforms with multiple memory address spaces, this version of the runtime [109] introduces two communication-reducing techniques: *i)* the workload is partitioned statically among the computational resources following a cyclic block data layout; *ii)* and the memory of each GPU is viewed as a local, fully-associative cache, and data coherence is preserved using *write-invalidate* and *write-back* protocols [69]. The outcome is a significant reduction of the volume of communications between CPU and GPU, diminishing the constraint imposed by the slow PCI-e bus. More details on the operation/implementation of the SuperMatrix runtime for multicore processors and multi-GPU platforms can be found in [110] and [109], respectively.

To close this brief introduction, note that it is precisely the decomposition of the operation into a discrete number of tasks and the introduction of a runtime that controls the scheduling of tasks to the computational resources (cores or GPUs) which permits the design of an energy-aware execution, as described later in this dissertation.

Improvements to the SuperMatrix runtime

Some extensions and techniques for the hybrid CPU–GPU SuperMatrix runtime have been proposed in order to improve performance [16]. To illustrate the different versions of the runtime, we use a multicore Intel platform with two quad-core Intel Xeon 5440. Attached to the platform, via a PCI Express 2.0 bus, there is a system consisting of four NVIDIA Tesla C2050 GPUs (Fermi); hereafter we refer this platform as TESLA2.

The original design of the SuperMatrix runtime for heterogeneous CPU–GPU architectures supported two basic execution modes: multicore and multi-GPU modes, depending on the available hardware resources. In the multicore mode, one worker thread is bound to a unique CPU core, executing and collaborating in the management of shared lists of ready and pending tasks. In multi-GPU mode, the runtime task devotes one control thread running on a CPU core, to instruct each GPU during the complete parallel execution. As the computation advances, ready tasks are executed in the GPUs using a specific implementation of the kernels for these devices. When a kernel is not appropriate for the GPU, the computation is carried out in the associated CPU core, and no data transfers are performed unless transfers are strictly required to maintain data consistency.

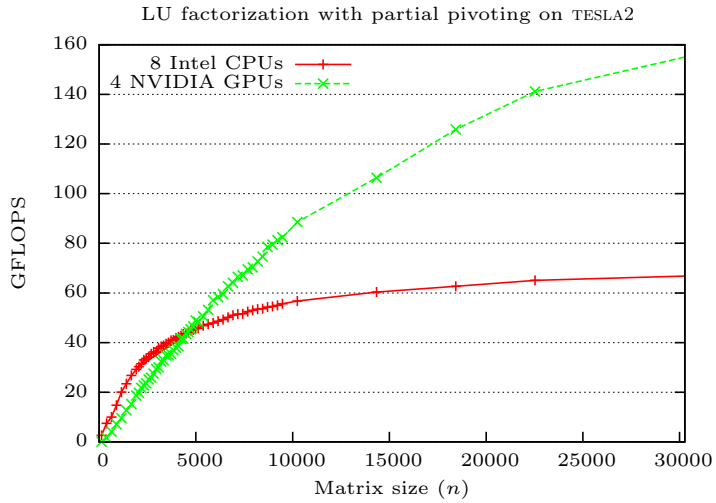


Figure 2.3: Performance of the LU factorization with partial pivoting on TESLA2, using 8 CPU cores (multicore mode) and 4 GPUs (multi-GPU mode).

Let us next illustrate the performance of the original modes of the SuperMatrix scheduler using the LU factorization with partial pivoting in terms of GFLOPS (i.e., billions of FLOPS), using 8 CPU cores (multicore mode) and 4 GPUs (multi-GPU mode) of TESLA2 (see Figure 2.3).

From this initial experiment it is possible to extract some conclusions. For small matrices the multicore mode outperforms the performance of the GPU-based alternative. This is a known result [28, 109] as GPUs need large volumes of computation in order to hide PCI-e data transfer overheads. However, for medium-size matrices both the multicore and multi-GPU modes deliver similar GFLOPS rates. This behavior, in which equivalent performance for different configuration modes appears, is common to other dense linear algebra operations. When dealing with large matrices, the multi-GPU setup clearly outperforms the multicore counterpart, basically due to the much higher hardware concurrency of the GPUs.

Tuning the scheduler. In past work, a number of heuristics have been applied to tune the performance of runtime task schedulers by reducing idle time and/or minimizing data movement among memory spaces. These techniques include data affinity [109], caching [42], work-stealing [74], and task renaming [105]. In the extension of the SuperMatrix runtime for hybrid CPU–GPU architectures, decisions taken at runtime address performance as well. From the experiment performed above, there exists different resource combinations that report optimal results for each problem size. To leverage this, the runtime system could modify, at execution time, the number of each type of computational resources (CPU or GPU) that are devoted to the actual task computation.

Leveraging Full Hardware Concurrency. In the original SuperMatrix implementation, the execution of tasks was performed either by the CPU cores (multicore mode) or by the GPUs (multi-GPU mode). As an exception, in the multi-GPU mode a few types of tasks could be executed on the CPU cores, due to their special properties. While this occurred, though, the corresponding GPUs remained stalled, waiting for the completion of the task. This improvement considers the GPUs and all the CPU cores as potential workers. In this case, each task type is bound to two different kernel instances, one for the GPU and one for the CPU. Depending on the type of thread a task is mapped to, the corresponding kernel instance is invoked.

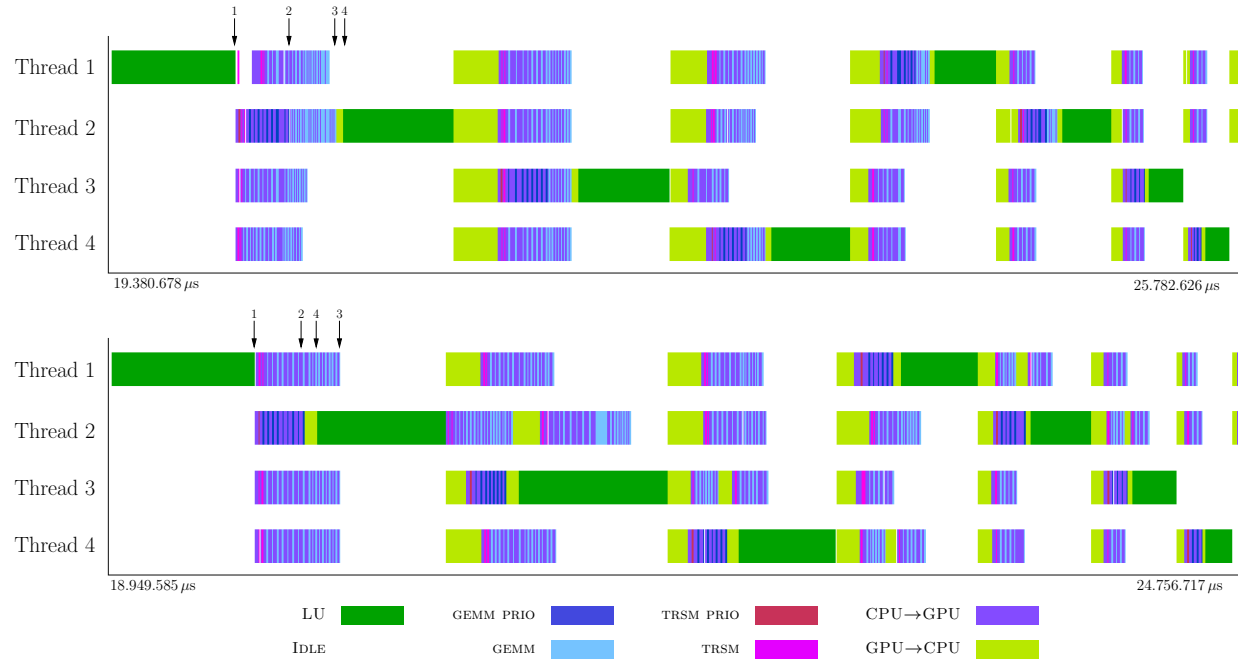


Figure 2.4: Traces of the execution of the LU factorization with partial pivoting of a matrix of dimension $n = 10{,}240$, with $b = 1{,}024$ using 4 GPUs of TESLA2, without priority tasks (top) and with priority tasks (bottom). Selected execution points: 1: End of LU(0); 2: End of UPDATE (0,1); 3: End of UPDATE $(0,s-1)$; and, 4: Start of LU(1).

Scheduling Critical Tasks. Let us analyze next in detail the task scheduling of the LU factorization with partial pivoting for a matrix, e.g., of size $n = 10{,}240$, with block size $b = 1{,}024$, using the multi-GPU mode on 4 GPUs. Given these dimensions, Algorithm 2 partitions the matrix into $s = 10{,}240/1{,}024 = 10$ panels. At each iteration $k = 0, 1, \ldots, s-1$, the algorithm proceeds by initially decomposing the k-th panel of the input matrix (LU(k)); and next updating the trailing submatrix panel-wise with respect to the factorization of this panel, which is performed as a sequence of triangular system solves and matrix-matrix updates (tasks TRSM(k,j) and GEMM(k,j), respectively, with $j = k+1, k+2, \ldots, s-1$). For simplicity, hereafter we refer to the combined application of TRSM and GEMM to the j-th panel as UPDATE(k,j).

Figure 2.4 (top) shows a trace of the execution of this LU factorization governed by the original SuperMatrix scheduler. Note how, in the operation of the original runtime, the factorization LU(k+1) does not commence till the update of the full trailing submatrix with respect to the factorization of the previous panel has been completed. An inspection of the order in which tasks are executed there (see the instants marked with numbers 1 to 4 in the trace) reveals that task LU(1) (execution point 4) does not proceed until the complete update of the trailing submatrix UPDATE$(0,s-1)$ (execution point 3) is completed.

However, the factorization of the current panel and the update of the first panel of the trailing submatrix both lie on the critical path of the DAG (see Figure 2.2) and, therefore, their execution should proceed as soon as possible. Indeed, task LU$(k+1)$ could effectively start as soon as task UPDATE(k,k+1) is completed, since the dependencies determine that it is unnecessary to wait for the update of all remaining panels in the trailing submatrix: UPDATE$(k,k+2)$, ..., UPDATE$(k,s-1)$. Thus, the goal of this optimization is to enforce a fast execution of those tasks in the critical

path, that in practice yields an overlapped execution of LU($k+1$) with UPDATE(k,$k+2$), ..., UPDATE(k,$s-1$).

To accomplish these goals, tasks in the critical path receive a different treatment in the enhanced version of the SuperMatrix scheduler. Specifically, *i)* critical (or priority) tasks are executed as soon as possible to avoid unnecessary stalls; and *ii)* they are mapped to the fastest computational resource (CPU or GPU) available. Generally, tasks that lie on the critical path are, therefore, signaled as priority tasks.

Once priority tasks are identified and marked in the DAG, the runtime remains in charge of performing the most adequate action when a ready critical task is encountered. In principle, the original implementation of SuperMatrix controls a single ready queue containing all tasks with their data dependencies satisfied. In the version enhanced with priorities, the scheduler is modified to introduce an additional *priority queue* (or one priority queue per thread in case data affinity is used).

The performance impact of the improvements introduced in the new runtime is reported in Figure 2.5. There, we offer a comparison between the efficiency of the original version of SuperMatrix and that of the new runtime using the multicore mode and the multi-GPU mode of the runtime scheduler. The results show that the performance improvement is especially remarkable for large matrices. For the multicore mode, the acceleration varies between 10 % and 12 %, while for the multi-GPU mode, the speedups range from 20 % to 25 % for the largest tested matrices

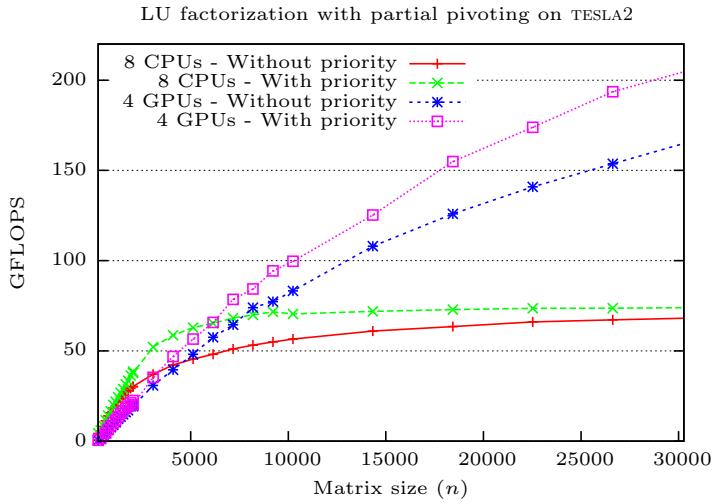


Figure 2.5: Impact of the use of priority tasks on the performance of the LU factorization with partial pivoting on TESLA2, using 8 CPU cores (multicore mode) and 4 GPUs (multi-GPU mode).

2.2.3 The SMP Superscalar framework

The SMP Superscalar (SMPSs) framework [118] consists of a source-to-source compiler and a runtime library. The supported programming model allows to develop sequential applications and the framework is able to exploit the existing concurrency and to use the different cores of a multicore or SMP performing an automatic parallelization at execution time.

SMPSs addresses the automatic exploitation of the functional parallelism in a sequential program on multicore and SMP environments. SMPSs prioritizes the portability, simplicity and flex-

ibility of the programming model. In particular, using a simple annotation of the source code, a source-to-source compiler generates the necessary code in common C, and a runtime library exploits the existing parallelism by building a task dependency graph at execution time. The runtime takes care of scheduling the tasks and handling the associated data.

Overview. The SMPSs project focuses on multicore and SMP architectures in general. The programming model allows programmers to write sequential applications and the framework is able to exploit the existing concurrency. The programmer employs annotations in the form of C macros alike those in OpenMP.

The SMPSs runtime builds a DAG where each node represents an instance of an annotated function and the edges between nodes denote data dependencies. Using the information in this graph, the runtime schedules independent nodes for execution in different processors. Techniques like data dependency analysis, data renaming and data locality exploitation are applied to increase performance of the applications. In summary SMPSs offers tools that propose a portable, flexible and high-level programming model for multicores and SMPSs.

Task Based Programming. SMPSs is a programming environment for parallel applications based on function-level parallelism. In this model, the programmer selects a series of functions called tasks that will run in parallel, and these functions are treated by the runtime as the unit of parallel computation.

Tasks are defined with *pragma* annotations before their function definition. The annotation indicates that the following function is a *task* and specifies the directionality of each task parameter. For example, the syntax of a task is the following:

```
#pragma css task input(input parameter list) \
                 output(output parameter list) \
                 inout(input and output parameter list)
   function definition
```

Partial Synchronization Points. SMPSs can only handle inter-task related data dependencies, but not dependencies with non-task code. For this reason, synchronization points or barriers may be needed. After a synchronization point, inline code is guaranteed to have all dependencies with the specified data resolved. In this sense, synchronization points are partial, since they wait for specific values instead of waiting for all tasks to finish. The basic syntax of the synchronization point annotation is the following:

```
#pragma css wait on(addresses of the variables)
```

Data dependency control. Tasks in a program operate on data that is generated and consumed from task to task. These relations define a certain control flow that must be respected in order to execute the tasks and obtain the same results as in the corresponding sequential execution. SMPs guarantees the consistency of the results by respecting the data dependencies between tasks. Dependency information is generated and kept in a task dependency graph at runtime.

The runtime is structured as a main thread that runs the *non-task* code and populates the graph, and a series of worker threads that consume and execute the tasks from the graph. As tasks are executed, their output parameters become available to the tasks that are dependent on them.

Data dependency reduction. Dependencies are one of the factors that determine how much parallelism can be extracted from of an application. SMPSs tries to reduce dependencies between instructions by performing register renaming at runtime.

The renaming technique consists of storing temporary definitions of a program variable into a temporary storage. That is, if a task writes to an array, renaming can replace that array by a temporary one and redirect all the following reads of that definition to the temporary array. This effectively eliminates all WaR[1] and WaW[2] dependencies.

Workload distribution. One of the goals of SMPSs is to provide good performance. In these sense, the scheduling algorithm is designed according to three principles: first, maximize parallelism; second, make task execution fast on the processors; and third, do not exceed the benefits of a simpler scheduling algorithm by applying a computationally expensive one.

SMPSs incorporates advanced techniques to exploit data locality by taking advantage of the information in the graph. Other techniques allow to advance tasks in the critical path and to steal work from other threads when their ready lists become empty; these enhancements improve load-balancing of the workload.

Tracing. Measuring is key to improve the performance. For this reason, SMPSs incorporates an integrated tracing facility. Applications compiled with the "tracing enabled" option record a series of events during their execution. These events are dumped into a trace file at the end of the execution for analysis. The file is formatted so that it can be processed with `Paraver` tool [104].

SMPSs in linear algebra

Since the SMPSs runtime can be used for any kind of parallel applications that exploit parallelism at the task-level, it can also be applied in the linear algebra domain. In this case, the framework for the semi-automatic detection of dependencies and seamless parallelization on multicore architectures proceeds internally using the same principles as the SuperMatrix runtime. As an example, the code in Listing 2.1 displays a simplified C code that computes the Cholesky factorization of an `n`×`n` matrix `A`, stored in column-major order, and with column leading dimension `Alda`. The code invokes high performance implementations of the *kernels* `dpotrf` (Cholesky factorization), `dtrsm` (triangular solve), `dgemm` (matrix-matrix product), and `dsyrk` (symmetric rank−b update) from LAPACK and BLAS. The lines starting with "`#pragma css task`" are the annotations the programmer needs to add to exploit task-parallelism using SMPSs.

2.3 The Cholesky Factorization

One of the most common strategies to solve a linear system of equations commences with the factorization of the coefficient matrix as the product of two triangular matrices. These factors are used in a subsequent stage to obtain the solution of the original system by solving the resulting

[1] Write-after-Read.
[2] Write-after-Write.

```
#define A_ref(i,j) A[((j)-1)*Alda+((i)-1)]

void dpotrf_smpss( int n, int b, double *A, int Alda, int *info )
{
  // Declaration of variables
  // ...
  for (k=1; k<=n; k+=b) {
    // Cholesky factorization
    dpotrf_u( b, &A_ref(k,k), Alda, info );

    // Triangular system solve
    for( j=k+b; k<=n; k+=b )
        dtrsm_lutn( b, &A_ref( k, k ), &A_ref( k, j ), Alda );

    // Update trailing submatrix: matrix-matrix products and
    // symmetric rank-b updates
    for( i=k+b; i<=n; i+=b ) {
      for( j=i+b; j<=n; j+=b )
        dgemm_tn( b, &A_ref( k, i ), &A_ref( k, j ),
                     &A_ref( i, j ), Alda );
      dsyrk_ut( b, &A_ref( k, i ), &A_ref( i, i ), Alda );
    }
  }
}

#pragma css task input( b, ldm ) inout( A[b*b], info[1] )
void dpotrf_u( int b, double A[], int ldm, int *info )
{
  dpotrf( "Upper", &b, A, &ldm, info );
}

#pragma css task input( b, A[b*b], ldm ) inout( B[b*b] )
void dtrsm_lutn( int b, double A[], double B[], int ldm )
{
  double done = 1.0;
  dtrsm( "Left", "Upper", "Transpose", "Non unit",
         &b, &b, &done, A, &ldm, B, &ldm );
}

#pragma css task input( b, A[b*b], B[b*b], ldm ) inout( C[b*b] )
void dgemm_tn( int b, double A[], double B[], double C[], int ldm )
{
  double dmone = -1.0, done = 1.0;
  dgemm( "Transpose", "No transpose", &b, &b, &b, &dmone,
         A, &ldm, B, &ldm, &done,  C, &ldm );
}

#pragma css task input( b, A[b*b], ldm ) inout( C[b*b] )
void dsyrk_ut( int b, double A[], double C[], int ldm )
{
  double dmone = -1.0, done = 1.0;
  dsyrk( "Upper", "Transpose", &b, &b, &dmone, A, &ldm, &done,  C, &ldm );
}
```

Listing 2.1: Blocked routine for the Cholesky factorization annotated with SMPSs parallelization directives.

triangular systems. The LU and the Cholesky factorizations are strategies of this type, with the latter being used when the coefficient matrix is *symmetric positive definite*.

Definition A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is positive definite if $x^T Ax > 0$ for all nonzero $x \in \mathbb{R}^n$.

The following theorem defines the Cholesky factorization of a symmetric positive definite matrix.

Theorem 2.3.1 *If $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, then there exists a unique lower triangular matrix $L \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that $A = LL^T$.*

The proof of this theorem can be found in classic literature [63]. The decomposition $A = LL^T$ is known as the *Cholesky factorization*, and L is usually referred to as the *Cholesky factor* or the *Cholesky triangle* of A. Alternatively, A can be decomposed so that $A = U^TU$, with $U \in \mathbb{R}^{n\times n}$ upper triangular.

The Cholesky factorization is the first step towards the solution of a system of linear equations $Ax = b$ with A symmetric positive definite. The system can be tackled by first computing the Cholesky factorization $A = LL^T$, then solving the lower triangular system $Ly = b$, and finally solving the upper triangular system $L^Tx = y$. The factorization of the coefficient matrix involves the major part of the arithmetic operations ($O(n^3)$ FLOPS for the factorization compared with $O(n^2)$ for the triangular system solutions), and thus its optimization can yield higher benefits in terms of performance gains.

The largest entries in a positive definite matrix A are on the diagonal (commonly referred to as a *weighty* diagonal). Thus, these matrices do not need pivoting during their factorization [63]. Linear systems with a positive definite coefficient matrix A constitute one of the most important and common cases of linear systems.

The Cholesky factor of a symmetric positive definite matrix is unique. Although the Cholesky factorization can only be computed for symmetric positive definite matrices, it presents some appealing features, basically with respect to computational cost and storage requirements. These advantages make the Cholesky decomposition of special interest compared with other decompositions such as the LU or QR factorizations.

Algorithm 1 presents a blocked (right-looking) procedure to compute the Cholesky factorization of A that overwrites the upper triangular part of the matrix with the contents of the upper triangular factor U. In the following algorithms, we consider a partitioning of this matrix into blocks of size $b \times b$ and $A \rightarrow (A_{i,j})$ denotes the (i,j) block in this partitioning. Each kernel in the algorithm is annotated to the right with its theoretical cost in FLOPS. The global cost of this factorization is $n^3/3$ FLOPS. (Hereafter we neglect lower order terms in the cost expressions.)

Algorithm 1 Right-looking blocked algorithm for the Cholesky factorization.

1:	**for** $k = 1,2,\ldots,s$ **do**		
2:	$A_{kk} = U_{kk}^T U_{kk}$	CHOLESKY FACTORIZATION	$b^3/3$ FLOPS
3:	**for** $j = k+1,k+2,\ldots,s$ **do**		
4:	$A_{kj} \leftarrow U_{kk}^{-T} A_{kj}$	TRIANGULAR SOLVE	b^3 FLOPS
5:	**end for**		
6:	**for** $i = k+1,k+2,\ldots,s$ **do**		
7:	**for** $j = i+1,i+2,\ldots,s$ **do**		
8:	$A_{ij} \leftarrow A_{ij} - A_{ki}^T A_{kj}$	MATRIX-MATRIX PRODUCT	$2b^3$ FLOPS
9:	**end for**		
10:	$A_{ii} \leftarrow A_{ii} - A_{ki}^T A_{ki}$	SYMMETRIC RANK-b UPDATE	b^3 FLOPS
11:	**end for**		
12:	**end for**		

Figure 2.6 shows the dependency graph (DAG) corresponding to the computation of the Cholesky factorization of a matrix consisting of 3×3 blocks using Algorithm 1. (We will assume hereafter that n is an integer multiple of b for simplicity.) In the graph, nodes stand for tasks and edges identify dependencies. Each task $A \rightarrow (A_{i,j})$, each of dimension $b \times b$, is labeled with a first letter representing its type (Chol for Cholesky, T for triangular system solve, G for general matrix-matrix product, and S for symmetric rank-b update).

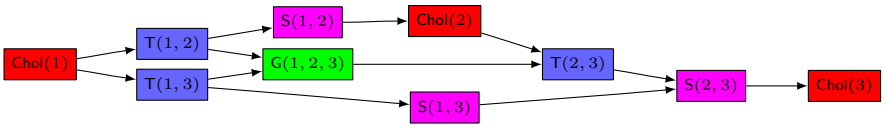


Figure 2.6: DAG with the tasks/data dependencies for the Cholesky factorization of a matrix consisting of 3×3 blocks using Algorithm 1.

2.4 The LU Factorization

The LU factorization, combined with a pivoting strategy (usually partial row pivoting), is the most common method to solve linear systems when the coefficient matrix A does not present any particular structure nor property. As for the Cholesky factorization, a necessary property of the coefficient matrix A is to be invertible.

The LU factorization of a matrix $A \in \mathbb{R}^{n \times n}$ involves the application of a sequence of $(n-1)$ Gauss transformations [124], that ultimately decompose the matrix into the product $A = LU$, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $U \in \mathbb{R}^{n \times n}$ is upper triangular.

Once the coefficient matrix is decomposed into the corresponding triangular factors, it is possible to obtain the solution of the linear system $Ax = b$ by solving two triangular systems: first, the solution of the system $Ly = b$ is obtained using progressive elimination; and then the system $Ux = y$ is solved by regressive substitution. As for the Cholesky factorization, these two stages involve lower computational cost, and thus the optimization effort is usually focused on the factorization stage.

Theorem 2.4.1 *A matrix $A \in \mathbb{R}^{n \times n}$ has an LU factorization if its $n-1$ leading principal submatrices of A are invertible. Moreover, if the LU factorization exists and A is invertible, then the factorization is unique.*

The proof for this theorem can be found in the literature [63].

2.4.1 The LU factorization with partial pivoting

As noted before, the solution of a linear system of the form $Ax = b$ exists and is unique if the coefficient matrix A is invertible. However, this is not the only requisite for the existence of the LU factorization. For example, if A is invertible, but any of the elements in the diagonal is zero, the factorization is not feasible, as a division by zero appears during the algorithm.

Row pivoting is a strategy that can be applied to solve this issue: if the row with the zero diagonal element is swapped with a different row below that in the matrix (without a zero element in the corresponding entry), the factorization can continue normally. These exchanges must be applied as well to vector b before solving the linear system. This problem can occur even with non zero diagonal entries, but with its absolute value being of small magnitude compared with the rest of the elements in its column. This situation involves a growth in the magnitude of the elements of U with the consequent rounding errors in the presence of limited precision arithmetic, and a catastrophic impact on the accuracy of the result to the linear system [124].

The row pivoting strategy thus consists of swapping the diagonal element with that of largest absolute magnitude from those in and below the diagonal in the current column. This method guarantees that every element in L is equal or smaller than 1 in magnitude, and handles properly the growth of the elements in U limiting the impact of the round-off errors. This technique is known as *partial row pivoting*.

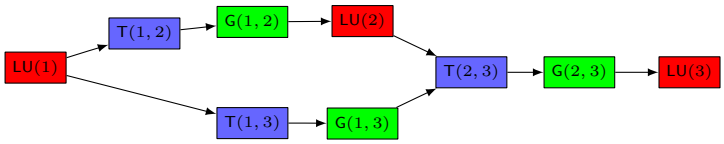


Figure 2.7: DAG with the tasks/data dependencies for the LU factorization with partial pivoting of a matrix consisting of 3×3 blocks using Algorithm 2.

Algorithm 2 shows a right-looking blocked algorithm for the LU factorization which overwrites the strictly lower/upper triangular parts of the matrix $A \rightarrow (A_{i,j})$ with the contents of the strictly lower/upper triangular factors L/U. For simplicity, partial pivoting is not included there. Each operation (i.e., task) in the algorithm is annotated to the right with its theoretical cost in FLOPS. This cost depends on the block size b and the number of blocks $s = n/b$. Note that

$$L_{k:s,k} = \begin{pmatrix} L_{k,k} \\ L_{k+1:s,k} \end{pmatrix},$$

with $L_{k,k}$ unit lower triangular and $L_{k+1:s,k}$ rectangular. The factorization of the current panel $A_{k:s,k}$ (kernel LU) can be obtained by calling an unblocked version of the algorithm ($b = 1$). Provided $1 \ll b \ll n$, the blocked algorithm in the figure performs $2n^3/3$ FLOPS, mostly cast in terms of the matrix-matrix product (GEMM) $A_{k+1:s,j} \leftarrow A_{k+1:s,j} - A_{k+1:s,k} \cdot A_{kj}$.

Algorithm 2 Right-looking blocked algorithm for the LU factorization.

1:	**for** $k = 1 : s$ **do**		
2:	$A_{k:s,k} = L_{k:s,k} \cdot U_{kk}$	LU FACTORIZATION	$(s - k + \frac{2}{3})b^3$ FLOPS
3:	**for** $j = k + 1 : s$ **do**		
4:	$A_{kj} \leftarrow L_{kk}^{-1} \cdot A_{kj}$	TRIANGULAR SOLVE	b^3 FLOPS
5:	$A_{k+1:s,j} \leftarrow A_{k+1:s,j} - A_{k+1:s,k} \cdot A_{kj}$	MATRIX-MATRIX PRODUCT	$2(s - k)b^3$ FLOPS
6:	**end for**		
7:	**end for**		

Figure 2.7 captures the tasks and dependencies that appear during the computation of the LU factorization with partial pivoting of a blocked 3×3 matrix using Algorithm 2. In the graph, nodes stand for tasks and edges identify dependencies. The label of each task represents its type: "LU" denotes the LU factorization via Gauss transforms, "T" the triangular system solve, and "G" the matrix-matrix product. The number after the letter uniquely identifies each task (and it can be derived from the loop indices k and j of the algorithm).

In practice, the introduction of pivoting slightly modifies the DAG in the figure, so that each pair of tasks/dependence $\mathsf{T}(k,j) \rightarrow \mathsf{G}(k,j)$ would become $\mathsf{LU}(k) \rightarrow \mathsf{T}(k,j) \rightarrow \mathsf{G}(k,j)$, with $\mathsf{T}(k,j)$ including the application of permutations to the block $A_{k:s,j}$. Given that these additional dependencies do not inhibit further the degree of parallelism in the algorithm and that the cost of applying the permutations $\mathsf{LU}(k)$ can be added to that of the triangular system solve $\mathsf{T}(k,j)$, we can safely avoid the explicit inclusion of these tasks.

2.4.2 The LU factorization with incremental pivoting

In [70] it is shown how the insights gained from studying the update an existing LU factorization yields the algorithm-by-blocks for the LU factorization with *incremental pivoting*. The key that allows the computational expense to be roughly the same as the standard LU factorization with partial pivoting is a careful orchestration of computation and pivoting so that the matrix on the

diagonal, after being factored itself, does not incur into significant fill-in as it is being used to zero elements in the blocks below it.

The *algorithm-by-blocks* for the LU factorization with incremental pivoting carries out a sequence of row permutations (corresponding to the application of pivots) which are different from those that would be performed in an LU factorization with partial pivoting. Therefore, the numerical stability of this algorithm is also different [70].

Algorithm 3 presents the blocked variant of the LU factorization with incremental pivoting [108]. Specifically, the algorithm overwrites the upper triangular part of the matrix $A \rightarrow (A_{i,j})$ with U, while the Gauss transforms that yield L are implicitly stored in the strictly lower triangle of A plus an additional workspace of small dimension. Although the LU factorization with incremental pivoting does not compute a clean unit lower triangular factor L, it still yields a decomposition that allows a high-performance BLAS-3-based solution of a linear system. The total number of FLOPS performed by the algorithm-by-blocks is approximately $2n^3/3$. Notice that there is some flexibility in the order in which the loops are arranged. Actually, in task-parallel implementations the runtime system rearranges the operations and therefore the exact order of the loops is not important.

Algorithm 3 Right-looking blocked algorithm for the LU factorization with incremental pivoting.

1:	**for** $k = 1,2,\ldots,s$ **do**		
2:	$A_{kk} = L_{kk} \cdot U_{kk}$	LU FACTORIZATION	$2b^3/3$ FLOPS
3:	**for** $j = k+1,k+2,\ldots,s$ **do**		
4:	$A_{kj} \leftarrow L_{kk}^{-1} \cdot A_{kj}$	TRIANGULAR SOLVE	b^3 FLOPS
5:	**end for**		
6:	**for** $i = k+1,k+2,\ldots,s$ **do**		
7:	$\begin{pmatrix} A_{kk} \\ A_{ik} \end{pmatrix} = \begin{pmatrix} L_{kk} \\ L_{ik} \end{pmatrix} \cdot U_{ik}$	2×1 LU FACTORIZATION	b^3 FLOPS
8:	**for** $j = k+1,k+2,\ldots,s$ **do**		
9:	$\begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix} \leftarrow \begin{pmatrix} L_{kk} & 0 \\ L_{ik} & I \end{pmatrix}^{-1} \cdot \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix}$	2×1 TRIANGULAR SOLVE	$b^3/2$ FLOPS
10:	**end for**		
11:	**end for**		
12:	**end for**		

Figure 2.8 illustrates the DAG corresponding to the computation of the LU factorization with incremental pivoting of a matrix consisting of 3×3 blocks using Algorithm 3. In the task names, LU stands for LU factorization, T for triangular system solve, G2 for 2×1 LU factorization, and T2 for 2×1 triangular system solve; this is followed by a number that uniquely identifies the task in the graph. The permutations required for stability can be easily merged with the other tasks, simplifying the presentation of the DAG without restricting the degree of concurrency of the algorithm.

2.5 The QR Factorization

The QR factorization is given by $A = QR$, where $A \in \mathbb{R}^{m \times n}$ is a real-valued matrix, $Q \in \mathbb{R}^{m \times m}$ is an orthogonal matrix (that is, $Q^T Q = QQ^T = I$ the identity matrix), and $R \in \mathbb{R}^{m \times n}$ is an upper triangular matrix. There are many different methods for computing the QR factorization, including those based on Givens rotations, orthogonalization via Gram-Schmidt and modified Gram-Schmidt, and Householder transformations [63]. To introduce this factorization we use the last one.

For dense matrices, the method of choice largely depends on how the factorization is subsequently used, the stability of the system, and the dimensions of the matrix. For problems where

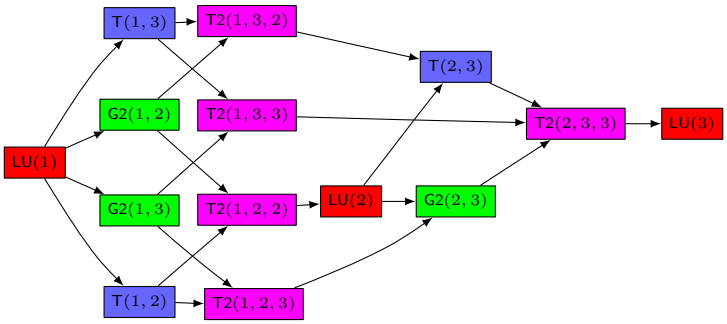


Figure 2.8: DAG with the tasks/data dependencies for the LU factorization with incremental pivoting of a matrix consisting of 3×3 blocks using Algorithm 3.

$m \gg n$, the method based on Householder transformations is typically the algorithm of choice, especially when Q does not need to be explicitly computed.

Householder transformations. Given a real-valued vector x of length m, partition $x = \begin{pmatrix} \chi_0 \\ \hline x_1 \end{pmatrix}$, where χ_0 equals the first element of x. The Householder vector associated with x is defined as the vector $u = \begin{pmatrix} 1 \\ \hline x_1/\nu_0 \end{pmatrix}$, where $\nu_0 = \chi_0 + \text{sign}(\chi_0)\|x\|_2$. If $\beta = \frac{2}{u^T u}$ then $(I - \beta u u^T)x = \eta e_0$, annihilating all but the first element of x, which becomes $\eta = -\text{sign}(\chi_0)\|x\|_2$. The transformation $I - \beta u u^T$ is referred to as a Householder transformation or *reflector*.

Let us introduce the notation $[u, \eta, \beta] \leftarrow h(x)$ as the computation of the above mentioned η, u, and β from vector x and the notation $H(x)$ for the corresponding transformation $(I - \beta u u^T)$. An important feature of $H(x)$ is that it is orthogonal and symmetric $(H(x)^T = H(x))$.

The idea is that Householder transformations are computed to successively annihilate all the subdiagonal elements of matrix A. The Householder vectors are stored by overwriting those elements that have been previously annihilated. Upon completion, matrix R overwrites the upper triangular part of the matrix while the Householder vectors are stored in the strictly lower trapezoidal part of the matrix. The scalars β discussed above are stored in a vector of length n.

If the matrix Q is explicitly desired, it can be formed by carefully accumulating the product $H_0 \cdot H_1 \cdots H_{n-1} = Q$, where H_k equals the $(k+1)$-th Householder transformation computed as part of the factorization described above. Frequently, Q does not need to be explicitly formed, and thus we will not discuss the issue further

Implementations of the QR factorization, whether unblocked or blocked, typically are written so that the bulk of the computation is performed by the BLAS, which export a standardized interface to common operations such as matrix-vector (BLAS-2) and matrix-matrix multiplication (BLAS-3).

2.5.1 The traditional QR factorization

The traditional slab-based QR factorization of a matrix $A \rightarrow (A_{i,j})$ proceeds in panels of b columns (slabs), as illustrated by Algorithm 4. Each operation in the algorithm is annotated to the right with its theoretical cost, which depends on the loop index k. The cost of this algorithm is $2n^2(m - n/3)$ FLOPS, or $4n^3/3$ FLOPS when $m = n$.

Figure 2.9 shows the DAG obtained when this algorithm is applied to a blocked 3×3 matrix A using Algorithm 4. There, QR represents the QR factorization and O the application of the orthogonal transformations.

Algorithm 4 Right-looking blocked algorithm for the traditional QR factorization.

1: **for** $k = 1 : s$ **do**
2: $\quad A_{k:s,k} = Q_{k:s,k:s} \cdot R_{kk}$ — QR FACT. — $2(s-k-\frac{1}{3})b^3$ FLOPS
3: $\quad$ **for** $j = k+1 : s$ **do**
4: $\quad\quad A_{k:s,j} \leftarrow Q^T_{k:s,k:s} A_{k:s,j}$ — APPLY ORTH. TRANSF. — $4(s-k-\frac{1}{2})b^3$ FLOPS
5: $\quad$ **end for**
6: **end for**

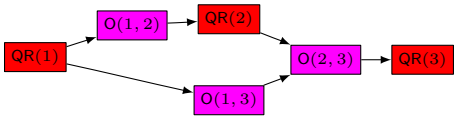


Figure 2.9: DAG with the tasks/data dependencies for the QR factorization of a matrix consisting of 3×3 blocks using Algorithm 4.

In this decomposition, the factorization of the current panel is an inherently sequential operation. Therefore, for this operation it is desirable to use a sequential blocked algorithm with a small block size and cast the computation in terms of BLAS-3. The fundamental question now becomes how to create as much parallelism as possible without the factorization of the current panel becoming a bottleneck because of dependencies. This problem is normally addressed by applying updates to future panels as early as possible so that the factorization of those panels can proceed in parallel with updates of later panels.

2.5.2 The incremental QR factorization

The blocked procedure for the incremental QR factorization is illustrated in Algorithm 5. In this case, the upper triangular part of the matrix $A \rightarrow (A_{i,j})$ is overwritten with the entries of R, while Q is not explicitly built but stored implicitly using the annihilated of A and a small workspace. A careful implementation of the building blocks for this factorization yields a global computational cost of $4n^3/3$ FLOPS [67], equivalent to that of the (unblocked) QR factorization via Householder transforms.

Algorithm 5 Right-looking blocked algorithm for the incremental QR factorization.

1: **for** $k = 1,2,\ldots,s$ **do**
2: $\quad A_{kk} = Q_{kk} R^T_{kk}$ — QR FACTORIZATION — $4b^3/3$ FLOPS
3: $\quad$ **for** $j = k+1, k+2, \ldots, s$ **do**
4: $\quad\quad A_{kj} \leftarrow Q^T_{kk} A_{kj}$ — APPLY ORTH. TRANSF. — $2b^3$ FLOPS
5: $\quad$ **end for**
6: $\quad$ **for** $i = k+1, k+2, \ldots, s$ **do**
7: $\quad\quad \begin{pmatrix} A_{kk} \\ A_{ik} \end{pmatrix} = \begin{pmatrix} Q_{kk} \\ Q_{ik} \end{pmatrix} R_{ik}$ — 2×1 QR FACTORIZATION — $2b^3$ FLOPS
8: $\quad\quad$ **for** $j = k+1, k+2, \ldots, s$ **do**
9: $\quad\quad\quad \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix} \leftarrow \begin{pmatrix} Q_{kk} & 0 \\ Q_{ik} & I \end{pmatrix}^T \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix}$ — 2×1 APPLY ORTH. TRANSF. — $4b^3$ FLOPS
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: **end for**

Figure 5.1 illustrates the tasks and dependencies obtained for the QR factorization of a blocked 3×3 matrix using Algorithm 5. There, G denotes the QR factorization, O the application of (orthogonal) transformations, G2 the 2×1 QR factorization, and O2 the 2×1 application of (orthogonal) transformations.

