University College Dublin

Optimization of Multithreaded Data-parallel Applications on Modern Multicore CPUs For Performance and Energy Using Application-level Decision Variables

Semyon Khokhriakov

UCD Student Number: 15204508

This thesis is submitted to University College Dublin in fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

School of Computer Science

Head of School: Assoc. Prof. Chris Bleakley

Research Supervisor: Assoc. Prof. Alexey Lastovetsky

Co-Supervisor: Dr. Ravi Reddy Manumachu

September 2019

Acknowledgements

Primarily I would like to express my deepest appreciation to my supervisor, Prof. Alexey Lastovetsky, for giving me the golden opportunity to do my PhD in the Heterogeneous Computing Laboratory, for his patience, motivation, faith, and immense knowledge he gave me.

Besides my supervisor, I would like to pay special thankfulness, warmth, and appreciation to my co-supervisor, Dr. Ravi Manumachu, whose help and sympathetic attitude at every point during my research helped me to work on time. I had a pleasure and honour to work with him during this research progression. Thank you, Ravi.

I thank all my colleagues from the UCD Heterogeneous Computing Laboratory, Hamidreza Khaleghzadeh, Ken O'Brien, Muhammad Fahad, Emin Nuriyev, Arsalan Shahid, Eric Gyamfi and Tania Malik, for all the fun and journeys we have had in the last four years.

I would also like to warmly thank Dr. Vladimir Rychkov and Ms. Julia Rychkova, who supported me at every bit and made me feel like home. To Dr. Sadegh Panahiazar and Mr. Morteza Matkan for giving me an opportunity to join their "Illumino" startup. To all my friends for always being there whenever I needed them.

This research was supported by Science Foundation Ireland (SFI) under Grant Number 14/IA/2474. I would like to thank SFI and University College Dublin for their financial support. To all the staff of the School of Computer Science, especially to my DSP members, Mr. Damian Dalton and Prof. Michela Bertolotto.

Last but not least, I would like to thank my mother, Ms. Valentina Khokhriakova, for raising me in a way I have become. To all my family for the encouragement, understanding, and love throughout my whole life.

DEDICATION

To

My Family

Abstract

Performance and energy are two most important objectives for optimization on modern parallel platforms such as supercomputers, high performance computing (HPC) clusters, and cloud computing infrastructures. These platforms are now ubiquitously equipped with multicore CPUs to address the twin critical concerns of performance and energy efficiency. The multicore CPUs feature tight integration of tens of cores organized in one or more sockets with multi-level cache hierarchy. Such tight integration, however, leads to several inherent complexities. The complexities are: a). Severe resource contention for shared on-chip resources such as last level lache (LLC), interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers; b). Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes; c). Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM).

The inherent complexities in these CPUs pose difficult challenges to solution methods solving the single- and bi-objective optimization problems of multithreaded data-parallel applications for performance and energy on such platforms. Recent researches demonstrate that performance and energy profiles of data-parallel applications executed on modern multicore CPUs to

manifest drastic variations and these variations are the principal cause for low average performance.

This thesis studies the influence of three-dimensional decision variable space on single- and bi-objective optimizations of applications for performance and energy on multicore CPUs. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups.

The thesis demonstrates the *workload distribution* to be an important decision variable that can no longer be ignored in performance optimization problem of data-parallel applications on modern multicore CPUs. The solution methods using *workload distribution* as a decision variable are proposed in this thesis. These methods employ model-based parallel computing technique and use load-imbalancing data partitioning.

The thesis proposes methods for single-objective optimization for performance and energy on modern multicore CPUs that use the *threadgroups* and the *number of threads* in each threadgroup as decision variables. The workload distribution is fixed so that a given workload is always partitioned equally between the threadgroups.

One of the key findings of this thesis is that energy proportionality of computing does not hold true for multicore CPUs thereby affording an opportunity for bi-objective optimization for performance and energy. Based on this finding, this thesis proposes the first application-level method for solving the bi-objective optimization problem for performance and energy on a single multicore CPU. The method uses two decision variables, the number of identical multithreaded kernels (*threadgroups*) executing the application in

parallel and the *number of threads* in each threadgroup. The workload distribution is not a decision variable. It is fixed so that a given workload is always partitioned equally between the threadgroups.

Finally, this thesis proposes a predictive dynamic energy model based on a non-negative linear regression and employing performance monitoring counters (PMCs) as predictor variables to explain the Pareto-optimal solutions determined by the solution method proposed in this thesis for modern multicore CPUs.

Contents

List of Figures

List of Tables

List of Acronyms

BOPPE Bi-objective optimization problem for performance and energy.

CPU Central processing unit.

DGEMM Double-precision general matrix multiplication.

DPM Dynamic power management.

dTLB Data translation lookaside buffer.

DVFS Dynamic voltage and frequency scaling.

FFT Fast Fourier transform.

FPM Functional performance model.

HPC High-performance computing.

IMKL Intel MKL.

LB Load balancing.

LIMB Load imbalancing.

LLC Last level cache.

NUMA Non-uniform memory access.

OS Operating system.

PAD Padding.

PFFT Parallel fast Fourier transform.

PMC Performance monitoring counters.

PMM Parallel matrix multiplication.

SOPPE Single-objective optimization problem for performance and energy.

TG Threadgroups.

TLB Translation lookaside buffer.

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

High-performance computing (HPC) has received lots of attention from the science and business industry with the advent of multi-core and cloud computing. HPC is essential in physical simulations, weather forecasting, quantum mechanics, data analytics, artificial intelligence (AI), etc., where large-scale problems need to be solved requiring massive computations to be performed. HPC gathers together a wide range of modern homogeneous and heterogeneous platforms (supercomputers [1], Grid'5000 [2]) to deliver higher performance. Multicore CPUs are the mandrel of such system, and any optimization focusing on the objectives such as performance and energy consumption of multicore CPUs, will optimize these objectives for the overall system.