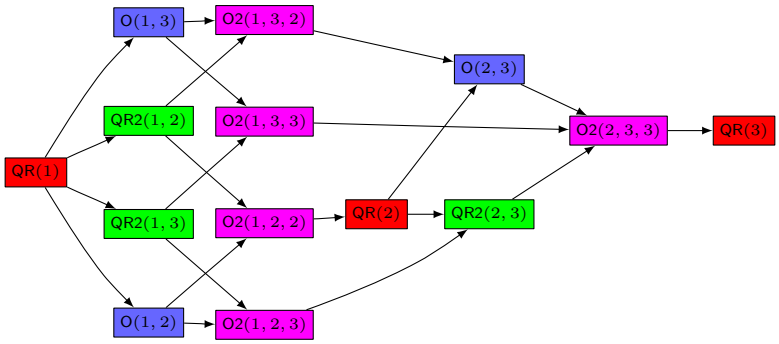


Figure 2.10: DAG with the tasks/data dependencies for the incremental QR factorization of a matrix consisting of 3×3 blocks using Algorithm 5.

The level of parallelism realized from the incremental QR factorization depends on block size b relative to the problem dimension, with smaller values of b unleashing a higher level of concurrency. Unfortunately, decreasing b may negatively affect the performance of the algorithmic subproblems since b must be reduced correspondingly if overhead is to be kept low. For these small block size situations BLAS-3 performance is less likely to be achieved due to the smaller matrix operands.

2.6 Sparse Linear Algebra

Large sparse linear systems arise in many application areas such as partial differential equations, quantum physics or problems from circuit and device simulations. These problems share the same central task, which consists in efficiently solving a large sparse linear system of equations. For a vast class of application problems, sparse direct solvers have proven to be extremely efficient. However, the enormous size of the applications arising in 3D 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 increases. This in turn demands for alternative approaches and, often, approximate factorization techniques, combined with iterative methods mainly based on Krylov subspaces, which reflect an attractive alternative for these kind of application problems.

2.6.1 ILUPACK: Incomplete LU factorization PACKage

The Incomplete LU factorization PACKage (ILUPACK) [76] is mainly built on incomplete factorization methods (ILUs) applied to the system matrix in conjunction with Krylov subspace methods. The main drivers can be called from C, C++ and FORTRAN. The ILUPACK hallmark is the so-called inverse-based approach. The package implements a multilevel incomplete factorization approach (multilevel 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 partial differential equations (PDEs). ILUPACK 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, which may or may not be positive definite, and general square matrices.

The approach connects the ILUs and their approximate inverse factors. 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. Thus, information

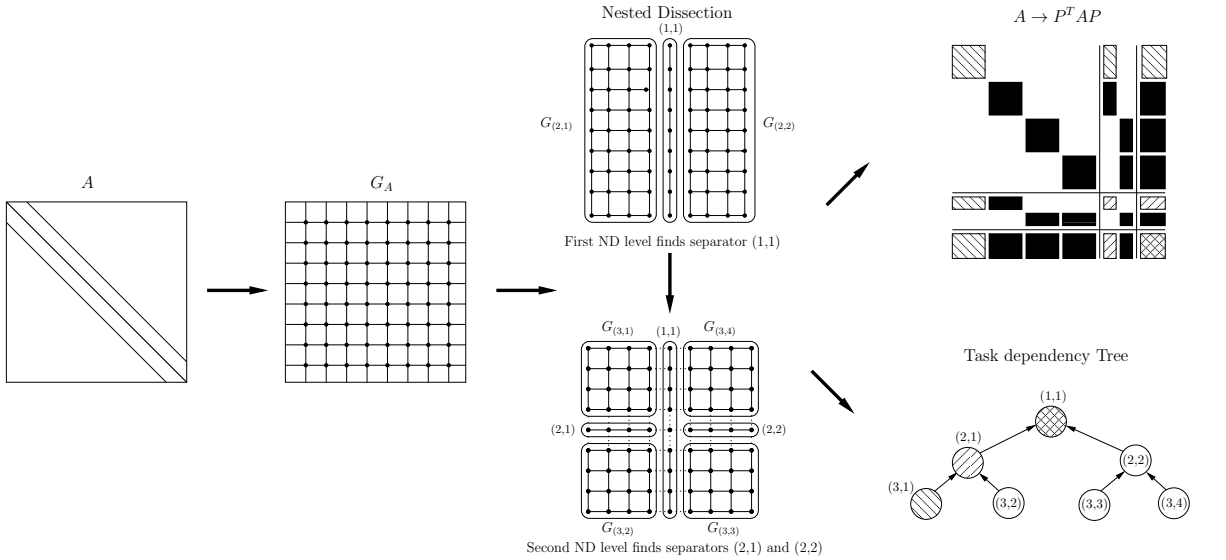


Figure 2.11: Nested dissection applied to the adjacency graph associated with a sparse matrix and the corresponding task dependency tree.

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 multilevel strategy is supported to limit the growth of the inverse factors. This has led to the inverse-based approach and hence the incomplete factorization process that has eventually been implemented in ILUPACK benefits from the information of bound inverse factors to being efficient in turn.

The approach to multilevel preconditioning in ILUPACK relies on the so-called inverse-based ILU factorizations. Unlike other classical threshold-based ILUs, this approach directly bounds the size of the preconditioned error and results in increased robustness and scalability, especially for applications governed by PDEs, due to its close connection with algebraic multilevel methods [7]. Specifically, for efficient preconditioning, only a small amount of fill-in is allowed during the factorization, resulting in a modest number of floating-point arithmetic operations per non-zero entry of the sparse coefficient matrix.

Parallelization. Concurrency in the computation of ILUPACK preconditioners is exposed by means of nested dissection applied to the adjacency graph representing the non-zero connectivity of the sparse coefficient matrix. Nested dissection is a partitioning heuristic which relies on the recursive separation of graphs. The graph is first split by a vertex separator into a pair of independent subgraphs and the same process is next recursively applied to each independent subgraph. The resulting hierarchy of independent subgraphs is highly amenable to parallelization. In particular, the inverse-based preconditioning approach is applied in parallel to the blocks corresponding to the independent subgraphs while those corresponding to the separators are updated. When the bulk of the former blocks has been eliminated, the updates computed in parallel within each independent subgraph are merged together, and the algorithm enters the next level in the nested dissection hierarchy. The same process is recursively applied to the separators in the next level and the algorithm proceeds bottom-up in the hierarchy until the root finally completes the parallel computation of the preconditioner; see Figure 2.11.

The type of parallelism described above can be expressed by a binary task dependency tree, where nodes represent concurrent tasks and arcs specify dependencies among them. The parallel execution of this tree on multicore processors is orchestrated by a runtime which dynamically maps tasks to threads (cores) in order to improve load balance requirements during the computation of the ILU preconditioner. This runtime keeps a shared queue of ready tasks (i.e., tasks with their dependencies fulfilled) which are executed by the threads in FIFO order. This queue is initialized with the tasks corresponding to the independent subgraphs. Idle threads have to wait for new ready tasks. When a given thread completes the execution of a task, its parent task is enqueued provided the sibling of the former task has been already completed. Further details on the mathematical foundations of the parallel algorithms and the runtime operation can be found in [7].

The most expensive operation involved in the preconditioned iterative solution of the linear system is the application of the multilevel preconditioner, which is in turn decomposed into two steps: the multilevel Forward Substitutions (FS) and Backward Substitutions (BS). The aforementioned task dependency tree also describes the parallelism available within both computations. However, while the FS proceeds bottom-up towards the root of the tree, the BS proceeds in the opposite direction. In order to maximize data locality during the parallel multithreaded execution of both operations, the mapping of threads to tasks resulting from the (dynamic load-balancing) computation of the preconditioner is re-used, so that each thread knows in advance the tasks it is in charge of (i.e., static mapping). The runtime uses a different task queue for each thread and substitution algorithm. For the FS, the queue of each thread is initialized with the leaves it is in charge of, and new (ready) tasks are enqueued on the corresponding queues as soon as their dependencies are fulfilled (i.e., as soon as their children tasks are completed). For the BS, only the root task is initially included in the corresponding queue. As soon as the root task is completed, its children are then enqueued on the corresponding queues, and the parallel execution (managed by the runtime) proceeds top-bottom while taking care of task dependencies until the computation of the leaves is completed. The other operations involved in the preconditioned iterative solution stage (i.e., sparse matrix-vector product and vector operations) are split and mapped conformally with the FS and BS steps in order to maximize data locality. Moreover, a careful management of shared data, by maintaining consistent or inconsistent copies of the vectors, avoids synchronization steps, except those reductions required in order to compute inner products. Further details on the mathematical foundations of the parallel algorithms, their implementation, and the runtime operation can be found in [7].

CHAPTER 3

Performance and Energy Measurement Framework

The development of Exascale systems exposed that current technologies, programming practices and performance metrics are not adequate. Existing tools for HPC mainly focus on monitoring and evaluation of performance metrics. Newer hardware has a wide range of sensors and measurement devices related to power consumption with varying granularity and informative value. Recently, new tools have been developed or have been adapted from other research areas (e.g. mobile computing) for analyzing power consumption. Most of these tools focus only on power consumption and disregard the performance aspects. Therefore, it is crucial to identify adequate sensors, hardware counters and measurement devices to gain detailed insights about the power consumption and the performance of the hardware.

In order to optimize energy consumption of scientific applications, enhanced profiling and tracing frameworks combining both power and performance metrics are needed. Moreover, to gain a better understanding of energy usage, performance metrics, such as performance counters or routine events, should be correlated with the power traces. Only analyzing these measurements, energy inefficiencies in the code can be localized and optimized. In this chapter, we propose an integrated framework with modular design to study power and energy profiles/traces of HPC scientific applications. This power-performance analysis framework supports different types of parallel applications: MPI/OpenMP and P-Thread.

This chapter is organized as follows. In Section 3.1, we provide a brief overview of the hardware energy-saving mechanisms and describe the different platforms and power measurement devices used in this work. Section 3.2 introduces the software leveraged as well as developed for performance-power analysis of applications. This basically includes the profiling and tracing tools, the PMLib library, and the integration of the suite in our framework. In Section 3.3 we illustrate the information provided by the framework with a detailed power and performance analysis of a representative dense linear algebra operation. Finally, some concluding remarks and future work are provided in Section 3.4.

3.1 Hardware

The efficient use of the energy-aware mechanisms available in current architectures is a key point that needs to be considered and explored by the scientific community in order to attain relevant

energy savings. These mechanisms can be encountered in different hardware components: CPUs, memory, HDDs, graphic accelerators, etc. For this reason, a preliminary step is the evaluation of these mechanisms from the performance and energy consumption point of view. In this section we revisit basic hardware techniques and introduce the architectures and power measurements devices leveraged along this work.

3.1.1 Architectures

To adjust the device performance and power consumption, most hardware devices in a high performance computing systems incorporate different power saving modes. An overview of these mechanisms along with CPU and memory levels is provided in this section.

The CPU

The CPU usually consumes a large portion of energy, around 50 % of the total of a high performance computing system [107]. Therefore, hardware manufacturers have special interest in the development of energy-saving mechanisms to reduce the power consumption of this component. In this sense, recent research efforts focus on exploring low consumption modes of the CPU. Before studying the energy saving techniques in detail, let us introduce the different sources of power consumption of the CPU.

In general, current CPUs and most digital circuits are constructed using CMOS circuits [71]. Therefore the analysis of power dissipation in CMOS circuits is essential to find out the relation between power, supply voltage, and clock frequency. The total power dissipation for CPUs is the summation of *dynamic* power, *static* power, and *short circuit* power. These ingredients of power dissipation are described as follows:

$P_{dynamic}$ is the power dissipation due to charging and discharging capacitors, composed of gate and interconnect capacitances. This dynamic flow switching allows charging and discharging thus enabling a better circuit performance.

P_{static} is the power dissipation due to reverse biased diodes. Normally it is due to the gradual loss of energy from charged capacitors or current leaks of the circuit. This component can be considered as constant and can be disregarded for dynamic power-saving modes.

$P_{shortcircuit}$ is the power dissipation due to switching direct path between V_{CC}-GND; i.e., the power which flows from the supply ground during a transition of period of input signals. This occurs during the signal transitions and is negligible for the total chip consumption.

These three factors are related by the formula:

$$P_{total} = P_{dynamic} + P_{static} + P_{shortcircuit} \ .$$

The dynamic power is the main portion of the CMOS power dissipation and can be expressed as

$$P_{dynamic} = \alpha \ \cdot C \ \cdot {V_{CC}}^2 \ \cdot f \ , \tag{3.1}$$

where f is the frequency, V_{CC} the voltage, α the level of chip activity, and C a factor dependent on the capacitance of the chip. The weight of the dynamic power varies depending on the usage of the CPU. In recent multicore processors, the variation of the dynamic power depends on the number of cores in use.

In order to meet size, performance and power constraints, recent generations of multicore processors have miniaturized and integrated more and more components into the same die. This trend unleashed a variety of advantages: latency reduction because of the integration of the memory controller, power efficiency, and better performance/scalability using multiple cores. In this sense, the uncore area has been enlarged to accommodate more elements. Normally, this uncore part of the die includes the memory-controller, the Last Level Cache (LLC) and interconnection links. Depending on the architecture and, more specifically, on the number of cores, the uncore area may consume an important percentage of the total power consumption generated by the chipset.

An approach to reduce $P_{dynamic}$ is to reduce voltage and thus, the frequency. The relation between voltage and frequency is direct: high frequencies require high voltages. According to (3.1), voltage increases quadratically, so its value plays an important role in the equation. A given voltage for a particular frequency depends on several factors related to the chip design and operation temperature. A reduced operating voltage close to the minimum threshold yields Near-Threshold Computing (NTC). Currently, hardware devices come with the ability of working under different power and performance configurations, a feature which can be exploited to improve energy efficiency. One mechanism is to dynamically select an operating frequency among an existing set of frequencies under which a processor can operate. Each frequency is assigned to a state named *performance state* or *P-state*. The goal of P-state management is to reduce the power loss due to leakage. On the processor side, the mechanism that controls the actual P-state is known as Dynamic Voltage and Frequency Scaling (DVFS) [86]. A complementary mechanism that can be leveraged to save energy is based on the *processor states* or *C-states*. With this technique it is possible disable the clock (clock gating) or interrupt the power consumption (power gating) to induce certain parts of the processor into a *sleep state* which results in energy savings.

Performance states

The P-states come originally from the mobile devices, in which the processing workload varies and requires low consumption hardware. Nowadays, DVFS is implemented in almost all desktop and server processors. Since the lower frequency also decreases performance, these frequency states are named as performance states (P-states) in the Advanced Configuration and Power Interface (ACPI) specification [72]. The performance states are enumerated as P0, ..., Pn, where P0 and Pn stand for the maximum and minimum allowable frequencies, respectively. Intel's DVFS mechanism is called Enhanced Intel SpeedStep Technology (EIST) [2] and AMD refers to DVFS as PowerNow! [1]. According to Intel, the primary goal of this technology is to provide multiple voltage and frequency operating points for optimal performance at the lowest power.

The voltage and frequency is controlled by software that writes into processor model-specific registers (MSR). If the target frequency is higher than the current frequency, the voltage is ramped up in steps, and the Phase Lock Loop (PLL) locks to the new frequency. If the target frequency is lower than the current frequency, the PLL locks to the new frequency and the voltage is changed. Software transitions are accepted at any time. The processor controls voltage ramp rates internally to ensure smooth transitions. The low transition latency leads to a large number of possible transitions per second. The processor core and the shared cache are unavailable for a few microseconds until the frequency transition is completed.

Apart from the reduction of the nominal frequency, it is also possible to increase the frequency above this threshold. This technology, identified as Turbo Boost by Intel [78] or Turbo Core by AMD, boosts the performance of the processor by increasing the working frequency when some specific conditions are met. For instance, when only one core is used, the mechanism may yield higher performance. This is also possible for a subgroup of cores providing higher performance

for one group and reducing performance for the rest, resulting in performance benefits for some workloads.

Recent developments in processor technology have resulted in the saturation of processor clock frequencies, larger static power consumption, smaller dynamic power range and better idle/sleep modes. Each of these developments limits the potential energy savings resulting from DVFS. While DVFS is effective on old platforms, it actually increases energy usage on the most recent platforms, even for memory-bound workloads [86].

The Linux CPU governors. The Linux `cpufreq` subsystem allows to dynamically scale the CPU frequency. Several `cpufreq` system governors may be used to manage the frequency of each CPU:

`ondemand`. The `ondemand` governor is the default governor and dynamically sets the frequency based on the current workload. During idle phases, the CPU will rest in the lowest frequency. When the current load surpasses a specified threshold (by default 95 %) the ondemand governor will switch the CPU to the highest frequency available. Once the load falls below that threshold, the ondemand governor will switch to the next lowest frequency and continue to do so until the lowest frequency is reached (if the load stays below the threshold). The default sampling frequency of core activity is 10 ms.

`conservative`. The `conservative` governor operates like the `ondemand` governor, based on the current workload, but it increases/decreases the frequency gradually. Once the load is higher than a threshold, the `conservative` governor only switches to the next highest frequency and not to the highest one. The frequency will be continually increased as long as the load stays above the threshold until the higher frequency is reached. For this case, the default sampling frequency is also 10 ms.

`powersave`. The `powersave` governor keeps the CPU always at the lowest frequency.

`performance`. The `performance` governor keeps the CPU at the highest frequency.

`userspace`. The `userspace` governor allows the user to take the control over the CPU frequencies.

Frequencies and governors can be set via `cpufreq-set` command and the `libcpufreq` library.

Processor states

The ACPI specification defines the power state of system processors as being either active (executing) or inactive (not executing). The processor states (C-states) are encoded as C0, C1, ..., Cm. The C0 power state is the active power state, i.e., the processor is executing instructions. The power consumption decreases with a higher C-state, however the latency increases to reach again the active power state C0. The C1-state is entered when all threads within a core execute a `HLT` (halt) instruction. The processor transits to C0 if an interrupt occurs. In C1 the clock of the core is gated and is thus able to maintain the context of the system caches. Based on the current implementation of the processor architecture, specific processor power states interfere also the cache behavior. If the state is deeper than C2, it is possible that the level 3 cache memory is turned off or flushed, and level 1 and 2 may be invalidated as well. Invalidation results in performance decrease since caches have to be repopulated again. Table 3.1 summarizes the C-states. Depending on the architecture and the power state, these states can be applied at the core or socket level.

State	Short name	Description
C0	Operating state	CPU fully turned on.
C1	Halt	Stops CPU main internal clocks via software; bus interface unit and APIC are kept running at full speed.
C1E	Enhanced halt	Stops CPU main internal clocks via software and reduces CPU voltage; bus interface unit and APIC are kept running at full speed. Depending on the model it may stop all CPU internal clocks only.
C2	Stop grant	Stops CPU main internal clocks via hardware; bus interface and APC are kept running at full speed.
C2	Stop clock	Stops CPU internal and external clocks via hardware
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.
C3	Sleep	Stops all CPU internal clocks.
C3	Deep sleep	Stops all CPU internal and external clocks.
C3	AltVID	Stops all CPU internal clocks and reduces CPU voltage.
C4	Deeper sleep	Reduces CPU voltage.
C4E/C5	Enhanced deeper sleep	Reduces CPU voltage even more and turns off memory cache.
C6	Deep power down	Reduces the CPU internal voltage to any value, including 0 V.

Table 3.1: Processor power states overview.

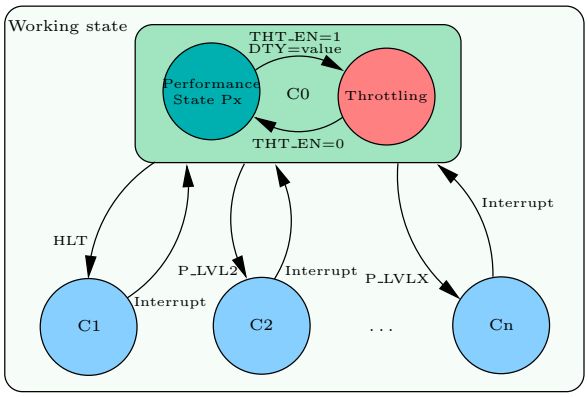


Figure 3.1: Transitions between P-/C-/T-states.

Throttling states

In addition to the aforementioned states, the CPU also supports the *processor throttling states* (T-states). The processor switches to lower frequencies in order to reduce thermal effects. This means that the CPU is forced to be idle a fixed percentage of its cycles per second. Throttling states range from T1 to Tp. In T1 the processor does not introduce idle cycles, meanwhile the percentage of idle cycles is increased with p. Throttling does not reduce voltage and, since the CPU is forced to be idle part of the time, processes will take longer to finish. Furthermore, working longer may consume more energy instead of saving it. T-states are only useful if the primary goal is to reduce thermal effects. Since the T-states can interfere with the C-states (preventing the CPU from entering a deeper C-state), they can even increase energy consumption in a modern CPU. Figure 3.1 depicts transitions between P-/C-/T-states.

Main memory

Memory power consumption is also critical in today's HPC platforms, mainly due to the system architecture and the amount of memory in current systems. In this sense, memory has also adopted DVFS, so that its frequency and voltage can be regulated to reduce power consumption. Supported operating points of frequency and voltage can only be selected before the system startup, via BIOS

settings. However, in some architectures, memory bandwidth is directly related to the processor frequency, thus the operating frequency and the power consumption can be adjusted at runtime. In these architectures, the memory controller frequency is reduced when the processor decreases its frequency. This fact is observed, for instance, in some AMD processors. Consequently, this reduces also the performance (in terms of memory bandwidth) and power consumption of the main memory. In general, the server memory modules, usually Registered Dual In-line Memory Modules (RDIMMs), support multiple electrical current levels for different operations of the module. Multiplying this current with the voltage yields the DIMM power consumption. These levels can be exploited by the memory controller that can keep only a subset of the memory ranks in a ready state, while the rest enters a sleep mode. Disabling the refreshing of main memory (Partial Array Self Refresh) is only possible for mobile DRAM which is used in power sensitive environments, e.g. mobile devices. In these environments, the utilization is usually much lower and thus high power savings can be attained.

3.1.2 Description of the target platforms

Three different systems have been used in the dissertation. Those systems are representative of different multicore architectures present nowadays. We will term them as WT_AMD, WT_ITL and TESLA2 throughout the rest of the thesis.

WT_AMD is a shared-memory multiprocessor equipped with two 8-core AMD Opteron 6128 processors, running at 2.00 GHz, with 48 GB of DDR3 RAM memory under Linux Ubuntu (kernel 2.6.32-220.4.1.el6.x86_64). These AMD processors feature 5 performance states or P-states, P0 to P4, corresponding to frequencies {2.00, 1.50, 1.20, 1.00, 0.80} GHz, and 3 power states or C-states: C0, C1 and C1E.

WT_ITL is a shared-memory multiprocessor equipped with Intel Xeon technology. It is composed of two quad-core Intel Xeon 5504 processors running at 2.00 GHz, with 32 GB of DDR3 RAM memory under Linux Ubuntu (kernel 2.6.32-220.4.1.el6.x86_64). There are 4 performance states or P-states, P0 to P3, in the Intel processor which correspond to frequencies {2.00, 1.87, 1.73, 1.60} GHz; and 4 power states or C-states: C0, C1, C3 and C6.

TESLA2 is a shared-memory multiprocessor also equipped with the Intel Xeon technology. It consists of two quad-core Intel Xeon 5440 (Harpertown) processors running at 2.83 GHz, with 16 GB of DDR2 RAM memory. Attached to the platform, via a PCI Express 2.0 bus, there is a system consisting of four NVIDIA Tesla C2050 GPUs (Fermi).

The processors in WT_AMD and WT_ITL are fair representants of current multicore technology, and adhere to the ACPI standard [72] for the CPU power-saving modes. Information on the voltage–frequency pairs ($V_{CC_i} - f_i$) associated with each P-state (Pi) is collected in Table 3.2. From the practical point of view, the WT_AMD and WT_ITL platforms differ in two important aspects:

- The frequency of the AMD cores can be adjusted independently while, on the WT_ITL platform, all cores in the same processor run at the same frequency. In particular, if the cores of a processor from WT_ITL operate at frequency f_i, and we instruct one of these cores to run at $f_j > f_i$ (using the Linux `cpufreq` utility), the remaining three cores in the same socket will also transition to operate at f_j. On the other hand, if the cores of a processor from WT_ITL run at frequency f_i, and we instruct one core to run at frequency $f_j < f_i$, there will be no change.

WT_AMD

P-state, Pi	VCC_i	f_i	BW_i
P0	1.23	2.00	30.29
P1	1.17	1.50	24.63
P2	1.12	1.20	20.46
P3	1.09	1.00	17.48
P4	1.06	0.80	14.00

WT_ITL

P-state, Pi	VCC_i	f_i	BW_i
P0	1.04	2.00	12.72
P1	1.01	1.87	12.58
P2	0.98	1.73	12.61
P3	0.95	1.60	12.55

Table 3.2: P-states, associated voltage–frequency pairs (V_{CC_i} in Volts and f_i in GHz), and core to memory bandwidth (BW_i, in GB/sec.) measured with the `stream` benchmark.

- On the WT_AMD platform, the bandwidth between the cores and the main memory varies with the processor frequency while, on the WT_ITL platform, this bandwidth is independent of the processor frequency. To illustrate this behavior, column BW_i of Table 3.2 reports the bandwidth to the main memory experienced by a single core running the `stream` microbenchmark [121] at different frequencies.

  We note that the bandwidth-frequency dependence is a design decision specific of each processor type: more recent processors as, e.g., the Intel Xeon E52670 "Sandy Bridge" follow AMD 6128's strategy and reduce the bandwidth with the processor frequency [68]; on the other hand, some other processors like the AMD 6274 "Interlagos" apparently abandon this approach [113].

3.1.3 Power measurement devices

Several power sampling devices have been used in the evaluation of the experiments of this work. They include two external commercial wattmeters, AP8653 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), and thus measure the external AC for the full platform. A different power sampling device is our own internal DC wattmeter, which is roughly based on a choice of current transducers that produce data for a commercial data acquisition system (DAS) from *National Instruments* (NI) or, alternatively, for an alternative ad-hoc design that uses a microcontroller to sample transducer data. Table 3.3 presents in detail the specifications of these wattmeters.

External AC wattmeters. The AP8653 PDU has 24 outlets and operates at a sampling rate of 1 Hz, employing the Simple Network Management Protocol (SNMP) to send data via Ethernet. The WattsUp? Pro .NET, hereafter WattsUp, also works at 1 Hz and returns samples to the server through an Universal Serial Bus (USB) 2.0 line.

Wattmeters using NI DAS. These measurement tools were developed bearing in mind that they had to measure currents ranging from 1 to 15 A, without introducing significant voltage drops. The selected transducer was the *LEM HXS 20-NP* Hall effect current sensor. The device exhibits high accuracy and linearity, and a very low internal resistance, while being able to measure current in the required ranges.

A number of these ad-hoc 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 (NI hereafter). The module features 32 16-bit resolution analog-to-digital (AD) channels which can sample data at 7 kHz. The NI `LabView` software runs in the tracing server reading the data captured by the DAS from a USB 2.0 port in the chassis. Due to the amount of generated data, we limit the sampling frequency to 1 kHz in the experiments that use this wattmeter.

MICRONTROLLER-BASED WATTMETERS. These designs [10] feature 10 and 25 channels and a Peripheral Interface Controller (PIC) 18 microcontroller from *Microchip* to perform AD conversion. Each channel consists of the aforementioned *LEM HXS 20-NP* transducer and a 10-bit resolution AD channel in the microcontroller. All the channels share a reference voltage of 2.5 V generated by the transducers. Data are sent to the host computer through an asynchronous RS232 port. The sampling rate is therefore limited by the speed of the communication link (115,200 bauds in the selected microcontroller). For the DCM device, the sampling rate is fixed to 28 Hz, and for the DC2M it is 1 kHz (depending on the number of selected lines).

	External AC		Internal DC		
Wattmeter	AP8653	WATTSUP	NI	DCM	DC2M
Manufactured by	APC	WattsUp? Pro .NET	National Instruments	Universitat Jaume I	Universitat Jaume I
# Channels	24	1	32	12	24
Channel type	Metered-by-Outlet Rack PDU	Standard power PC cord	12 V ATX-related lines	12 V ATX-related lines	12 V ATX-related lines
Power nature	Average	Average	Instantaneous	Instantaneous	Instantaneous
Microcontroller	-	-	NI9205 NIcDAQ-9178	*Microchip* PIC 18	*Microchip* Atmel 16
Power sensors	-	-	*LEM HXS 20-NP* transducers	*LEM HXS 20-NP* transducers	*LEM HXS 20-NP* transducers
Sampling rate (Hz) per channel	1	1	7000	28	1000
Accuracy	$< \pm 1.5\,\%$	$< \pm 1.5\,\%$	$\pm 1\,\%$	$\pm 1\,\%$	$\pm 1\,\%$
Interface	Ethernet	USB/Ethernet	USB	RS232	USB
Price	1200 €	200 €	2700 €	Not commercialized	Not commercialized

Table 3.3: Specifications of the wattmeters.

3.2 Framework Environment

In this section we describe the software of the built-in framework for performance and energy profiling and tracing of applications developed as part of this work. This approach is based in post-mortem offline analysis, since the recorded data is accessed after the application execution. The main advantage of this methodology is that the data can be analyzed many times and compared with other data.

Due to the vast amount of data generated, sophisticated tools are required to localize performance and power issues of the system and to correlate them with the application behavior and the finally source code. The tools can operate either manually, i.e. the user must inspect the data by himself, or automatically, i.e. the tools try to analyze the data. The tools could also give hints to the user about where abnormalities or inefficiencies are (semi-automatic tools). Tool environments that localize and tune code automatically without user interaction are good options for all

programmers. However, due to the system and application complexities, automatic tools are only applicable for a reduced set of problems.

The tracing mechanism normally proceeds out by collecting and analyzing data in order to characterize the application execution and the system behavior. The approach to perform this analysis is realized in terms of statistics storage, comprising, e.g., absolute values for the number of invoked routines, the execution time of routines and the hardware counters. Profiling tools that output these statistics are very useful to analyze the application behavior. Tracing tools are, as well, important to analyze the different phases and the behavior of the application over time. Its extension to the power analysis also drives us to include data from the power measurement devices with the aim at correlating them with the application traces. We use a combination of all these methods.

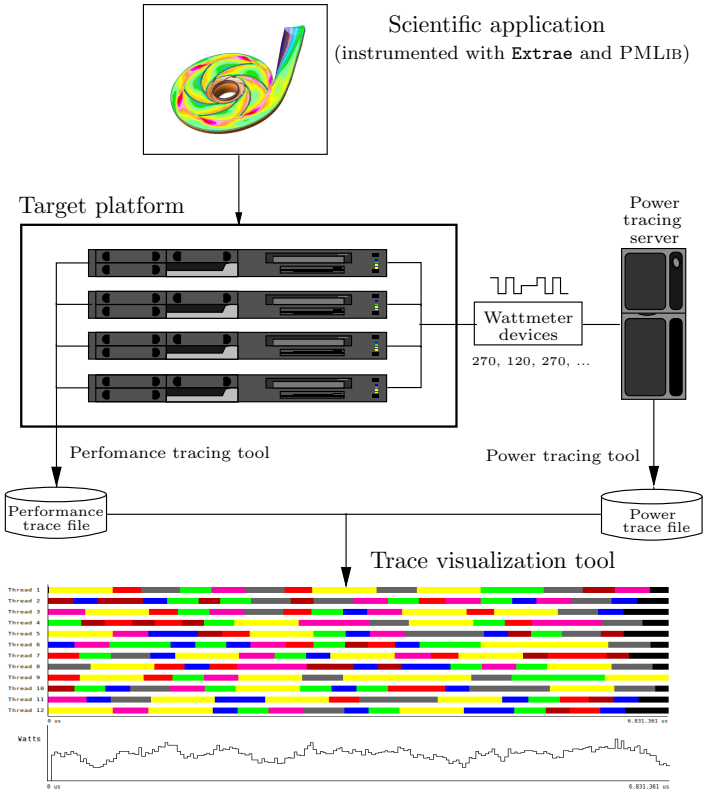


Figure 3.2: Collecting traces at runtime and visualization of power-performance data.

The framework for performance and energy/power analysis of applications developed in this work is composed of several components. Figure 3.2 depicts a representation of the framework for power-performance tracing and analysis. The starting point is a concurrent *scientific application*, instrumented with our power measurement (PMLIB) software, that runs on a parallel *target platform* (e.g., a cluster, a multicore architecture, or a hybrid computer equipped with one or more GPUs) and gives rise to a certain power consumption. Attached to the target platform there is a *wattmeter device* (or more) —either internal DC or external AC— that steadily samples power and sends the output to a *tracing server*. Calls to routines of the power measurement library, from the application which is running on the target platform, allow to perform different and complementary tasks: instruct the tracing server to start/stop collecting data captured by the wattmeters; dump the samples into a disk file (*power trace*) in a particular format; query different properties of the wattmeters; etc. Upon completion of the application execution, the power trace can be inspected,

optionally hand by hand with a performance trace, using some *visualization tool*. Our current setting allows a smooth integration of the framework power-related traces and the performance traces obtained with `Extrae`. The combined traces can be visualized with `Paraver`. Nevertheless, the modular design of the framework and the PMLib library can easily accommodate other tracing tools like TAU [116], VampirTrace [95], etc.

3.2.1 Profiling, tracing and visualization tools

`Extrae`

The `Extrae` library is a dynamic instrumentation software, developed at Barcelona Supercomputing Center (BSC), to trace programs compiled and run with the shared memory model (like OpenMP and Pthreads), the message passing (MPI) programming model or both programming models (different MPI processes using OpenMP or Pthreads within each MPI process) [57].

This package intercepts calls to MPI, OpenMP and Pthreads and records the information (events) into several tracing files that are later merged to produce a final file that can be visualized with `Paraver`.

Interposition mechanisms. `Extrae` takes advantage of multiple interposition mechanisms to add monitors into the application and collect performance metrics to provide the performance analyst a correlation between performance and the application execution. The first one, DynInst, is an instrumentation library that allows modification of the application by injecting code at specific code locations. The second method uses linker preload to inject a shared library into an application that provides the same symbols or routines as those contained in the libraries used to inject code in these calls, basically acting as a wrapper.

As a third method, `Extrae` provides other instrumentation mechanisms. They basically take advantage of some parallel programming runtimes that have their own instrumentation (or profile) mechanisms available for performance tools. The most widely known example is the Message Passing Interface (MPI), which provides the Profile-MPI (PMPI) layer. Another example is the programming model OmpSs (successor of SMPSs) which provides an instrumentation version of its compiled binaries. The result is that `Paraver` tracing files do not only contain information regarding the application evolution but also more specific information regarding the parallel programming model. Finally, `Extrae` offers the possibility of manually instrumenting the application to emit its own events in case the previous mechanisms do not fulfill the user's needs.

Sampling mechanisms. `Extrae` can be used to instrument the application code. It also offers sampling mechanisms to gather performance data. While adding monitors into specific locations of the application yields information that can be easily correlated with source code, the resolution of such data is directly related with the application control flow. By adding sampling capabilities into `Extrae` the tool allows to obtain performance information from specific regions of the code that have not been explicitly instrumented.

Performance data gathered. The monitors added by `Extrae` gather different types of information. Each monitor can be taught to collect specific information. The most common information gathered are timestamps, as well as performance and hardware counters numbers (like the PAPI [94] interface) that account for the microprocessor performance at specific sampling points of the application under test.

`Paraver`

`Paraver` is a flexible performance visualization and analysis tool developed at BSC that can be used to graphically display and examine a variety of parallel applications based on frequently used parallel tools (OpenMP, MPI, OpenMP+MPI, etc.) [104]. It provides a powerful environment to inspect the parallelism and scalability of an application as well as to obtain a number of metrics that characterize the program and its performance.

`Paraver` was developed responding to the basic need of having a qualitative global perception of the application behavior by visual inspection, and to be able to focus on the detailed quantitative analysis of the problem. It gathers and provides a large amount of information regarding different aspects of the behavior of the application under study. This information directly improves the decisions on whether and where to invest the programming effort to optimize the application. The result is a reduction of the development time as well as a minimization of the hardware resources required for it.

Some `Paraver` features are the support for:

- 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 (networks of SMPs).
- 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 states transitions, events and communications with an associated timestamp. These three elements can be used to build traces that capture the behavior over time of very different types of systems. The `Paraver` distribution includes, either in its own distribution or as an additional package, the following instrumentation tools:

- Sequential application tracing: it is included in the `Paraver` distribution and can be used to trace the value of certain variables, procedure invocations, etc., in a sequential program.
- Parallel application tracing: a set of modules that capture the activity of parallel applications using shared-memory (OpenMP directives), message-passing (MPI library), or a combination of them.
- System activity tracing in a multiprogrammed environment: an application to trace processor allocations.

`Paraver` allows users to develop their own tracing facilities according to their own interests and requirements. The visualization, semantic and quantitative modules are powerful enough to allow users to analyze and understand the behavior of the traced systems.

3.2.2 The power measurement library: PMLib

The power measurement library (PMLib) software package is developed and maintained at the Universitat Jaume I to investigate power usage of HPC applications. The current implementation of this package provides an interface to utilize the wattmeters described in Section 3.1.3 and a

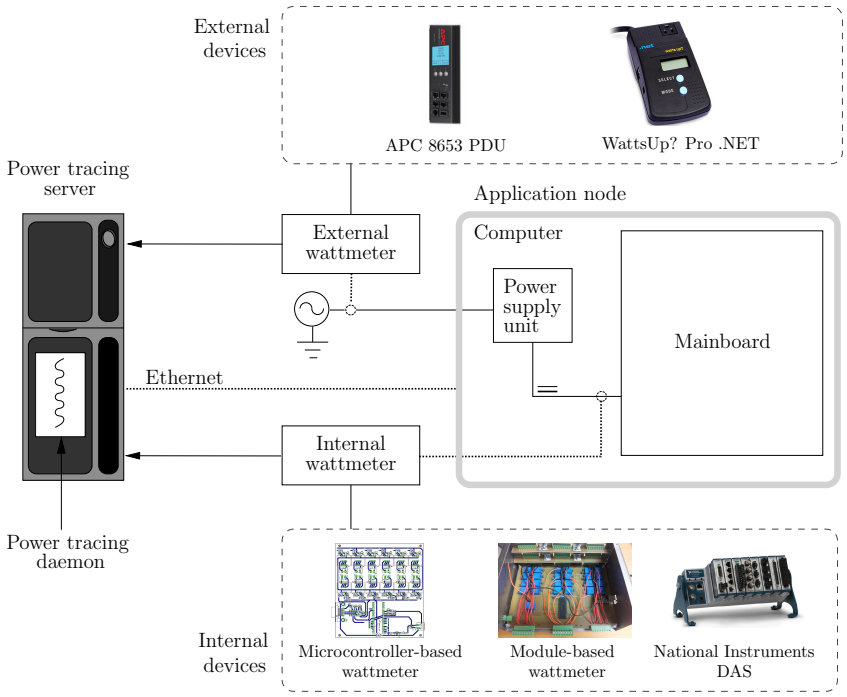


Figure 3.3: Single-node application system and sampling points for external and internal wattmeters.

number of tracing tools. The system and sampling points for external and internal wattmeters are illustrated in Figure 3.3.

Next we portray the interface of PMLib using a practical example (user's view), offer a few key implementation details (developer's view), and describe the functionality of a module to gather information about power-related states of the processor cores. We close this section by illustrating the type of information provided by the framework using a simple parallel application.

User's view

Power measurement is controlled from the application using a collection of routines that allows the user to query information on the power measurement units, create counters associated to a device where power 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 as well as returning the appropriate answers, via the interface of the PMLib routines, to the invoking application (client). The basic routines from the PMLib API are summarized in Table 3.4. The global client-server interaction is exposed in Figure 3.4.

Listings 3.1 displays a detailed example that illustrates 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, located in a separated machine to avoid interfering with the parallel application, returns power samples. C-states are recorded using the second server, which is placed in the same machine where the parallel application runs, so that it can query the files of this machine containing the requested data on C-states. The invocation to function `pm_get_devices` establishes a communication with the server to obtain a list with the names of the wattmeters

`int pm_set_server( char *svrip, int port, server_t *svr )`
Purpose: Initializes the server's IP address and port to be used for the communication with PM server. svrip: IP address of PM server. port: Communication port with PM server. svr: Server address specification as return parameter.
`int pm_get_devices( server_t *svr, char** ldev, int *ndev )`
Purpose: Queries for the current connected measurement devices. svr: Server address specification as return parameter. ldev: List of current connected devices. ndev: Number of devices.
`int pm_get_device_info( server_t *svr, char* devn, device_t * dev )`
Purpose: Returns the maximum sampling frequency and number of lines of device. svr: Server address specification as return parameter. ndev: Number of devices. dev: Device structure with frequency and avail. lines.
`int pm_create_counter( char *devn, mask_t lin, int aggr, int freq, server_t svr, counter_t *pm_ctr)`
Purpose: Sends a request to the PMLib server (PS) in order to create a new power counter. PS creates a counter. devn: String which identifies the selected wattmeter. lin: Mask which selects the lines of wattmeter to be measured. aggr: Boolean which tells to PS if return an aggregate power for selected lines or not. freq: Integer with the desired working frequency. svr: Server address specification. pm_ctr: Counter structure as return parameter.
`int pm_start_counter( counter_t *pm_ctr )`
Purpose: Starts the power measurement. pm_ctr: Counter structure.
`int pm_continue_counter( counter_t *pm_ctr )`
Purpose: Continues the power measurement preserving previous power data of counter. pm_ctr: Counter structure.
`int pm_stop_counter( counter_t *pm_ctr )`
Purpose: Stops the power measurement. pm_ctr: Counter structure.
`int pm_get_counter_data( counter_t *pm_ctr )`
Purpose: Dumps power data onto memory. pm_ctr: Counter structure.
`int pm_print_data_stdout( counter_t *pm_ctr )`
Purpose: Dumps power data into stdout. pm_ctr: Counter structure.
`int pm_print_data_paraver( char *file, counter_t *pm_ctr, char *unit )`
Purpose: Dumps power data into `Paraver`-like trace file format. file: Output filename. pm_ctr: Counter structure. unit: Time unit.
`int pm_finalize_counter( counter_t *pm_ctr )`
Purpose: Finalizes the counter. pm_ctr: Counter structure.

Table 3.4: Basic routines of the PMLib API.

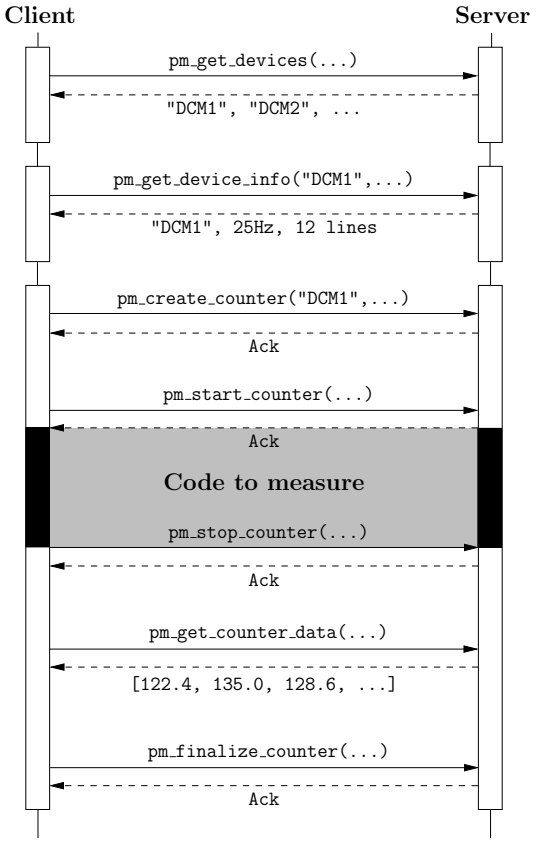


Figure 3.4: Diagram of the communication between client (running a scientific application) and the (PMLib) server.

connected. A call to `pm_get_device_info` returns more specific information on a given device from the set of wattmeters detected.

With the next two calls to `pm_set_lines`, we select the lines to measure since distinct wattmeters may have different numbers of lines. Next, we also call function `pm_create_counter` 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 kernel `dgemm`. The sampling process is momentarily interrupted then, by invoking `pm_stop_counter`, and continued later, with `pm_continue_counter`, to record only power samples for kernel `dsyrk`. Finally, routine `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); and the counters are destroyed using routine `pm_finalize_counter`.

Developer's view

The PMLib software is written in Python and consists of two modules: the *settings* file and the *server*. Figure 3.5 depicts how the server operates. The daemon starts by initially reading the settings file, which contains configuration information on the wattmeters available in the system. After that, a new thread is created per wattmeter in order to manage and receive data from these devices. The server then creates the number of counters (i.e., new thread instances) required by the clients.

```
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\nMax freq: %d\nNumber of lines: %d\n",
            disp.name,disp.max_frecuency,disp.n_lines);

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

    // Creates a counter for wattmeter DCMeter1
    pm_create_counter("DCMeter1", lines1, !aggr1, freq1, server1, &counter1);

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

    // Starts 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 );

    // Stops to collect samples
    pm_stop_counter(&counter1);
    pm_stop_counter(&counter2);

    // ...  Some other nonsampled   ...
    // ... aplication 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);

    //Stops to collect samples
    pm_stop_counter(&counter1);

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

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

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

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

Listing 3.1: Example of use of PMLib.

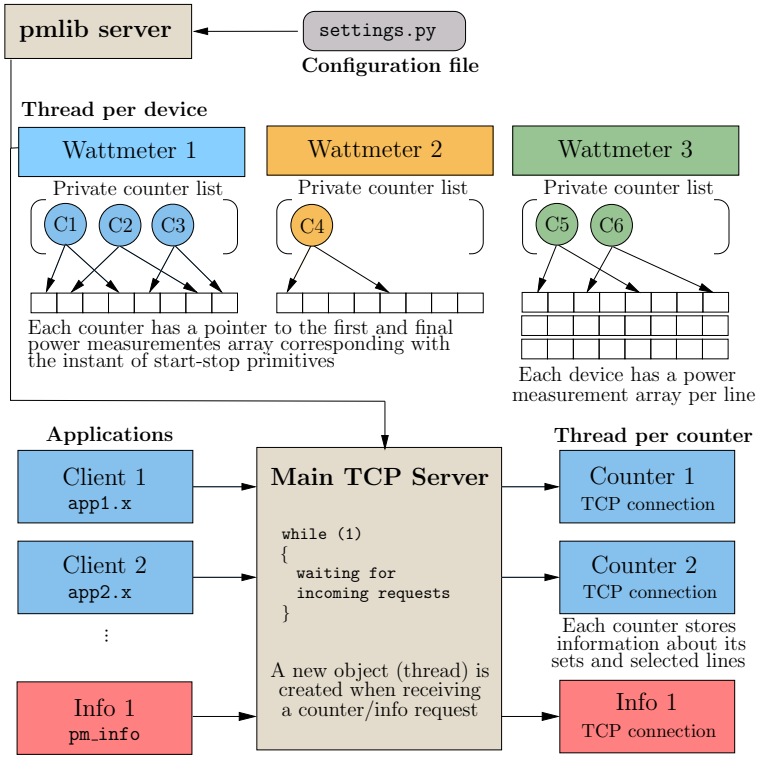


Figure 3.5: Internal workings of the PMLib server.

The main threaded classes implemented by the server are:

`Device.` This class reads data from a specific wattmeter and stores a list of object-pointers into all the active counters.

`Counter.` This class manages all the operations performed on a counter. It is stored in the `Device` object it is associated with. The class contains data acquired while the counter is running.

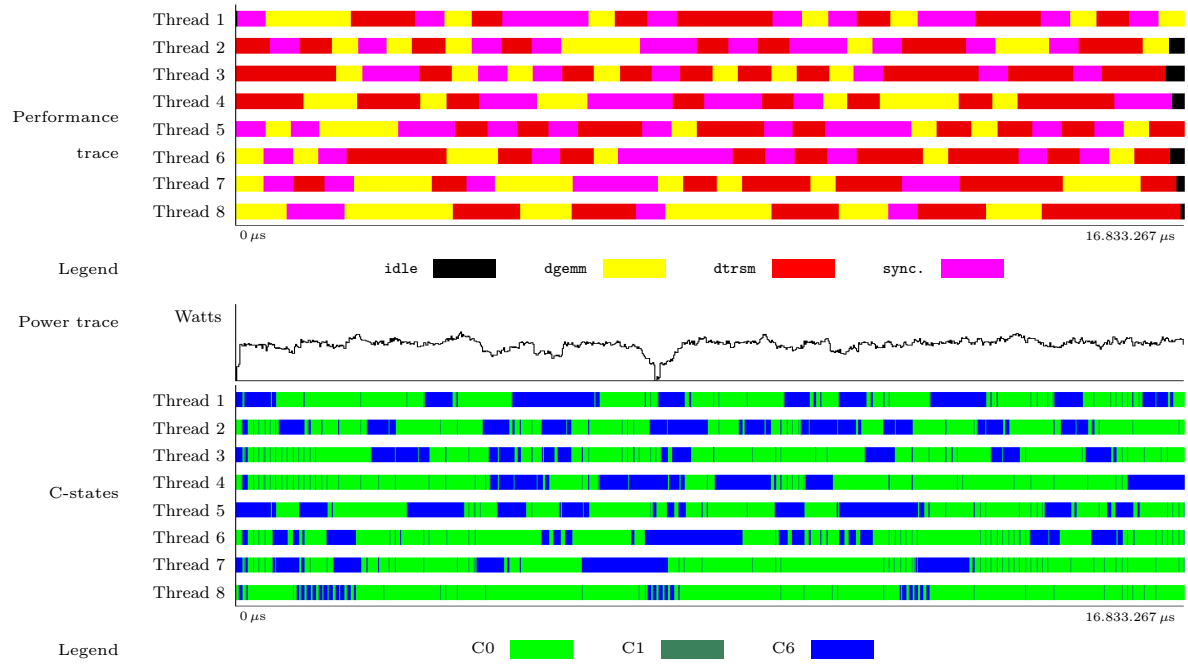
`Info.` This class comprises information about the devices and their configuration.

As shown in Figure 3.5, the server can receive two types of *requests*, either a query on information about a device or an operation on a counter. In the first case, the server creates an `Info` object to obtain the required data from the settings file and sends them back to the client.

If the operation is a request to create a counter, the server allocates a `Counter` object, which will manage all subsequent operations on it as well as store the structure in the appropriate `Device` object. After creation of a counter, the client should invoke `pm_start_counter` to instruct the server to start recording samples and `pm_stop_counter` to stop counting. The client can also use `pm_continue_counter` to restart the recording process and force the server to record samples from other fragments of the application code in the same counter, generating different `sets` of data. Finally, all collected data can be retrieved by invoking `pm_get_counter_data`.

3.2.3 Power-related states module

Our power framework obtains a trace of the C- and P-states for each core. For example, in order to obtain information on the C-states, a daemon integrated into the power framework accesses the model-specific registers (MSR) of the cores, with a user-configured frequency. The daemon reads



	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 3.6: Example of performance and power traces captured by `Extrae` and the proposed power framework, visualized with `Paraver`.

values from these registers corresponding to the total time 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.

Note that the state-recording daemon necessarily has to run on the target 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 non-negligible. To avoid this effect, the user is advised to adjust experimentally the sampling frequency of this daemon carefully.

Figure 3.6 shows a graphical example of the type of the information that can be collected with our power-tracing framework when combined with the performance tracer `Extrae` and the visualization tool `Paraver`. The view depicted corresponds 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 WT_ITL. The performance trace in the top plot displays 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 that cores experience between processor states C0, C1, C3 and C6 (with a sampling frequency of 10 Hz). The final part reports the same information contained in the performance and C-states traces in numerical format.

3.2.4 Example of use

Listing 3.2 displays a simplified version of a C routine for the computation of the LU factorization with partial pivoting of a matrix `A`. The blocked routine loops over variable `j` processing `b` columns of the matrix at each iteration. The code invokes the numerical kernels `dgetf2`, `dlaswp` (twice), `dtrsm`, and `dgemm`. Routines `Extrae_init` and `Extrae_fini`, from the `Extrae` API, initialize and finalize the tracing tool. Routine `Extrae_event` records an event into the trace file and a reference to the point in the source code. If the second argument is not 0, it marks the beginning of the event; otherwise, it marks the end. The first argument indicates the event type and it is simply set to `500000001` in our example. For the second argument we used values $1, 2, \ldots, 4$ to uniquely identify the different numerical kernels invoked from the code. The measurement is initiated and terminated from the application via routines `pm_start_counter` and `pm_stop_counter`, respectively. The results are retrieved using `pm_get_counter_data`, and printed in `Paraver` format afterwards. Finally, performance and power traces are merged in only one `Paraver`-format trace file.

3.3 Experimental Results

In this section we provide a detailed power and performance analysis of a dense linear algebra code to demonstrate the use of our performance-power framework on a multicore technology platform. This study offers a vision of the power drawn by the system during the execution of a few high-quality implementations the LU factorization [63]. This factorization is the key to the solution of dense linear systems.

In order to collect and illustrate this information, we bind a trace of the algorithm execution obtained by using the proposed framework `Extrae`+`Paraver` with our own power evaluation setup and using our power measurement library PMLib and the internal DCM wattmeter.

3.3.1 Environment setup

The following experiments were carried out using IEEE double-precision arithmetic on platform WT_AMD. The implementation of BLAS was that provided in Intel MKL (v10.3.4). Tracing and visualization were obtained with `Extrae` (v2.2.0) and `Paraver` (v4.1.0).

In our evaluation, power measurements are collected using the internal DCM wattmeter. To obtain the nodal power, the internal wattmeter is directly attached to the 12 V lines connecting the PSU with the motherboard (chipset plus processors) of the test platform. Therefore, the results are not affected by inefficiencies of the PSU, or the "noise" due to the operation of other hardware components like fans, disks, network interfaces, etc.

Three implementations were evaluated for the LU factorization:

- LAPACK: The legacy code available at netlib [84] for the LU factorization with partial pivoting (`dgetrf`). In this case parallelism is exploited within the invocations to multithreaded MKL BLAS. Except where otherwise stated, the block size was $b = 128$; for the problem sizes, architecture and BLAS employed in our experiments, b was always close to the optimal.

```
#define Aref(i,j) A[((j)-1)*Alda+((i)-1)]

void dgetrf( int m, int n, int b, double *A, int Alda, int *ipiv, int *info ) {

  // Declaration of variables (omitted)

  pm_start_counter(&c);
  Extrae_init();
  for (j= 1; j<= min(m, n); j+= b) {

    Extrae_event(500000001,1);
    // Factor current panel
    dgetf2( m-j+1, b, &Aref(j,j), Alda, &ipiv[j-1], info );
    Extrae_event(500000001,0);

    Extrae_event(500000001,2);
    // Apply permutations to left and right of panel
    dlaswp( j-1, A, Alda, j, j+b-1, ipiv, 1 );
    dlaswp( n-j-b+1, &Aref( 1, j+b ), Alda, j, j+b-1, ipiv, 1 );
    Extrae_event(500000001,0);

    Extrae_event(500000001,3);
    // Triangular solve
    dtrsm( "L", "L", "N", "U", b, n-j-b+1, done, &Aref(j, j), Alda, &Aref(j, j+b), Alda );
    Extrae_event(500000001,0);