Reviewing the history of computers, for more than three decades prior to mid-2000s called the single-core era, performance doubled every 18 months due to Moore's law [3] and Dennard scaling ([4]). Moore's law states that the number of transistors per square inch on integrated circuits doubles every year since the integrated circuit was invented. Dennard scaling is a scaling model whereby the power density of a transistor based processor of a unit area remains constant due to voltage and current scaling down with the length of the transistor. However, since 2004, designers of processors started facing physical constraints of the integrated circuit containing the transistors. Both power dissipation and power density trends have essentially required

designers to remain within a particular power budget and density requirements. All these limitations, associated with voltage supply scaling, threshold scaling, and clock frequency scaling, along with design complexity, forced companies to look for an alternative to the single core paradigm [5]. Thus, in 2005, AMD released their first dual-core processor (Athlon 64 X2) and from that time onwards, microprocessor architecture entered multicore era. Multicore processors integrate many cores into one chip to overcome the physical constraints of uniprocessor architecture and deliver high computing power with a single chip.

Modern parallel platforms are composed of tightly integrated multicore CPUs with a hierarchical arrangement of cores into sockets with multi-level cache hierarchy. This tight integration has resulted in the cores contending for various shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intel's Quick Path Interconnect [6], AMD's Hyper Transport [7]), leading to resource contention and non-uniform memory access (NUMA). NUMA happens where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes. Figure 2.1 shows the most general architecture of multicore CPUs. It comprises of two sockets (NUMA node 0 and NUMA node 1) with four physical cores each.

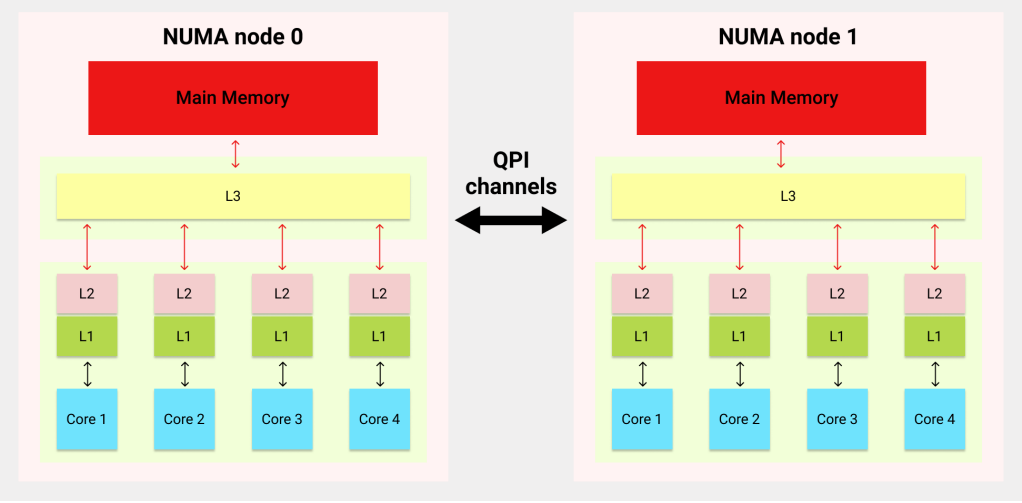


Figure 1.1: The most general architecture of processors nowadays.

Each core has its own L1 and L2 caches. All the cores in a socket share the last level cache (L3). The time taken to access a data item depends on where it is in the multi-level cache and memory hierarchy. The closer the memory to the core, the less the access time. For example: time to access data in the L1 cache is considerably less than that for L2 and L3 caches. Time is longer for access to the memory of the neighbour NUMA node since in this case the slow on-chip interconnect is used. Furthermore, all cores share the same last level cache (L3) leading to severe resource contention for it between threads. Efficient portable parallel programming on platforms composed of such multicore CPUs must address daunting challenges posed by the inherent complexities.

1.1 Motivation Behind This Thesis

To explain the motivation of this thesis, the author elucidates the challenges posed by the inherent complexities in multicore CPU platforms to solving single-objective optimization of data-parallel applications for performance and energy, and bi-objective optimization for performance and energy on such platforms. The challenges are illustrated using two well-known highly optimized scientific kernels, matrix-matrix multiplication (DGEMM) and 2D fast Fourier transform (2D-FFT).

1.1.1 Performance Optimization on Modern Multicore CPUs

This section presents the challenges posed to performance optimization on modern multicore CPUs. This is followed by explanation why the state-of-the-art dominant technique of load balancing fails to address the challenges. Finally, it proposes solution methods to address the challenges.

Figure 1.2 shows the performance profile of multithreaded matrix-matrix multiplication employing DGEMM routine provided by the Intel Math Kernel Library v.2017. The application computes the matrix product ($C = \alpha \times A \times B + \beta \times C$) of two dense square matrices A and B of size $N \times N$. It is executed on a modern Intel Haswell server consisting of 36 cores. The number of threads