    Extrae_event(500000001,4);
    // Update trailing submatrix
    dgemm( "N", "N", m-j-b+1, n-j-b+1, b, -done, &Aref(j+b, j), Alda,
                     &Aref(j, j+b), Alda, done, &Aref(j+b, j+b), Alda );
    Extrae_event(500000001,0);
  }
  Extrae_fini();
  pm_stop_counter(&c);
}
```

Listing 3.2: Blocked routine for the LU factorization annotated with the `Extrae` and the PMLIB libraries.

- MKL: The code from the Intel library for the LU factorization with partial pivoting (using the same naming convention as in the LAPACK case).
- SMPSs: C code for the LU factorization with incremental pivoting linked to the sequential MKL BLAS, with task-level parallelism extracted by the SMPSs runtime system [26]. In this routine, the data matrix is partitioned into square $t \times t$ blocks (tiles), with $t = 256$ and the inner block size b is set to 64.

3.3.2 The LU factorization

In the experiments with the LU factorization we set $m = n = 10{,}240$ and used 12 threads from a single socket of WT_AMD. We neglect lower order terms in the cost expressions, through in order to cast the operation in terms of high performance kernels, the LU factorization with incremental pivoting requires a certain amount of additional operations (under mild conditions) of minor order. Similar results were obtained with other problem dimensions and different number of cores.

Figure 3.7 reports the kernel invocation/core activity and power consumption obtained with the LAPACK routine `dgetrf` for the LU factorization with partial pivoting. The different colors identify events (kernels) invoked from the routine: `dgetf2` (factorization the current panel),

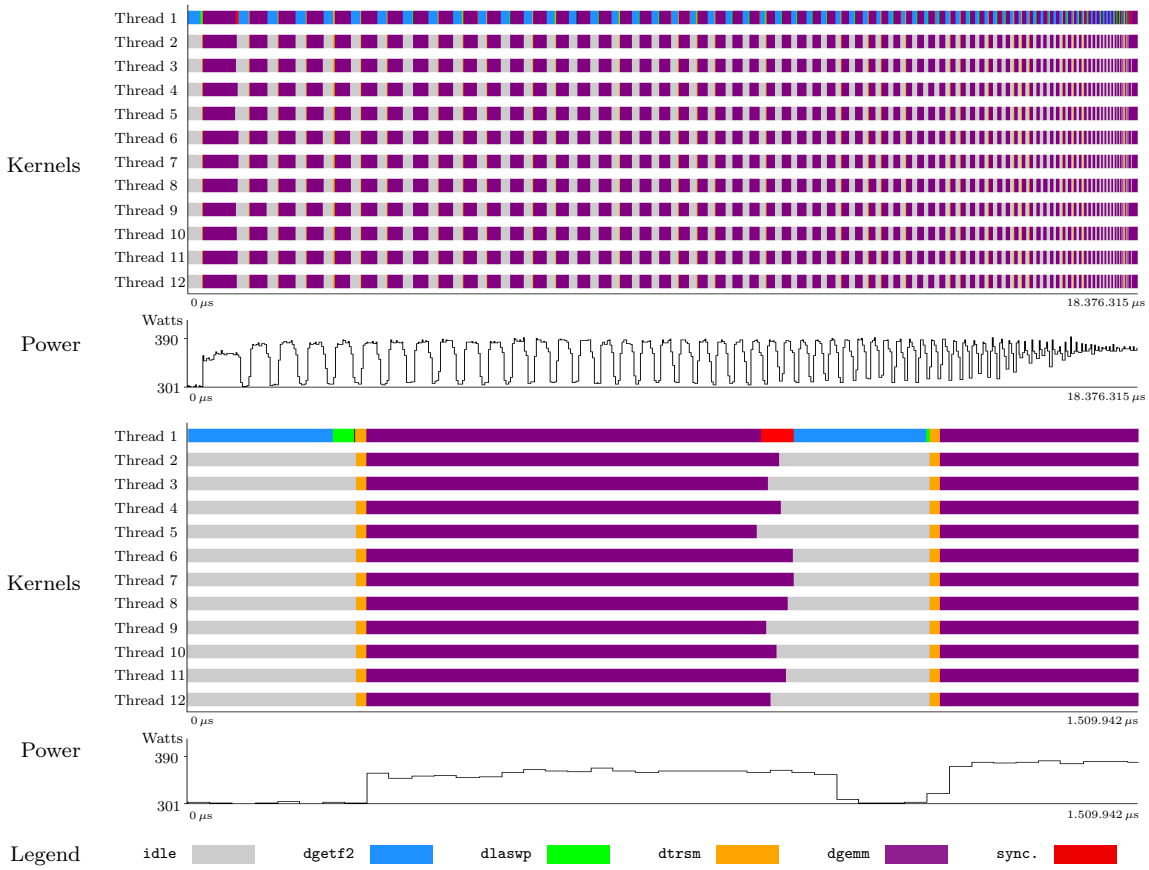


Figure 3.7: Trace of LAPACK `dgetrf`. Top: full. Bottom: first two iterations.

`dlaswp` (application of permutations), `dtrsm` (triangular system solve), `dgemm` (update the trailing submatrix via a matrix-matrix product) —see the code in Listing 3.2—, as well as idle time and other system activity (e.g., thread synchronization). The top half of the figure (first two plots) offers the complete trace while the bottom one (third and fourth plots) zooms into the first two iterations of the routine. These results illustrate that the LAPACK routine, combined with the multithreaded MKL BLAS, interleaves the execution of sequential and concurrent phases in this platform. The sequential ones correspond to the execution of kernels `dgetf2` and `dlaswp`, while `dtrsm` and `dgemm` proceed in parallel and involve all the cores. The performance traces also reveal that kernels `dgetf2` and `dgemm` dominate the execution time of the routine. The bottom half of the figure indicates that a synchronization occurs at each iteration after the execution of kernel `dgemm` due to an unbalanced distribution of the workload among the cores during this operation. From the power consumption perspective, the interlaced sequential and concurrent activity leads to periods of low and high power, respectively, which vary between 301 and 390 W.

Figure 3.8 evaluates the implementation of the MKL routine `dgetrf` for the LU factorization with partial pivoting. In this case we have no access to the source code, only to the binary included in the library. Therefore we could not identify the kernels being invoked from within. In order to face this fact, we apply reverse-engineering to learn more about what is brewing inside the MKL routine by sampling the hardware counters for the L2 cache misses and the MFLOPS rate (`PAPI_L2_DCM` and `PAPI_FP_INS`, respectively). Then, we confront these data with some known properties of the four kernels potentially involved in the factorization. In particular, kernel `dgetf2`

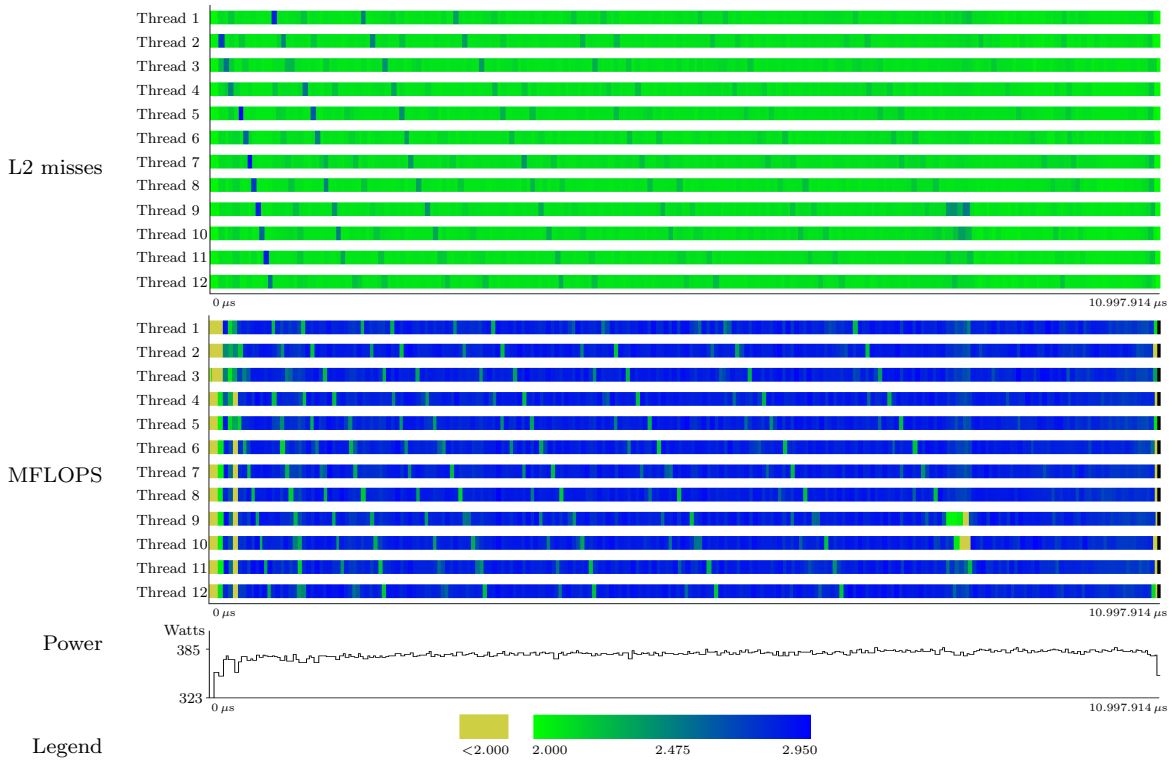


Figure 3.8: Trace of MKL `dgetrf`.

is composed of BLAS-1 and BLAS-2 operations and, thus, it can be expected to deliver a low MFLOPS (or a high L2 cache miss) rate; `dlaswp` performs no FLOPS at all; on the contrary, `dgemm` and `dtrsm` are both BLAS-3 operations that should deliver high values for the MFLOPS counter. From the performance trace in the bottom of the figure, we can deduce that, in the MKL routine, the factorization of the current panel performed by one single core via kernel `dgetf2` is overlapped with the updates to the rest of the matrix. (In sophisticated implementations, this is usually attained via the application of look-ahead to some depth [120].) Besides, the time required by this factorization is much smaller than that of the LAPACK code, which implies the use of a narrower panel width (b) and/or a more efficient implementation. Furthermore, the core in charge of this factorization rotates during the execution of the routine, while in the LAPACK implementation this operation was always performed by the same core. The trace also identifies a synchronization point towards 3/4 of the execution time.

Figure 3.9 shows the activity and power consumption of the SMPSs task-parallel C implementation of the LU factorization with incremental pivoting. This algorithm is composed of four basic kernels: `dgetrf` (LU factorization with partial pivoting of a tile); `dtrsm` (triangular system solve involving two tiles: the coefficient triangular matrix and the right-hand side); `dgetrf2x1` (LU factorization of a matrix consisting of 2×1 tiles, with the top one being upper triangular); and `dgemm2x1` (application of the Gauss transforms resulting from the previous kernel to a matrix of 2×1 tiles); see [108, 26] as well as Section 2.4.2 for details. The last kernel dominates the theoretical cost and, as the figure clearly exposes, the execution time of the implementation. There are very little synchronization points, due to the higher concurrency of this particular algorithm, which is leveraged by SMPSs to maintain all cores/threads executing tasks (kernels) most of the

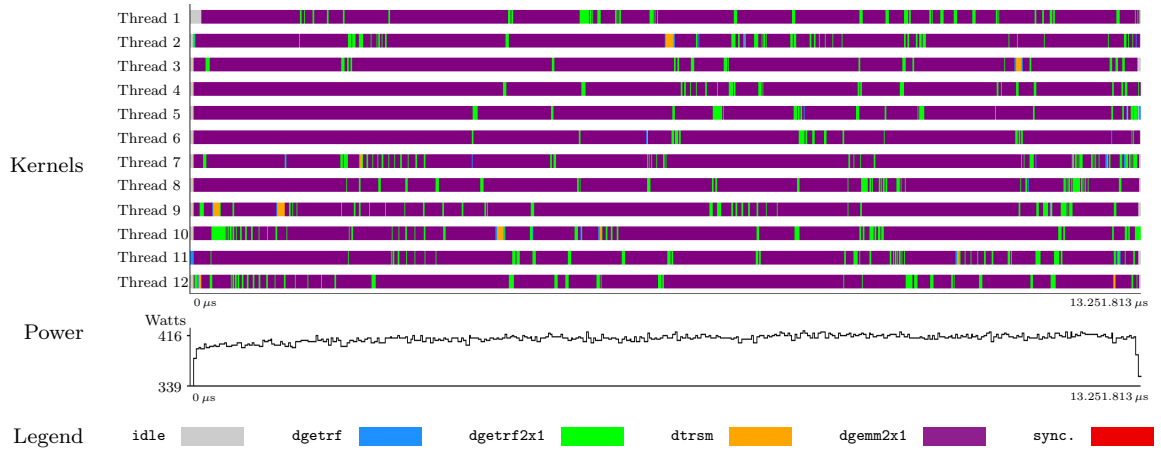


Figure 3.9: Trace of the C implementation of the LU factorization with incremental pivoting parallelized with SMPSs.

	LAPACK	**MKL**	**SMPSs**
T (s)	18.37	10.99	13.25
GFLOPS	38.96	65.13	54.02
P_{max} (W)	390.70	385.78	392.81
P_{min} (W)	301.64	294.37	328.12
P_{avg} (W)	359.72	377.94	385.56
P_{wrk} (W)	112.22	130.44	138.06
E_{tot} (J)	6,608.60	4,155.61	5,109.44
E_{wrk} (J)	2,061.48	1,433.54	1,829.30

Table 3.5: Performance, power and energy of the different implementations of the LU factorization.

time. The power line of the SMPSs routine is quite homogeneous, corresponding to the dominance of a BLAS-3 kernel like `dgemm2x1` and the lack of significant idle periods during the execution.

Table 3.5 compares the three factorization algorithms using several parameters: Time (T, in seconds); GFLOPS (10^9 FLOPS); minimum, maximum and average power (P_{min}, P_{max}, P_{avg}, respectively, in Watts); the average workload power (P_{wrk}, in Watts) obtained by subtracting the power consumed by the platform when idle (247.50 W) from P_{avg}; total energy and workload energy (E_{tot} and E_{wrk}, in Joules), with $E_{\text{tot}} = P_{\text{avg}} \cdot T$ and $E_{\text{wrk}} = P_{\text{wrk}} \cdot T$. In principle, this last variable captures the energy cost of running the application, because it eliminates the fixed power that has to be paid to keep the machine active doing nothing. This alternative to bridge system power yields more accurate figures to compare the energy efficiency of the different algorithms.

Overall, the higher concurrency of the MKL routine and the lack of synchronization points lead to its superior performance in terms of execution time over the LAPACK implementation. On average, the power usage of the MKL routine is close to 380 W which, combined with its shorter execution time, dictates the superiority of the MKL routine from the viewpoint of total energy consumption. For this particular problem size, number of threads, and platform the execution time of the SMPSs algorithm is longer than that of the MKL routine, basically because of the higher number of FLOPS that are required by the LU factorization with incremental pivoting (which can be regarded as overhead, if compared with the LU factorization with partial pivoting) and the inferior performance (GFLOPS rate) of some of the kernels involved.

3.4 Concluding Remarks

In this chapter we have reviewed the existing power-aware mechanisms in the CPU and the memory and we have described the platforms that will be used in the experiments performed in this dissertation. We have also 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 highly 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 multithreaded codes that execute on a multicore+GPU platform.

Finally, we have performed a study of the computational performance, power profile and energy consumption of a few efficient implementations of the LU factorization (with partial pivoting, `dgetrf`, and with incremental pivoting). The LAPACK implementation of `dgetrf` exhibits interlaced sequential and concurrent phases, which yield low and high peaks in the power profile. The analysis of the equivalent routine from MKL is more difficult, due to the black-box nature of this library. Nevertheless, the use of a hardware counter for the MFLOPS (or the L2 cache misses) rate offers partial information on the internal structure of the implementation which can then be used to bind this information to the power profile. Specifically, during most of the time, the implementation of `dgetrf` of MKL maintains a high level of occupancy (i.e., MFLOPS) in all but one core, and thus reduces the execution time while maintaining the power rate. The SMPSs parallel implementation of the LU factorization with incremental pivoting delivers levels of concurrency similar to those of MKL. By keeping the hardware resources doing useful work most of the time, the SMPSs codes reduce the execution time and maintain a stable power usage.

The final conclusion of this analysis ties execution time and energy. The MKL/SMPSs routines present a higher average power than their LAPACK counterpart, but also a more reduced execution time which determines their superior energy efficiency. In other words, by keeping the cores busy most of the time, the algorithms in MKL/SMPSs benefit from "race-to-idle". Given the high energy cost of keeping the machine "active" when there is no workload to run, this approach clearly pays off for such computationally-intensive algorithms.

CHAPTER 4

Modeling Power and Energy Consumption

In this chapter we introduce simple yet accurate models for the power dissipation and energy consumption during the execution task-parallel linear algebra algorithms on multicore and multi-threaded platforms. The kernels of these algorithms operate with one or more blocks of the data matrix, and can be executed in a certain order dictated by data dependencies among them. In order to exploit the data parallelism existing in the matrix factorizations transparently, the SuperMatrix and SMPSs runtimes execute task-parallel implementations of the corresponding dense linear algebra algorithms (see Section 2.2.3).

While there is a collection of previous work [37, 80, 35, 62] that also introduce reliable models of power consumption for general benchmarks, in general these approaches employ hardware counters to do so. Our methodology departs from these other work in that we leverage the particular properties of dense linear algebra operations to estimate power and energy only from the execution time and theoretical cost of the kernels that compose the operation, thus eliminating the need for cumbersome, platform-dependent hardware counters. Our models are shown to cover a significant part of the functionality offered by current dense linear algebra libraries like LAPACK or `libflame`, in general delivering power/energy estimates within a small factor of the real consumption. Several models for different architectures are proposed in this chapter. The simple model is able to capture power and energy consumed by dense linear algebra algorithms, and it is further refined with a contention-aware instance. We also introduce three more models that address platforms with DVFS, multi-socket and hybrid CPU–GPU platforms.

The chapter is structured as follows. In Section 4.1 we introduce the power and energy models, and describe the methodology to gather and assemble the necessary data. Sections 4.2 and 4.3 present two variants which correspond to the simple and contention-aware approaches. In Section 4.4 we provide an additional power model that can handle the P-states. Next, Section 4.5 introduces two new instances of the model for multi-socket and hybrid CPU–GPU platforms. Finally, we offer some concluding remarks in Section 4.6.

4.1 Formulation of the Power Model

Consider a task-parallel algorithm for a linear algebra operation, say *Op*(eration), that can be decomposed into r different types of tasks (kernels) that operate on (square) blocks of size b.

Assume the algorithm runs concurrently on a parallel system consisting of c cores with one thread per core. In order to predict the power dissipated by the execution of the proposed linear algebra operation, at a given instant of time t, we will adopt the following aggregate model [23]:

$$\begin{aligned} P_{Op}(t) &= P^Y + P^C(t) \\ &= P^Y + P^S + P^D_{Op}(t) \\ &= P^Y + P^S + \sum_{k=1}^{c} \sum_{j=1}^{r} P^D_{j,b} \cdot N_{k,j}(t) \ , \end{aligned} \tag{4.1}$$

where the terms of the model are defined as follows:

- P^C is the power dissipated by the CPU;
- P^Y is the power dissipated by the remaining components (system power corresponding, e.g, to RAM, mainboard, etc.);
- P^S and $P^D_{Op}(t)$ are, respectively, the static power (mainly due to leakage) and dynamic power for the CPU;
- $P^D_{j,b}$ is the dynamic power of a task of type j that operates with blocks of size b; and
- $N_{k,j}(t) = 1$ if the k-th thread/core is executing a task of type j at time t or equals 0 otherwise.

Furthermore, we will make the following considerations:

- Linear algebra operations (e.g., Cholesky, LU or QR factorizations) mainly exercise the floating-point arithmetic units of the processors and the main memory. Therefore, we can focus on the power dissipated by the components integrated in the mainboard (e.g., CPU and RAM chips), discarding other power sinks due, e.g., to network interface, disk, inefficiencies of the power supply unit, etc.
- P^Y and P^S remain constant during the execution of the algorithm. In practice, starting from an idle (cold) platform, P^S grows with the system temperature till it reaches a plateau [23]. To avoid this effect, we consider there is a continuous compute-bound workload to run in the platform and, in order to mimic this situation, all our tests are performed on a "hot" system, with this state reached by initially warming the cores with a compute-intensive benchmark.
- $P^D_{j,b}$ depends on "static" properties of the corresponding kernel such as computational cost, type of arithmetic operations and memory access pattern, as well as the algorithmic block size b chosen for the operation.
- The energy consumption of the algorithm is obtained by integrating its power dissipation over its execution time T; i.e.

$$\begin{aligned} E_{Op} &= \int_{t=0}^{T} P_{Op}(t)\ dt \\ &= (P^Y + P^S) \cdot T + \int_{t=0}^{T} P^D_{Op}(t)\ dt \ . \end{aligned} \tag{4.2}$$

4.2 The Simple Power Model

We first consider the derivation of a simple power model designed for a system equipped with a single multicore socket. To guide the formulation of the model we use the WT_ITL platform, considering only a single socket, and the NI internal wattmeter. In our experiments we set the

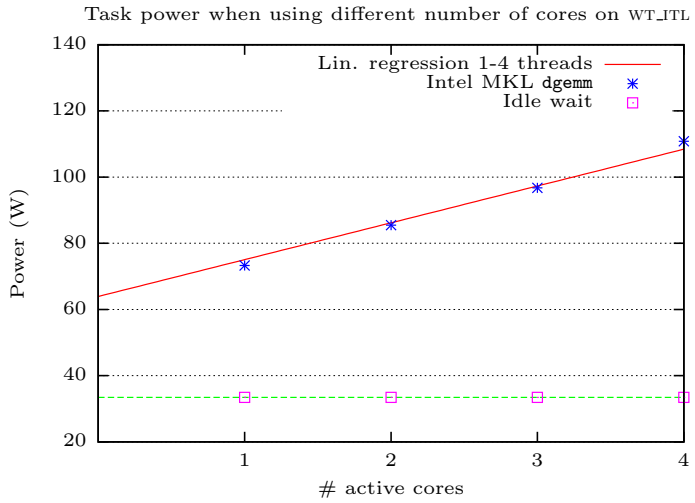


Figure 4.1: Power dissipated as a function of number of active cores on WT_ITL.

sampling frequency to 1 kHz, which was judged to be sufficient to attain reliable measures for the model. We next employ the kernels of the Cholesky factorization to obtain static estimations of the dynamic power. Finally, we consider the SMPSs runtime to exploit task parallelism in this factorization.

4.2.1 System and static power

Let us first estimate the system and static parameters of the proposed power model. In order to derive P^Y for (4.1), we directly measure the power with the platform completely idle. This yields a power dissipation of 42.61 W, which can be taken as an approximation for P^Y

In order to obtain the P^S component we apply the following methodology. We first obtain the power consumption of WT_ITL with c cores executing each an instance of the kernel `dgemm` (real, double-precision parallel matrix-matrix product from Intel MKL) using square operands (matrices) of order 512. In order to obtain a sample with already warm components, the tests were run during 750 s. before the power was measured. We next apply linear regression to adjust the power samples to the linear model $P^T_{\texttt{dgemm}}(c) = \alpha + \beta \cdot c$. The linear model for WT_ITL platform leads to $P^T_{\texttt{dgemm}}(c) = 63.93 + 11.12 \cdot c\,\text{W}$, as depicted in Figure 4.1. In the linear model, β captures the power dissipated per core invoking the matrix-matrix product, and α accounts for the power needed to maintain the different components in the socket in a power-active mode. (Analogous tests with a variety of kernels showed negligible differences for α.) We therefore approximate $P^S \approx \alpha - P^Y = 30.50\,\text{W}$; see Table 4.1. Note that P^Y and P^S are directly related to the platform, and thus they are independent of the modeled operation.

Platform	P^Y	α	β	P^S
WT_ITL	33.43	63.93	11.12	30.50

Table 4.1: Parameters for the simple power model of WT_ITL.

4.2.2 Simplistic estimate of the task dynamic power

Let us start noting that the dynamic power is directly related to the nature of the kernel, thus different dynamic powers per kernel will be obtained for a linear algebra operation with different kinds of kernels. To guide our methodology we approximate the dissipated power during the execution of each one of the four kernels (building blocks) of the Cholesky factorization (see Section 2.3), —`dpotrf` (P^D_{C}), `dtrsm` (P^D_{T}), `dsyrk` (P^D_{S}), and `dgemm` (P^D_{G})— on the WT_ITL platform. For that purpose, we perform an experiment where one single thread continuously invokes one of these kernels, e.g. `dgemm`, till the power stabilizes and then sample this value, say $P^T_{\texttt{dgemm}}$; we then set $P^D_{\mathsf{G}} = P^T_{\texttt{dgemm}} - P^S - P^Y = P^T_{\texttt{dgemm}} - 63.93$ W. Table 4.2 collects the average power determined from this experiment for the four task types and different values of the block size b (kernel/task granularity). Note that there is one additional line in the table, labeled as "P^D_{B} (`busy`)". This reflects the average power dissipated by one thread polling for work in a busy-wait in the SMPSs runtime. Our practical experiments determined that this query process also dissipates a considerable amount of power and, therefore, must be taken into account. A separate experiment determined the value in the table, which is independent of the task granularity.

	Block size, b			
Task	128	192	256	512
P^D_{P} (`dpotrf`)	10.26	10.35	10.45	11.28
P^D_{T} (`dtrsm`)	10.12	10.31	10.32	10.80
P^D_{S} (`dsyrk`)	11.22	11.47	11.67	12.60
P^D_{G} (`dgemm`)	11.98	12.54	12.72	13.30
P^D_{B} (`busy`)	7.62	7.62	7.62	7.62

Table 4.2: Dynamic power (in Watts) of the Cholesky factorization kernels and busy-wait on WT_ITL.

4.2.3 Formulation of the simple power model

Consider e.g. the task-parallel execution of the Cholesky factorization, with the algorithm decomposed into s different types of tasks or building blocks (see Table 4.2), using c threads/cores. The total power dissipation of the algorithm, at an instant of time t, is given by the composition of the system power, the static power, and the dynamic power of all tasks being executed at that instant:

$$\begin{aligned} P_{Op}(t) &= P^Y + P^S + \sum_{k=1}^{c}\sum_{j=1}^{r} P^D_{j,b}\cdot N_{k,j}(t) \\ &= 33.43 + 30.50 + \sum_{k=1}^{c}\sum_{j=1}^{r} P^D_{j,b}\cdot N_{k,j}(t)\ ; \end{aligned} \tag{4.3}$$

see Tables 4.1 and 4.2.

The energy consumption of the algorithm for the Cholesky factorization can be written as a function of the time spent by the threads on each type of task as follows:

$$\begin{aligned} E_{Op} &= \int_{t=0}^{T}(P^Y + P^S + P^D_{Op}(t))\ dt \\ &= (P^Y + P^S)\cdot T + \int_{t=0}^{T} P^D(t)\ dt \\ &= (P^Y + P^S)\cdot T + \sum_{k=1}^{c}\sum_{j=1}^{r} P^D_{j,b}\left(\int_{t=0}^{T} N_{k,j}(t)\ dt\right) \\ &= (33.43 + 30.50)\cdot T + \sum_{k=1}^{c}\sum_{j=1}^{r} P^D_{j,b}T_{k,j}\ , \end{aligned} \tag{4.4}$$

where $T_{k,j}(t)$ is the time that the k-th thread spends running tasks of type j. Thus, in order to obtain the total energy, we only need a *profile* of the task activity that informs on the total execution time per task type.

4.3 The Contention-Aware Power Model

We present next the derivation of our contention-aware power model for the particular case of multicore platforms that contain only one socket. To guide the explanations we use a single socket of WT_ITL measured with the NI internal wattmeter. Alike the previous section, we continue with the SMPSs runtime, but now consider the three major dense matrix factorizations (Cholesky, LU and QR) to illustrate this version of the power model.

4.3.1 System and static power

We leverage the same methodology described in Section 4.2.1 to estimate system and static parameters of WT_ITL, which appear summarized in Table 4.1.

4.3.2 Contention-aware estimation of the task dynamic power

In the simple model we used "static" values for the dynamic power which depended on the kernel type and block size (see Table 4.2). The problem with this strategy is that, in a concurrent execution of a task-parallel dense linear algebra operation on a multicore processor, the power drawn by a particular kernel/thread/core is not constant (static) but greatly depends on the memory contention, which cannot be estimated *a priori*. In other words, at a given instant of time during the concurrent execution of a dense linear algebra operation, there will be a number of active threads executing tasks/kernels (note that one or more threads may be blocked, waiting for "work", if the number of ready tasks is limited). The active threads will compete to access the data in memory so that, even if several threads are working on the same type and dimension of kernel, the dynamic power they draw will be different, depending on the physical core they are mapped to and where are the data being accessed. This results in the use of static estimations to dynamic power being oversimplistic for a practical, concurrent execution of task-parallel dense linear algebra algorithms on current multicore architectures.

The rationale is as follows. At a certain instant of time, the dynamic power drawn by a core is mostly due to the execution of instructions on the arithmetic units, the branch logic, or simply the core having to wait for data that are in the memory. This is leveraged by hardware counter-based models, which select a reduced, platform-dependent number of counters that capture activity (of these three types) in the core. Now, consider the kernels that are involved in the particular case of a dense linear algebra operation. Given a fixed block dimension, the number of FLOPS that need to be executed is known a priori. Furthermore, provided highly-tuned implementations of the kernels are employed (like, e.g., those in Intel MKL), we can assume that the power consumption of the branch logic is negligible. The reason is that, in general, dense linear algebra codes consist of simple, floating-point intensive loops that are aggressively unrolled for high performance, and contain no branches other than a minor number due to loop completion tests (which, besides, can be accurately predicted). Thus, we can expect that the power drawn during the execution of a dense linear algebra kernel will be mainly due to the floating-point arithmetic and memory contention. Therefore, given that the number of FLOPS of a kernel is known in advance, and that we can measure its execution time, it is easy to estimate the memory contention experienced by the kernel at run time from the difference between its theoretical execution time and the actual one.

Let us now formally expose our refined approach to model the dynamic power of a particular task as a function of its execution time and theoretical cost. Specifically, consider the algorithmic block size is b, and assume $t_{\{i:j\}}$ denotes a concrete task instance i, of type j. Given the theoretical number of FLOPS of this task type and the processor frequency, we can easily estimate the the-

oretical shortest execution time of this task, $T_j(b)$, by considering that the floating-point units of a conventional core can deliver floating-point 4 (double-precision) FLOPS/cycle. (This is the case for many general-purpose cores, from Intel and AMD.) Hence, for instance, in the matrix-matrix product this time is given by $T_{\texttt{dgemm}}(b) = 2b^3/(4 \cdot f)$, where f is the processor frequency.

Assume next that task $t_{\{i:j\}}$ has experienced a real execution time $R_{\{i:j\}}$. The (relative) difference

$$\delta_{\{i:j\}} = \frac{R_{\{i:j\}} - T_j(b)}{R_{\{i:j\}}} \quad (\leq 1) \; , \tag{4.5}$$

provides an estimate of the delay that this task has suffered due to memory contention. We then relate this deviation with the dynamic power dissipated by this task $t_{\{i:j\}}$ as:

$$P^D_{\{i:j\}} = \delta_{\{i:j\}} \cdot P^M_j + (1 - \delta_{\{i:j\}}) \cdot P^F_j \; , \tag{4.6}$$

where P^M_j is the power dissipated when the processor is waiting for data from memory and P^F_j is the power dissipated when the processor is doing useful work (FLOPS). Therefore, this model considers that the average dynamic power of a particular task (instance) is a function of the ratio of time that the core is stalled versus performing FLOPS ($\delta_{\{i:j\}}$), as well as the power rates dissipated in these two states (P^M_j and P^F_j, respectively); and that these power rates depend on the type of task. One could argue that P^M_j and P^F_j should not depend on the task type, being instead constant for any type of kernel. Our justification is that the kernel types differ in the ratio of FLOPS to memory operations, composition of arithmetic operations, memory access patterns, etc., which may well account for the different power rates.

The question thus becomes how to obtain reliable estimates for P^M_j and P^F_j. For this purpose, we execute repeatedly (say m_j times) a kernel of type j on a single core, for a given block size b, obtaining a sequence of execution times $R_{1:j}$,$R_{2:j}$, ...,$R_{m_j:j}$. We then obtain average experimental values of the execution time, deviation and dynamic power for this specific block size, $\bar{R}_{j,b}$, $\bar{\delta}_{j,b}$ and $\bar{P}^D_{j,b}$, respectively, with $\bar{R}_{j,b} = \sum_{i=1}^{m_j} R_{\{i:j\}}/m_j$. With these data, we then apply linear regression, solving the overdetermined system:

$$Ax = b \quad \equiv \quad \begin{pmatrix} \bar{\delta}_{j,128} & (1 - \bar{\delta}_{j,128}) \\ \bar{\delta}_{j,160} & (1 - \bar{\delta}_{j,160}) \\ \vdots & \vdots \\ \bar{\delta}_{j,1024} & (1 - \bar{\delta}_{j,1024}) \end{pmatrix} \begin{pmatrix} P^M_j \\ P^F_j \end{pmatrix} = \begin{pmatrix} \bar{P}^D_{j,128} \\ \bar{P}^D_{j,160} \\ \vdots \\ \bar{P}^D_{j,1024} \end{pmatrix} \; , \tag{4.7}$$

for the sought-after values of P^M_j and P^F_j.

Figure 4.2 illustrates the difference between average measured dynamic power and modeled (predicted) dynamic power for a variety of block sizes of the four building kernels arising in the Cholesky factorization, with the model constructed from (4.5)–(4.7). In all cases, the relative error is below 6 %, demonstrating the reliability of this approach to estimate the dynamic power of an isolated task instance.

Table 4.3 summarizes the FLOP and contention powers (in Watts) for the building kernels that appear in the Cholesky, LU and QR factorizations. The table also displays the range of variability observed for the deviations. These results demonstrate that, as could be expected, the power rate while performing FLOPS is higher than that dissipated when the core is issuing stall cycles waiting for data located in memory.

4.3.3 Formulation of the contention-aware power model

Consider a task-parallel algorithm for a linear algebra operation Op, decomposed into r different types of tasks (kernels), with n_j tasks of type j, all of them operating on blocks of size b. Assume

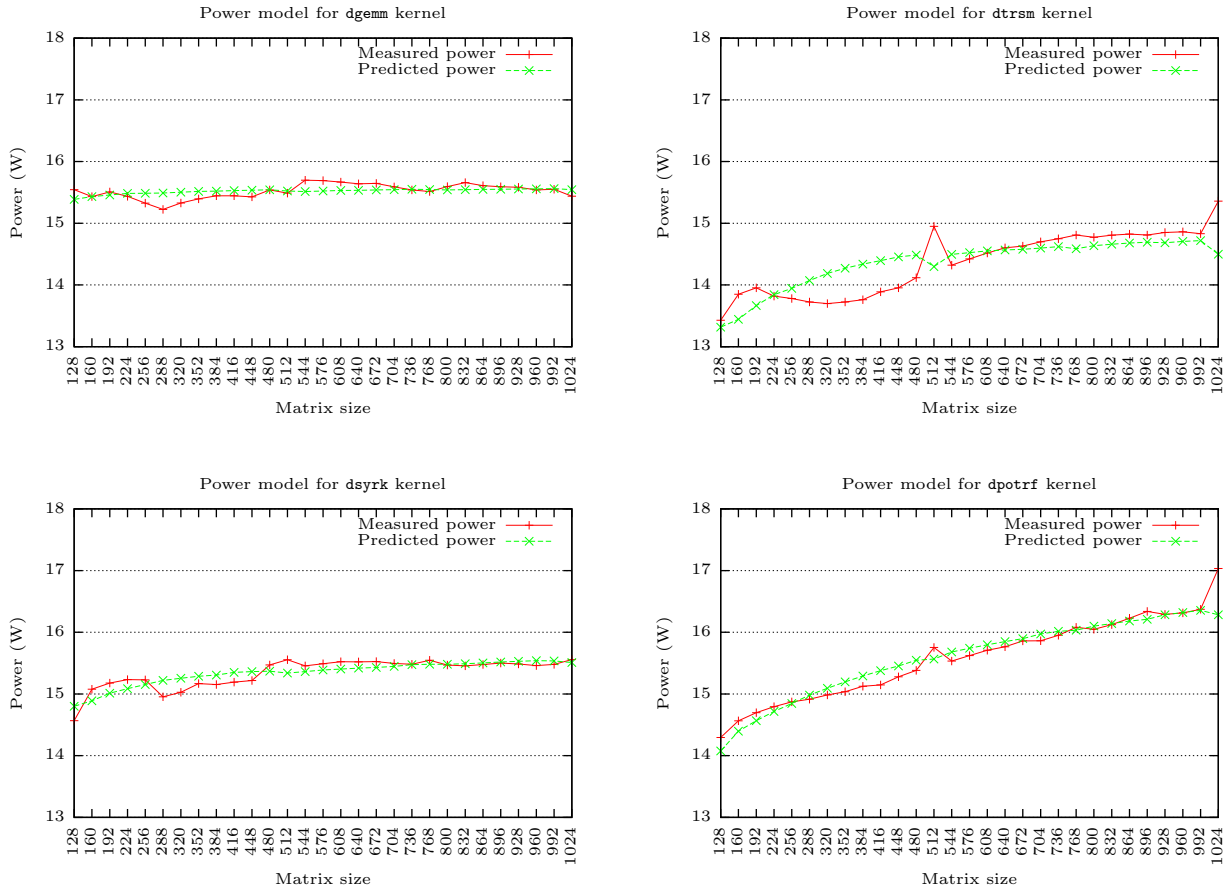


Figure 4.2: Measured and modeled power for building kernels `dpotrf`, `dtrsm`, `dsyrk` and `dgemm` of the Cholesky factorization on WT_ITL.

the algorithm runs concurrently on a parallel system consisting of c cores with one thread per core. The contention-aware model for power dissipation is then given by:

$$\begin{aligned} P_{Op}(t) &= P^Y + P^C(t) \\ &= P^Y + P^S + P^D_{Op}(t) \\ &= P^Y + P^S + \sum_{k=1}^{c}\sum_{j=1}^{r}\sum_{i=1}^{n_j} P^D_{\{i:j\}} \cdot M_{k,\{i:j\}}(t) \\ &= P^Y + P^S + \sum_{k=1}^{c}\sum_{j=1}^{r}\sum_{i=1}^{n_j} \left(\delta_{\{i:j\}} \cdot P^M_j + (1-\delta_{\{i:j\}}) \cdot P^F_j\right) \cdot M_{k,\{i:j\}}(t)\ . \end{aligned} \tag{4.8}$$

Thus, this model aggregates the system and static components of the power with those dissipated by each task instance $t_{\{i:j\}}$ on the system, taking into account the specific memory contention experienced by this particular task at execution time, $\delta_{\{i:j\}}$, and the power rates of memory contention and FLOPS of the corresponding kernel type, P^M_j and P^F_j, respectively. $M_{k,\{i:j\}}(t)$ is just the binary function that equals 1 if core k is running task $t_{\{i:j\}}$ at instant t or 0 otherwise.

Task	P_j^M	P_j^F	$\min_b \bar{\delta}_{j,b}$–$\max_b \bar{\delta}_{j,b}$
Cholesky factorization	13.32	18.72	45–86
Triangular solve	7.47	15.66	14–28
Symmetric rank-b update	12.83	16.00	15–38
Matrix-matrix product	14.67	15.70	7–15
LU factorization	12.83	17.75	75–95
Triangular solve	12.12	19.40	55–80
2x1 LU factorization	12.54	16.54	33–76
2x1 triangular solve	12.53	19.55	81–86
QR factorization	15.30	16.88	62–85
Apply orth. transf.	12.10	26.98	76–86
2x1 QR Factorization	13.91	19.18	65–82
2x1 Apply orth. transf.	6.84	16.72	16–32
Busy wait	0	9.21	–

Table 4.3: Parameters used to obtain contention-aware estimations of the task dynamic power.

Furthermore, the energy consumption of the algorithm can be obtained by integrating the power model (4.8) over the total execution time, T, resulting in:

$$\begin{aligned} E_{Op} &= \int_{t=0}^{T} (P^Y + P^S + P_{Op}^D(t))\ dt \\ &= (P^Y + P^S) \cdot T + \int_{t=0}^{T} P_{Op}^D(t)\ dt \\ &= (P^Y + P^S) \cdot T + \sum_{k=1}^{c} \sum_{j=1}^{r} \sum_{i=1}^{n_j} \left(P_{\{i:j\}}^D \cdot \int_{t=0}^{T} M_{k,\{i:j\}}(t)\ dt \right) \\ &= (P^Y + P^S) \cdot T + \sum_{j=1}^{r} \sum_{i=1}^{n_j} \left((\delta_{\{i:j\}} \cdot P_j^M + (1 - \delta_{\{i:j\}}) \cdot P_j^F) \cdot R_{\{i:j\}} \right) \\ &= (P^Y + P^S) \cdot T + \sum_{j=1}^{r} \sum_{i=1}^{n_j} \left((R_{\{i:j\}} - T_j(b)) \cdot P_j^M + T_j(b) \cdot P_j^F \right)\ , \end{aligned} \tag{4.9}$$

where $R_{\{i:j\}}$ refers to the actual execution time of task $t_{\{i:j\}}$. Note how, from the point of view of energy, the dynamic component of the model depends on the energy spent waiting for data (memory contention), $(R_{\{i:j\}} - T_j(b)) \cdot P_j^M$, which is specific to each task, and the energy corresponding to actual computations, $T_j(b) \cdot P_j^F$, which is general for the task type.

4.3.4 Experimental evaluation

We next evaluate the accuracy of the contention-aware power and energy model using high-performance implementations for three operations dense linear algebra: the Cholesky factorization, the LU factorization with incremental pivoting, and the incremental QR factorization, on matrices of dimension $n = 4096, 8192, \ldots, 32768$, with block sizes $b = 256$ and 512, and $c = 1, 2, 3$ and 4 threads/cores executed on WT_ITL. Implementations from Intel MKL 10.3.9 for the kernels and other basic suboperations appearing in the matrix factorizations were used. Concurrency was leveraged using SMPSs (v2.5). For the measurement of power we used the NI internal wattmeter combined with our framework and the PMLib library.

In order to obtain the estimated energy consumption from the model (4.9), we employ the values of P^Y and P^S calculated in Section 4.2.1 (see Table 4.1), and the dynamic power estimates for memory contention and floating-point arithmetic for each kernel type in Table 4.3. Furthermore, the execution times of each task, $R_{\{i:j\}}$, and total execution time, T, are measured using the `Extrae` tracing framework, which is fully integrated with SMPSs and provides highly accurate data.

Figures 4.3, 4.4 and 4.5 contain the results of this evaluation for the Cholesky, LU and QR factorizations, respectively. In all cases, the left-hand side plots report the relative error between the

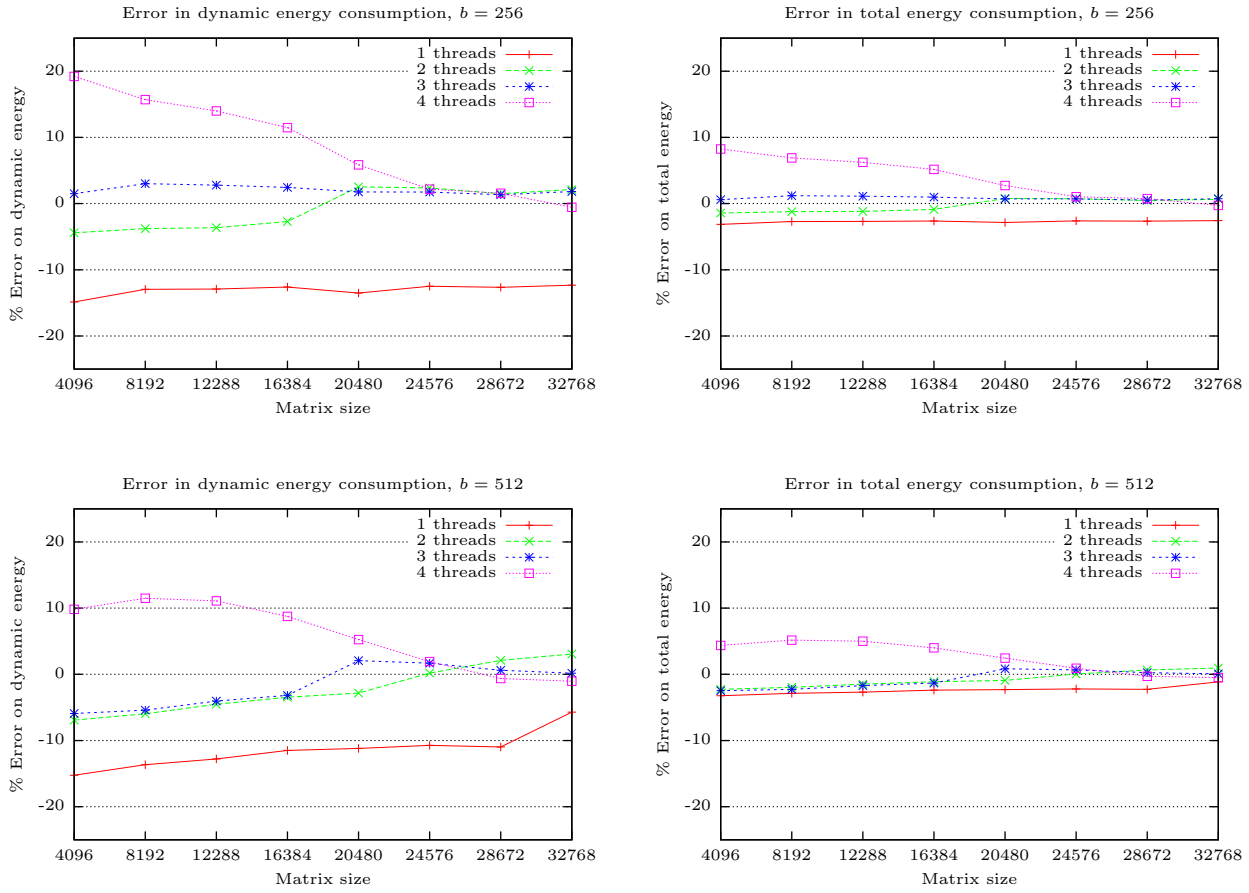


Figure 4.3: Relative error in the estimated dynamic (left) and total (right) energy consumption for the execution of the SMPSs-based task-parallel Cholesky factorization on 1–4 threads/cores of WT_ITL.

estimated dynamic energy consumptions given by the energy model (4.9) and the actual measurements. Note that, as the dynamic component of the real total energy is unknown, we assume that P^Y and P^S are those given by our experiments and subtract them to obtain the "real" dynamic energy. The right-hand side plots report the relative error for the total energy consumption.

This experimental evaluation demonstrates the high accuracy of the estimated total energy model, which is within $\pm 5\,\%$ of the real energy consumption, for most problem dimensions, block sizes, and number of threads/cores. If we subtract the constant estimations of the system and static energy from both the total real and estimated energy consumptions we obtain an approximation of the error of the estimation for the dynamic component of the total energy. As could be expected, the relative error is now larger, but still remains within a bound of $\pm 20\,\%$ of the real dynamic energy, being much lower in most cases.

4.4 Power Modeling and P-states

Many past studies have analyzed the effect of DVFS on the performance-power trade-off; see, e.g., [60]. In order to perform a similar study for the specific case of linear algebra operations on

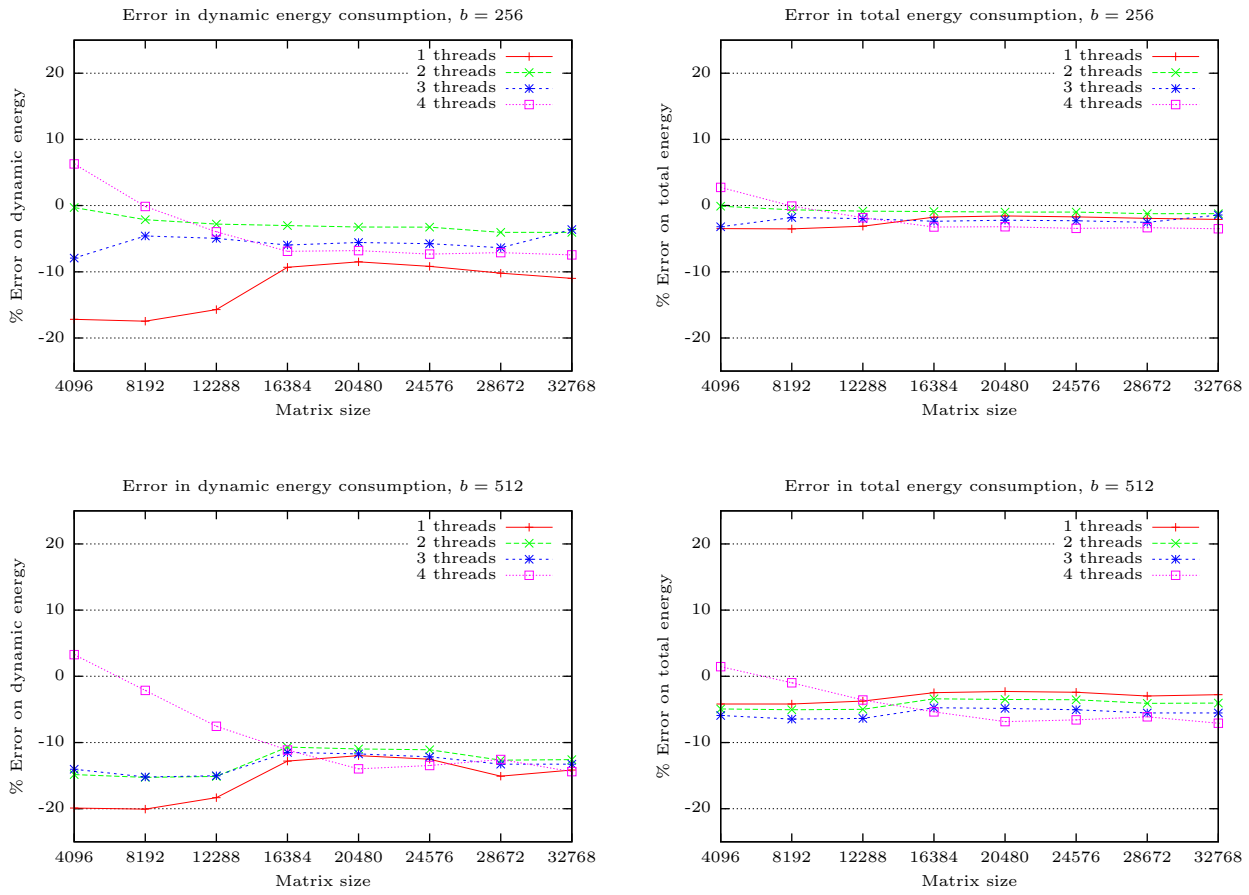


Figure 4.4: Relative error in the estimated dynamic (left) and total (right) energy consumption for the execution of the SMPSs-based task-parallel LU factorization with incremental pivoting on 1–4 threads/cores of WT_ITL.

current multicore processors, we employ the simple model (4.1) for the total (aggregate) power dissipated and extend it to handle the effect of the P-states.

Let us start by noting that all considerations described in Section 4.1 are also taken into account in this variant of the model. Also, we will assume that P_i^S, where the subscript i denotes the specific P-state, is independent of the application that runs in the platform. In practice, this is not the case, but our results will show that the errors introduced by this simplification are small, and can be easily accommodated into the model. We will further assume that the variations in the consumption of DDR RAM are small compared with those experienced by the CPU cores. In our experiments we consider both the WT_ITL and WT_AMD platforms, measured using the NI internal wattmeter in combination with our framework and the PMLib library.

To obtain practical values for the power model (4.1), we proceed as follows. For simplicity, let us assume that all c active cores of the platform run the same type of task (kernel) k, in the same state Pi, during all the execution time, as then we can easily consider that the total power at instant t equals the average power. For P_i^Y, we thus simply set all the cores of the platform to each P-state using `cpufreq`, and then measure the power with the platform idle for 30 seconds and average the results: between 71.86 and 84.83 W, depending on the state Pi, for WT_AMD; and 33.43 W for all P-states in WT_ITL; see column P_i^Y in Table 4.4.

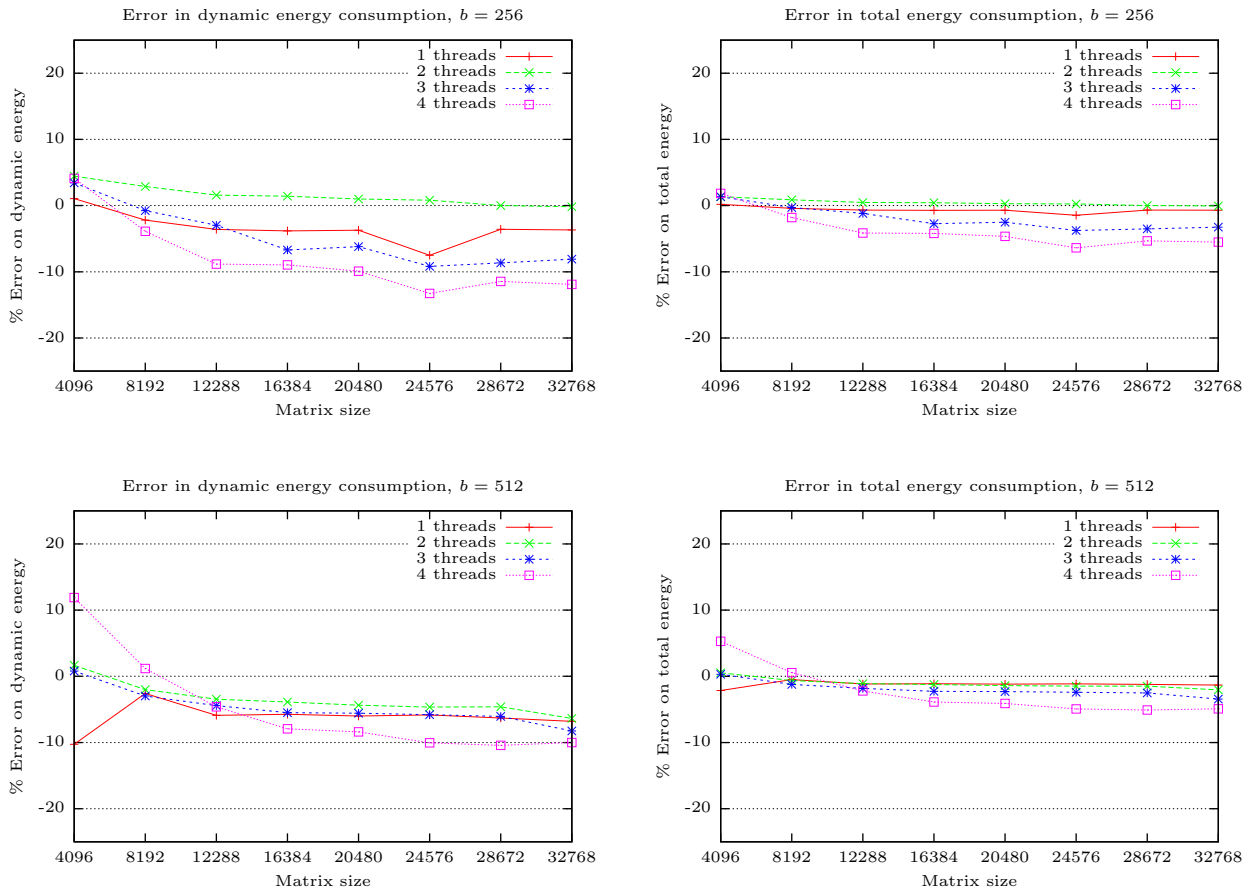


Figure 4.5: Relative error in the estimated dynamic (left) and total (right) energy consumption for the execution of the SMPSs-based task-parallel incremental QR factorization on 1–4 threads/cores of WT_ITL.