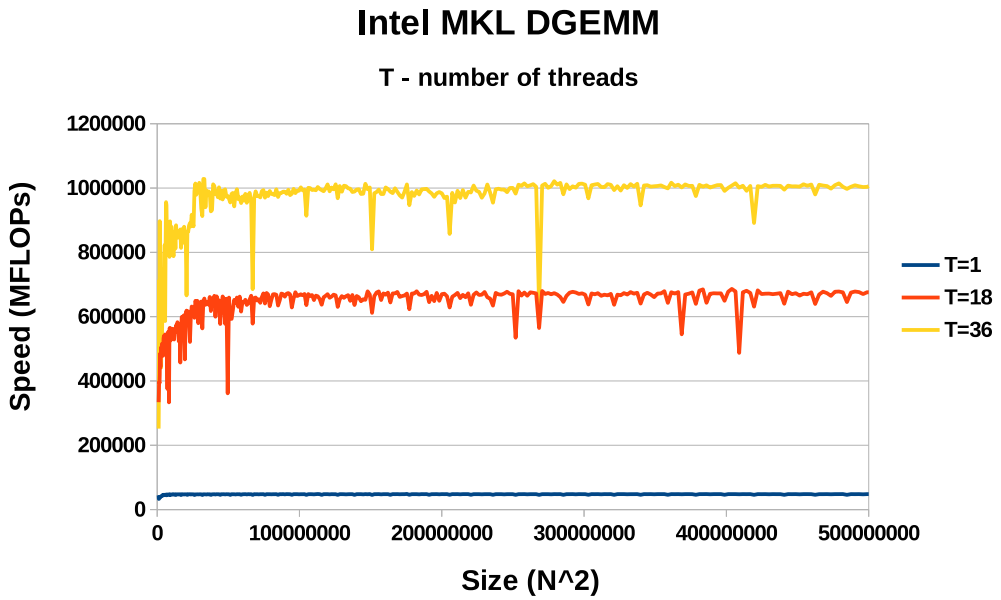


Figure 1.2: Speed function of IMKL DGEMM application executing varying number of threads (T) on the Intel Haswell server.

employed during the execution of the DGEMM routine is configurable.

The crucial observation is that for one thread the profile is smooth. However, drastic variations in the performance can be observed with increasing number of threads. The variation is related to the difference of speeds between two subsequent local minima (s_1) and maxima (s_2) and is defined as: $variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100$. The maximum width of variations with 36 threads is more than 40%. There are several sizes where the width of variations reaches more than 20%.

Figure 1.3 illustrates the performance profile of 2D-FFT offered by the same Intel Math Kernel Library v.2017. The 2D-FFT application is executed with 36 threads on the same Intel Haswell server. It computes the 2D-DFT of the signal matrix of size $N \times N$. The number of threads employed during the execution of the 2D-FFT routine is also configurable. The variations happen for the whole range of problem sizes. The maximum width of variations is around 89%. The detailed study of performance profiles of the 2D-FFT application using three vendor packages, FFTW-2.1.5, FFTW-3.3.7 and IMKL

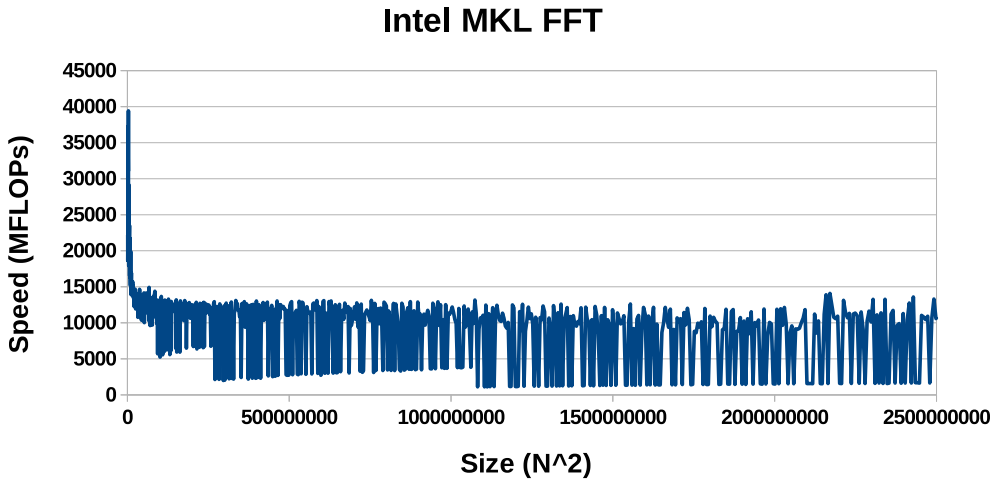


Figure 1.3: Speed function of IMKL FFT application executing with 36 threads on the Intel Haswell server.

FFT, can be found in the Appendix A, where also is show that the FFT routines in the packages demonstrate low average performance due to these variations.

To make sure the experimental results are reliable and not noise, a statistical methodology described in Appendix B is used. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%).

The variations cannot be explained by the constant and stochastic fluctuations due to OS activity or a workload executing in a node in common networks of computers. In such networks, a node is persistently performing minor routine computations and communications by being an integral part of the network. Examples of such routine applications include e-mail clients, browsers, text editors, audio applications, etc. As a result, the node will experience constant and stochastic fluctuations in the workload. This changing transient load will cause a fluctuation in the speed of the node in the sense that the speed will vary for different runs of the same workload. One way to represent these inherent fluctuations in the speed is to use a speed

band rather than a speed function. The width of the band characterizes the level of fluctuation in the speed due to changes in load over time [8], [9], [10]. For a node with uniprocessors, the width of the band has been shown to decrease as the problem size increases. For a node with a very high level of network integration, typical widths of the speed bands were observed to be around 40% for small problem sizes and narrowing down to 3% for large problem sizes. Therefore, as the problem size increases, the width of the speed band is observed to decrease. Therefore, for long running applications, one would observe the width to become quite narrow (3%). However, this is not the case for variations in the presented graphs. Hence, these variations are consequences of the inherent complexities posed by the tight integration which has resulted in the cores contending for various shared on-chip resources such as Last Level Cache (LLC) and interconnect (NUMA). They pose a daunting challenge to performance optimization of multi-threaded applications on modern multicore CPUs.

Load balancing is a well known and still the dominant technique for performance optimization of scientific applications on parallel platforms. Load balancing algorithms can be classified as static or dynamic. Static algorithms (for example, those based on data partitioning) [11], [12] require a priori information about the parallel application and platform. Dynamic algorithms (such as task scheduling and work stealing) [13]–[15] balance the load by moving finegrained tasks between processors during the calculation. Dynamic algorithms do not require a priori information about execution but may incur significant communication overhead due to data migration.