The estimation of P_i^S and P_i^C is more elaborated, as it is difficult to separate these two components, and the second depends also on the application that is being run. Refining (4.1), the total power for the execution of c copies of task k, with the active cores in state Pi, becomes

$$P_{k,i}^T(c) = P_i^Y + P_i^S + P_{k,i}^C(c) \approx P_i^Y + P_i^S + P_{k,i}^{C1} \cdot c\,, \tag{4.10}$$

where $P_{k,i}^{C1}$ denotes the power dissipated by a single core in state Pi running task k.

To estimate the missing parameters in (4.10), P_i^S and $P_{k,i}^{C1}$, we will leverage three compute-intensive kernels: the cpu*burn* microbenchmark[1], a simple *busy*-wait test consisting of a "`while (1);`" loop, and the general dense matrix-matrix product (`dgemm`) with double-precision data from Intel MKL. Specifically, we execute these tests for 60 seconds and average the power draft, for an increasing number of cores c (all in the same P-state) of the machines (8 cores of both WT_AMD and WT_ITL), while the remaining cores remain idle in an inactive C-state.

Applying linear regression to the data obtained from this experimental evaluation, we obtain linear models for the total power of the form

$$P_{k,i}^T(c) = \alpha_{k,i} + \beta_{k,i} \cdot c, \tag{4.11}$$

[1] `http://manpages.ubuntu.com/manpages/precise/man1/cpuburn.1.html`.

Platform	P-state, Pi	P_i^Y	α_i	$\beta_{burn,i}$	$\beta_{busy,i}$	$\beta_{\texttt{dgemm},i}$	P_i^S
WT_AMD	P0	84.83	140.94	14.70	13.50	13.82	56.11
	P1	77.85	137.47	7.71	6.10	10.01	59.62
	P2	73.83	131.35	5.48	4.58	7.40	57.52
	P3	71.86	127.87	4.12	3.33	5.46	56.01
	P4	72.46	124.89	3.10	2.28	4.27	52.43
WT_ITL	P0	33.43	63.93	9.48	7.10	11.12	30.50
	P1		63.38	8.19	6.16	9.84	29.95
	P2		64.10	7.33	5.43	9.03	30.67
	P3		64.72	6.34	4.64	7.81	31.29

Table 4.4: Parameters for the simple power model for the cpu*burn*, *busy* and `dgemm` benchmarks (kernels) on WT_AMD and WT_ITL. Note that $P_i^S = \alpha_i - P^Y$ and $P_{k,i}^C(c) = P_{k,i}^{C1} \cdot c = \beta_{k,i} \cdot c$, with i, k and c denoting the kernel type, P-state and number of cores, respectively.

with the values for $\alpha_{k,i}$ and $\beta_{k,i}$ in the corresponding columns of Table 4.4, and the relation between the models and the experimental data graphically captured in Figure 4.6.

These regression models show an almost perfect fit to the experimental data, offering the same (rough) value $\alpha_{k,i}$ for all three kernel types (the largest variation between the three was 2.11 % and the average difference 0.61 %.) Therefore, in the following we will use $\alpha_i - P_i^Y$ as an estimation for P_i^S (see Table 4.4), the system power dissipated by a socket in state Pi (which agrees with our assumption that the system power is independent of the kernel type); and we will set $P_{k,i}^C(c) = \beta_{k,i} \cdot c$ so that $P_{k,i}^{C1} = \beta_{k,i}$.

4.5 Other Target Platforms

In this section we present two other extensions for our simple power model (4.1). The first one has been designed for multicore platforms that have more than one socket. The second one models the power and energy consumed by the CPUs of a hybrid platform with graphic hardware accelerators (GPUs) attached.

4.5.1 Power model for multi-socket platforms

This extension of the simple power model supports multicore platforms with multiple sockets. In particular the model controls if there are kernels executing in different sockets. In that case, it applies a different dynamic power that is, in general, higher than that of the case where there is work performed on a single socket only. This is due to the promotion of the idle socket to a hardware-controlled, energy-saving PC-state.

To guide the explanation we use the TESLA2 platform considering its two sockets (but discarding the GPU attached to it, which will remain idle). We employ the kernels of the LU factorization with partial pivoting to obtain the static estimations of the dynamic power. For the power measurements we used the AP8653 external wattmeter.

System and static power

The basic methodology to estimate the system and static parameters on a multi-socket platform is closely similar to that described in Section 4.2.1. First, we estimate the system and static parameters of the power model (4.1). A direct measurement using our external AP8653 with

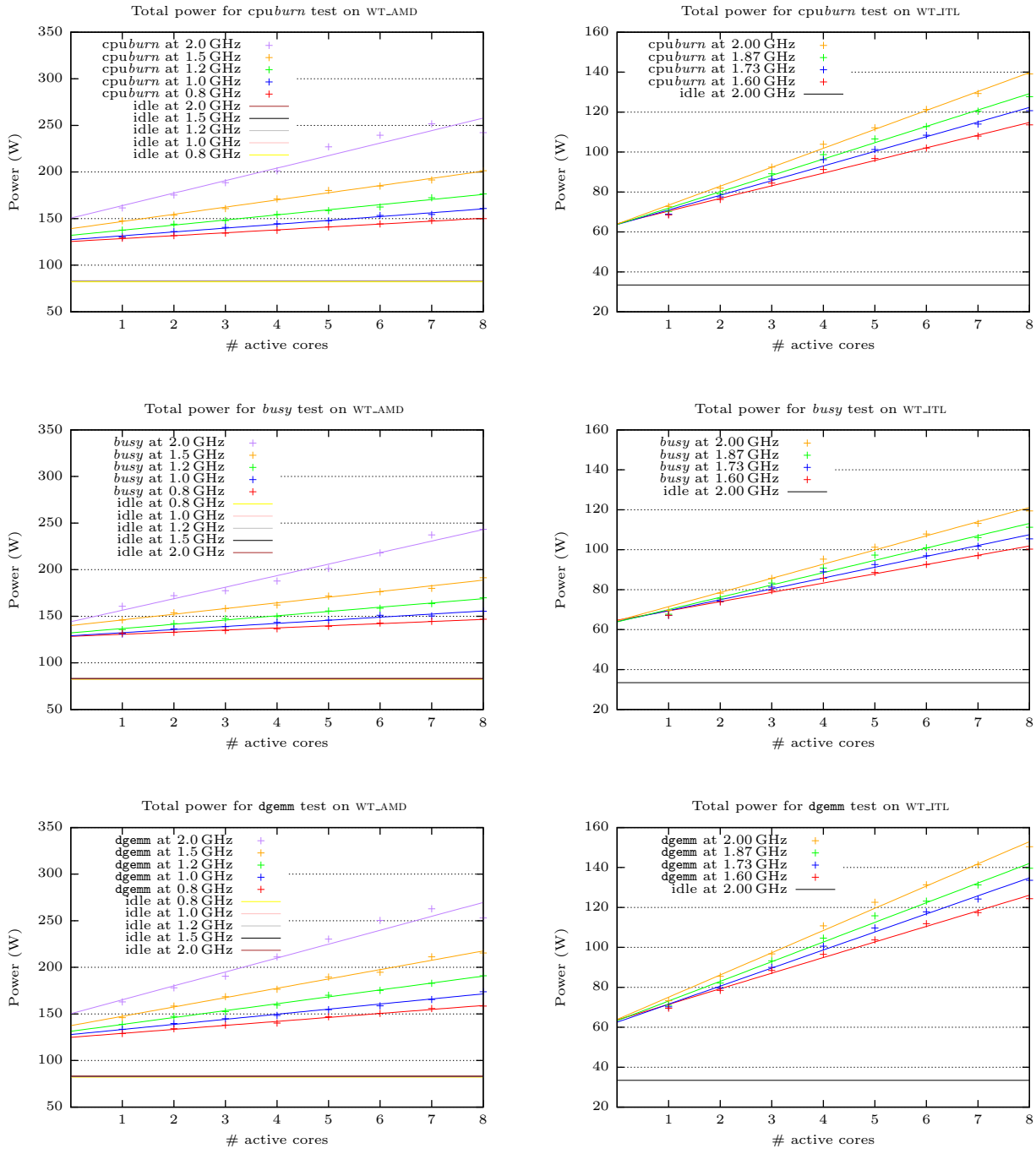


Figure 4.6: Power dissipated as a function of the number of active cores for kernels cpu*burn*, *busy* and `dgemm` (top, middle and bottom, respectively) on WT_AMD (left) and WT_ITL (right).

the platform completely idle revealed a power dissipation of 290.47 W, which can be taken as an approximation for P^Y.

For multi-socket platforms, multiple linear regression models per socket may be required. For example, TESLA2 is equipped with 2 quad-core sockets which required two models in order to attain higher accuracy, and thus two different values for the static powers. Figure 4.7 reports the power consumption obtained by the AP8653 wattmeter, when executing a matrix-matrix product (routine `dgemm` from Intel MKL) with square operands of size 1,024. We vary the number of CPU threads/cores of TESLA2 from 1 to 4 when operating with a single socket, and from 5 to 8 when running on the two sockets. Applying a linear regression to adjust the total power, two linear models are obtained: $P^T_{\texttt{dgemm}}(c) = \alpha + \beta \cdot c = 297.16 + 26.78 \cdot c$ W for the case of 1–4 threads; and $\tilde{P^T}_{\texttt{dgemm}}(c) = \tilde{\alpha} + \tilde{\beta} \cdot c = 329.02 + 19.69 \cdot c$ W for the case of 5–8 threads. Therefore, the static power dissipated by the system can be approximated as $P^{S1} \approx \alpha - P^Y = 6.68$ W when there is only one socket active; and $P^{S2} \approx \tilde{\alpha} - P^Y = 38.55$ W otherwise. The basic methodology to estimate system and static parameters on a multi-socket platform differs from the mono-socket in that it needs one linear model per socket. The results of applying the linear regressions to the TESLA2 platform are summarized in Table 4.5.

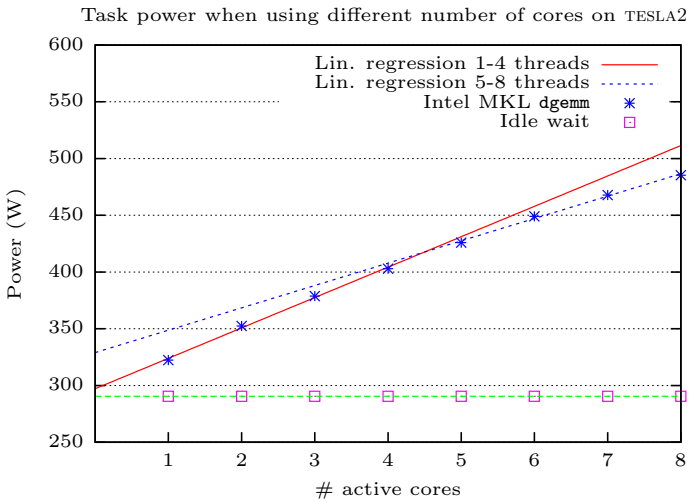


Figure 4.7: Power dissipated as a function of number of active cores on TESLA2.

Platform	Sockets	P^Y	α	β	P^S
TESLA2	1	290.47	297.16	26.78	6.68
	2		329.02	19.69	38.55

Table 4.5: Parameters for the multi-socket power model of TESLA2.

Dynamic power

To illustrate this case, we estimate the power dissipated during the execution of each one of the three kernel types of the LU factorization with partial pivoting —`dgetrf`, `dtrsm` and `dgemm`—. The previous experiment in Section 4.5.1 revealed that the dynamic power increases linearly with the number of threads mapped to a single socket that execute a certain task (the matrix-matrix

product in that case). Besides, there we could also observe that the use of cores from two sockets changes the arguments which define this linear function.

To account for this difference during the estimation of each one of the kernels, we performed an experiment where one single thread continuously invokes `dgemm`, till the total power stabilizes; we then sampled this value, say $P^T_{\texttt{dgemm}}$, and set $P^{D1}_{\texttt{dgemm}} = P^T_{\texttt{dgemm}} - P^S - P^Y = P^T_{\texttt{dgemm}} - 297.16\,\text{W}$. We next repeated the experiment with two concurrent CPU threads invoking the same kernel on different sockets of the target platform and, when the power stabilized, we sampled the value, $P^{\tilde{T}}_{\texttt{dgemm}}$, and set $P^{D2}_{\texttt{dgemm}} = (P^{\tilde{T}}_{\texttt{dgemm}} - 329.02)/2\,\text{W}$. The estimations of the dynamic power for each kernel type and number of active sockets are collected in Table 4.6. These values were obtained employing operands of square dimension 1,024 in all cases.

Task	`dgetrf`	`dtrsm`	`dgemm`
P^{D1}	27.92	36.32	26.03
P^{D2}	14.53	16.03	19.43

Table 4.6: Dynamic power (in Watts) of the task types involved in the LU factorization with partial pivoting, estimated when placing 1 and 2 threads in different sockets of TESLA2.

Formulation of the power model for multi-socket platforms

Consider finally the task-parallel execution of a dense linear operation Op, with the algorithm decomposed into r different types of tasks j. The total power dissipation of the algorithm, at an instant of time t, is given by the composition of the system power, the static power, and the dynamic power of all tasks being executed at that instant:

$$\begin{aligned} P_{Op}(t) &= P^Y + P^S(t) + P^D(t) \\ &= P^Y + P^{S1} M(t) + P^{S2} \tilde{M}(t) \\ &+ \sum_{k=1}^{c} \sum_{j=1}^{r} \left(P_j^{D1} M(t) + P_j^{D2} (1 - M(t)) \right) N_{k,j}(t) \; . \end{aligned} \tag{4.12}$$

Here, $M(t)$ is a function that, at time t, returns 1 in case there are active threads running in a single socket only or equals 0 otherwise; and $N_{k,j}(t) = 1$ if k-th thread is running a task of type j at time t.

The energy consumption of the algorithm is a function of the time spent in each type of task:

$$\begin{aligned} E_{Op} &= \int_{t=0}^{T} P^T(t)\; dt \\ &= P^Y\, T + P^{S1}\, S + P^{S2}\, (T - S) \\ &+ \sum_{j=1}^{r} \sum_{i=1}^{r_j} P_j^{D1}\, T_j\, \gamma_{i,j} + P_j^{D2}\, T_j\, (1 - \gamma_{i,j}) \end{aligned} \tag{4.13}$$

where S is the period of time when there is a single socket active; T_j is the aggregated execution time of tasks of type j; P_j^{D1}, P_j^{D2} are the dynamic powers given in Table 4.6, with j referring to the different types of tasks (columns of the table); and $\gamma_{i,j}$ is the ratio of time that there is one socket active during the execution of task (i,j).

4.5.2 Power model for hybrid platforms

In this section we extend the our simple power model (4.1) to tackle hybrid CPU–GPU platforms. We use the TESLA2 platform measured with the AP8653 external wattmeter to illustrate

this case. Note that the wattmeter does not measure the power consumption of Tesla C2050 module attached to TESLA2. The main contribution of this extension thus consists in that we integrate the dynamic power related to the transfer operations.

System and static power

The methodology followed to estimate system and static parameters is the same described in Section 4.5.1. Parameters for the TESLA2 platform are summarized in Table 4.5.

Dynamic power

The characterization of the dynamic power dissipated by a task-parallel dense linear operation on an hybrid platform is more challenging. This extension of the model for hybrid CPU–GPU platforms only contemplates the power consumed by the CPUs, but takes into account the energy consumption required for data transfers between the host (CPU) and the device (GPU). These transfers appear each time a kernel is off-loaded to the accelerator. Specifically before starting the kernel, the CPU has to transfer some data into the GPU. Once the execution has finished, the CPU has to transfer back the results from the GPU.

In order to estimate static values for the dynamic power of the transfers, we performed an experiment where a single thread continuously invokes kernels to transfer data till the total power stabilizes; we then sampled this value, say $P^T_{CPU\rightarrow GPU}$, and set $P^D_{CPU\rightarrow GPU} = P^T_{CPU\rightarrow GPU} - P^S - P^Y = P^T_{CPU\rightarrow GPU} - 297.16\,\mathrm{W}$. The results are offered in Table 4.7.

Task	CPU→GPU	GPU→CPU
P^D	50.33	49.63

Table 4.7: Dynamic power (in Watts) of the CPU–GPU transfer tasks on TESLA2.

Formulation of the power model for hybrid CPU–GPU platforms

The power and energy models, (4.3) and (4.4), respectively, can be applied as well to hybrid CPU–GPU platforms.

4.6 Concluding Remarks

In this chapter we have introduced several models to estimate the energy consumption of linear algebra operations that can be applied to task-level parallel dense or sparse linear algebra algorithms. First, we introduced a simple (static) power model that can be applied in multicore architectures. Next, we refined this initial version to employ dynamic measures that take into account memory contention during the actual execution, yielding more accurate global results. We also designed an additional power model that accounts for the total (aggregate) power dissipated by an application that can handle the P-states. The last two extended models deal with multi-socket and hybrid CPU–GPU platforms.

We have performed an experimental validation using the contention-aware model on the WT_ITL platform and the NI power measurement device. The experiments confirm the reliability of the energy models for the SMPSs-parallel Cholesky, LU and QR factorizations on this platform, which offer estimates for the total energy consumption that are, in general, within 5 % of the actual values, comparable with those obtained with elaborate models that rely on hardware counters.

Two key properties of the proposed models are portability and, to some extent, generality. Specifically, we believe that our contention-aware model can be applied to integer or floating-point intensive task-parallel applications, not necessarily related to dense linear algebra, when the following three conditions are met: *i)* there exist fairly good estimations of the theoretical cost of each type of task; *ii)* the target platform is instrumented with some sort of power measurement device; and *iii)* accurate measures of task execution time can be obtained, e.g., with `Extrae` or any other instrumentation tool (TAU, VampirTrace, Score-P, etc.). On the other hand, the models do not require access to low-level, platform-dependent hardware counters.

The ultimate goal of the models is to guide the operation of a runtime, as e.g. SuperMatrix or SMPSs, in order to unleash a more energy efficient execution of task-parallel algorithms. For example, there exists different mappings of tasks to computational resources that may result in equivalent or nearly equivalent performance on hybrid/heterogeneous platforms equipped with multicore processors and hardware accelerators (graphic accelerators, Intel Xeon Phi, etc.). In these situations, an energy-aware runtime can leverage the prediction provided by our models to automatically select the most energy-efficient configuration without a negative impact on performance.

CHAPTER 5

Theoretical Analysis of Slack Reduction and Race-to-Idle

In this chapter we investigate how to leverage DVFS and energy-friendly processor states during the execution of dense linear algebra algorithms on state-of-the-art multicore processors. In particular, we consider a DAG that represents a collection of tasks and data dependencies among these, corresponding to the computation of a dense linear algebra operation. The goal of our *Slack Reduction Algorithm* (SRA) is to detect which tasks lie in non-critical paths and reduce the frequency/voltage (DVFS) of the processor cores in charge of their execution, and thus save energy. We also present an alternative approach, the *Race-to-Idle Algorithm* (RIA), which pursues the power-conservation goal but from a totally opposite approach: it executes all tasks at the highest frequency and relies on the power savings attained via a reduction of the operating frequency during idle periods.

We consider here the analysis of the four algorithms: the LU factorization with partial and incremental pivoting, and the traditional and incremental QR decompositions, described in Sections 2.4 and 2.5, respectively. These algorithms are challenging from the point of view of slack control, as their tasks exhibit varying sizes and costs during the execution. We target the WT_AMD and WT_ITL platforms, in which the DVFS is applied at the core and socket level, respectively. Our simulations will show the impact of this strategy on the attainable theoretical gains. For the experimentation we use *real experimental* data for performance and energy consumption of the computational tasks appearing in these algorithms on these two architectures. Thus, we can expect that the results reflect the trade-off between performance and energy accurately.

The chapter is organized as follows. Section 5.1 reviews the Critical Path Method and its application to identify tasks which can be delayed without negatively affecting the total execution time. The Slack Reduction and Race-to-Idle energy-saving algorithms, introduced in Section 5.2 and 5.3, respectively, are followed by a brief discussion of the energy-aware simulator, in Section 5.4. Experiments are reported in Section 5.5 and a few concluding remarks close the chapter in Section 5.6.

5.1 The Critical Path Method

The *Critical Path Method* (CPM) is commonly used in management and project planning [114] to control the duration of a project by carefully scheduling so-called "critical" tasks (that is, those tasks which are likely to delay the project execution time in case of a late start/finish). We next

discuss how to apply CPM to detect slacks in a DAG that captures the dependencies existing in one of the algorithms for the QR factorization.

5.1.1 Demonstration example

We illustrate the CPM using the incremental QR factorization (see Section 2.5.2) and base our explanation using Algorithm 5 which presents the blocked procedure of this algorithm. Remember that each operation (i.e., task) in the algorithm is annotated to the right with its theoretical cost in floating-point arithmetic operations, or FLOPS.

Figure 5.1 illustrates the tasks and dependencies that appear in the incremental QR factorization of a blocked 3×3 matrix using Algorithm 5. The name of each task is followed, inside parenthesis, by parameters that uniquely identify the task in the graph from the loop indices k, i and j of the algorithm. Inside brackets, we include the execution time of the corresponding task (in milliseconds, ms) for a block size $b = 256$ on WT_AMD running at 2.00 GHz (the maximum available frequency).

We define the *slack* of a task as the amount of time that it can be delayed without increasing the total execution time of the algorithm.

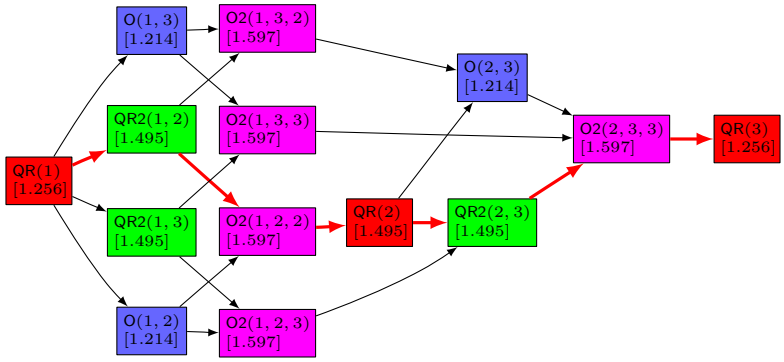


Figure 5.1: DAG with the tasks/data dependencies for the incremental QR factorization of a matrix consisting of 3×3 blocks using Algorithm 5. Red arrows identify the critical path of the algorithm.

Task	C	ES	LF	S
QR(1)	1.256	0.000	1.256	0
O(1, 3)	1.215	1.256	4.290	1.819
O(1, 2)	1.215	1.256	2.752	0.281
QR2(1, 3)	1.496	1.256	4.008	1.256
QR2(1, 2)	1.496	1.256	2.752	0
O2(1, 3, 3)	1.597	2.752	7.102	2.752
O2(2, 3, 3)	1.597	7.102	8.699	0
QR(3)	1.256	8.699	9.955	0
O2(1, 2, 3)	1.597	2.752	5.606	1.256
QR2(2, 3)	1.496	5.606	7.102	0
O2(1, 2, 2)	1.597	2.752	4.350	0
QR(2)	1.256	4.350	5.606	0
O(2, 3)	1.215	5.606	7.102	0.281
O2(1, 3, 2)	1.597	2.752	5.887	1.538

Table 5.1: Application of CPM to the DAG capturing the data dependencies in the computation of the incremental QR factorization of a matrix consisting of 3×3 blocks using Algorithm 5.

CPM can now be applied to the task-node graph in order to detect slacks, which yields the profile in Table 5.1. The application of this method obtains the following values for each task T_i:

- Its cost C_i (execution time).
- The *earliest time* at which the task can start its execution,

$$ES_i = \max_k(ES_k + C_k) \tag{5.1}$$

  for all task (node) k connected to task (node) i by an edge from task k to task i.

- The *latest time* at which the task can finish its execution without increasing the total time of the algorithm, given by

$$LF_i = \min_k(LF_k - C_k) \tag{5.2}$$

  for all task k connected to task i by an edge from i to k.

- Its slack, obtained as

$$S_i = LF_i - ES_i - C_i \ . \tag{5.3}$$

CPM identifies slack times, but does not provide an explicit strategy (procedure) to exploit them. In the following section, we introduce an algorithm that conveniently tailors the operating frequency of the processor cores, to tune the slack of those tasks with $S_i > 0$, yielding a lower power usage. In an ideal case where the cores can operate in any of an infinite (continuous) range of frequencies and the transition time (overhead) between any two frequencies is zero, the slack could be adjusted very accurately. In a real case, processor cores run at a limited (discrete) number of frequencies, and switching between any two given frequencies is not immediate, so that the slack can only be adjusted sub-optimally. Furthermore, in some architectures (e.g., those from Intel), changes can only be made at the socket level instead of at the CPU core level.

5.2 The Slack Reduction Algorithm

The Slack Reduction Algorithm (SRA) [17, 12] assigns a tentative operating frequency to each task, among a discrete number of these, at which it will be executed. The algorithm is preceded by an initialization stage that decomposes a given dense linear algebra operation into a collection of tasks, and identifies the dependencies among these, resulting in a DAG (alike that in Figure 5.1).

The SRA is composed of three stages, with the second and third stages being iterative procedures. To illustrate the algorithm, in the following discussion we will refer to the DAG in Figure 5.1. We will also consider the discrete collection of frequencies FR={2.00, 1.50, 1.20, 1.00, 0.80} (in GHz, according to the values of WT_AMD).

Frequency assignment. Initially, all tasks are assigned to be run at the maximum frequency.

Critical subpath extraction. In this step the graph is decomposed into a number of critical subpaths. First, the critical path is identified. Next, the graph edges that belong to the critical path (as well as the nodes, if they have no other edge arising at or leaving from them) are eliminated from the graph. A new critical subpath is then extracted for this subgraph; and the process is repeated until the graph is empty. Figure 5.2 details the application of the procedure to the DAG contained in Figure 5.1. For each iteration of the extraction procedure, we indicate the sequence of nodes in the critical path/subpaths (CP_0/CP_1, CP_2, CP_3) and the execution time in the bottom right corner. Note that this decomposition automatically sorts critical subpaths according to descending execution time.

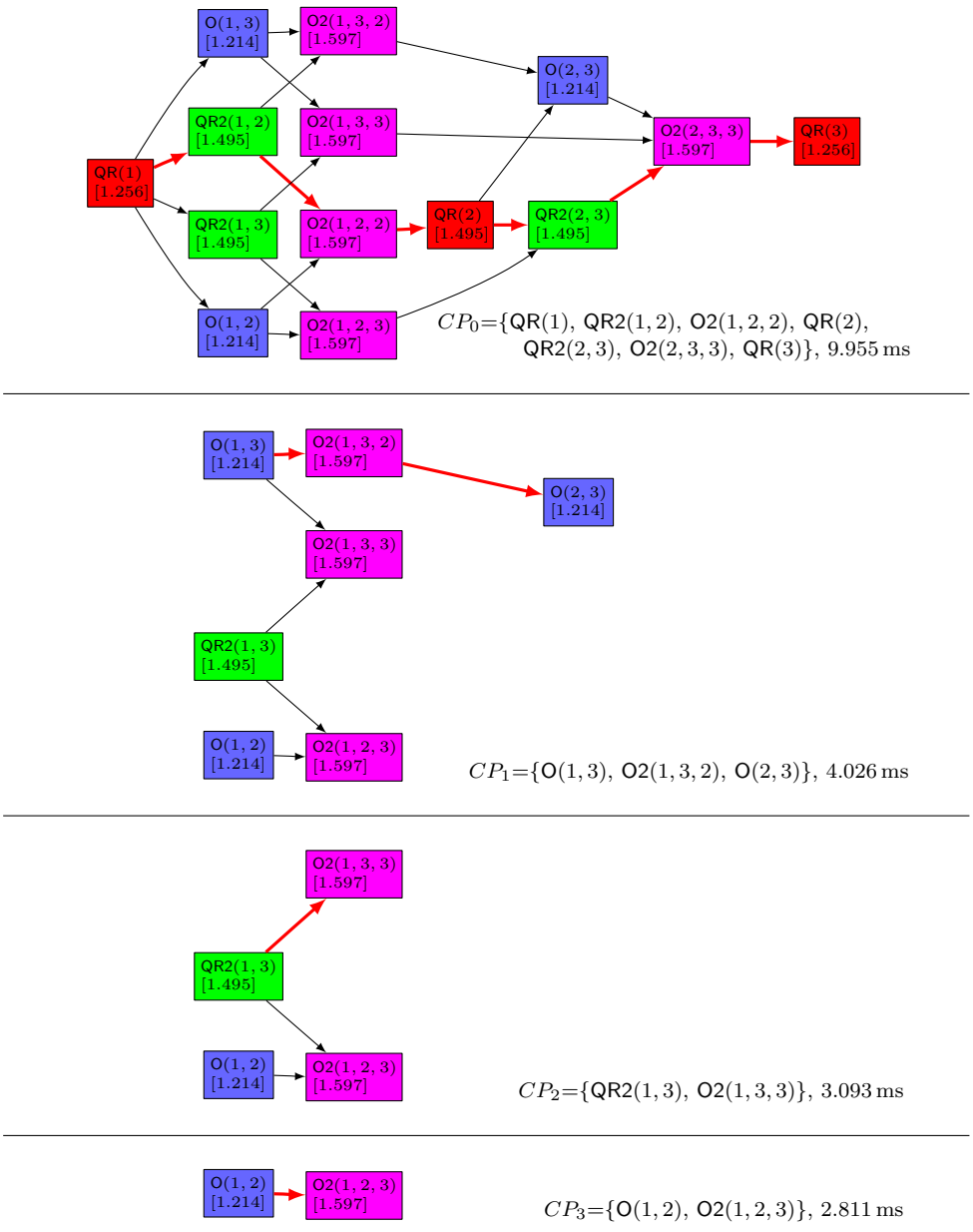


Figure 5.2: Critical subpath decomposition of the DAG capturing the data dependencies in the computation of the incremental QR factorization of a blocked 3×3 matrix using Algorithm 5.

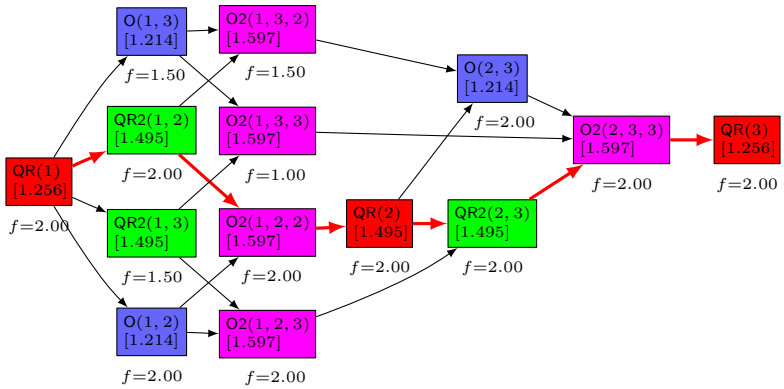


Figure 5.3: Frequency assignment and time of the DAG with the tasks/data dependencies for the incremental QR factorization of a blocked 3×3 matrix using Algorithm 5. Red arrows identify the critical path of the algorithm.

Frequency tuning. This stage calculates the (recommended) operating frequency and, therefore, dictates the execution time of the tasks and the overall execution time of the algorithm. The procedure starts by processing the first critical subpath, CP_1={O(1, 3), O2(1, 3, 2), O(2, 3)}, trying to annihilate the slack of the tasks embedded in this subpath (see Table 5.1). For this purpose, the procedure initially computes the lengths of the longest paths (LF) from task QR(1) to the first and the last task in the subpath, O(1, 3) and O(2, 3), respectively. These values are 1.256 and 7.102 ms, respectively, and their difference, 5.846 ms, provides a bound on the maximum duration of the execution of the tasks in CP_1. Given that the execution time of the tasks in this subpath is $1.214 + 1.597 + 1.214 = 4.026$ ms (column C in Table 5.1), this implies that there is a slack of $5.846 - 4.026 = 1.820$ ms, which can be distributed among O(1, 3), O2(1, 3, 2) and O(2, 3). How the slack is shared among these tasks is completely arbitrary, provided the individual slacks for these tasks (column S in Table 5.1) are not exceeded. In our specific implementation, the ratio $5.846/4.026 = 1.452$ recommends a rough increase of 45 % to the execution time of the tasks in the subpath.

The procedure is then repeated for CP_2, CP_3 and so on, yielding a recommended execution time for each one of the tasks of the DAG; see Figure 5.3.

Under mild conditions, a user can be interested in trading execution time for energy consumption. To accommodate this, the SRA utilizes a user-defined *excess ratio* which specifies the maximum increase in execution time that is acceptable. This makes the previous algorithm slightly more complicated, but the basic structure remains the same. For simplicity, we skip the exposition of the modified algorithm, and refer to the experimental evaluation for a practical demonstration of its effects.

5.3 The Race-to-Idle Algorithm

The following alternative strategy to save energy consumption, the Race-to-Idle Algorithm (RIA), leverages the fact that current processors are quite efficient at reducing power when idle. Therefore, it may be more convenient to complete the execution of tasks as soon as possible, so as to "enjoy" longer periods of inactivity. In other words, RIA executes the tasks of the algorithm at the highest possible frequency, while reducing frequency to the lowest possible during idle periods. This strategy can be combined with the use of energy-friendly processors states.

This strategy requires no processing of the DAG that reflects the tasks and dependencies of a blocked algorithm, as all tasks are executed at the highest frequency, while during the idle periods, the CPU frequency is reduced to the lowest value. Whether this is advantageous or not depends on the specific trade-off between performance and energy of the computational kernels, the existence and the length of idle periods in the dense linear algebra algorithms, and the overhead incurred to change the operating frequency, which is basically dictated by the target hardware platform.

5.4 Simulator

In order to evaluate the performance of the strategies, we employ a flexible, energy-aware simulator [17, 12] which uses the information obtained with SRA and RIA to produce a schedule, for a particular target architecture. It also records all frequency variations occurred during the execution, and displays statistics on energy savings in terms of percentage of time that each computing resource has operated at a given frequency. Therefore, this tool can help to analyze the theoretical performance and energy savings produced by the application of SRA and RIA in different scenarios: DAGs associated with different task-based algorithms, platform setups, excess ratios, frequency ranges, etc. In general, the static (*a priori*) schedule produced by the simulator is not applicable in practice, as even tiny variations during task execution may render it useless. However, it serves as a demonstrator of the effect that a technique like an energy-aware DVFS-based strategy can yield for the execution of dense linear algebra kernels.

In the following, we describe the possibilities and properties of the simulator in more detail.

5.4.1 Input parameters

The simulator receives the following inputs:

- A DAG reflecting the tasks and dependencies implicit in the blocked algorithm as well as the frequencies recommended by SRA/RIA to execute the tasks (in case of SRA, the frequencies are precisely those obtained from the application of that procedure; in RIA, all tasks are run at the highest frequency).
- A simple description of the target architecture that specifies the *number of sockets* (or physical processors) and the *number of cores per socket*. To mimic technologies like AMD PowerNow! and Intel SpeedStep, the simulator can adjust the frequency at core/socket level, respectively. Furthermore, core frequency cannot be changed if there is a task running on it at the moment.
- A discrete range of core frequencies, FR.
- A collection of real power consumption for each combination of frequency, {idle/busy} state per core. To obtain these values in our simulations, we executed a benchmark that varied the frequency/load of the cores, testing all the combinations.
- The cost (overhead) incurred to perform a frequency variation.

5.4.2 Scheduler

As starting point, we chose a static priority list scheduler [89, 93]. The reason for this is twofold. First, the approximate duration of the tasks is known in advance (as, in general, is the case for dense linear algebra operations). Second, the execution of tasks that lie on the critical path must be prioritized. Thus, tasks to be executed at higher frequency are assigned raised priority. Among

the tasks which have to be run at the same frequency, those which belong to a critical subpath (CP_j) with smaller index (j) are sorted first.

Consider the execution of a task T_i with recommended frequency f_i (set either by the SRA- or RIA-based strategies). The scheduling algorithm maps the execution of T_i to the first idle core that satisfies one of the following conditions, checking them in the order they appear next:

1. The core socket is operating at frequency f_i.
2. The core socket is varying its operating frequency to $f_o = f_i$. (The task will commence execution when the change is completed.)
3. The core socket is operating at a frequency $f_o > f_i$.
4. The core socket is varying its operating frequency to $f_o > f_i$.
5. All cores in the same socket are idle. If the socket is operating at a frequency $f_o \neq f_i$, a change of frequency to f_i is requested. The socket is reserved so that T_i will be the first task that will run on it when the change is completed.

If none of the above conditions is satisfied, the task remains in the pending queue, waiting for subsequent frequency variations in the system. This strategy will ensure that the execution time of a task does not take longer than dictated by SRA. For that purpose, the schedule guarantees that the task is executed in a core running at the desired frequency or, if not possible, at a higher one. For RIA all tasks are run at the highest frequency.

5.5 Simulation Results

We next evaluate the performance of SRA and RIA combined with the energy-aware simulator/scheduler using several blocked algorithms for the LU and the QR factorizations.

5.5.1 Benchmark algorithms

In our experiments, we employ the right-looking blocked algorithmic variants for the incremental and traditional QR factorizations (Algorithm 5 and 4, respectively). We also use the blocked algorithms for the LU factorization with incremental and partial pivoting (Algorithm 3 and 2, respectively).

5.5.2 Environment setup

We emulate WT_AMD and WT_ITL as target platforms in our experiments. Details about available frequencies and voltage/frequency pairs for these architectures are introduced in Section 3.1.2. Table 5.2 reports the latencies incurred to change between any two frequencies in these two platforms, determined experimentally.

Our experiments consider a variety of (square) matrix dimensions ranging from 384 to 2,944 with the block size $b = 128$. This size was experimentally determined to be close to the optimal for most kernels involved in these factorizations. For the incremental QR and LU with incremental pivoting we measured the execution time of their four different types of tasks running on a single core of WT_AMD and WT_ITL (using only a single socket for WT_ITL); i.e., an AMD Opteron 6128 and an Intel Xeon E5504, respectively, at each possible frequency. For the traditional QR and

WT_AMD

Source freq. \ Destination freq.	2.00	1.50	1.20	1.00	0.80
2.00	–	40.36	43.18	43.77	49.85
1.50	302.5	–	50.98	54.00	58.19
1.20	301.7	302.7	–	61.60	66.05
1.00	297.4	302.3	306.0	–	74.70
0.80	291.6	292.7	294.0	295.80	–

WT_ITL

Source freq. \ Destination freq.	2.00	1.87	1.73	1.60
2.00	–	125.6	122.0	46.18
1.87	143.6	–	120.9	45.98
1.73	144.7	143.3	–	48.23
1.60	167.3	156.9	167.7	–

Table 5.2: Frequency change latency between each available frequency (in μsec.) on WT_AMD and WT_ITL platforms.

LU with partial pivoting factorizations, the execution time of tasks QR/O, and LU/T, respectively, depends on the iteration.

To estimate the execution time in this case, we evaluated the time of 1 FLOP for each type of task involved in the algorithm, and then we determined the execution time of a given task by replacing the FLOP time in its theoretical cost formula. Using this data, we apply SRA to adjust the execution frequency of the tasks.

To obtain an approximation of the power required by the computational tasks, we evaluated the `dgemm` kernel (from Intel MKL 11.0) executed at all possible combinations of frequencies on 1, 2, 3, ..., p cores, with $p = 16$ on the AMD and $p = 8$ on the Intel. Thus, we measure the power using p cores, simultaneously running p copies of the kernel at frequencies f_1, f_2, ..., f_p, possibly different, while all remaining cores are idle, at the lowest frequency if SRA/RIA are in place, or at the highest frequency when no energy-saving strategy is applied. Power measures were obtained using the internal DCM wattmeter.

5.5.3 Metrics

In order to assess the benefits of the proposed solution we employ the following metrics:

- The *Impact of SRA/RIA on time ($IT_{SRA/RIA}$)* measures the ratio between the execution time of the algorithm operating at the frequencies dictated by SRA/RIA and that obtained when all tasks are run at the highest frequency. This ratio is obtained by dividing the execution time of both variants:

$$IT_{SRA/RIA} = \frac{T^{exec}_{SRA/RIA}}{T^{exec}}$$

  Ideally, this ratio should be 1. In RIA, the overhead due to the frequency changes may render a ratio higher than 1. In SRA, there may appear a certain overhead due to frequency changes as well, but when this occurs, it is mostly due to the algorithm being oblivious to the real number of available resources (cores). A resource-aware implementation of SRA would solve this issue.

- The *Impact of SRA/RIA on consumption ($IC_{SRA/RIA}$)* measures the ratio between the energy consumption of the algorithm operating at the frequencies dictated by SRA/RIA and that obtained when running all tasks at the highest frequency. This ratio is obtained by dividing the energy consumption of both variants:

$$IC_{SRA/RIA} = \frac{C^{exec}_{SRA/RIA}}{C^{exec}}$$

  Ideally, this ratio should be close to 0. A longer execution time and the overhead introduced by frequency changes may yield an energy ratio higher than 1.

5.5.4 The traditional QR factorization

Figures 5.4 and 5.5 report the results for the traditional QR factorization on the two platforms. The usage of SRA produces an increase in the execution time for the largest problem sizes (from $n =$ 2,304 on WT_AMD and $n =$ 1,536 on WT_ITL) due to the resource-oblivious current implementation. On the other hand, RIA maintains the execution time for all problem dimensions on both platforms, demonstrating that the overhead due to frequency changes is negligible compared with the cost (i.e., time) of the individual tasks (at least, for such block size). From the point of view of energy, SRA outperforms RIA as long as there is no increase in the execution time (small to moderate problem sizes). Comparing both architectures, it is important to emphasize the benefits that the flexibility of operating the frequency per core in WT_AMD yields. While the energy savings estimated for this platform reaches up to 20 % (SRA for $n =$ 1,792), the reduction for the Intel platform are much more modest, with a peak around 5 % (SRA for $n = 640$).

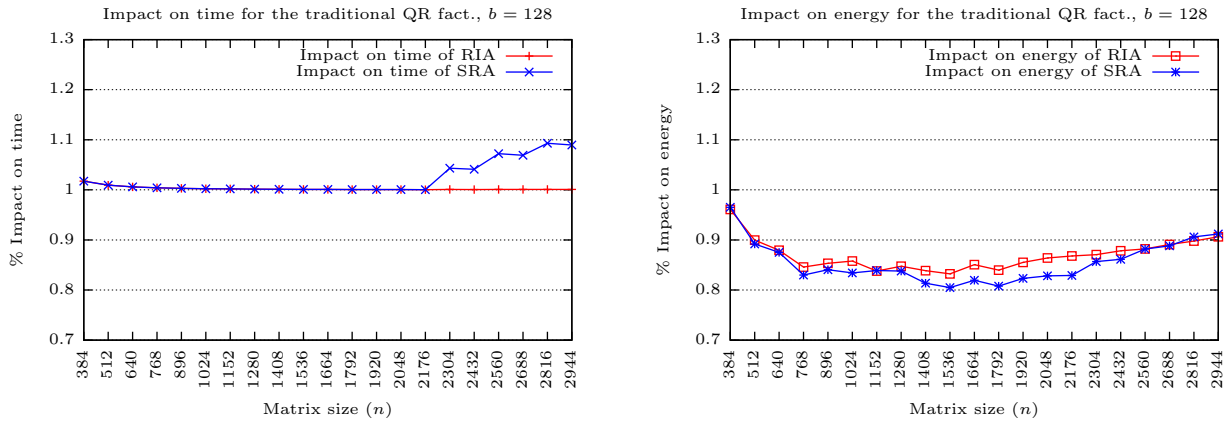


Figure 5.4: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the traditional QR factorization on WT_AMD.

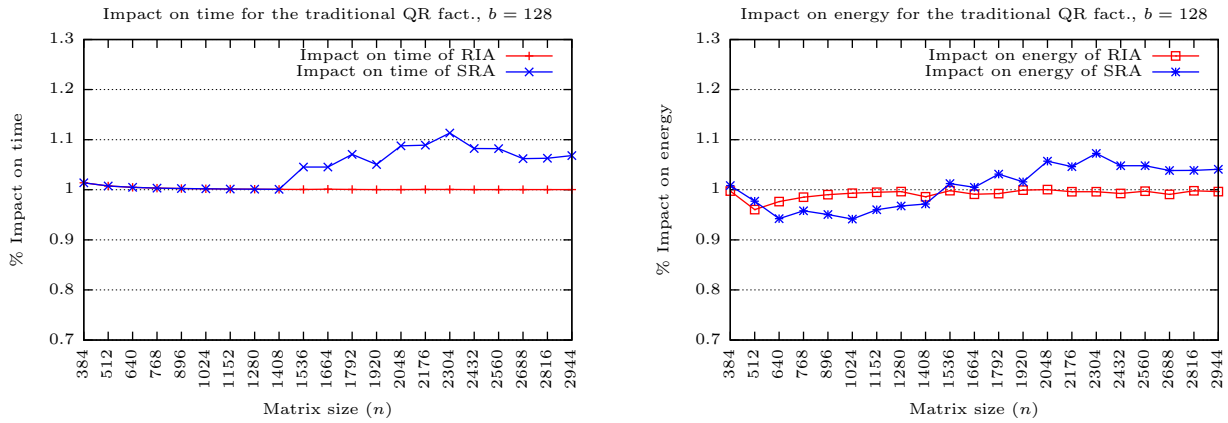


Figure 5.5: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the traditional QR factorization on WT_ITL.

5.5.5 The incremental QR factorization

Due to the cost of the simulation and the higher complexity of the DAG associated with the algorithm for the incremental QR factorization, in this case we could only evaluate the impact of SRA for problems of dimension n up to 1,408. Figures 5.6 and 5.7 report the results on both platforms. In general, the behavior of the execution time for SRA is similar to that already observed in the previous algorithm. The benefits of SRA on energy consumption are evident for the AMD platform, especially for the smallest problem sizes, while this strategy only yields low energy savings for the two smallest problem sizes on the Intel architecture. With the exception of one particular case ($n = 896$, WT_AMD), RIA keeps the ratio of execution time close to 1. The energy savings yielded by RIA are always competitive or superior to SRA on AMD, as well as on Intel when $n \geq 896$. In general, the poor results obtained on the Intel platform with any of the two energy-saving strategies are due to the granularity of DVFS which limits frequency changes to occur at the socket-level.

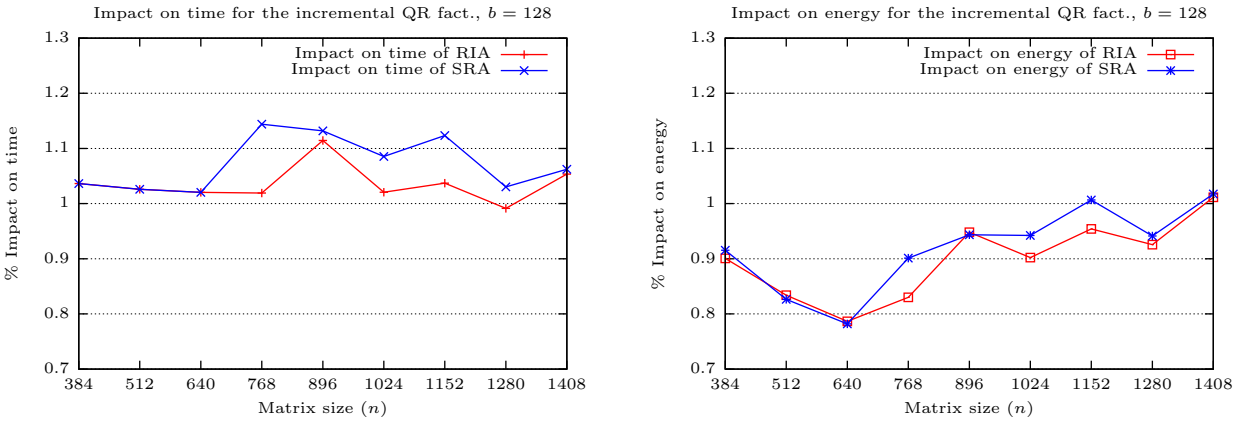


Figure 5.6: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the incremental QR factorization on WT_AMD.

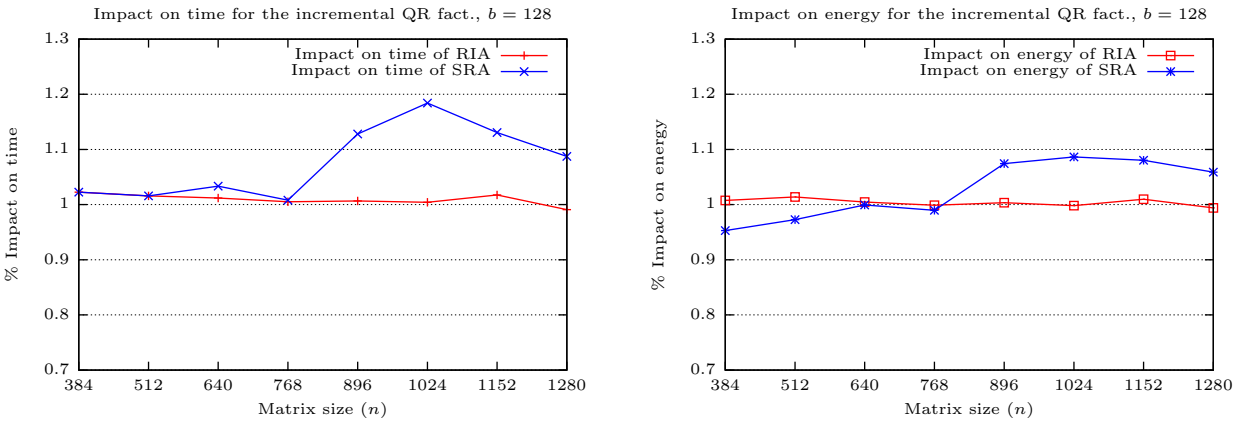