The most advanced load balancing algorithms use functional performance models (FPMs), which are application-specific and represent the speed of a processor by continuous function of problem size but satisfying some assumptions on its shape [9]. These FPMs capture accurately the real-life behavior of applications executing on nodes consisting of uniprocessors (single-core CPUs). The assumptions require them to be smooth enough in order to guarantee that optimal solutions minimizing the computation time are always load balanced. However, as can be seen from the figures 1.2 and 1.3, due to complex nodal architectures with a highly hierarchical arrangement

and tight integration of cores the shape of the performance profiles of real scientific applications on the modern multicore CPUs is not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions.

Lastovetsky et al. [16], [17] study the variations in performance profile for a real-life data-parallel scientific application, Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), on a Xeon Phi co-processor. They geometrically prove the limitations of the FPM-based load balancing algorithms to modern performance profiles executed on multicore CPUs. Based on FPMs, the authors propose a novel optimization technique that distributes workload among cores unequally but gaining better performance in comparison with traditional load balancing. Furthermore, Lastovetsky et al. in [18] propose new model-based methods and algorithms for minimization of time and energy of computations for the most general shapes of performance and energy profiles of data parallel applications observed on the modern homogeneous multicore clusters.

The methods [16]–[18] show that workload distribution has become an important decision variable for performance optimization on modern multicore CPUs. The methods are, however, theoretical works and target homogeneous clusters of multicore CPUs and not a single multicore CPU.

There are three solution approaches that can be employed to remove the performance variations.

Manual code optimization is typically the first approach adopted to improve the performance of an application. The roofline model [19] is used to visually depict the trend of performance gains accrued from code tuning towards the theoretical peak performance of a multicore processor. Using this model, the highly optimized scientific applications such as Intel Math Kernel Library (IMKL) (BLAS, FFT) consistently demonstrate the superior performance of their codes for new platforms.

However, manual code optimization is a time-consuming process and programmers who can program such techniques are rare because they should be experts in both hardware and software domain. This approach involves different techniques such as loop transformation, use of pointers,

use of SIMD registers, blocking etc. [20], [21], [22], to avoid the unprofitable use of cache resources and improve CPU utilization that in turn leads to higher performance. For this, data from performance monitoring counters (PMCs) is required, that demands additional knowledge about hardware specific architecture. PMCs are special-purpose registers provided in modern microprocessors to store the counts of software and hardware activities. We will use the acronym PMCs to refer to software events, which are pure kernel-level counters such as *page-faults*, *context-switches*, etc. as well as micro-architectural events originating from the processor and its performance monitoring unit called the hardware events such as *cache-misses*, *branch-instructions*, etc.

Besides, PMCs in some cases are not reliable based on additivity test proposed in [23]. Moreover, such efficient tuning for one architecture can be inefficient for the other that damages code portability. Some vendors such as Intel do not disclose the source code of their applications which makes code modification impossible at the kernel level.

The second approach constructs solutions for an input workload size by employing solutions to larger workload sizes with better performance. From the figures 1.2 and 1.3 can be seen that two subsequential workload sizes have different performance where sometimes a larger problem size has better performance. The basic idea is to increase the input workload size (by padding, for example) to a bigger workload size with better performance, solve the padded workload size, and use its solution to construct the solution for the input workload size. This is a portable approach.

Finally, the third approach is optimization using model-based parallel computing method [16]–[18]. The key idea behind this approach is to design and implement a parallel version of the application that can be executed using identical abstract processors named threadgroups in parallel. The performances of the threadgroups are represented by realistic and accurate performance models of computation. The models are input to a data partitioning algorithm to determine the optimal workload distribution maximizing the performance during the parallel execution of the application. The main advantages of this approach are:

- It is portable when the performance models of computation used in the data partitioning algorithms do not use architecture-specific parameters.
- It does not require source code modification of the optimized package.
- The programming effort is less time-consuming, which is to distribute the workload between identical already optimized and well-tested multithreaded routines (abstract processors) and execute them in parallel.

This thesis proposes novel single-objective optimization methods specifically designed for performance optimization of 2D fast Fourier transform based on FFTW and IMKL (PFFT) and dense matrix-matrix multiplication written using OpenBLAS DGEMM and IMKL (PMM).

The solution methods employ workload distribution as the decision variable and are based on model-based parallel computing method using load-imbalancing data partitioning technique. The technique determines optimal solutions (workload distributions) that may not load-balance the application in terms of execution time. The methods take as inputs, the discrete functions of the performance of the processors against problem size. Based on the experiments conducted on a dual-socket Intel Haswell CPU consisting of 36 physical cores, the average and maximum speedups observed for *PFFT* using *FFTW-3.3.7* are 2.3x and 9.4x and the average and maximum speedups observed using *IMKL FFT* are 1.4x and 5.9x. The average and maximum speedups observed for *PMM* using *OpenBLAS DGEMM* are 1.2x and 1.4x and the average and maximum speedups observed using *IMKL DGEMM* are 1.1x and 1.3x.

Then an application-level method, SOPPETG, for solving performance optimization problem on a single multicore CPU is proposed. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application in parallel and the number of threads in each threadgroup. The workload distribution is not a decision variable. It is fixed so that a given workload is always partitioned equally between the threadgroups. Based on the experiments conducted on a single-socket Intel

Skylake CPU consisting of 22 physical cores, the average and maximum performance improvements of SOPPETG using *OpenBLAS DGEMM* are 7% and 26.3% and the average and maximum performance improvements using *IMKL DGEMM* are 4.1% and 6.5%. The average and maximum performance improvements of SOPPETG using *IMKL FFT* are 7% and 13% and using *FFTW-3.3.7* are 25% and 51% respectively.

On a dual-socket Intel Haswell CPU consisting of 36 physical cores, the average and maximum performance improvements of SOPPETG using *OpenBLAS DGEMM* are 19% and 31.7% and the average and maximum performance improvements using *IMKL DGEMM* are 7% and 42.1%. The average and maximum performance improvements of SOPPETG using *FFTW-3.3.7* are 85% and 90%.