Figure 5.7: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the incremental QR factorization on WT_ITL.

5.5.6 The LU factorization with partial pivoting

Figure 5.8 reports the results for the LU factorization with partial pivoting on WT_AMD. The first aspect to notice is the increase of execution time that the usage of SRA produces for the largest problem sizes. RIA, on the other hand, maintains the execution time for all problem dimensions, demonstrating that the overhead of frequency changes is negligible compared with the cost (i.e., time) of the individual tasks (at least, for such block size). If we focus on the energy, the higher execution times required by SRA yield an increase of consumption as well, and this is not compensated by the reduction that, in principle, an execution at a slower pace (frequency) should entail. A deeper investigation revealed that the increase in execution time of SRA that appears for $n \geq 2{,}560$ is actually due to the algorithm being oblivious to the real number of available resources (cores), and this could be solved by a resource-aware implementation of SRA.

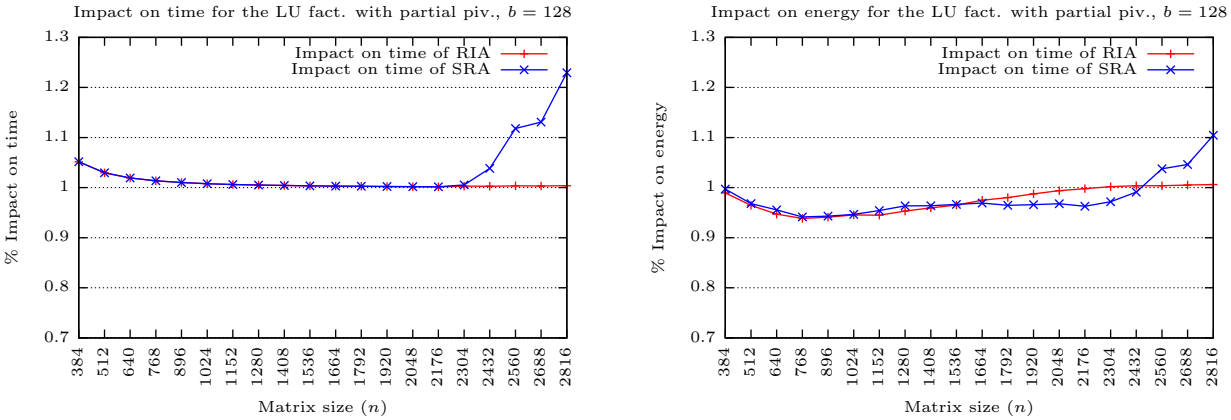


Figure 5.8: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the LU factorization with partial pivoting on WT_AMD.

5.5.7 The LU factorization with incremental pivoting

Due to the cost of the simulation and the higher complexity of the DAG associated with this algorithm, in this case we could only evaluate the impact of SRA for problems of dimension n up to 2,816. Figure 5.9 reports the results for the LU factorization with incremental pivoting on WT_AMD. In general, both strategies render an important increase of the execution time, especially for small problem sizes. This is due to the frequency changes that these strategies require during their operation. The energy in this case also increases since the time needed to execute is higher, around 10 %.

5.6 Concluding Remarks

In this chapter, we provide evidence that it is possible to improve energy consumption during the execution of dense linear algebra algorithms while, in some cases, maintaining their performance. Following the current trend for multicore parallelization (adopted, e.g., in libraries `libflame`, PLASMA, SMPSs, etc.), our algorithms exploit task-level parallelism, considering dense linear algebra operations that are partitioned into a number of tasks with dependencies among them. Our energy-conservation strategies, SRA and RIA, start from the DAG representing the operation and are based on two key observations. First, if all the tasks run at full speed (highest

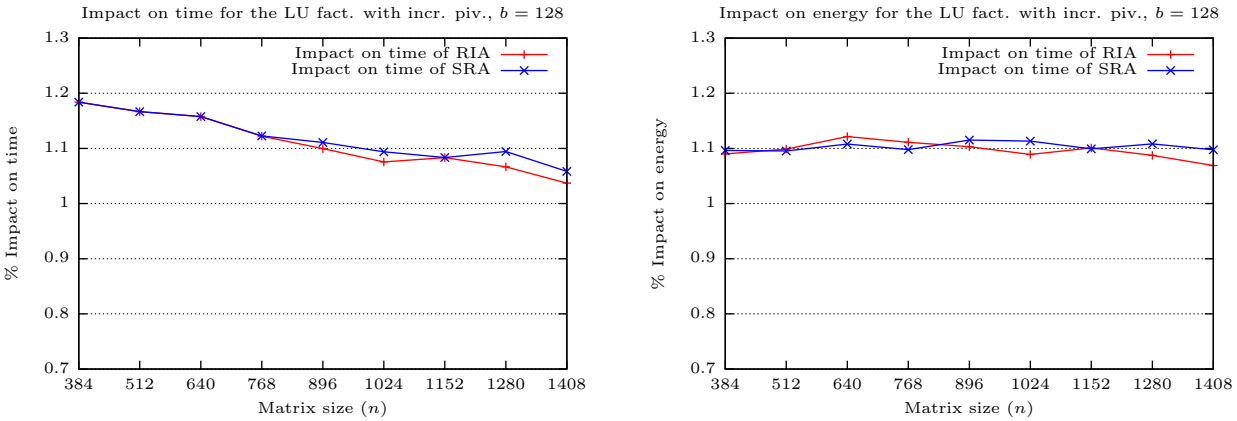


Figure 5.9: Impact of SRA and RIA on the execution time and energy of the blocked algorithm for the LU factorization with incremental pivoting on WT_AMD.

frequency), idle times appear during their execution. Second, present processors include efficient mechanisms to dynamically adjust frequency/voltage (DVFS) and hence the power consumed.

We have evaluated two alternative strategies that leverage DVFS to save energy during the execution of dense linear algebra algorithms on multicore architectures: SRA and RIA. The SRA is inspired by concepts and methods of project planning theory. In fact, we first apply CPM to determine the individual slack of each task, and then employ SRA to conveniently slow down the execution frequency of the appropriate tasks, while potentially maintaining the global execution time. On the other hand, RIA pursues energy-conservation from a totally opposite approach; specifically, this strategy generates periods of inactivity during the execution of the DAG by executing all tasks at the highest frequency, and relies on the energy savings attained via a reduction of the operating frequency during these idle periods. In the end, both alternatives leverage the trade-off between energy and performance. In the chapter, the results from these techniques are fed to a simulator, which is used to produce a feasible schedule of the tasks as well as tune their execution frequencies to a particular target architecture, assessing the benefits that can be obtained for a given operation.

We have evaluated these two energy-control policies using two algorithms of the QR and LU factorizations which are representative of many other high-performance BLAS-3-based dense linear algebra operations. The results of this experimental analysis under realistic conditions show a reduction in energy consumption under certain conditions, and some interesting insights. First, they show the superior performance of the RIA strategy over SRA from the point of view of execution time for problems of large dimension. Second, SRA, which is a more elaborate strategy than RIA, can potentially deliver higher energy savings than RIA under certain circumstances. Third, the study illustrates the importance of selecting the appropriate energy-saving policy, depending on the algorithm, problem dimension and target architecture. Finally, the results demonstrate the impact that a flexible hardware which allows operating with DVFS frequencies at the core level (instead of the socket level) has on energy savings.

CHAPTER 6

Energy-Aware Techniques for Dense and Sparse Linear Algebra

In Chapter 5 we presented the Slack Reduction (SRA) and the Race-to-Idle (RIA) policies for the execution of dense matrix factorizations on multithreaded architectures. In that study we observed that RIA generally offers larger energy savings than SRA for large problems, preserving the execution time of the tested algorithms. For this reason, in this chapter we move one step forward and incorporate the RIA approach into a production runtime for the domain of dense linear algebra, which allows us to provide experimental evidence of the impact of this policy on energy consumption. Specifically, we address the energy-aware execution of dense linear algebra operations on platforms equipped with general-purpose multicore processors and, sometimes, one or more GPUs. (For simplicity, we will refer to both types of systems as multithreaded architectures.) Our solution features a new design of the SuperMatrix runtime that controls the task-parallel execution of the operations, eliminating busy-waits (polling) when the general-purpose cores remain idle, with a minimal impact on the execution time. The current underlying hardware has the necessary mechanisms to switch to low-energy states. Our methodology is based on a redesign of the software so that it can provide the necessary conditions for the hardware to switch to low-energy states and, thus, to leverage the processor power states or C-states.

In addition, our methodology is also leveraged and applied in combination with others that employ DVFS in the sparse linear algebra domain. In this sense, we extend the methodology applied to dense linear algebra operations, in order to demonstrate performance benefits for the task-parallel solution of sparse linear systems on multicore processors in ILUPACK (see Section 2.6). Unfortunately, current implementations are energy-oblivious, in spite of the significant assets that energy-aware software can yield. Therefore, the aim of the second part of the chapter is to analyze how to seize the CPU power-saving mechanisms in the execution of ILUPACK. Specifically, we provide a new version of this runtime that is designed to leverage the P-/C-states in order to yield energy-aware executions for the solution of sparse systems of linear equations.

The chapter is organized as follows. Section 6.1 introduces our energy-aware techniques and describe how to accommodate them into a task-parallel runtime. In Section 6.2, we analyze the potential savings of dense linear algebra operations using our energy-aware SuperMatrix runtime, evaluating its practical performance and energy consumption over a representative set of dense linear algebra operations. In Section 6.3, we present results of our energy-aware runtime for the solution of sparse linear algebra systems using ILUPACK, providing theoretical and experimental

results and analyzing the impact of the P-/C-states on the time-power-energy balance. Finally, the chapter is closed with a discussion of the results for both linear algebra domains in Section 6.4.

6.1 Energy-Aware Techniques

Modern Linux distributions harness DVFS by providing different *governors* (`ondemand`, `powersave`, etc.) which set idle threads into power-hungry/power-save modes by increasing/reducing their operating frequency and voltage scaling. Operations as those in the BLAS-3 (e.g., GEMM) are CPU-bound so that, in most architectures, reducing the operating frequency/voltage incurs an increase of the execution time and, therefore, yields higher energy consumption, blurring all benefits of a lower-paced execution [41]. Despite being a BLAS-3-based operation, the picture is different, in example, for the LU factorization with partial pivoting. Due to the existence of task dependencies, idle periods may appear during the computation of this operation. While this can be exploited by setting a fixed governor for the entire application, a more effective approach in our case can be applied instead by integrating this mechanism in the runtime system.

Let us first illustrate the benefits of introducing energy-saving strategies into a task-parallel execution of a dense linear algebra operation. Although we use the Cholesky factorization as a guiding example, the ideas and techniques described here also apply to other dense linear algebra algorithms. Actually, we report experimental results for the LU factorization with partial pivoting as well.

The results reported in this chapter were obtained using IEEE double-precision arithmetic on WT_AMD and TESLA2. Highly tuned implementations of BLAS and LAPACK were provided by MKL 10.0.1, and the SuperMatrix runtime in `libflame` release 5.0–r6719 was employed in the evaluation.

Figure 6.1 shows a fragment of a trace (obtained using TAU and `Jumpshot` [116, 127]) corresponding to the execution of the routine for the Cholesky factorization in `libflame` (`FLASH_Chol`), parallelized with SuperMatrix, on TESLA2. The four (general-purpose) cores and four GPUs of the platform collaborate in the factorization. Due to their complexity, kernels of type CHOL are executed only by the CPU cores; the remaining three types of kernels, TRSM, GEMM and SYRK, run on the GPUs. Idle time (in white color) corresponds to periods within which the cores are doing no useful work and are waiting for an event instead. They occur mainly for two reasons. First, at a given moment, there may be no tasks ready to be executed (because of dependencies). Second, when a task is derived to the GPU, the associated core remains idle, waiting for the completion of the job.

The question that naturally arises is how to leverage these idle periods to reduce energy consumption. We describe in the following the modifications introduced in the SuperMatrix runtime to replace, whenever possible, busy-waits by more power-friendly states during the execution of the dense linear algebra codes in `libflame`.

6.1.1 EA1: Reduce the operating frequency when there are no ready tasks

Our first energy-saving technique works as follows: when a thread samples whether there is work in an empty ready list, the runtime immediately sets the operating frequency of the associated core to the lowest possible (using routine `cpufreq_set_frequency` from the `cpufreq` Linux library). Later, when the poll receives a positive answer, the frequency is raised back to the highest, in preparation for the execution of the corresponding job. This operational mode implies a reduction in the polling rate which is profitable (polling in itself can be viewed as a waste of energy). Figure 6.2 illustrates the difference in power consumption between a thread that performs polling at 2.00 GHz

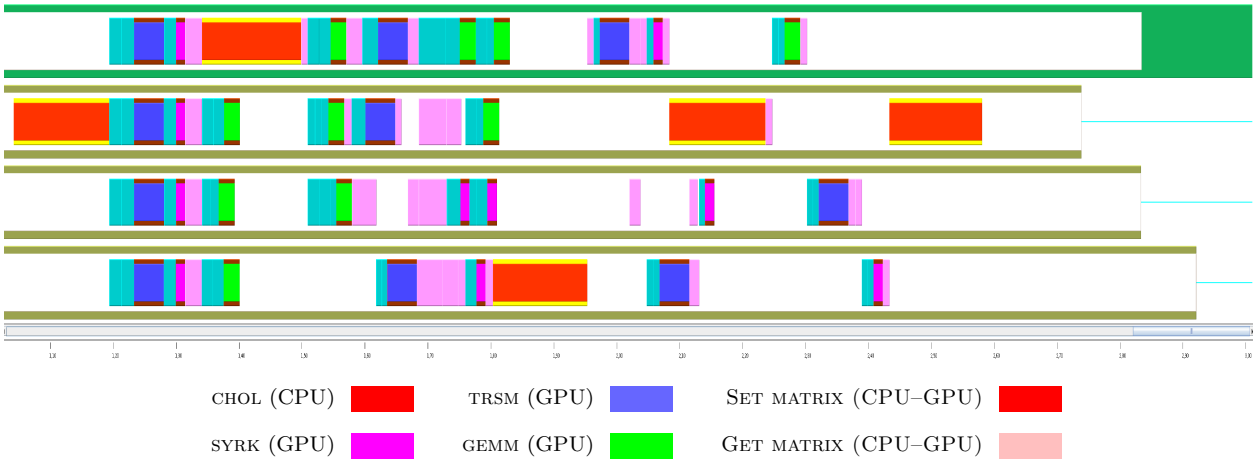


Figure 6.1: Execution trace of routine `FLASH_Chol` for the Cholesky factorization of a $7{,}680 \times 7{,}680$ matrix, using the SuperMatrix runtime, on TESLA2 (4 cores and 4 GPUs). The "white space" in the trace corresponds to idle time. The remaining colors identify kernels following the same pattern used in Algorithm 1 (page 26).

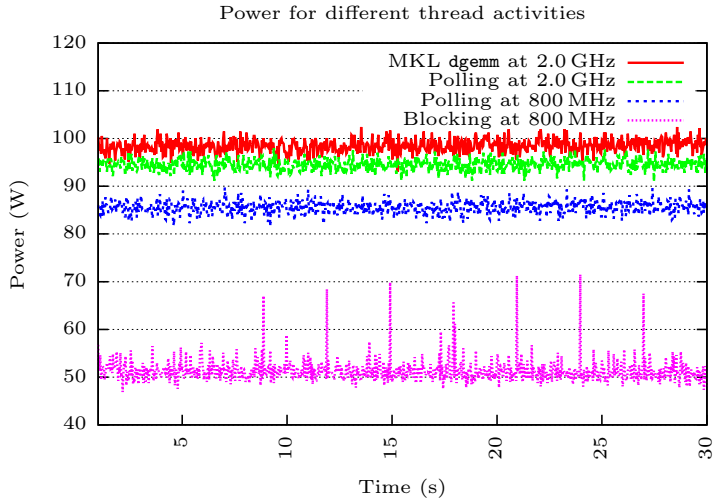


Figure 6.2: Power consumption of different actions performed by threads on WT_AMD.

and one that does the same at 800 MHz on WT_AMD (when all remaining cores are idle): from around 95 W to less than 90 W. Note also that the power demand of a thread performing polling at the highest frequency is only slightly lower than that of a thread performing useful work like, e.g., a matrix-matrix product (MKL `dgemm`). The results in that plot also point in the direction of a complementary/alternative strategy. In particular, observe that a thread performing the busy-wait corresponding to polling, even at 800 MHz, still requires a considerable amount of power. However, when the same thread is blocked, the consumption is decreased significantly, to 50–55 W.

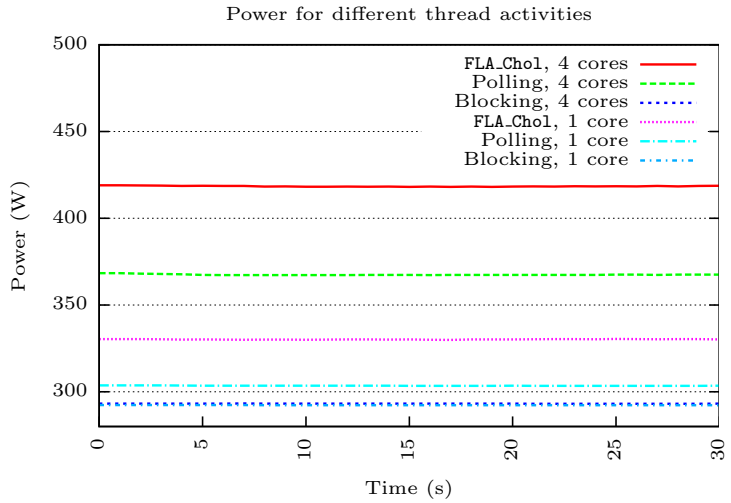


Figure 6.3: Power consumption of different actions performed by threads on TESLA2.

6.1.2 EA2: Remove polling when there are no ready tasks

Our second energy-saving technique replaces the polling state of "inactive" threads by an energy-friendly, blocking one, thus promoting the associated core to a C-state. 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. Whether these theoretical savings yield an actual gain or not will depend, however, on the existence of idle long periods during the execution of the algorithm, and the overhead of blocking/activating a thread. In SuperMatrix, upon completion of the execution of a kernel, a thread queries the list of ready tasks for new work. When there are no ready tasks, this query results in an active polling by the thread that produces a waste of energy. Our second energy-aware technique avoids this situation using POSIX semaphores to control the activity of "idle" threads. Now, when a thread that polls the ready list for a new job receives a negative answer (there is no task ready for execution at the moment), it blocks itself (with the system call `sem_wait()`). On the other hand, when a thread completes the execution of a task, it updates the dependencies of the tasks in the pending list. In case this implies moving k tasks from the pending list to the ready list, this thread will also enforce that there exist k active threads, including itself. (Using system call `sem_post()` to activate other threads, if necessary.) This mechanism ensures that there is basically one active thread per task in the ready list (to avoid introducing delays in the execution of tasks as well as deadlock) and that no continuous polling is being done on an empty list (to economize energy).

Although current architectures incorporate DVFS that can be used to reduce power during these idle periods, Figure 6.3 illustrates that we can do better using our technique which, in addition, is also useful for hardware where DVFS technology is not available. In particular, the figure reports the power usage of one/four cores on TESLA2 platform when repeatedly running kernel CHOL (lines labeled as "`FLA_Chol`"), performing a busy-wait ("Polling") and blocked ("Blocking"). Thus, depending on the number of active cores, approximately 10–80 W can be saved by avoiding polling, provided this does not affect the execution time. On more recent processors, the difference is also large, up to 30 W or 40 % for a single core (see Figure 6.2).

6.1.3 EA3: Remove polling when the GPU is running

Our third saving-energy technique replaces the polling state of the busy-waits performed when a thread waits until the completion of a GPU task. Therefore it is only suitable for multi-GPU platforms. When a thread encounters a task that has to be executed on the bound GPU, it invokes the corresponding CUBLAS kernel. The calling thread does not block and, thus, queries the ready list for more work. In this moment, if the thread obtains a second task to be run in the GPU, it tries to execute it, invoking the corresponding CUBLAS kernel. As a result and because the thread associated to the GPU can only serve one kernel at a time, it remains in a busy-wait until the first kernel completes its execution on the GPU.

To solve this problem our modified energy-aware runtime *i)* invokes `cudaSetDeviceFlags` with the `cudaDeviceBlockingSync` parameter, which blocks the CPU thread on a synchronization primitive when waiting for the device to finish work; and *ii)* adds the corresponding synchronization primitive after the CUBLAS routine performed by `libflame`. Routine `cudaSetDeviceFlags` allows to specify the behavior of the active host thread when it executes device code. To achieve this, each thread calls this routine before the CUDA runtime is initialized. After that, all synchronizations using `cudaThreadSynchronize` will suspend the execution of the calling thread until the device finalizes its work, thus "sleeping" the core to avoid the potential energy-consuming state.

To evaluate the impact of this option, we have invoked the CUBLAS kernel for GEMM 100 times, doing a polling and a blocking wait (setting accordingly the `cudaSetDeviceFlags`) with the `cudaThreadSynchronize`. The results showed that, on average, the blocking synchronization only increases the total execution time around 2 %.

6.2 Dense Linear Algebra

In this section we report the performance/energy impact when leveraging the techniques presented in the previous section to the SuperMatrix runtime using a representative set of dense linear operations executed on multicore and hybrid CPU–GPU platforms.

6.2.1 Multicore architectures

The experiments reported in this section were obtained using IEEE double-precision arithmetic on WT_AMD. A modified version of the SuperMatrix runtime in `libflame` version 5.0–r5587 was designed to leverage the three energy-saving techniques described in the previous section. Execution times/power measurements were obtained for routine `FLASH_LU_piv` (for the blocked right-looking variant of the LU factorization with partial pivoting) from `libflame`, linked to the original and energy-aware implementations of the runtime. Matrices were generated with random entries uniformly distributed in [0,1], so that pivoting actually occurs during the computation of the triangular factors. Our evaluation includes a variety of square matrices whose dimensions range from 2,048 to 12,288 and the block size is set to $b = 512$. This block dimension was close to optimal for most kernels involved in this factorization. Power was measured using our internal DCM wattmeter (see Table 3.3).

Our first experiment evaluates the existence and length of idle periods during the computation of the LU factorization with partial pivoting on an 8 cores of the AMD processor of WT_AMD; see Figure 6.4. Let us examine the two extreme cases: when the problem size is $n = 2{,}048$, 53 % of the time there is a single active thread and only during 9 % of the time all threads are performing work. On the contrary, if the problem size is much larger, e.g. $n = 10{,}240$, about 26 % of the time there is one active and most of the remaining period all the 8 threads are running. The conclusion

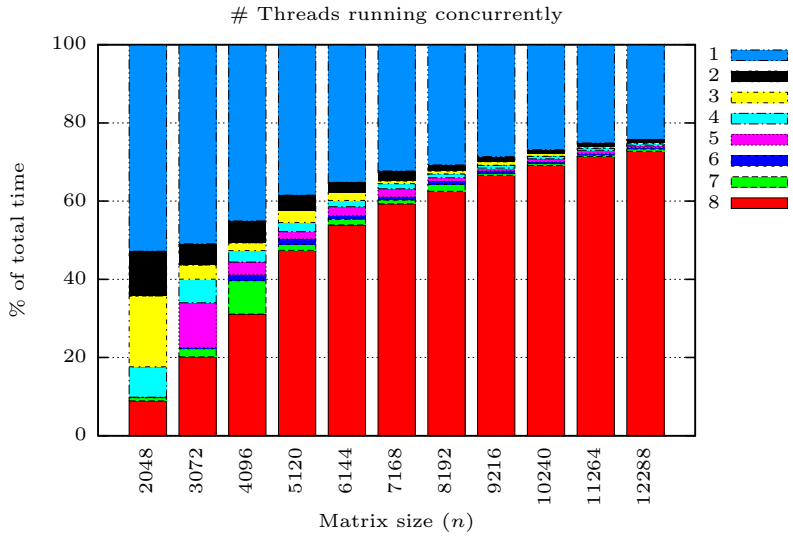


Figure 6.4: Thread activity during the execution of the LU factorization with partial pivoting.

from this experiment is that, indeed, there exists the opportunity of saving energy by carefully controlling the level of activity of idle threads.

The second experiment measures the actual gains that can be attained by our energy-aware approaches, evaluating the real savings introduced by techniques EA1 and EA2. To evaluate the impact of these techniques, we compare the original SuperMatrix runtime with three modified variants: the first one reduces the operating frequency during polling periods via DVFS (EA1); the second one blocks idle threads to avoid polling (EA2); finally, the third variant combines both techniques (EA1+EA2). While the original SuperMatrix uses the `performance` Linux governor to control frequency for maximum performance, we manually control the operating frequency in the experiments when using EA1 with the `userspace` Linux governor. The EA2 technique leaves the operation of DVFS in the hands of the OS with the `ondemand`[1] governor. The combination of both techniques removes polling intervals and delegates the control of DVFS to SuperMatrix.

Figure 6.5 illustrates the effect of the energy-saving strategies on the execution time, the energy consumption and the "application energy consumption". The graphs on the left-hand side report absolute values, while the ones on the right-hand side correspond to relative results with respect to the original runtime. The data (sequence of power samples) for the application energy consumption are obtained from those of the total energy consumption, subtracting the power when the machine is idle (51.13 W). The results demonstrate that these techniques introduce a minimal overhead (in terms of longer execution time, due e.g. to the period required to "wake-up" blocked threads) in the execution time. Concretely, there is an increase of 2.5 % at most for the smallest problem sizes while, for others, there is no appreciable difference between the results obtained with the original runtime and those of the different energy-saving variants. On the other hand, the effect on energy efficiency is much more relevant. For the smallest problem sizes, the number of tasks is relatively low compared with the number of threads, which results in gaps (idle periods) during the execution of the algorithm, and this translates into significant energy savings. Specifically, EA1 potentially leads to savings of 2 % in energy for largest problem sizes, while for EA2 the gain is higher (5 %). In combination, the use of both techniques produces a similar energy-savings than using only EA2.

[1] Although we evaluated several other governors (`powersave`, `conservative`, etc.), they all offered a significant increase in the execution time which resulted in a higher energy consumption.

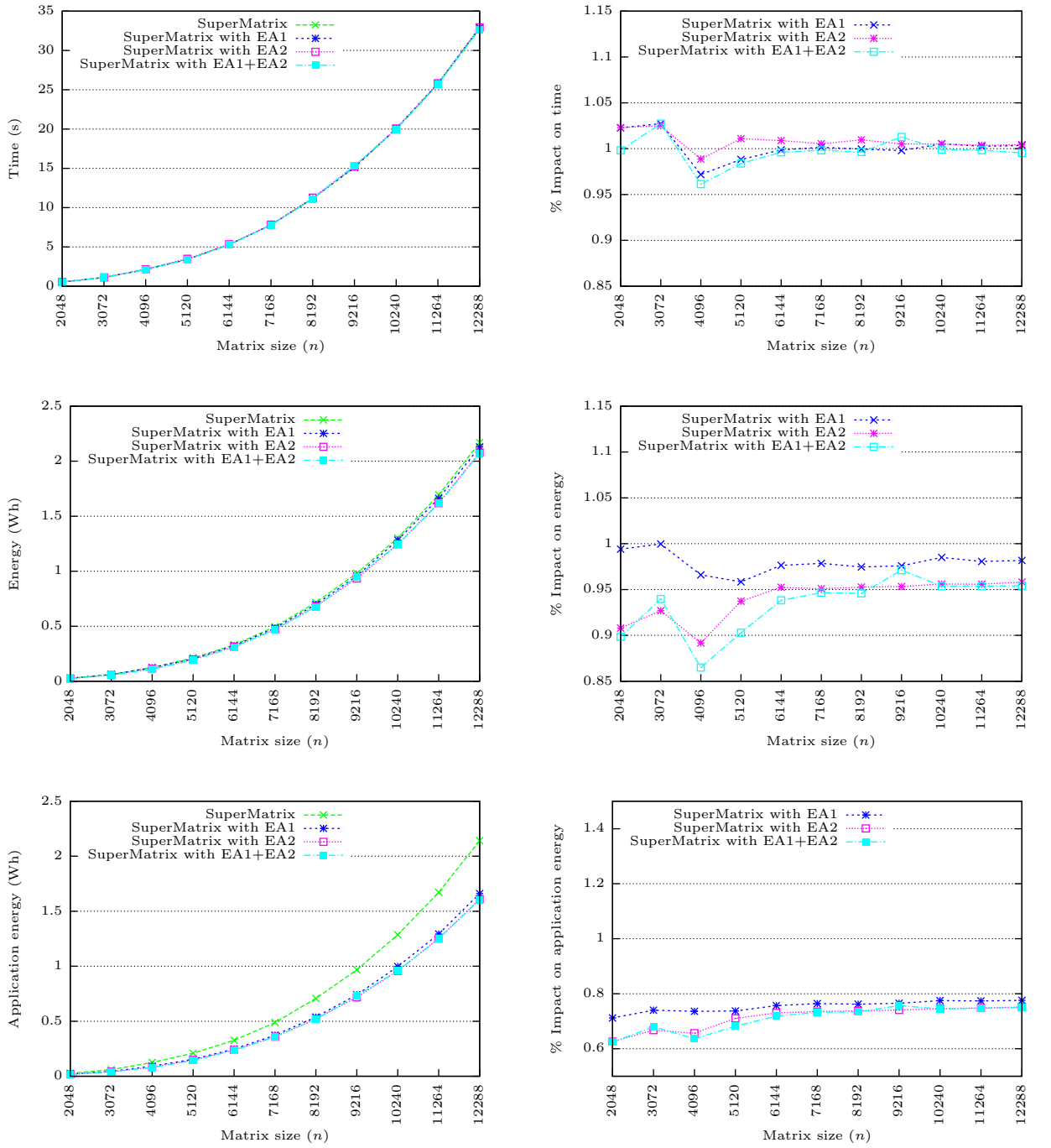


Figure 6.5: Impact on time and energy of the E1/EA2 energy-aware techniques of the LU factorization with partial pivoting.

6.2.2 Hybrid CPU–GPU architectures

We evaluate the original implementation of the SuperMatrix runtime for multi-GPU platforms in `libflame` and the modified versions that incorporate the EA2 and EA3 energy-aware techniques described in the previous section. Execution times and power measurements are obtained in TESLA2, and they correspond to those of routines `FLASH_Chol` (blocked right-looking variant of the Cholesky

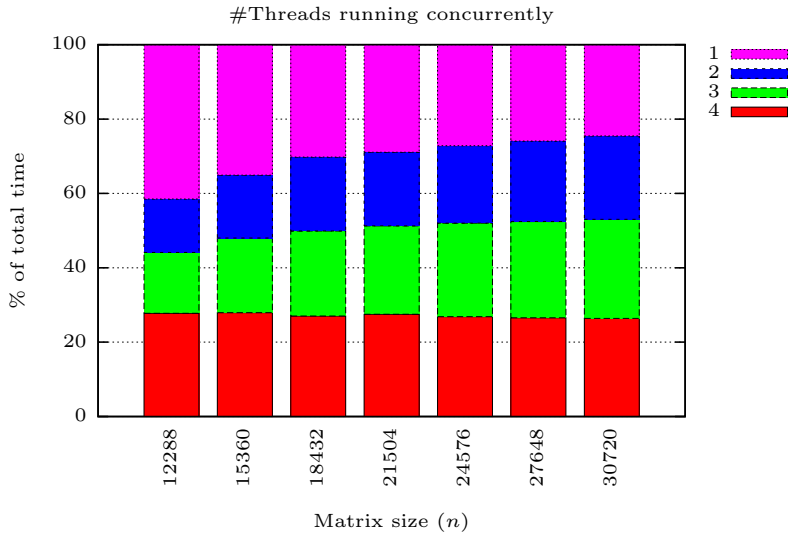


Figure 6.6: Thread activity on CPU during the execution of the Cholesky factorization.

factorization) and `FLASH_LU_piv` (right-looking version of the LU factorization with partial pivoting) from the `libflame` library, linked to the original and energy-aware versions of the runtime. Our experimental study includes a variety of matrix dimensions, ranging from $n = 6{,}144$ to 30,720, with block size $b = 1{,}536$. (This block dimension was close to optimal for most GPU kernels involved in these factorizations.) Power was measured using the WattsUp wattmeter. The case reported in the figures corresponds to the average from the point of view of execution time.

The Cholesky factorization

Our first experiment determines the periods during which there exist idle cores, because there are no tasks in the ready list or the linked GPU is performing work. (In this experiment, a core is performing "useful" work when it is executing a kernel of type CHOL, handling the structures that control the dependencies, or transferring data to/from the GPU memory address space.) The results in Figure 6.6 show that, for the smallest problem dimension $n = 12{,}288$, the percentage of time that 1–3 threads are used is above 70 % of the total. (In other words, the four cores of the platform are being used simultaneously only 30 % of the time.) When the problem size grows up to $n = 30{,}720$, this ratio sightly decreases to 25 %. These numbers clearly identify the energy-saving potential in the task-parallel execution of the algorithm.

The second experiment analyzes the theoretical and actual gains that are attained with the energy-aware techniques, contrasting them with the potential and real savings introduced by techniques EA2 and EA3. In order to do this, we compare the original SuperMatrix runtime with three modified variants: the first one employs semaphores to block threads when there are no CPU tasks (EA2); the second one integrates the idle-wait `cudaThreadSynchronize` calls in order to suspend thread activity during execution of GPU kernels (EA3); finally, the third variant combines both techniques (EA2+EA3). In our experiments, we do not modify the operating frequency of sockets/cores via DVFS (related to EA1) because the version of Linux kernel of the platform –CentOS 5.3– does not feature this capability.

Figure 6.7 illustrates the impact of the energy-aware techniques on the execution time, the total energy consumption and the "application energy consumption". The graphs on the left-hand side report absolute values, while the ones on the right-hand side correspond to relative results

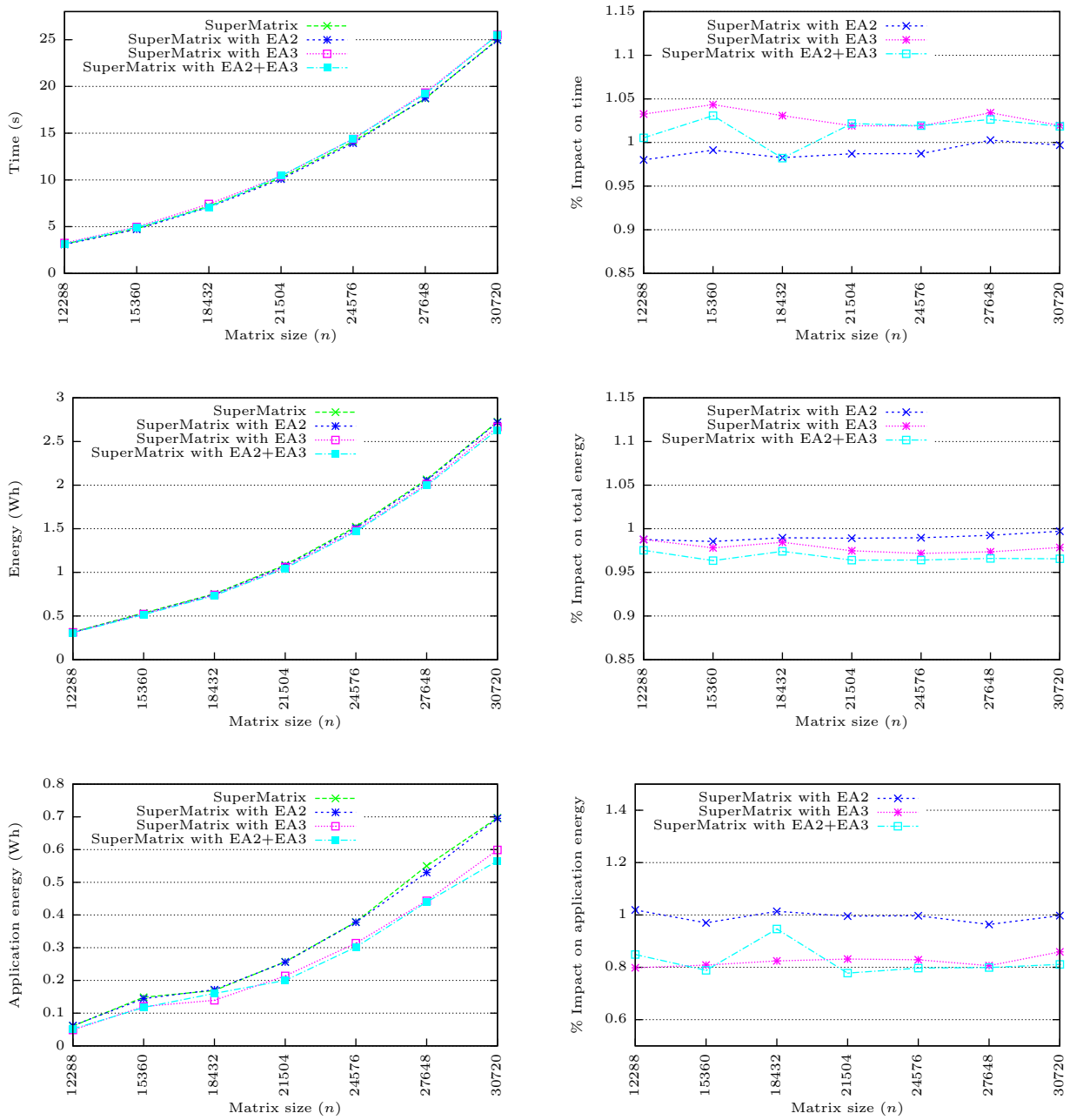


Figure 6.7: Impact on time and energy of the EA2/EA3 energy-aware techniques of the Cholesky factorization.

with respect to the original runtime. The data (sequence of power samples) for the application energy consumption are obtained from those of the total energy consumption, subtracting the power when the machine is idle (292 W). The results demonstrate that these techniques introduce a minimal overhead (in terms of delays, due to the time required to "wake-up" blocked threads) in the execution time. Concretely, there is an increase of around 2–4 % for all problem sizes when using EA3 while, for EA2, no difference can be appreciated between the results obtained with the original runtime and those of the different energy-saving version. On the other hand, from the energy-consumption perspective, the graphs in the middle row of the figure show that, on average,

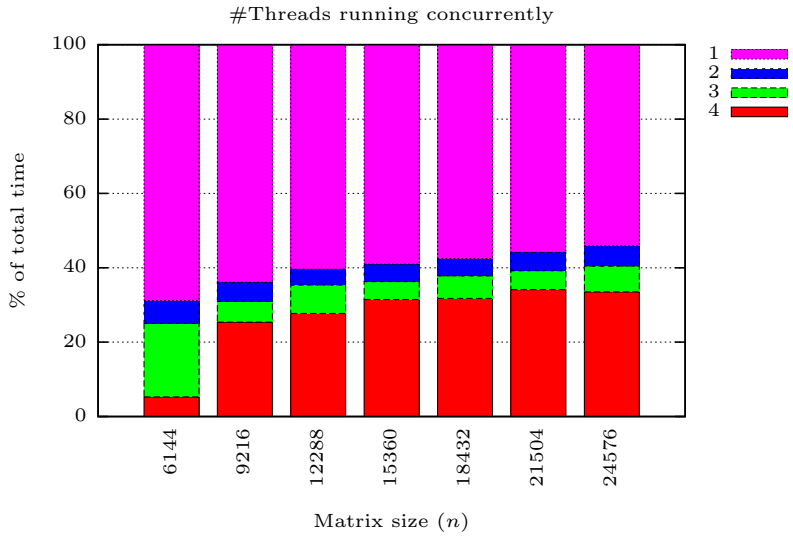


Figure 6.8: Thread activity on CPU during the execution of the LU factorization with partial pivoting.

EA2 potentially leads to an energy-saving of 1 %, while EA2 could provide slightly higher gains, around 3 %. In combination, the use of both techniques yields a practical saving around 4 %. The graphs in the last row only consider consumption due to the application and show that, on average, EA2 does not provide any practical energy saving for this factorization. However, EA3 provide significant gains, around 18 %, and the combination of both techniques renders energy savings that, on average, are close to 20%.

The LU factorization with partial pivoting

We next repeat the experiment to illustrate the energy-saving potential in the hybrid GPU–GPU task-parallel execution of the algorithm for the LU factorization with partial pivoting. For this purpose, we determine the periods during which cores are idle (there are no tasks in the ready list or the bound GPU is performing work). The results in Figure 6.8 show that, for the smallest problem dimension, $n = 6{,}144$, the percentage of time that 1–3 threads are used is above 92 % of the total while the four cores are being used only 8 % of the time. When the problem size grows up to $n = 24{,}576$, these ratios change to 45–55 %.

Figure 6.9 shows the benefits of the energy-aware techniques on the energy consumption as well as the impact on the execution time. In this case, EA3 yields a minor increase of the execution time (1 %), while EA3 and the combination of both techniques does not affect the time. On the other hand, from the energy-consumption perspective, the graphs in the middle row of the figure show that, on average, EA3 leads to 4 % savings of the total energy in some matrix sizes, while EA3 attains slightly higher gains, around 7–9 %. In combination, the use of both techniques deliver an energy saving around 9 %. Finally, the graphs in the last row show that, on average, EA3 does not provide significant savings in the application energy, while EA2 provide remarkable gains, around 35 %. The combination of both techniques shows energy savings reaching 38 %.

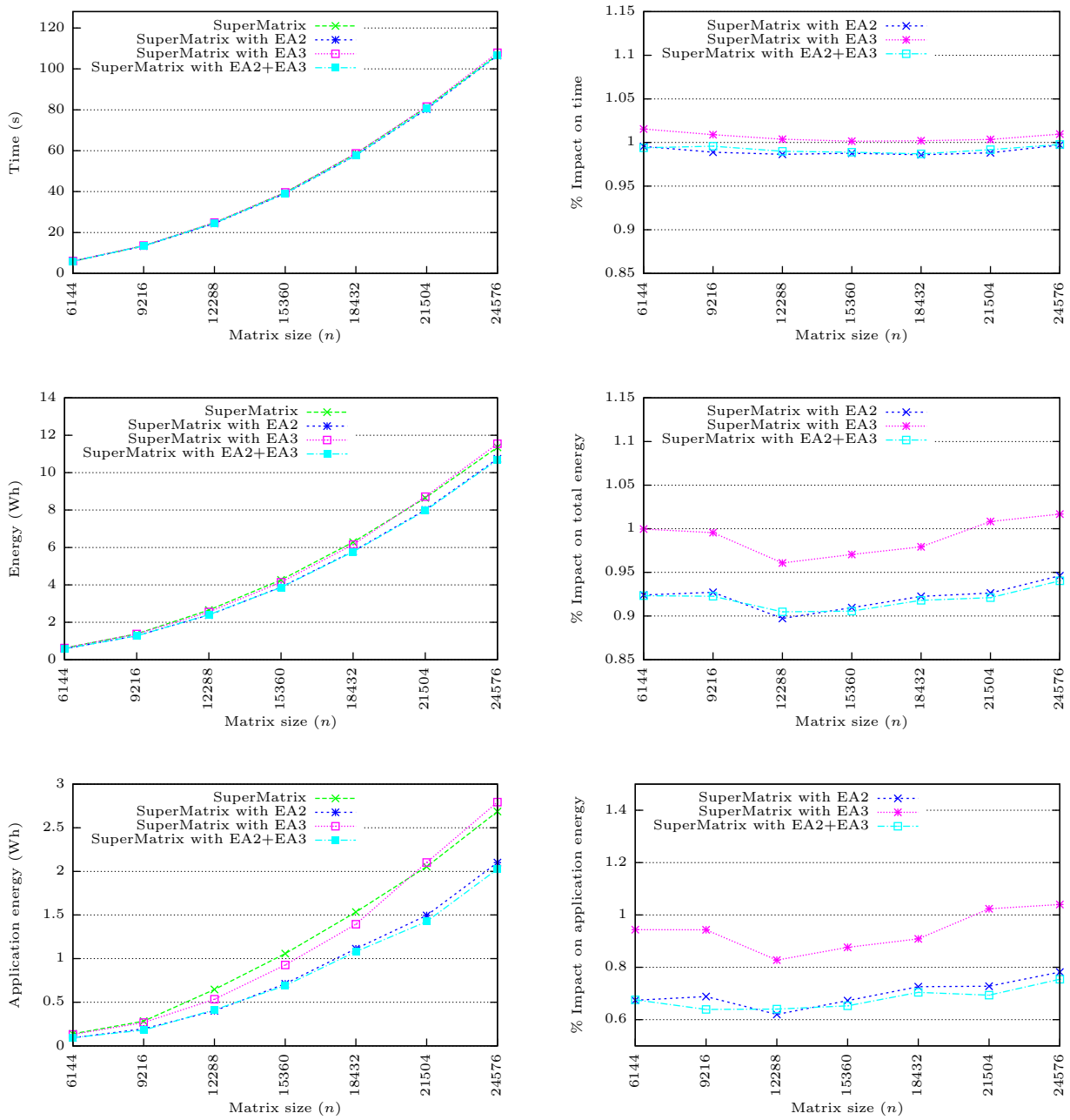


Figure 6.9: Impact on time and energy of the EA2/EA3 energy-aware techniques of the LU factorization with partial pivoting.

Improved multi-GPU SuperMatrix runtime

In this section we first present the energy-saving techniques introduced in the runtime, and their practical outcome on the execution of the LU and the Cholesky factorization using the improved version of SuperMatrix runtime that allows to advance critical tasks introduced in Section 2.2.2.

Let us first evaluate the gains of the EA2 and EA3 energy-saving techniques (presented in Section 6.1) for the task-parallel execution of `libflame` routines on hybrid CPU–GPU platforms using the tuned SuperMatrix runtime that advances critical tasks.

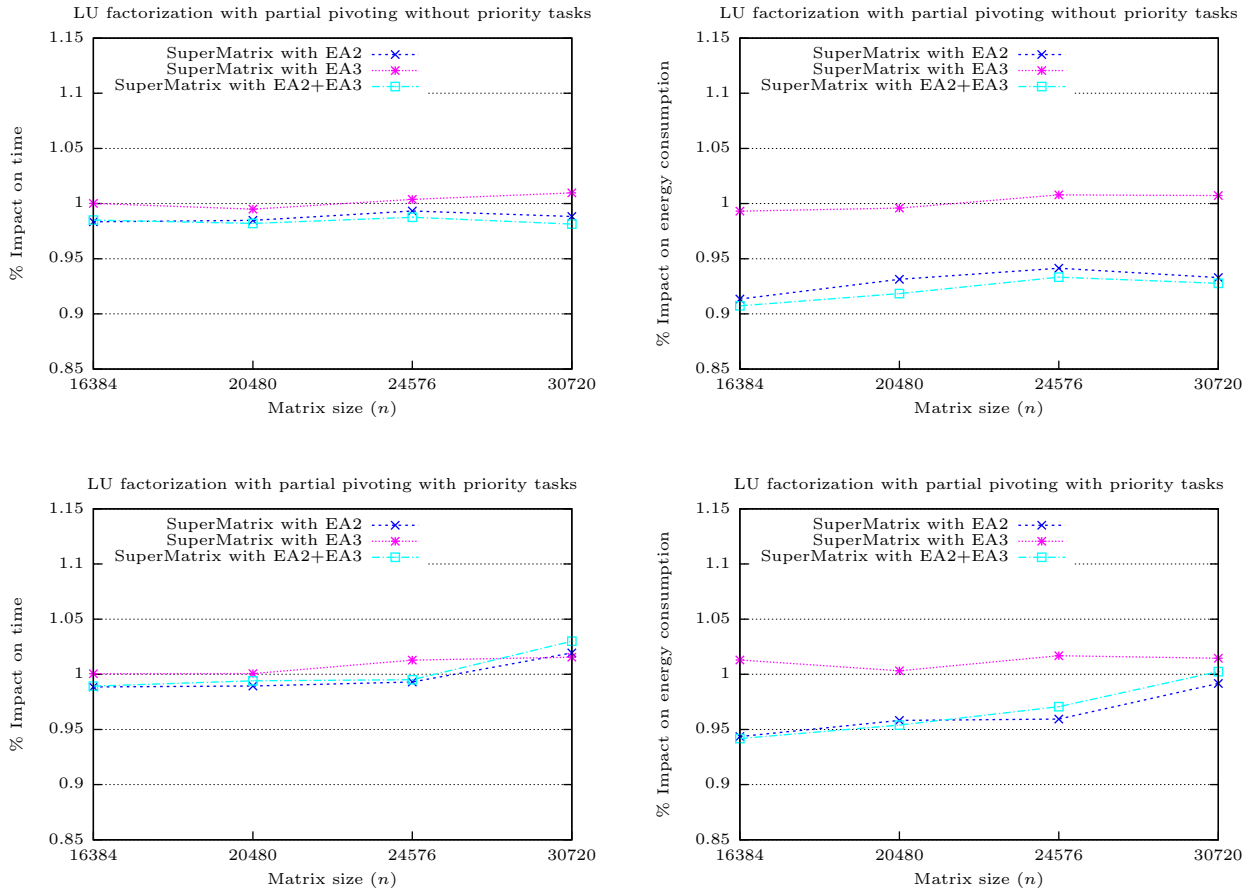


Figure 6.10: Impact on time and energy of the energy-aware techniques of the LU factorization with partial pivoting without (top plots) and with priority tasks (bottom plots).

Figure 6.10 reports the impact on the execution time and on the energy consumption when the energy-aware version of SuperMatrix with and without priority tasks is employed to execute the LU factorization with partial pivoting. Figure 6.11 reports the same information for the Cholesky decomposition using both versions with and without priority tasks of SuperMatrix. In all cases, we employed the multi-GPU mode with 4 GPUs. The combination of both techniques is referred to as "EA2+EA3". These results show a variety of energy gains, from close to 10 % in some cases to even a waste of energy in a couple of cases, depending on the dense linear algebra operation, matrix dimension, and technique. In the following section we relate these results with the actual consumption of dynamic power and the length of idle periods.

It is of special interest the behavior of the power-aware runtime for the LU factorization with and without task priorities. The reduction in execution time when task priorities are used yields an important reduction in energy consumption (compare right plots in Figure 6.10); however, this improvement in execution time is mainly due to a reduction in the amount of idle periods in the parallel execution. As our power-aware techniques exploit idle periods, the expected improvements from the application of these mechanisms are less significant as idle time decreases. That is the main reason why the percentage of reduction in energy consumption is smaller when priorities are applied (less than 6 %) than when priorities are not used (up to 9 %).

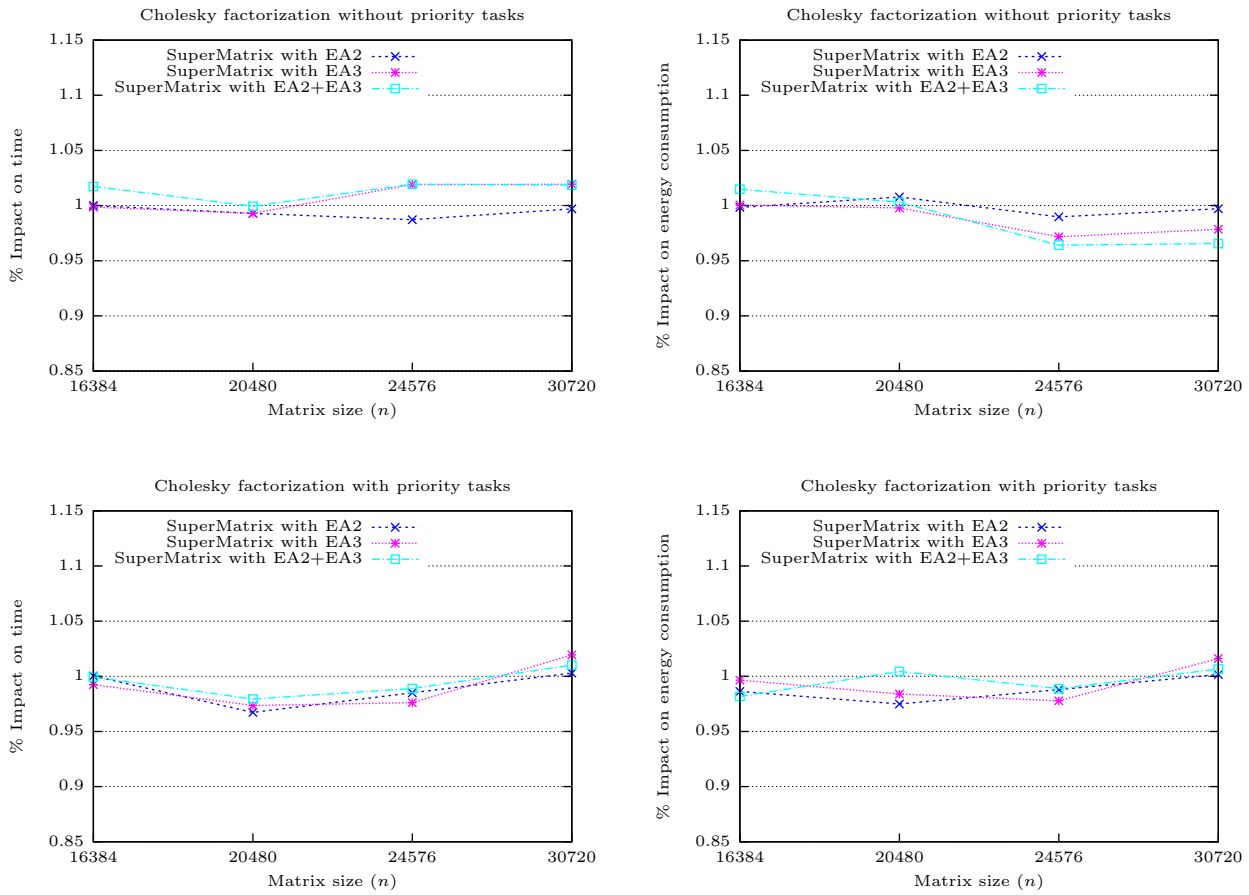


Figure 6.11: Impact on time and energy of the energy-aware techniques of the Cholesky factorization with partial pivoting without (top plots) and with priority tasks (bottom plots).

6.2.3 Leveraging power models in SuperMatrix

We next employ a combination of the power/energy models for hybrid and multi-socket platforms presented in Sections 4.5.1 and 4.5.2, respectively, to assess the quality of the energy-savings observed in the previous experiments in the context of the hybrid CPU–GPU platforms addressed in this work. Given that the energy savings due to EA2 and EA3 techniques are produced in the CPU cores, we only take into account the power consumption in the host. The proposed model can eventually be incorporated into the runtime to dynamically determine the optimal number of computational resources to employ for the execution of a dense linear algebra operation. Since there exist different configurations of resources that result in equivalent or nearly equivalent performance, the *hybrid* SuperMatrix runtime can make use of the model to automatically select the most energy-efficient configuration.

Consider, e.g. the execution of the LU factorization with partial pivoting, for a matrix of dimension $n = 20{,}480$ with block size $b = 1{,}024$, using the multi-GPU mode and 4 GPUs. The total execution time employing the runtime enhanced with priorities is 39 seconds, of which CPU cores are inactive during 50.9 %. On the other hand, the total energy consumption is 3.88 Wh, with about 82.9 % corresponding to the sum of system and static power dissipation. Thus, the highest savings that we could expect by exploiting those periods during which CPU cores are idle is approximately

$0.509 \cdot (1 - 0.829) \approx 7.6\,\%$, which is consistent with the savings reported in Figure 6.10 for that particular problem size.

6.3 Sparse Linear Algebra

The solution of sparse systems of linear equations is an ubiquitous problem in scientific and engineering applications which has been tackled in many projects during the past decades [111]. One ongoing effort has resulted in ILUPACK (*Incomplete LU decomposition PACKage*) [76], a package that combines ILU factorizations with iterative Krylov subspace methods. Compared with sparse direct solvers, this class of methods have proven to be quite competitive for a wide range of applications (specially those arising from 3D PDEs) because of their moderate computational and memory requirements [111].

Due to the scale of the linear systems appearing in many applications, and the computational cost of the numerical methods, most solvers target parallel architectures. Following this trend, the performance benefits of exploiting task-parallelism within ILUPACK for the solution of sparse linear systems on multicore processors have been demonstrated in [7] and [8] which present two concurrent versions of ILUPACK, for multicore architectures and distributed-memory (message-passing) platforms respectively. More details of ILUPACK can be found in Section 2.6.1.

One of the goals of this section is to analyze the effect that the exploitation of our energy-aware techniques provide on an energy-aware version of the runtime for the complete preconditioner+iterative solve process in ILUPACK. Afterwards, we explain the energy gains of the new runtime and its effects when the different P-states are applied.

6.3.1 Environment setup

In all our experiments, we employ a scalable symmetric positive definite sparse linear system of dimension $n = N^3$, resulting from a partial differential equation $-\Delta u = f$ in a 3D unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions $u = g$ on $\delta\Omega$, discretized using a uniform mesh of size $h = \frac{1}{N+1}$. We set $N = 252$, which yields a linear system with about $16 \cdot 10^6$ unknowns and $111 \cdot 10^6$ nonzero entries in the coefficient matrix. All tests were performed using IEEE double-precision arithmetic.

We employ two target servers in our experiments, WT_AMD and WT_ITL. Details on the voltage–frequency pairs ($V_{CC_i} - f_i$) associated with each P-state (Pi) are collected in Table 3.2. In our experiments, power samples were obtained using the NI internal wattmeter.

The multithreaded implementation of ILUPACK is built on top of the OpenMP interface available with Intel `icc` (version 12.1.3) on both platforms. Performance (core activity) traces were captured using our power-performance framework introduced in Section 3.2. Traces of CPU power modes were recorded using our PMLIB plug-in described in Section 3.2.3.

6.3.2 Leveraging the C-states in ILUPACK

We first investigate the exploitation of the C-states by the introduction of the EA2 technique in the runtime underlying ILUPACK, and relate its effect with the power model presented in Section 4.4. In the experiments in this section, we employ all the cores of the target platforms; and we set the Linux governor to `ondemand`, operating all the active cores in the same state P0 during all the execution (i.e., we do not allow voltage-frequency changes).

In Section 2.6.1 we exposed that the task-parallel calculation of the preconditioner in ILUPACK is organized as a directed task graph, with the structure of a binary tree and bottom-up

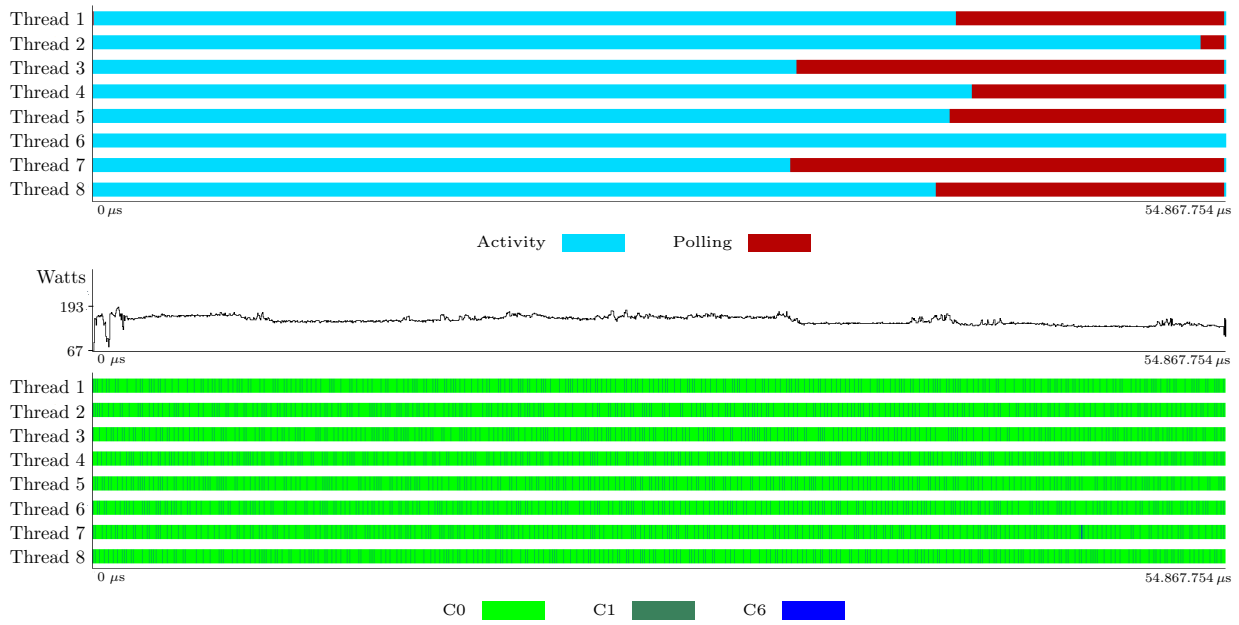


Figure 6.12: Traces of core activity, power and C-states (top, middle and bottom, respectively) during the computation of the ILU preconditioner using the performance-oriented, power-oblivious runtime, with all cores of WT_ITL in state P0.