1.1.2 Energy Optimization on Modern Multicore CPUs

Reducing energy consumption is of paramount concern to the HPC community since its pervasiveness in data centers and cloud computing infrastructures. Energy in HPC is now an environment concern not only because of the maintenance cost of HPC systems but also of high carbon footprint which affects environmental sustainability as modern data centers already can rival cities in power consumption. This was not an issue in the past since until now we have followed Moore's Law enhancements in photolithography techniques which are proportional reductions in dynamic power consumption per transistor and consequent improvements in clock frequency at the same level of power dissipation. However, below 90 nm, the static power dissipation can be greater than the dynamic power dissipation. This effect summons clock frequency freezing in order to stay within thermal power emission limits [24].

The optimization of energy consumption of multicore CPUs is more complex than that of a single- or dual-core CPUs. The new complexities such as tight integration with severe contention on shared resources (Last level caches (LLC), main memory, PCI-E links, etc.) and NUMA pose tremendous challenges to the energy optimization of data-parallel applications on modern

multicore CPUs.

In contrast to single-core optimization, where energy profiles follow the fully polynomial-time scheme for task partitioning, i.e. energy consumption with a higher workload is larger than that with a lower workload [25], the energy profiles of real scientific applications executed on modern multicore CPUs demonstrate highly non-linear relationship between workload size and energy consumption.

As an example, figure 3.2 depicts the dynamic energy consumption profile of 2D-FFT employing IMKL FFT on the Intel Xeon Platinum server consisting of 56 cores. The dynamic energy consumption is measured with Yokogawa WT310 power meter. It can be seen that the graph is highly non-linear. The maximum width of variations can be up to 73%. It represents the maximum amount of energy savings possible.

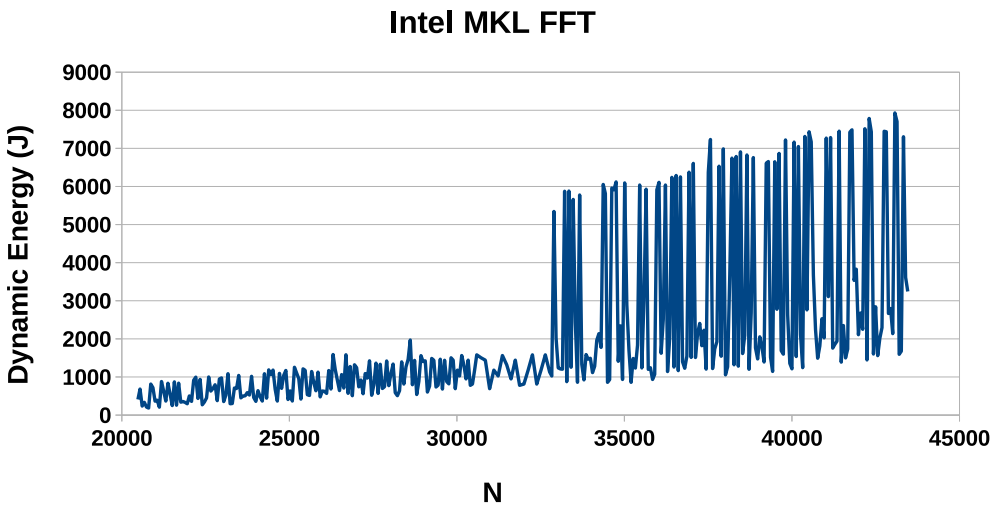


Figure 1.4: Dynamic Energy Consumption of IMKL FFT application executing with 56 cores on the Intel Xeon Platinum server.

The research works [26], [27] propose model-based data partitioning methods to minimize the total dynamic energy consumption during the execution of a data-parallel application on homogeneous clusters of multicore CPUs. They take as input discrete dynamic energy functions with no shape assumptions (for example, the discrete profile in the Figure 1.4), which

accurately and realistically account for resource contention and NUMA inherent in modern multicore CPU platforms. The research works are theoretical demonstrating energy improvements based on simulations of clusters of homogeneous nodes containing multicore CPUs.

This thesis proposes an application-level method, SOPPETG, for solving energy optimization problem on a single multicore CPU. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application in parallel and the number of threads in each threadgroup. The workload distribution is not a decision variable. It is fixed so that a given workload is always partitioned equally between the threadgroups. Based on the experiments conducted on a single-socket Intel Skylake CPU consisting of 22 physical cores, the average and maximum energy savings of SOPPETG using *OpenBLAS DGEMM* are 7.9% and 30% and the average and maximum energy savings using *IMKL DGEMM* are 35.7% and 67%. The average and maximum energy savings of SOPPETG using *FFTW-3.3.7* are 30% and 63%.

On a dual-socket Intel Haswell CPU consisting of 24 physical cores, the average and maximum energy savings of SOPPETG using *OpenBLAS DGEMM* are 10% and 24.5% and the average and maximum energy savings using *IMKL DGEMM* are 13% and 67%. The average and maximum energy savings of SOPPETG using *FFTW-3.3.7* on a dual-socket Intel Skylake CPU consisting of 56 cores are 23% and 43%.

1.1.3 Bi-Objective Optimization for Performance and Energy

Energy proportionality is the key design goal pursued by architects of modern multicore CPU platforms [28]. One of its implications is that optimization of an application for performance will also optimize it for energy. Modern multicore CPUs however have several inherent complexities, which are: a) Severe resource contention due to tight integration of tens of cores organized in multiple sockets with multi-level cache hierarchy and contending for shared on-chip resources such as last level lache (LLC), interconnect (For example:

Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers; b) Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes; and c) Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM). This thesis shows that due to these complexities, energy proportionality does not hold true for multicore CPUs. This finding creates the opportunity for bi-objective optimization of applications for performance and energy.

Solution methods solving the bi-objective optimization problem for performance and energy BOPPE can be broadly classified into *system-level* and *application-level* categories. System-level methods aim to optimize performance and energy of the environment where the applications are executed. The methods employ application-agnostic models and hardware parameters as decision variables. They are principally deployed at operating system (OS) level and therefore require changes to either the OS or the hardware. The key decision variable employed is Dynamic Voltage and Frequency Scaling (DVFS).

In the second category, solution methods optimize applications rather than the executing environment. The methods use application-level decision variables and predictive models for performance and energy consumption of applications to solve BOPPE. The dominant decision variables include the number of threads, loop tile size, workload distribution, etc. Following the principle of energy proportionality, a dominant class of such solution methods aim to achieve optimal energy reduction by optimizing for performance alone. Definitive examples are scientific routines offered by vendor-specific software packages that are extensively optimized for performance. For example, Intel Math Kernel Library [29] provides extensively optimized multithreaded basic linear algebra subprograms (BLAS) and 1D, 2D, and 3D fast Fourier transform (FFT) routines for Intel processors. Open source packages such as [30]–[32] offer the same interface functions but contain portable optimizations and may exhibit better average performance than a heavily optimized vendor package [33], [34]. The optimized routines in these software packages allow

employment of one key decision variable, which is the number of threads. A given workload is load-balanced between the threads.

The works [26], [27], [35] propose model-based data partitioning methods that take as input discrete performance and dynamic energy functions with no shape assumptions, which accurately and realistically account for resource contention and NUMA inherent in modern multicore CPU platforms. Using a simulation of the execution of a data-parallel matrix multiplication application based on OpenBLAS DGEMM on a homogeneous cluster of multicore CPUs, [26] show that optimizing for performance alone results in average and maximum dynamic energy reductions of 24% and 68%, but optimizing for dynamic energy alone results in performance degradations of 95% and 100%. For a 2D fast Fourier transform application based on FFTW, the average and maximum dynamic energy reductions are 29% and 55% and the average and maximum performance degradations are both 100%. Research work [35] proposes a solution method called *ALEPH* to solve BOPPE on homogeneous clusters of modern multicore CPUs. ALEPH is shown to determine a diverse set of globally Pareto-optimal solutions whereas existing solution methods give only one solution when the problem size and number of processors are fixed. The methods target homogeneous HPC platforms. Khaleghzadeh et al. [36] propose a solution method solving the bi-objective optimization problem on heterogeneous processors. The authors prove that for an arbitrary number of processors with linear execution time and dynamic energy functions, the globally Pareto-optimal front is linear and contains an infinite number of solutions out of which one solution is load balanced while the rest are load imbalanced. A data partitioning algorithm is presented that takes as an input discrete performance and dynamic energy functions with no shape assumptions. The research works [26], [27], [35], [36] are theoretical demonstrating performance and energy improvements based on simulations of clusters of homogeneous and heterogeneous nodes.

All these works done on bi-objective optimization for performance and energy do not consider the optimization on a single multicore CPU. Furthermore, the works [26], [27], [35], [36] are theoretical and use only workload distribution as a decision variable. However, one of the findings of

this thesis is that modern multicore CPUs are not energy proportional and a trade-off between energy and performance can be found on such platforms. This finding opens an opportunity for bi-objective optimization for performance and energy on a single multicore CPU and makes it meaningful. To the best of author's knowledge, this is the first work studying bi-objective optimization for performance and energy consumption on a single multicore CPU.

This thesis studies the influence of three-dimensional decision variable space on bi-objective optimization of applications for performance and energy on multicore CPUs. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups. The author focuses exclusively on the first two decision variables in this work. The number of possible workload distributions increases exponentially with increasing number of threadgroups employed in the execution of a data-parallel application and it would require employment of threadgroup-specific performance and energy models to reduce the complexity. It is a subject of future work.

The thesis proposes the first application-level method for bi-objective optimization of multithreaded data-parallel applications on a single multicore CPU for performance and energy. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application in parallel and the number of threads in each threadgroup. The workload distribution is not a decision variable. It is fixed so that a given workload is always partitioned equally between the threadgroups. The method allows full reuse of highly optimized scientific codes and does not require any changes to hardware or OS.

Based on the experiments conducted on a dual-socket Intel Skylake CPU consisting of 56 cores, it was observed that the number of Pareto optimal solutions can be up to 11 for *FFTW-3.3.7.*. Figure 1.5 shows these solutions for problem size $m = n = 30464$. One can observe, choosing the best configuration for performance (g,t)=(1,96), increases the dynamic energy

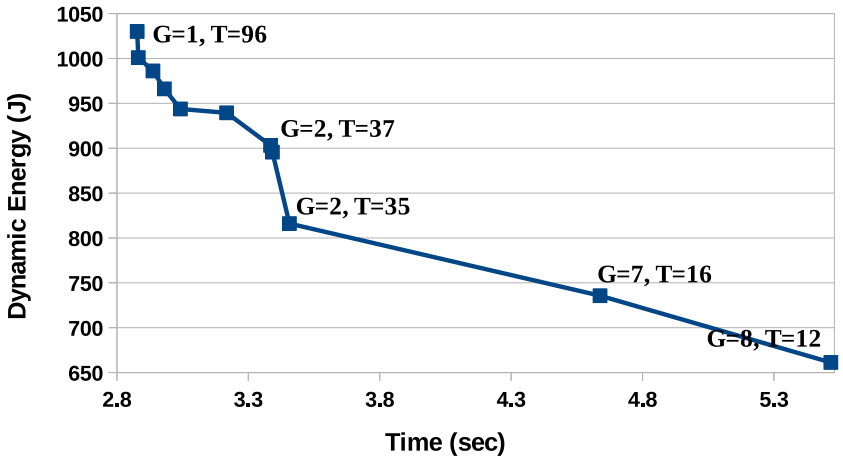


Figure 1.5: Pareto frontier of FFTW PFFTTG application on HCLServer4 (S4) for workload size $m = n = 30464$.

consumption by 35% in comparison with the optimal configuration for energy (8,12), and choosing the optimal configuration for energy (8,12), degrades the performance by 49% in comparison with the optimal configuration for performance (1,96). The average number of globally Pareto-optimal solutions for *FFTW-3.3.7* is 3. On a single-socket Intel Skylake CPU consisting of 22 physical cores, the average and the maximum number of globally Pareto-optimal solutions for *IMKL DGEMM* and *IMKL FFT* are (2.3,3) and (2.6,3).