dependencies, from the nodes (tasks) at each level to those in the level immediately above it. The subsequent iterative process basically requires the solution of (lower and upper) triangular linear systems per iteration, with tasks that are also organized as binary task-trees, but with bottom-up (lower triangular system) or top-down (upper triangular system) dependencies. In any case, when the tasks in these trees are dynamically mapped to a multicore platform by the runtime, the execution should result in periods of time during which certain cores are idle, depending on the number of tasks of the tree, their computational complexity, the number of cores of the system, etc. It is basically these idle periods that we could expect that the operating system leverages, by promoting the corresponding cores into a power-saving C-state employing our EA2 energy-aware technique.

Figure 6.12 presents the execution trace, power consumption, and C-states observed during the computation of the ILUPACK preconditioner, using the original (power-oblivious) runtime, with all cores of WT_ITL in state P0. Surprisingly, the results are quite different from what we could expect: Idle periods do not show a transition of the corresponding core to a power-saving C-state and the associated reduction of the power rate. Figure 6.13 reports an analogous behavior for the (preconditioned) iterative solution stage on WT_ITL (and similar results were also obtained for both stages on WT_AMD). A closer inspection of the runtime that leverages the task-parallelism in ILUPACK reveals the reason for these results. Concretely, in the original implementation of ILUPACK runtime, upon encountering no tasks ready to be executed, "idle" threads simply perform a "busy-wait" (polling) on a condition variable, till a new task is available. This strategy thus prevents the operating system from promoting the cores into a power-saving C-state because the threads are not actually idle (but doing useless work).

As an alternative to the previous power-hungry strategy, we integrate the principles of our EA2 technique into the runtime underlying ILUPACK. Thus, the energy-aware version applies an "idle-wait" (blocking) whenever a thread does not encounter a task ready for execution and, thus, becomes inactive. As in the original version of the runtime, upon completing the execution of a task,

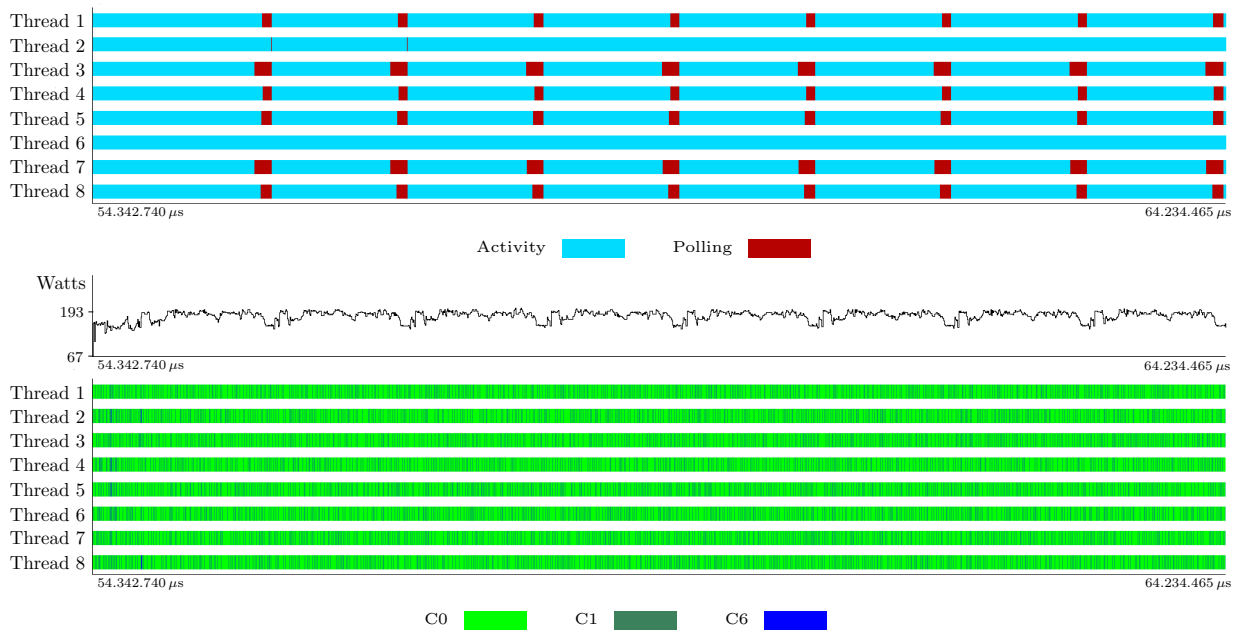


Figure 6.13: Traces of core activity, power and C-states (top, middle and bottom, respectively) during (part of) the iterative solution stage using the performance-oriented, power-oblivious runtime, with all cores of WT_ITL in state P0.

a thread updates the corresponding dependencies identifying those tasks, if any, that have become ready for execution. However, in the power-aware runtime, the thread also ensures that the number of active (non-blocked threads) is, at least, equal to the number of ready tasks, releasing blocked threads if needed. The effect of idle-wait on the power trace and use of the C-states of WT_ITL is illustrated in Figure 6.14, for the calculation of the preconditioner, and Figure 6.15, for the iterative solution stage. Compared with the performance-oriented (but power-hungry) implementation of the runtime (see Figures 6.12 and 6.13), the new runtime effectively allows inactive cores to enter a power-saving C-state, thus yielding the sought-after power reduction.

The pending question, however, is whether the adoption of the power-aware runtime comes with a performance penalty which may blur the energy benefits, as in most cases the key factor is energy instead of power. Table 6.1 compares the execution time, average power, and energy consumption of the two runtimes, showing that fortunately this is not the case for the computation of the preconditioner and iterative solution, on any of the two target platforms when operating in state P0. For example, consider platform WT_AMD: for a minimal increase in the total execution time, from 286.28 s to 287.91 s, we observe reductions in the (average) power from 240.17 W to 227.16 W for the preconditioner; and from 269.27 W to 230.80 W for the solver. The outcome is a decrease of the total energy from 75,163.74 J to 67,758.84 J (−10.88 %). The power reductions attained by the power-aware implementation with respect to the power-oblivious case are given in form of ratios (in %) in the columns labeled as "Experimental" of Table 6.2 (averaged for 10 repeated executions). Combined with the negligible impact of the runtime on the execution time, these power figures also justify similar energy savings.

Let us now relate the power-energy reductions attained by the reimplementation of the runtime with the EA2 technique that leverages the CPU C-states to the power model presented in Section 4.4. For this purpose, we need to *i)* to account for the periods of "idle" time during the execution of ILUPACK, with both the original and energy-aware variants; as well as *ii)* to assess

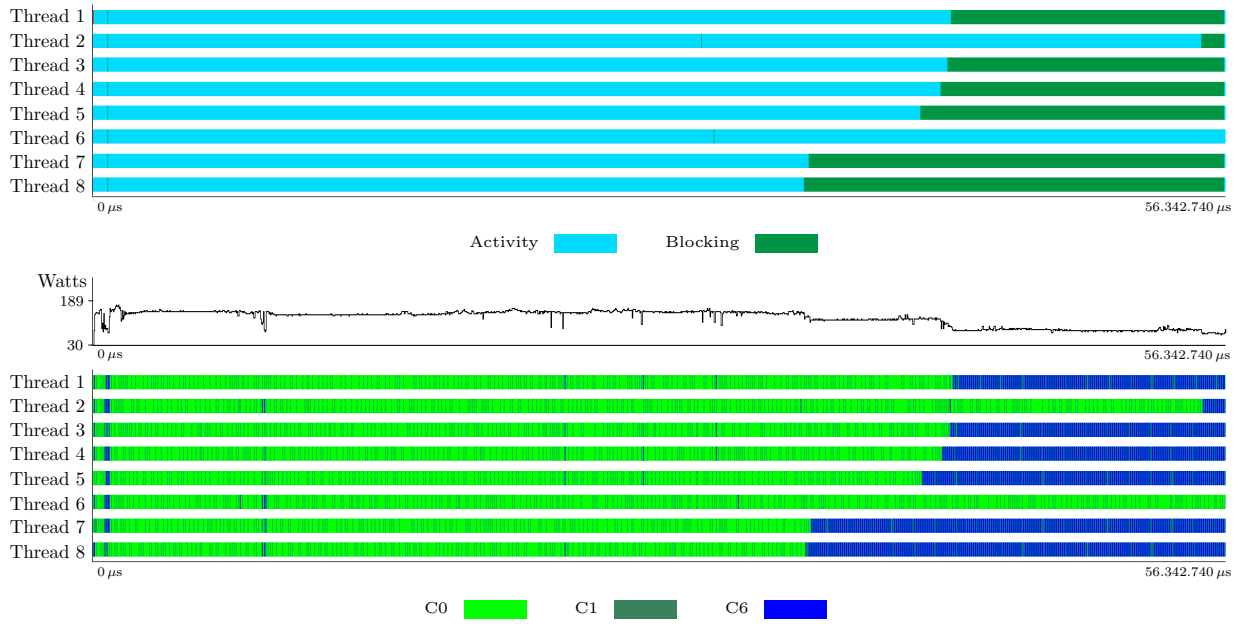


Figure 6.14: Traces of core activity, power and C-states (top, middle and bottom, respectively) during the computation of the ILU preconditioner using the power-aware runtime, with all cores of WT_ITL in state P0.

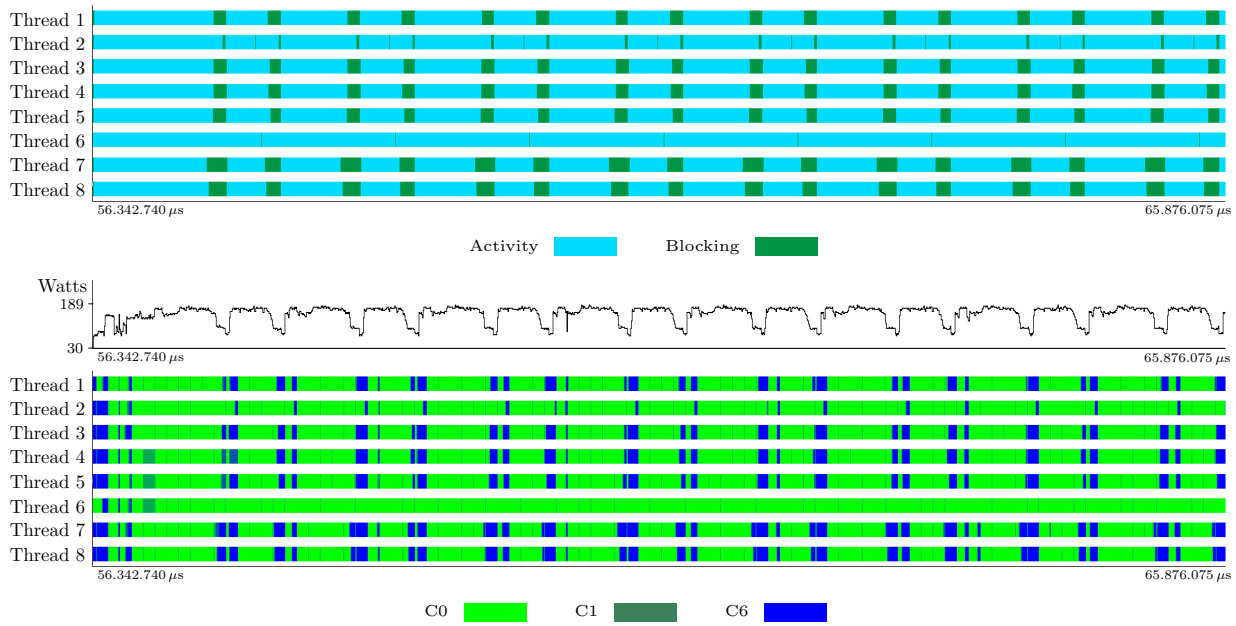


Figure 6.15: Traces of core activity, power and C-states (top, middle and bottom, respectively) during (part of) the iterative solution stage using the power-aware runtime, with all cores of WT_ITL in state P0.

the impact of the EA2 technique, i.e., replacing a busy-wait (polling) for an idle-wait (blocking). In order to tackle *i)*, we follow a pragmatic approach, and simply execute the codes and measure the actual idle and computation times in our case, e.g. using our power-performance tracing framework.

Platform	Runtime type	Preconditioner			Iterative solver			Total	
		Time	Power	Energy	Time	Power	Energy	Time	Energy
WT_AMD	Oblivious	66.11	240.17	15,878.82	220.17	269.27	59,284.92	286.28	75,163.74
	Aware	66.52	227.16	15,112.06	221.39	237.80	52,646.41	287.91	67,758.84
WT_ITL	Oblivious	54.01	137.58	7,431.67	146.67	162.34	23,809.87	200.68	31,241.69
	Aware	54.47	125.70	6,847.47	148.15	150.17	22,247.93	202.62	29,095.40

Table 6.1: Execution time, average power and energy of the power-oblivious and power-aware implementations of the runtime (top and bottom, respectively) with all cores operating in state P0.

Platform	Preconditioner		Iterative solver	
	Theoretical	Experimental	Theoretical	Experimental
WT_AMD	89.58	93.88	91.47	87.36
WT_ITL	90.11	90.64	93.65	92.05

Table 6.2: Expected and observed (theoretical and experimental, respectively) power ratios (%) between the power-aware implementation of the runtime and the power-oblivious one, with all cores running in state P0.

For *ii)*, we use the data in Table 4.4 for P_0^Y, P_0^S; and estimate $P^{C1}_{ilu,0} = \beta_{ilu,0}$ using a procedure for ILUPACK analogous to that exposed for the three benchmark kernels combined with linear regression. Finally, we assume that a core promoted to a sleep state does not dissipate any core power.

Consider $P^T_{pilu,0}(c)$ and $P^T_{bilu,0}(c)$ denote, respectively, the total power dissipated during the execution of ILUPACK, using the power-oblivious (polling *pilu*) and power-aware runtimes (blocking *bilu*), with c cores in state P0. Since now some cores may be inactive during a certain part of the execution, we need the power model (4.10) of Section 4.4: which now becomes

$$P^T_{pilu,0}(c) = f_{pilu,0} \cdot (P_i^Y + P_i^S + P^{C1}_{ilu,0} \cdot c) + (1 - f_{pilu,0}) \cdot (P_i^Y + P_i^S + P^{C1}_{polling,0} \cdot c)\ . \qquad (6.1)$$

The first term of the addition captures the cost of the cores performing useful work during the computation of ILUPACK (alike (4.10)), and appears multiplied by $f_{pilu,0}$, which corresponds to the ratio of the total time that this computation occupies. Thus, the second part of the addition represents the remaining fraction of the total time, $(1 - f_{pilu,0})$, and captures the power dissipation of the cores performing polling. In our evaluation, we set $P^{C1}_{polling,0} = P^{C1}_{busy,0}$, as the underlying procedures are similar.

On the other hand, for $P^T_{bilu,0}(c)$, we have

$$P^T_{bilu,0}(c) = f_{bilu,0} \cdot (P_i^Y + P_i^S + P^{C1}_{ilu,0} \cdot c) + (1 - f_{pilu,0}) \cdot (P_i^Y + P_i^S)\ , \qquad (6.2)$$

as we assumed that a core in blocking mode wastes no power (i.e, $P^{C1}_{blocking,0} = 0$).

Table 6.2 compares the theoretical ratios $P^T_{bilu,0}(c)/P^T_{pilu,0}(c)$ with the experimental data (averaged for 10 different executions), showing a close matching between the two, below 2 % for WT_ITL and slightly larger, about 4 % for WT_AMD. These results illustrate the benefits of the power-aware runtime, but also the accuracy of the power model. For all other frequencies, as we will see next, the model always predict the power-ratio with an error below 2 %.

Platform	P-state, Pi	Preconditioner			Iterative solver		
		Time	Power	Energy	Time	Power	Energy
WT_AMD	P0	66.52	227.17	15,112.06	221.39	237.80	52,656.41
	P1	81.56	197.77	16,131.31	252.54	207.98	52,525.50
	P2	97.16	172.68	16,778.25	288.31	187.14	53,954.38
	P3	113.25	160.16	18,138.44	326.11	176.10	57,426.43
	P4	137.62	151.52	20,852.36	284.34	167.36	64,321.79
WT_ITL	P0	54.47	125.70	6,847.47	148.15	150.17	22,247.94
	P1	57.31	119.23	6,833.73	147.16	145.37	21,392.83
	P2	60.65	114.16	6,924.03	151.35	140.75	21,302.70
	P3	65.29	108.95	7,114.07	164.85	132.35	21,819.16

Table 6.3: Execution time, power and energy of the power-aware implementation of the runtime, with all cores in state Pi.

Platform	Pi/P0	Δf_i	ΔBW_i	Preconditioner			Iterative solver		
				ΔTime	ΔPower	ΔEnergy	ΔTime	ΔPower	ΔEnergy
WT_AMD	P1/P0	−25.00	−18.68	22.60	−12.94	6.74	14.07	−12.54	−0.22
	P2/P0	−40.00	−32.45	46.06	−23.99	11.02	30.23	−21.30	2.48
	P3/P0	−50.00	−42.29	70.24	−29.50	20.02	47.30	−25.94	9.07
	P4/P0	−60.00	−53.78	106.88	−33.30	37.98	73.60	−29.62	22.17
WT_ITL	P1/P0	−6.50	−1.10	5.21	−5.15	−0.20	−0.67	−3.20	−3.84
	P2/P0	−13.50	−0.86	11.35	−9.18	1.12	2.16	−6.27	−4.25
	P3/P0	−20.00	−1.33	19.86	−13.33	3.89	11.27	−11.86	−1.93

Table 6.4: Variations of frequency, bandwidth, execution time, power and energy ratios (%), of the power-aware implementation of the runtime, between state Pi and state P0.

6.3.3 Impact of the P-states on ILUPACK

In this section we evaluate the effect of the different frequencies or P-states available for each processor on the performance-power-energy trade-off of ILUPACK. For that purpose, we set the Linux governor mode to `userspace`, and operate all the cores of the platforms in the same P-state. In the following, we always employ the power-aware version of the runtime. Therefore, we assume that, when idle, a core will remain in one of the deep power-saving C-states (C1 or higher), consuming a negligible amount of power.

The general consensus is that, for a memory-bound computation, some benefit may result from operating the system cores at low frequencies. The reason is that, although there exists a linear dependence between the core performance and the frequency, the effect on the execution time of a memory-bound algorithm should be minor because the key for this type of computation is not core performance but memory bandwidth. On the other hand, for current multicore technology, a reduction of frequency is associated with a decrease of voltage (see Table 3.2) and, because of the relation between static power to $V_{CC}{}^2$ and dynamic power to $V_{CC}{}^2 \cdot f$, in principle we can expect a significant reduction of the power draw. However, the balance between these two factors, time and power, on the energy efficiency is delicate, and other elements also play a role. Whether these variations of time and power yield a loss or a gain in energy for ILUPACK is thus the question to investigate in this section.

Table 6.3 reports the impact of the P-states on the time, (average) power consumption and energy efficiency of the two stages of ILUPACK, calculation of the preconditioner and iterative solution, on both platforms. To help with the analysis of these results, Table 6.4 offers the variation of bandwidth and results that are experienced when moving from state P0 to state Pi, calculated as

$100 \cdot (M_i - M_0)/M_0$, where M_0 and M_i denote, respectively, the values of the magnitudes (parameters or results) in states P0 and Pi.

The first aspect to notice is that the presumed independence between execution time and core frequency does not hold on WT_AMD. This should not be a surprise as our experiment in Table 3.2 already revealed that there is a strong connection between the core frequency and the memory bandwidth in this platform (see also column ΔBW_i in Table 6.4). The combined decreases of frequency and memory bandwidth when moving from P0 to a higher P-state (between $-25\,\%$ and $-60\,\%$ for the former and from $-18.68\,\%$ to $-53.78\,\%$ for the latter) explain the increases of execution time for both the preconditioning stage (22.60–106.88 %) and the iterative solver (12.54–73.60 %) in this platform. The behavior of WT_ITL is quite different, which is partially explained because now the reduction of frequency does not bring a decrease of memory bandwidth. Still, for the preconditioner, the reduction of frequency when moving from P0 to a higher P-state ($-6.50\,\%$ for P1, $-13.50\,\%$ for P2 and $-20.00\,\%$ for P3) basically matches the increase of execution time for this stage (5.21, 11.35 and 19.86 %, respectively). We can take this as an indicator that the computation of the preconditioner (or, at least, parts of it) is not such a memory-bound computation as one could, in principle, presume. The results are different for the iterative solver. In this case, there is no significant difference in the execution time when running the computations in states P0 or P1, but the time increase when moving from P0 to P2/P3 is 2.16/11.27 %, which is still lower than what could be explained by the reduction of frequency alone.

From the performance point of view, the major conclusion of this analysis is that the best solution is to always run ILUPACK with all the cores operating at the highest frequency (i.e., in state P0), though in some cases —in particular, the iterative solver executed in frequencies P0, P1 and P2—, the differences are small on WT_ITL.

Performance is crucial and, under some circumstances, energy efficiency is also vital. From that point of view, a reduction of power is beneficial only if it does not yield an increase of execution time that blurs the positive effects on energy consumption. For the particular case of ILUPACK, the results in Table 6.3 show that, on WT_AMD, the most energy efficient solution is to execute the preconditioner with all cores in state P0 but the iterative solver in state P1. For WT_ITL, however, using states P1, P2 or P3 for the iterative solver results in small significant energy savings, from $-1.93\,\%$ to $-4.25\,\%$.

Let us connect again the power variations attained with the different P-states and the models for total power. For this purpose, we relate $P^T_{bilu,i}(c)$ and $P^T_{bilu,0}(c)$, using

$$P^T_{bilu,i}(c) = f_{bilu,i} \cdot (P^Y_i + P^S_i + P^{C1}_{ilu,i} \cdot c) + (1 - f_{bilu,i}) \cdot (P^Y_i + P^S_i)\ , \tag{6.3}$$

and the experimental data. Table 6.5 reports the accuracy of our model to capture the experimental behavior due to the variations of the P-state on ILUPACK, with an error of at most 3.08 % for WT_AMD and even smaller for WT_ITL.

6.4 Concluding Remarks

Performance-oriented decisions, like busy-waiting (polling) till new work is available, are far from uncommon, being adopted in runtimes like OmpSs (SMPSs) [118] or SuperMatrix and `libflame` [59] as well. Furthermore, the same performance-oriented but power-oblivious behavior appears, for example, when a synchronous GPU kernel is invoked with the default operation mode of CUDA [101] (the CPU remains in an active polling, waiting for the GPU to finish), or with the polling mode of certain MPI implementations (e.g., MVAPICH [96]). In all these cases, these energy-oblivious

Platform	Pi/P0	Preconditioner		Iterative solver	
		Theoretical	Experimental	Theoretical	Experimental
WT_AMD	P1/P0	88.05	85.48	84.17	87.64
	P2/P0	78.96	76.38	76.56	79.65
	P3/P0	73.50	70.78	71.60	74.85
	P4/P0	69.73	66.65	68.77	70.80
WT_ITL	P1/P0	95.62	95.54	96.47	96.47
	P2/P0	90.84	90.80	91.25	91.84
	P3/P0	87.21	86.94	85.96	87.77

Table 6.5: Expected and observed (theoretical and experimental, respectively) power ratios (%), of the power-aware implementation of the runtime, between state Pi and state P0.

strategies prevent the operating system from promoting the hardware into a power-saving C-states because the application is not idle but doing useless work. In this sense, solutions are needed to minimize the energy impact of the previous power-hungry strategies.

In this chapter we have introduced three energy-aware software techniques that aim at promoting hardware to low-consumption states and illustrate them through a series of examples using the SuperMatrix runtime. Furthermore, we integrate these techniques into the SuperMatrix runtime for the parallel execution of dense linear algebra operations on multicore and hybrid CPU–GPU platforms, in order to demonstrate the trade-off between energy consumption and execution time of both energy-oblivious/-aware runtime versions. The experimental results on platforms WT_AMD and TESLA2 demonstrate that, by avoiding active polling (busy-waits) in the runtime, it is possible to leverage periods when the general-purpose cores are performing no useful work, with a minor impact on the execution time. One crucial aspect is that, because of the clear separation between the linear algebra codes in `libflame` and the SuperMatrix runtime in charge of their task-parallel execution, our energy-aware techniques can be automatically applied to all routines of `libflame`. Also, we expect that the same energy-aware techniques are valid for other runtimes, in particular those of the OmpSs, PLASMA and MAGMA projects [118, 5], yielding similar benefits.

We have also implemented an energy-aware runtime for the complete preconditioner+iterative solve process in ILUPACK using the techniques developed throughout the chapter. In the second part of the chapter, we explain/justify the variations observed for the new energy-aware runtime and the effect of the different P-states for this particular application. We provide experimental results in two different multicore architectures and relate these to the model presented in Section 4.4. The introduction of the energy-aware runtime results from the experimental observation that, in an energy-oblivious execution of the original runtime for ILUPACK, idle threads with no useful task to execute, simply poll till new work is available. As a result, these threads dissipate a significant amount of power in current processors, for no practical performance benefit for the particular case of ILUPACK. Our energy-aware implementation replaces this behavior with a more power-friendly implementation, that blocks idle threads till new work is available. This requires a careful reorganization of the underlying runtime, to avoid deadlocks and ensure a rapid response that does not impair performance. As a result, we observed experimental savings in the energy usage between 7 and 13 %, at practically no cost from the performance point of view, which are clearly connected to the impact of the C-states by our power models. In theory, there exists a linear relation between performance and frequency, which could be expected to be even sublinear (at least on the Intel processor) for a presumably memory-bound computation like ILUPACK, and a quadratic/cubic relation between energy and voltage-frequency. However, the analysis of the time-power-energy trade-off when the cores operate in a certain P-states, with the energy-aware

version of the runtime, reveals the high impact of idle and, to a minor degree, uncore power which clearly favor shorter execution time over lower power dissipation rates. This is also contrasted to and accurately captured by our power model.

CHAPTER 7

Conclusions

7.1 Conclusions and Main Contributions

The main goal of the dissertation was the *design, development and evaluation of strategies and techniques to improve the energy consumption of existing linear algebra libraries.*

At the conclusion of this work, the main contributions of this dissertation are the following:

- A review of the state-of-art techniques, methods and algorithms related to energy efficiency that can be applied on multicore and hybrid platforms.
- The development of an integrated framework for power-performance analysis of applications on multicore and hybrid platforms, consisting of profiling/tracing tools in combination with power measurement devices.
- The introduction of a power model in applications that exploit parallelism at the task-level based on a series of consumption parameters that are directly related to the architecture and the modeled application.
- The analysis of power-aware techniques from the theoretical point of view and their simulation via linear algebra codes on a variety of hardware configurations to assess their impact on performance and energy consumption.
- The integration and validation of the energy-aware techniques into a set of dense and sparse linear algebra libraries and runtimes.

A main contribution of this dissertation is the integrated power-performance analysis framework. Due to the lack of environments that provide both energy/power and performance metrics, we developed a full framework which has been the base for all the experiments performed in this dissertation. This suite is the combination of an already existing performance profiling/tracing tool with a new library to interact with a series of internal and external devices. This allowed us to account energy usage and relate these data with performance parameters for the execution of state-of-the-art linear algebra codes.

An additional contribution is the development of models in order to predict the power/energy usage of parallel applications. Our contribution is different from already existing models in that it

does not rely on the hardware counters. Starting from estimations of the theoretical costs and real execution times involved in the tasks of the linear algebra codes, our memory-contention approach predicts the energy consumption with very high resolution.

Thirdly, we propose and evaluate two key energy-saving techniques. This theoretical study determined the best approach for current architectures, the Race-to-Idle Algorithm, which has been incorporated into two state-of-the-art linear algebra runtimes: SuperMatrix as part of the `libflame` library for dense linear algebra; and the ad-hoc runtime for ILUPACK, that implements a complete preconditioner+iterative solver for sparse linear algebra. As a major contribution, we provide two new energy-aware versions of these runtimes that yield, in average, savings between 7–9 % without a negligible impact on the performance. We also demonstrate that our model can be easily incorporated into these runtimes to provide better scheduling of the tasks on multithreaded architectures.

The following sections further detail those specific contributions and summarize the corresponding conclusions.

7.1.1 Power-performance profiling/tracing tools

A full environment for power-performance analysis of scientific applications in the HPC field was developed and evaluated using several dense and sparse linear algebra algorithms. The framework offers useful information on power and performance for different kinds of parallel workloads, from MPI codes that operate on moderate-scale clusters, to multithreaded applications that exploit the benefits of multicore+GPU platforms. In addition, we also developed a complementary package, PMLib, that allows to gather power data and eases the interaction with the power measurement devices attached to the target platforms. Its integration into the framework contributes to the detailed analysis of parallel applications.

We also presented new modules for the PMLib library, in particular, a new appliance which collects informations on processors energy states, like the P-/C-states.

To demonstrate the capabilities of our framework we provided a detailed power and performance analysis of the LU factorization on a platform equipped with multicore technology. Different implementations of this algorithm have been analyzed, in particular the LAPACK and MKL implementations of the LU factorization with partial pivoting, and the SMPSs task-parallel implementation which computes the same factorization with incremental pivoting.

The conclusion from this study is that the routines that present less idle time, and thus perform a better usage of the hardware resources, significantly reduce the execution time and energy consumption. In other words, by keeping the cores busy most of the time, compute-intensive algorithms that apply a "race-to-idle" strategy provide always lower execution times and energy consumption. Given the high energy cost of keeping the machine active when there is no workload to run, the "race-to-idle" approach clearly pays off for such algorithms.

7.1.2 Power and energy models

An accurate model to estimate the energy consumption for task-parallel implementations of dense matrix factorizations was provided as a key contribution of this dissertation. With this result, we demonstrate that it is possible to systematically model the power and energy consumed by these kind of operations in multicore architectures. Modeling the power dissipated by a high performance parallel implementation of this operation on a multicore platform is an initial step towards the more ambitious goal of modeling power for a large collection of message-passing numerical codes on large-scale clusters equipped with multicore processors.

In order to deliver accurate estimations, we accommodate memory contention in the prediction of dynamic power dissipation, based on the execution time variability during the actual the global execution of the algorithm.

After a careful experimental validation using a fine grain power measurement device, we confirmed the reliability of the model using a variety of task-parallel dense linear algebra algorithms: the Cholesky, LU and the QR factorizations. Our total energy estimations are, in average, within 5 % of the real values, and are thus comparable with those obtained from more elaborated models that rely on hardware counters [35].

Two properties of the proposed model are portability and generality, since it does not require access to low-level, platform-dependent hardware counters. Our approach can be applied to model power/energy consumption of integer- or floating-point intensive task-parallel applications, not necessarily related to dense linear algebra, when the following three conditions are met: *i)* there exist fairly good estimations of the theoretical cost of each type of task; *ii)* the target platform is instrumented with some sort of power measurement device; and *iii)* there exist accurate measures of task execution time of the tasks.

In addition to provide accurate estimations of energy consumption, our model aims at guiding and helping the operation of a runtime to unleash a more energy efficient execution of task-parallel algorithms. For instance, in situations where there exist different mappings of tasks to computational resources that may result in equivalent or nearly equivalent performance on multithreaded platforms, our model can be used by a runtime to automatically predict the energy of different hardware configurations and select the most efficient one from this point of view.

7.1.3 Energy-aware techniques

A contribution of this dissertation consisted in the analysis of two key energy-aware techniques that handle DVFS and aim at reducing energy consumption in linear algebra operations. The *Slack Reduction Algorithm* (SRA) is designed and implemented to exploit *slacks* (idle periods) existing in the DAG that represents a dense linear algebra operation by carefully tuning frequency execution of certain tasks. We also study an alternative approach, the *Race-to-Idle Algorithm* (RIA), which pursues the power-conservation goal but from a totally opposite approach; specifically, this strategy generates inactive times during the execution of the DAG by executing all tasks at the highest frequency, and relies on the power savings attained via a reduction of the operating frequency during idle periods.

To validate the theoretical energy gains that can be attained with SRA and RIA, we developed a simulator to emulate different types of architectures. The simulator calculates a schedule of the tasks for both strategies, taking into account practical constraints like actual number of resources (processor cores), the cost of varying processor frequency, the discrete range of frequencies, the granularity of DVFS operation (core- or socket-level), etc.

Thanks to this simulator, we provide a complete energy performance analysis of both strategies using current blocked dense linear algebra algorithms for the LU and QR factorizations. The results demonstrate that, under realistic environments and certain conditions, a reduction in energy consumption is possible. In general, superior performance of the RIA policy over the SRA one from the point of view of execution time for problems of large dimension is observed. However, SRA, which is a more elaborate strategy than RIA, can potentially deliver higher energy savings than RIA for small problems. Finally, we demonstrate the impact on the energy savings of hardware that operates with DVFS at the core level (instead of the socket level).

7.1.4 Dense linear algebra

A contribution was the design and implementation of energy-aware techniques that, integrated into the SuperMatrix runtime, report consistent reductions of the total energy consumption. Additionally, we leverage DVFS to further enhance the energy-efficient execution of the Cholesky factorization and LU decomposition with partial pivoting, where idle threads are set into a blocking state and the corresponding cores are promoted into a low-power mode, without compromising the computational performance of the execution.

Experimental results with state-of-the-art multicore and multithreaded platforms demonstrate that, by avoiding active polling (busy-waits) in the runtime, it is possible to leverage periods where the general-purpose cores are performing no useful work, with a minor impact on the time-to-solution. One key observation is that, because of the clear separation between the linear algebra codes in `libflame` and the SuperMatrix runtime in charge of their task-parallel execution, the proposed energy-aware techniques can be automatically applied to all routines of `libflame`. We expect that the same energy-aware techniques are valid for other runtimes, in particular those of the StarSs [118], PLASMA [5] and MAGMA [5] projects, yielding similar benefits.

During the experimental evaluation of certain dense matrix decompositions on heterogeneous CPU–GPU platforms, it was observed that the panel factorization that lies in the critical path of the algorithm is a major obstacle to attain high performance. To deal with this problem, we leverage task priorities of a new version of the runtime, that advances the computation of critical operations. The experimental results show a significant reduction of idle time in this type of platforms, thus reducing the energy consumption. Furthermore, in this particular setup, we can leverage the power model to pass valuable information to the scheduler, so that at run time it can make decisions on the best configuration from the energy perspective.

Finally, we analyze the actual impact of the performance and energy-saving enhancements using the improved version of the SuperMatrix runtime that advances critical tasks using two key dense linear algebra operations, namely the LU factorization with partial pivoting and the Cholesky decomposition, that are representative of many other dense BLAS-3-based matrix operations.

7.1.5 Sparse linear algebra

Two main contributions in the domain of sparse linear algebra were: *i)* the implementation of energy-aware strategies for the complete preconditioner+iterative solve process in ILUPACK; and *ii)* the characterization of the power consumption in the complete solution of symmetric positive definite sparse linear systems.

A general conclusion from this study is that, for a mildly memory-bound operation, the reduction of power attained by lowering the voltage/frequency does not necessarily result in energy savings due to the increase of execution time. The computation of the complete preconditioner+iterative solve process in ILUPACK is just an example where a reduction of voltage/frequency renders an increase in energy consumption. This is partly due to the large fraction of power dissipation that corresponds to the system and static components, which do not benefit or do little benefit from a reduction of the frequency.

Therefore, any effort at reducing the energy consumption of these computations must carefully leverage the processor performance (or P-) states so as to avoid increasing the execution time. Fortunately, in the case of our parallel code, the operation is already divided into well-defined tasks, which allows us to avoid busy-waits and exploit the presence of inactive periods by promoting cores executing vacant threads into a power-friendly state.

A contribution is the development of an energy-aware runtime. The experimental observations demonstrate that the original runtime for ILUPACK is energy-oblivious in that idle threads, with no useful task to execute, simply poll till new work is available. As a consequence, these threads dissipate a significant amount of power in current processors, for no practical performance benefit in the particular case of ILUPACK. The introduction of energy-aware techniques replace this behavior with a power-friendly implementation, which blocks idle threads till new work is available. These techniques require a careful reorganization of the underlying runtime, in order to avoid deadlocks and ensure a rapid response that does not impair performance. A complete study reveals savings in the energy usage between 7 and 13 %, at practically no cost from the performance point of view, which are clearly connected to the impact of the C-states by our power model.

In this study we demonstrate that, for a presumedly memory-bound computation like ILUPACK, the power savings yielded by the higher P-states are mostly blurred by the increase of execution time. However, the analysis of the time-power-energy trade-off when the cores operate in a certain P-state, with the energy-aware version of the runtime, reveals the high impact of idle and, to a minor degree, uncore power, which clearly favors shorter execution time over lower power dissipation rates. This is also contrasted to and accurately captured by our power model.

7.2 Related Publications

The scientific contributions of this thesis have been validated with several peer-reviewed publications in national and international conferences, as well as international journals. Each one of these contributions is supported by, at least, one international publication.

The following sections list the main publications derived from the thesis. We divide them into papers directly and indirectly related to the thesis topics, and unrelated topics with a certain connection to energy efficiency. For the first group of publications, we provide a brief abstract of the main contents of the paper. Only international conferences and journals are listed.

7.2.1 Directly related publications

Chapter 3. Performance and Energy Measurement Framework

The first approach towards the development of a performance and power measurement framework was introduced in [10]. The framework presented in this paper includes support for wattmeters to measure internal DC power consumption in combination with the `Extrae` and `Paraver` instrumentation/visualization tools. In [27] we extended this framework with support for new measurement devices, including an internal device to monitor DC power consumption at a high frequency. We also defined the interface of PMLib and included a module to record information on processor states related to power consumption. To demonstrate the use of our framework in [31, 30] we analyzed different dense linear algebra algorithm implementations from the performance and power consumption point of views.

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 [10]

> 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 multithreaded 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.