Finally, this thesis proposes a predictive dynamic energy model based on non-negative linear regression and employing performance monitoring counters (PMCs) as predictor variables to explain the Pareto-optimal solutions determined by solution method proposed in this thesis for multicore CPUs.

1.2 Thesis Contributions

The main contributions of this thesis are the following:

1. Demonstration of the challenges posed by inherent complexities in modern multicore CPUs such as severe resource contention and

NUMA to the performance of multi-threaded data-parallel applications executing on such platforms.

2. Studying the performance profiles of multithreaded 2D FFT and matrix-matrix multiplication provided in highly optimized packages, FFTW-3.3.7, IMKL FFT, OpenBALS DGEMM and IMKL DGEMM on a modern Intel Haswell multicore processor consisting of thirty-six cores. It is shown that all routines demonstrate drastic performance variations and that their average performances therefore are considerably lower than their peak performances.

3. Three novel optimization methods specifically designed for optimization of 2D-FFTW, 2D-FFT-IMKL, OpenBLAS-DGEMM and IMKL-DGEMM for performance. The methods employ workload distribution as the decision variable and are based on model-based parallel computing method using load-imbalancing data partitioning technique. The technique determines optimal solutions (workload distributions) that may not load-balance the application in terms of execution time.

4. Application-level methods for single-objective optimization of multithreaded data-parallel applications for performance and energy. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) and the number of threads in each threadgroup.

5. Detection and demonstration of that the energy proportionality does not hold true for multicore CPUs thereby affording an opportunity for bi-objective optimization for performance and energy.

6. The first application-level method for bi-objective optimization of multithreaded data-parallel applications for performance and energy. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) and the number of threads in each threadgroup. The method is demonstrated using four highly optimized data-parallel applications. It is shown that the proposed

method determines good numbers of globally Pareto-optimal configurations of the applications allowing for a better balance between performance and energy consumption.

7. Predictive dynamic energy model based on linear regression and employing PMCs as predictor variables to explain the Pareto-optimal solutions determined by the method proposed in this thesis for dual-socket multicore CPUs.

1.3 Thesis Structure

The rest of the thesis is organized as follows: chapter 2 covers the review of state-of-the-art methods of single-objective optimization for performance and energy, bi-objective optimization for performance and energy on modern multicore CPUs, and performance and energy models of computation. Chapter 3 presents novel methods for single-objective optimization performance and energy using three decision variables - workload distribution, the number of threadgroups and the number of threads in each threadgroup. Chapter 4 proposes bi-objective optimization for performance and energy on modern multicore CPUs using the number of threadgroups and the number of threads per threadgroup as decision variables. The conclusion of this thesis is in chapter 5.

Chapter 2

Background and Related Work

This section starts with survey of research works focusing on single-objective optimization for performance and energy on modern multicore CPUs. Then, solution methods for bi-objective optimization for performance and energy and multi-objective optimization on multicore CPU platforms are presented.

2.1 Multicore CPUs: Performance Optimization

In the era of uniprocessors prior to the advent of multicore CPUs, computer users came to expect performance doubling every 18 months due to Moore's law and Dennard scaling. However, achieving the high performance of applications on multicore CPUs is not as simple. There are complex issues surrounding the migration forward to multicore architecture which affect the performance. They are: a). Severe resource contention due to tight integration of tens of cores organized in multiple sockets with multi-level cache hierarchy and contending for shared on-chip resources such as last level cache (LLC), interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers, b). Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes, and c). Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM).

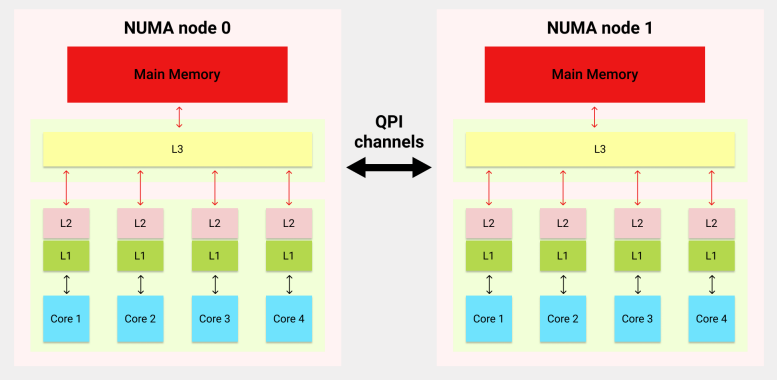


Figure 2.1: A representative architecture of modern multicore CPU processors.

These complexities pose serious challenges to model and algorithm designers.

Figure 2.1 depicts a representative architecture of modern multicore CPU processors. It comprises two sockets (NUMA node 0 and NUMA node 1) with four physical cores each. Each core has its own L1 and L2 caches. All the cores in a socket share the last level cache (L3). The time taken to access a data item depends on where it is in the multi-level cache and memory hierarchy. The closer the memory to the core, the less is the access time. For example: time to access data in the L1 cache is considerably less than that for L2 and L3 caches. Time is longer for access to the memory of the neighbour NUMA node since in this case the slow on-chip interconnect is used. Furthermore, all cores share the same last level cache (L3) leading to severe resource contention for it between threads. Hence, programmers and algorithm designers must take into account the inherent complexities to port or write high performance codes for such architectures.

Next, we look at the performance models designed to analyze or predict the performance of applications on multicore CPUs, wich is followed by the most popular methods for performance optimization on modern multicore processors.