Conference Proceedings [27]

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.

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.

Conference Proceedings [31]

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.

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.

Conference Proceedings [30]

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.

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.

Chapter 4. Modeling Power and Energy Consumption

In [18], we introduced the first results to model the energy consumption of the Cholesky factorization for multicore processors. We assumed a task-parallel execution of the factorization process, with concurrency exploited via a run-time. Experimental results validated the precision of the model and reported estimations of the power and energy dissipation of the global algorithm. In [20] we introduced a contention-aware model that accommodates for the variability of power consumption due to memory contention. The model showed the reliability for the Cholesky, LU and QR factorizations.

The following is a detailed list of the main publications related to this topic:

Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development* (2012), pp. 1–8. Journal [18]

We introduce a model for the total energy consumption of the Cholesky factorization on a multicore processor. Our model assumes a task-parallel execution of the factorization process, with concurrency leveraged via a run-time as those recently proposed in projects like SMPSs, PLASMA or `libflame`, and decomposes the power usage into its system, static and dynamic components. A few simple experiments provide experimental data (parameters) with enough accuracy to assemble the model, which can then be used to estimate the actual power dissipation and energy consumption of the global algorithm. Experimental results on an 8-core platform equipped with Intel Xeon processors reveal the precision of the model.

Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Modeling power and energy consumption of dense matrix factorizations on multicore processors. *Concurrency and Computation: Practice and Experience* (2013). To appear. Journal [20]

We propose a model for the energy consumption of the concurrent execution of three key dense matrix factorizations, with task-parallelism leveraged via the SMPSs runtime, on a multicore processor. Our model decomposes the power dissipation into the system, static and dynamic components, with the former two being estimated from basic, off-line experiments. The dynamic power, on the other hand, requires significantly more care, and we introduce a contention-aware model that accommodates for the variability of power consumption due to memory contention. Experimental results on an Intel Xeon E5504 processor with four cores, using an internal wattmeter that samples the power drawn by the mainboard with a frequency of 1 kHz, show the reliability of the energy model for the Cholesky, LU and QR factorizations on this platform.

Chapter 5. Theoretical Analysis of Slack Reduction and Race-to-idle

The work in [17] was the first contribution in which we introduced our Slack Reduction Algorithm to optimize the execution frequency of a collection of tasks (in which many dense linear algebra algorithms can be decomposed) on multicore architectures. In [12, 19] we analyzed the

impact on power consumption of two DVFS-control strategies: the Slack Reduction Algorithm (SRA) and the Race-to-Idle Algorithm (RIA) from the theoretical and experimental point of views. A power-aware simulator, in charge of scheduling the execution of tasks to processor cores, and the real execution of the algorithms was employed to evaluate the performance benefits of these power-control policies for different dense linear algebra algorithms.

The following is a detailed list of the main publications related to this topic:

Conference Proceedings [17]

Alonso, P., Dolz, M., Mayo, R., and Quintana-Ortí, E. S. Improving power efficiency of dense linear algebra algorithms on multicore processors via slack control. In *International Conference on High Performance Computing and Simulation (HPCS)* (2011), pp. 463–470.

> We address the efficient exploitation of task-level parallelism, present in many dense linear algebra operations, from the point of view of both computational performance and energy consumption. In particular, we consider a procedure, the Slack Reduction Algorithm (SRA), to optimize the execution frequency of a collection of tasks (in which many dense linear algebra algorithms can be decomposed) on multicore architectures. The results from this procedure are modulated by an energy-aware simulator, which is in charge of scheduling/mapping the execution of these tasks to the cores, leveraging dynamic frequency voltage scaling featured by current technology. Simultaneously, the simulator evaluates the performance benefits of the solution. Experiments with these tools show significant energy gains for two key dense linear algebra operations: the Cholesky and QR factorizations.

Journal [12]

Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. DVFS-control techniques for dense linear algebra operations on multicore processors. *Computer Science - Research and Development* (2011), pp. 1–10.

> This paper analyzes the impact on power consumption of two DVFS-control strategies when applied to the execution of dense linear algebra operations on multicore processors. The strategies considered here, prototyped as the Slack Reduction Algorithm (SRA) and the Race-to-Idle Algorithm (RIA), adjust the operating frequency of the cores during execution of a collection of tasks (in which many dense linear algebra algorithms can be decomposed) with a very different approach to save energy. A power-aware simulator, in charge of scheduling the execution of tasks to processor cores, is employed to evaluate the performance benefits of these power-control policies for two reference algorithms for the LU factorization, a key operation for the solution of linear systems of equations.

Journal [19]

Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Energy-efficient execution of dense linear algebra algorithms on multicore processors. *Cluster Computing 16*, 3 (2013), pp. 497–509.

> We address the efficient exploitation of task-level parallelism, present in many dense linear algebra operations, from the point of view of both computational performance and energy consumption. The strategies considered here, referred to as the Slack Reduction Algorithm (SRA) and the Race-to-Idle Algorithm (RIA), adjust the operating frequency of the cores during the execution of a collection of tasks (in which many dense linear algebra algorithms can be decomposed) with very different approaches to save energy. The procedures are evaluated using an energy-aware simulator, which is in charge of scheduling/mapping the execution of these tasks to the cores, leveraging dynamic frequency voltage scaling featured by current technology. Experiments with these tools show significant energy gains for two versions of the QR factorization.

Chapter 6. Energy-Aware Techniques for Dense and Sparse Linear Algebra

In [11], we introduced our research on the application of power-control techniques to the execution of dense linear algebra operations on modern multicore processors and hybrid CPU–GPU architectures, based on the SuperMatrix runtime system. In [14, 13] we analyzed the data-parallel execution of different dense linear algebra factorizations in multicore and multi-GPU platforms using different energy-aware versions of the SuperMatrix runtime. These techniques leverage DVFS by activating/blocking idle threads and enable the transition to a more energy-friendly state. Furthermore, in [16] we extended the runtime scheduler by accommodating hybrid CPU–GPU executions and managing task priorities for dense linear algebra operations in combination with the aforementioned energy-saving techniques. In [15] we also proposed a power consumption model that can be leveraged by runtime to make decisions on energy considerations. In [9], we analyzed the energy performance of a task-parallel computation of an ILU-based preconditioner for the solution of sparse linear systems on multicore processors. In this work we introduced two energy-saving mechanisms incorporated into the runtime. We also employed a theoretical model that allows to explore the effect of the power states. We extended this work in [6] to use the energy-saving techniques not only in the preconditioner but also in the resolution phase of the iterative solver. The results were evaluated on different architectures to explore the impact of the P-/C-states.

The following is a detailed list of the main publications related to this topic:

Alonso, P., Dolz, M. F., Igual, F. D., Marker, B., Mayo, R., Quintana-Ortí, E. S., and van de Geijn, R. Power-aware dense linear algebra implementations on multicore and many-core processors. In *MARC Symposium* (2011), KIT Scientific Publishing, Karlsruhe, pp. 103–106. Conference Proceedings [11]

> This paper outlines our research on the application of power-control techniques to the execution of dense linear algebra operations on modern multicore processors and hybrid CPU–GPU architectures. The framework is based on the SuperMatrix runtime system which exploits the inherent task-parallelism present in most blocked dense linear algebra algorithms. As part of the on-going work, we analyze the possibility of extending the power-aware techniques to novel many-core architectures, such as the Intel SCC processor.

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 multicore processors. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2012), pp. 353–358. Conference Proceedings [14]

> We analyze the trade-off between energy and performance for a data-parallel execution of the LU factorization with partial pivoting on a multicore processor. To improve energy efficiency, we adapt the runtime in charge of controlling the concurrent execution of the algorithm so as to leverage DVFS by activating/blocking idle threads. For a CPU-bound operation like the LU factorization, experiments on an AMD 8-core processor report an average reduction around 5 % in energy consumption in exchange for a minor, in some cases negligible, increase in the execution time.

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. Conference Proceedings [13]

We investigate the balance between the time-to-solution and the energy consumption of a task-parallel execution of the Cholesky and LU factorizations on a hybrid platform, equipped with a multicore processor and several GPUs. To improve energy efficiency, we incorporate two energy-saving techniques in the runtime in charge of scheduling the computations, to block idle threads and enable the transition to a more energy-friendly state of the general-purpose cores. Experiments on an Intel Xeon-based platform connected to an NVIDIA Tesla server report an average reduction of the energy consumption close to 9 % (38 % when only the consumption associated with the application is considered), for a minor increase in the execution time of the algorithm.

Conference Proceedings [16]

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 (EE-LSDS)*, Lecture Notes in Computer Science, Vol. 8046. Springer-Verlag (2013), pp. 153–167.

We enhance the SuperMatrix runtime scheduler from the `libflame` library for dense linear algebra in two different directions that address high performance and energy. First, we extend the runtime scheduler by accommodating hybrid CPU–GPU executions and managing task priorities for dense linear algebra operations, with remarkable performance improvements. Second, we introduce techniques to reduce energy consumption during idle times inherent to parallel executions, attaining fair energy savings. While our techniques are applicable to the complete `libflame` library, in this paper we use the LU factorization with partial pivoting to illustrate the actual impact on performance and energy consumption of the adopted techniques.

Journal [15]

Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Enhancing performance and energy consumption of runtime schedulers for dense linear algebra. *Concurrency and Computation: Practice and Experience* (2013). In revision.

The road towards Exascale Computing requires a holistic effort to address three different challenges simultaneously: high performance, energy efficiency, and programmability. The use of runtime task schedulers to orchestrate parallel executions with minimal developer intervention has been introduced in the past years to tackle the programmability issue while maintaining, or even improving, performance. We enhance the SuperMatrix runtime task scheduler integrated in the `libflame` library in two different directions that address high performance and energy efficiency. First, we extend the runtime by accommodating hybrid parallel executions and managing task priorities for dense linear algebra operations, with remarkable performance improvements. Second, we introduce techniques to reduce energy consumption during idle times inherent to parallel executions, attaining important energy savings. In addition, we propose a power consumption model that can be leveraged by runtime task schedulers to make decisions based not only on performance, but also on energy considerations.

Conference Proceedings [9]

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*, Lecture Notes in Computer Science, Vol. 7453. Springer-Verlag, (2012), pp. 55–63.

We analyze the energy-performance balance of a task-parallel computation of an ILU-based preconditioner for the solution of sparse linear systems on multicore processors.

In particular, we elaborate a theoretical model for the power dissipation, and employ it to explore the effect of the processor power states on the time-power-energy interaction for this calculation. Armed with the insights gained from this study, we then introduce two energy-saving mechanisms which, incorporated into the runtime in charge of the parallel execution of the algorithm, improve energy efficiency by 6.9 %, with a negligible impact on performance.

Aliaga, J. I., Barreda, M., Dolz, M. 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. *Concurrency and Computation: Practice and Experience* (2013). In revision. Journal [6]

For the solution of sparse linear systems, we investigate the benefits that an energy-aware implementation of the runtime in charge of the concurrent execution of ILUPACK, an iterative solver with a sophisticated and effective preconditioner, 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.

7.2.2 Indirectly related publications

A parallel research was performed into the energy-efficient to explore the capabilities of power measurement devices and new modules for the PMLib library. In [16, 45] we explored first some power monitoring approaches for energy and power analysis of computers. In [29] we presented an extension of our PMLib framework for power-performance analysis that permits a rapid and automatic detection of power sinks during the execution of concurrent scientific workloads.

The following is a detailed list of the main publications related to that topic:

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. 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 (EE-LSDS)*, Lecture Notes in Computer Science, Vol. 8046. Springer-Verlag, 2013, pp. 3–18. Conference Proceedings [16]

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and S. Quintana-Ortí, E. S. Assessing power monitoring approaches for energy and power analysis of computers. *Journal of Sustainable Computing* (2013). In revision. Journal [45]

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), pp. 1–9. Journal [29]

7.2.3 Other publications

As an orthogonal research line, several publications in this field were obtained during the development of this thesis. These publications are focused on an energy-saving module for HPC clusters that implement energy-aware policies which, taking into account past and future users' request, allows to switch on and shutdown the system nodes. This software was presented in [49] and evaluated in [48]. In [46, 47] we extended this work by presenting a simulator for this tool that allows the evaluation and analysis of the benefits under realistic workloads.

The following is a detailed list of the main publications related to that topic:

CONFERENCE PROCEEDINGS [49] DOLZ, M. F., FERNÁNDEZ, J. C., MAYO, R., AND QUINTANA-ORTÍ, E. S. EnergySaving Cluster Roll: Power saving system for clusters. In *Architecture of Computing Systems - ARCS 2010*, Lecture Notes in Computer Science, Vol. 5974. Springer Berlin Heidelberg, (2010), pp. 162–173.

CONFERENCE PROCEEDINGS [48] DOLZ, M. F., FERNÁNDEZ, J. C., ISERTE, S., MAYO, R., QUINTANA-ORTÍ, E. S., COTALLO, M. E. AND DÍAZ, G. EnergySaving Cluster experience in CETA-CIEMAT. In *5th Iberian Grid Infrastructure Conference (IBERGRID)*, (2011), pp. 39–50.

CONFERENCE PROCEEDINGS [46] DOLZ, M. F., FERNÁNDEZ, J. C., ISERTE, S., MAYO, R., AND QUINTANA-ORTÍ, E. S. A flexible simulator to evaluate a power saving system for HPC clusters. In *2nd International Workshop on Green Computing Middleware (GCM)*, (2011), pp. 2:1–2:6.

JOURNAL [47] DOLZ, M. F., FERNÁNDEZ, J. C., ISERTE, S., MAYO, R., AND QUINTANA-ORTÍ, E. S. A simulator to assess energy saving strategies and policies in HPC workloads. *SIGOPS Operating Systems Review 46*, 2 (2012), pp. 2–9.

In a different research line we worked on developing efficient parallel algorithms for Toeplitz matrices that produced the following contribution:

JOURNAL [21] ALONSO, P., DOLZ, M. F., AND VIDAL, A. M. Block pivoting implementation of a symmetric Toeplitz solver. *Journal of Parallel and Distributed Computing*, (2014), To appear.

7.3 Open Research Lines

Energy-efficiency is a relatively novel discipline in computer science, and thus many research questions remain open after the conclusion of this thesis. The following list details some of the research lines related to this thesis that deserve further investigation:

- Integration of energy-aware techniques into runtimes that exploit task-level parallelism and/or linear alebra libraries, such as SMPSs [118], OmpSs [102], PLASMA [5] and MAGMA [5].
- Development of new energy-aware mapping techniques and scheduling heuristics, integrated into a number of practical runtimes, that can improve the parallel performance as well as reduce the energy consumption of current as well as future numerical HPC libraries, for a wide variety of architectures.
- New scheduling heuristics with tasks dependencies in environments with limited number of resources.
- Redesign of the Slack Reduction Algorithm to incorporate a dynamic policy, which operates at run-time, dynamically adapting to variations on the conditions.

Bibliography

[1] AMD PowerNow! Technology. Available at: `http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx`.

[2] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, March 2004. White Paper. Available at: `http://download.intel.com/design/network/papers/30117401.pdf`.

[3] The Green500 list, 2013. Available at: `http://www.green500.org`.

[4] The Top500 list, 2013. Available at:`http://www.top500.org`.

[5] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series 180*, 1 (2009), 12–37.

[6] Aliaga, J. I., Barreda, M., Dolz, M. 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. *Concurrency and Computation: Practice and Experience* (2013). In revision.

[7] 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.

[8] 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*, vol. 7133 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 162–172.

[9] 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.

[10] 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.

[11] Alonso, P., Dolz, M. F., Igual, F. D., Marker, B., Mayo, R., Quintana-Ortí, E. S., and van de Geijn, R. Power-aware dense linear algebra implementations on multi-core and many-core processors. In *MARC Symposium* (2011), KIT Scientific Publishing, Karlsruhe, pp. 103–106.

[12] Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. DVFS-control techniques for dense linear algebra operations on multi-core processors. *Computer Science - Research and Development* (2011), 1–10.

[13] 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.

[14] 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.

[15] Alonso, P., Dolz, M. F., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Enhancing performance and energy consumption of runtime schedulers for dense linear algebra. *Concurrency and Computation: Practice and Experience* (2013). In revision.

[16] 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.

[17] Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *International Conference on High Performance Computing and Simulation (HPCS)* (2011), pp. 463–470.

[18] Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development* (2012), 1–8.

[19] Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Energy-efficient execution of dense linear algebra algorithms on multi-core processors. *Cluster Computing 16*, 3 (2013), 497–509.

[20] Alonso, P., Dolz, M. F., Mayo, R., and Quintana-Ortí, E. S. Modeling power and energy consumption of dense matrix factorizations on multicore processors. *Concurrency and Computation: Practice and Experience* (2013), To appear.

[21] Alonso, P., Dolz, M. F., and Vidal, A. M. Block pivoting implementation of a symmetric Toeplitz solver. *Journal of Parallel and Distributed Computing* (2014). To appear.

[22] AMD Core Math Library (ACML). `http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/`.

[23] AnandTech Forums. Power-consumption scaling with clockspeed and Vcc for the i7-2600K. http://forums.anandtech.com/showthread.php?t=2195927, 2011.

[24] Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[25] Ashby., S., et al. The opportunities and challenges of exascale computing. Tech. rep., U.S. Department of Energy, Advanced Scientific Computing Advisory Committee, 2010.

[26] Badia, R. M., Herrero, J. R., Labarta, J., Pérez, J. M., Quintana-Ortí, E. S., and Quintana-Ortí, G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience 21* (2009), 2438–2456.

[27] 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.

[28] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, G. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2457–2477.

[29] Barreda, M., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Quintana-Ortí, E. S. Automatic detection of power bottlenecks in parallel scientific applications. *Computer Science - Research and Development* (2013), 1–9.

[30] 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.

[31] 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.

[32] Baskiyar, S., and Palli, K. Low power scheduling of DAGs to minimize finish times. In *High Performance Computing*, Y. Robert, M. Parashar, R. Badrinath, and V. Prasanna, Eds., vol. 4297 of *Lecture Notes in Computer Science*. Springer, 2006, pp. 353–362.

[33] Bedard, D., Porterfield, A., Fowler, R., and Lim, M. Y. PowerMon 2: Fine-grained, integrated power measurement. Tech. Rep. TR-09-04, RENCI, North Carolina, 2009.

[34] Bekas, C., and Curioni, A. A new energy aware performance metric. *Computer Science - Research and Development 25* (2010), 187–195.

[35] Bertran, R., Tallada, M. G., Martorell, X., Navarro, N., and Ayguade, E. A systematic methodology to generate decomposable and responsive power models for CMPs. *IEEE Transactions on Computers 62*, 7 (2013), 1289–1302.

[36] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software 31*, 1 (2005), 1–26.

[37] Bircher, W. L., and John, L. K. Complete system power estimation: A trickle-down approach based on performance events. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2007), 158–168.

[38] Borkar, S., and Chien, A. A. The future of microprocessors. *Communications of the ACM 54* (May 2011), 67–77.

[39] Brewer, O., Dongarra, J., and Sorensen, D. Tools to aid in the analysis of memory access patterns for FORTRAN programs. LAPACK Working Note 6, Technical Report MCS-TM-120, Argonne National Laboratory, June 1988.

[40] Buttari, A., Langou, J., Kurzak, J., , and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35*, 1 (2009), 38–53.

[41] Castillo, M., Dolz, M., Fernández, J. C., Mayo, R., Quintana-Ortí, E. S., and Roca, V. Evaluation of the energy performance of dense linear algebra kernels on multi-core and many-core processors. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)* (2011), pp. 846–853.

[42] Chan, E., van de Geijn, R., and Chapman, A. Managing the complexity of lookahead for LU factorization with pivoting. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2010), ACM, pp. 200–208.

[43] Cilk project. `http://supertech.csail.mit.edu/cilk/`.

[44] Demmel, J., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., and Sorensen, D. Prospectus for the development of a linear algebra library for high-performance computers. Tech. Rep. MCS-TM-97, Argonne National Laboratory, Sept. 1987.

[45] Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R., and Quintana-Ortí, E. S. Assessing power monitoring approaches for energy and power analysis of computers. *Sustainable Computing* (2013). In revision.

[46] Dolz, M. F., Fernández, J. C., Iserte, S., Mayo, R., and Quintana-Ortí, E. S. A flexible simulator to evaluate a power saving system for HPC clusters. In *2nd International Workshop on Green Computing Middleware (GCM)* (2011), GCM '11, ACM, pp. 2:1–2:6.

[47] Dolz, M. F., Fernández, J. C., Iserte, S., Mayo, R., and Quintana-Ortí, E. S. A simulator to assess energy saving strategies and policies in HPC workloads. *SIGOPS Operating Systems Review 46*, 2 (2012), 2–9.

[48] Dolz, M. F., Fernández, J. C., Iserte, S., Mayo, R., Quintana-Ortí, E. S., Cotallo, M. E., and Díaz, G. EnergySaving Cluster experience in CETA-CIEMAT. In *5th Iberian Grid Infrastructure Conference (IBERGRID)* (2011), pp. 39–50.

[49] Dolz, M. F., Fernández, J. C., Mayo, R., and Quintana-Ortí, E. S. EnergySaving Cluster Roll: Power saving system for clusters. In *23rd International Conference on Architecture of Computing Systems (ARCS)* (2010), pp. 162–173.

[50] Dongarra, J., et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications 25*, 1 (2011), 3–60.

[51] Dongarra, J., and Walker, D. LAPACK working note 58: The design of linear algebra libraries for high performance computers. Tech. rep., University of Tennessee, Knoxville, TN, USA, 1993.

[52] Dongarra, J. J., Bunch, J. R., Moler, C. B., and Stewart, G. W. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[53] Du Croz, J., Mayes, P., and Radicati, G. Factorization of band matrices using level 3 BLAS. LAPACK Working Note 21, Technical Report CS-90-109, University of Tennessee, July 1990.

[54] Duranton, M. *et al.* The HiPEAC vision for advanced computing in horizon 2020, 2013.

[55] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *38th annual International Symposium on Computer Architecture (ISCA)* (2011), pp. 365–376.

[56] Etinski, M., Corbalán, J., Labarta, J., and Valero, M. Utilization driven power-aware parallel job scheduling. *Computer Science - Research and Development 25*, 3-4 (2010), 207–216.

[57] Extrae: User guide manual for version 2.4.1. `http://www.bsc.es/computer-sciences/extrae`.

[58] Feng, W.-c., Feng, X., and Ge, R. Green supercomputing comes of age. *IT Professional 10*, 1 (jan.-feb. 2008), 17 –23.

[59] FLAME project home page. `http://www.cs.utexas.edu/users/flame/`.

[60] Freeh *et al.*, V. W. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Transactopms Parallel Distributed Systems 18* (2007), 835–848.

[61] Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K. W. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems 99* (2009), 658–671.

[62] Goel, B. Per-core power estimation and power aware scheduling strategies for CMPs. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, 2011.

[63] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, 1996.

[64] Goto, K. `http://www.tacc.utexas.edu/resources/software`.

[65] 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.

[66] Gruian, F., and Kuchcinski, K. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *Asia and South Pacific Design Automation Conference (ASP-DAC)* (2001), ACM, pp. 449–455.

[67] Gunter, B. C., and van de Geijn, R. A. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software 31*, 1 (2005), 60–78.

[68] Gunther, S., Deval, A., and Burton, T. Energy-efficient computing: Power-management system on the Intel Nehalem family of processors. *Intel Technology Journal 15*, 1 (2011).

[69] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Pub., San Francisco, 2012.

[70] Henry, G. BLAS based on block data structures. Tech. rep., Cornell University, Ithaca, NY, USA, 1992.

[71] Hong, I., Kirovski, D., Qu, G., Potkonjak, M., and Srivastava, M. B. Power optimization of variable-voltage core-based systems. *IEEE Transactions Computer-Aided Design 18* (1999), 1702–1714.

[72] HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., and Toshiba Corp. Advanced configuration and power interface specification, revision 5.0, 2011.

[73] IBM. Engineering Scientific Subroutine Library.

[74] Igual, F. D., Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., van de Geijn, R. A., and Zee, F. G. V. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing 72*, 9 (2012), 1134–1143.

[75] Igual, F. D., Quintana-Ortí, G., and van de Geijn, R. Scheduling algorithms-by-blocks on small clusters. *Concurrency and Computation: Practice and Experience* (2012). Available online.

[76] ILUPACK project home page. `http://ilupack.tu-bs.de`.

[77] Intel. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2009.

[78] Intel Corporation. Intel Core Microarchitecture (Nehalem) based processors incorporate a new feature: Intel Turbo Boost technology, November 2008.

[79] Intel Math Kernel Library (MKL). `http://software.intel.com/en-us/intel-mkl/`.

[80] Isci, C., and Martonosi, M. Runtime power monitoring in high-end processors: Methodology and empirical data. In *36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), MICRO 36, IEEE Computer Society, p. 93.

[81] Kimura, H., Sato, M., Hotta, Y., Boku, T., and Takahashi, D. Empirical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster. In *IEEE International Conference on Cluster Computing* (2007), pp. 1–10.

[82] King, D., Ahmad, I., and Sheikh, H. F. Stretch and compress based re-scheduling techniques for minimizing the execution times of DAGs on multi-core processors under energy constraints. In *International Conference on Green Computing (ICGC)* (2010), pp. 49–60.

[83] Kunkel, J. HDTrace - a tracing and simulation environment of application and system interaction. Tech. Rep. 2, Department of Informatics, Scientific Computing. Universität Hamburg, 2011.

[84] LAPACK project home page. `http://www.netlib.org/lapack`.

[85] Laros III, J., Pedretti, K., Kelly, S., Shu, W., Ferreira, K., Vandyke, J., and Vaughan, C. Energy delay product. In *Energy-Efficient High Performance Computing*, Springer Briefs in Computer Science. Springer London, 2013, pp. 51–55.

[86] Le Sueur, E., and Heiser, G. Dynamic voltage and frequency scaling: the laws of diminishing returns. In *International Conference on Power-Aware Computing and Systems (HotPower)* (2010), pp. 1–8.

[87] Lee, B. C., and Brooks, D. M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGPLAN Not. 41*, 11 (2006), 185–194.

[88] Lee, Y. C., and Zomaya, A. Y. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)* (2009), IEEE Computer Society, pp. 92–99.

[89] Li, R., and Huang, H. C. List scheduling for jobs with arbitrary release times and similar lengths. *Journal of Scheduling 10*, 6 (2007), 365–373.

[90] 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.

[91] Manzak, A., and Chakrabarti, C. Variable voltage task scheduling for minimizing energy or minimizing power. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (2000), IEEE Computer Society, pp. 3239–3242.

[92] Martin, S. M., Flautner, K., Mudge, T., and Blaauw, D. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2002), ACM, pp. 721–725.

[93] Mtibaa, A., Ouni, B., and Abid, M. An efficient list scheduling algorithm for time placement problem. *Computers & Electrical Engineering 33*, 4 (2007), 285–298.

[94] 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.

[95] Müller, M. S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., and Nagel, W. E. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *PARCO* (2007), vol. 15 of *Advances in Parallel Computing*, IOS Press, pp. 637–644.

[96] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[97] Netlib.org. http://www.netlib.org/blas.

[98] Netlib.org. http://www.netlib.org/eispack.

[99] Netlib.org. http://www.netlib.org/lapack.

[100] NVIDIA. *NVML Reference Manual*, 2013.

[101] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 ed., August 2009.

[102] OmpSs project home page. http://pm.bsc.es/ompss/.

[103] Project home page for OpenCL - the open standard for parallel programming of heterogeneous systems. project home page. http://www.khronos.org/opencl/.

[104] Paraver project. http://www.bsc.es/computer-sciences/performance-tools/paraver.

[105] Pérez, J. M., Bellens, P., Badía, R. M., and Labarta, J. CellSs: Programming the Cell/B.E. made easier. *IBM Journal of Research and Development 51*, 5 (2007).

[106] PLASMA project home page. http://icl.cs.utk.edu/plasma/.

[107] Pouwelse, J., Langendoen, K., and Sips, H. Dynamic voltage scaling on a low-power microprocessor. In *7th International conference on Mobile Computing and Networking (MobiCom)* (2001), ACM, pp. 251–259.

[108] Quintana-Ortí, E. S., and van de Geijn, R. A. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software 35*, 2 (July 2008), 11:1–11:16.

[109] 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.

[110] 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.

[111] Saad, Y. *Iterative methods for sparse linear systems*, 3rd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[112] Saunders, W. Rethinking supercomputer performance and efficiency for exascale. https://communities.intel.com/community/datastack/blog/2011/11/22/linking-efficiency-and-performance-another-look-at-the-latest-top500-and-green500, 2011.

[113] Schöne, R., Hackenberg, D., and Molka, D. Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors. In *USENIX conference on Power-Aware Computing and Systems (HotPower)* (2012), p. 9.

[114] Shaffer, L. R., Ritter, J. B., and Meyer, W. L. *The critical-path method.* McGraw-Hill, 1965.

[115] Shekar, V., and Izadi, B. Energy aware scheduling for DAG structured applications on heterogeneous and DVS enabled processors. In *International Conference on Green Computing (ICGC)* (2010), IEEE, pp. 495–502.

[116] Shende, S. S., and Malony, A. D. The TAU parallel performance system. *International Journal of High Performance Computing Applications 20* (May 2006), 287–311.

[117] Singh, K., Bhadauria, M., and McKee, S. A. Prediction-based power estimation and scheduling for CMPs. In *23rd International Conference on Supercomputing (ICS)* (2009), pp. 501–502.

[118] SMP superscalar project home page. `http://www.bsc.es/computer-sciences/programming-models/smp-superscalar`.

[119] StarPU project home page. `http://runtime.bordeaux.inria.fr/StarPU/`.

[120] Strazdins, P. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[121] The STREAM benchmark: Computer memory bandwidth. `http://www.streambench.org/`.

[122] Susan L. Graham, Marc Snir, C. A. P., Ed. *Getting up to speed: the future of supercomputing*. The National Academies Press, 2004.

[123] Wang, S., Chen, H., and Shi, W. SPAN: A software power analyzer for multicore computer systems. *Sustainable Computing: Informatics and Systems 1*, 1 (2011), 23–34.

[124] Watkins, D. S. *Fundamentals of Matrix Computations*, 2nd ed. John Wiley and Sons, inc., New York, 2002.

[125] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *International Conference on Supercomputing (ICS)* (1998).

[126] Yao, F., Demers, A., and Shenker, S. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science (FOCS)* (1995), pp. 374–382.

[127] Zaki, O., Lusk, E., Gropp, W., and Swider, D. Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications 13*, 3 (1999), 277–288.

[128] Zee, F. G. V. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[129] Zhang, Y., Hu, X. S., and Chen, D. Z. Task scheduling and voltage selection for energy minimization. In *39th Design Automation Conference (DAC)* (2002), ACM, pp. 183–188.