2.1.1 Performance Models of Computation

Roofline model [19] is a well known and standard model for performance estimation of a given compute kernel or application running on multi-core, many-core, or accelerator processor architectures. This model visually depicts inherent hardware limitations and potential benefit of optimization by determination of an upper bound on feasible performance. More complex performance models employ the informaton from hardware and/or software performance counters [37], [38], [39], [40], [41]. They build a linear or non-linear relationship between performance and counter's information. The state-of-the-art models include the functional performance model (FPM) of a given application [9], [12], [13], [14], [15], [42], [43], [44]. This model represents the processor speed by a function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application.

2.1.2 Code Tuning

Code optimization is typically the first approach adopted to improve the performance of an application. The roofline model [19] is used to visually depict the trend of performance gains accrued from code tuning towards the theoretical peak performance of a multicore processor. Using this model, the highly optimized scientific applications such as Intel Math Kernel Library (IMKL) (BLAS, FFT) consistently demonstrate the superior performance of their codes for new platforms.

The code optimization is usually performed at two levels: at the algorithm level, and the microkernel level. The algorithm level implies different implementation of the same function to gain better performance. For instance, in matrix multiplication, one of the most optimized packages, OpenBLAS [30], uses the GotoBLAS [45] algorithm, that is far faster than a naive ijk algorithm. At the kernel level, the code is tuned for the specific architecture of the processors. Usually, it requires coding in low-level programming languages, for example, assembly language. Both approaches involve different techniques such as loop transformation, pointers, SIMD

registers, blocking, etc., to avoid the unprofitable use of cache resources and improve CPU utilization that in turn leads to higher performance [20], [21], [22].

Furthermore, based on hardcoded pieces of code, vendors provide autotuning, which involves automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation [46]. The simplest approach in autotuning is to execute each code variant, measure its runtime (or other objective function), evaluate the performance of all variants and select the best one to be run. For instance, ATLAS [47] provides self-adapting implementations that search for optimal parameters and code structure to find the optimal configuration for the target platform. FFTW [31] has its own "wisdom" implementation that tracks the performance of execution candidates and chooses the optimal for execution. However, it is shown in chapter 3 using FFTW as an example that such kind of optimization can be very time-consuming and may perform poorly compared to one that does not use the optimization.

Other examples of autotuning are based on model-based approaches. Here, autotuners perform complete enumeration of all possible or pruned parameter configurations obtained by knowledge about architecture-specific and/or application-specific information [48], [49], [21]. Koliai et al. [48] introduce a combined methodology for the optimization process. Their strategy combines static assembly analysis using MAQAO [50] with dynamic information from hardware performance monitoring (HPM) and memory traces. Then they present a technique called decremental analysis (DECAN) to iteratively identify the individual instructions responsible for performance bottlenecks.

Hashimito et al. [51] use evidence-based performance tuning (EBT) that aims at helping performance engineers gain and share evidence of performance improvement to make better decisions. They developed a tool, CCA/EBT, which assists performance engineers in comprehending source code written in Fortran, especially to identify loop kernels. Eventually, their goal is to construct a database of facts extracted from performance tuning

histories of computational kernels that is then used for promising optimization patterns that fit a given computational kernel.

To summarise, all these methods require source code modification at the initial stage that makes them time-consuming and requires enormous human efforts. Porting code to a different processor architecture requires the repetition of pattern search. There is no specific number of attempts and configuration parameters which should be tried, to find the best configuration. Moreover, some verdors, for example Intel, do not disclose the source code of their applications that makes code modification impossible at the kernel level.

2.1.3 Scheduling

As processor architects have turned to multi-core designs, where cores share resources, scheduling is a promising approach for performance optimization. Choosing which threads to run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core since simultaneous multithreaded cores (SMT) share low-level hardware resources such as TLBs among all threads [52]. Traditional scheduling uses an explicit thread and data placement. Most thread library implementations provide support for pinning threads to assign threads to specific CPUs (i.e., hardware threads) and to restrict their migration. Others use performance data that is provided by hardware and software counters, that are embedded in all popular processor architectures.

D. Chandra et al. [53] study the impact of L2 cache sharing on threads that simultaneously share the cache, on a Chip Multi-Processor (CMP) architecture. They propose a performance model which uses L2 cache profile, for predicting which thread slows down the performance.

Radojkovic et al. [54] propose BlackBox scheduler, a systematic method for thread assignment of multithreaded network applications running on multicore processors. Their method requires minimum information about the target processor architecture and no data about the hardware requirements of the application under study. The method demonstrates performance

improvements from 5 to 48 percent over the load balancing algorithms for threads assignment.

Ebrahimi et al. [55] propose a memory scheduling algorithm designed specifically for parallel applications. Their approach has two main components: locks and barriers. The idea is to design a memory scheduler that reduces parallel application execution time by managing inter-thread DRAM interference. They show that by intelligently prioritizing requests in a thread-aware manner, memory controller significantly improves the performance of parallel applications compared to state-of-the-art memory controller designs.

Jeong et al. [56] propose to partition the internal memory banks between cores to isolate their access streams and eliminate locality interference. They extend the physical frame allocation algorithm of the OS such that physical frames mapped to the same DRAM bank can be exclusively allocated to a single thread. They use together bank partitioning and memory sub-ranking that balance the conflicting demands for rowbuffer locality and bank parallelism. This combination, unlike using each separately, is able to simultaneously increase overall performance and significantly reduce memory power consumption.

Matthew DeVuyst et al. [57] study both, balanced and unbalanced schedules. They show that unbalanced schedules often outperform balanced schedules when threads are distributed unevenly between cores. Higher performance is obtained by clumping badly behaving threads together on the same core than by spreading them around. They conclude that the best scheduling policies are those that consider both balanced and unbalanced schedules.

Wen et al. [58] consider heterogeneous systems consisting of multiple CPUs and GPUs. They propose an efficient OpenCL task scheduling scheme which schedules multiple kernels from multiple programs on CPU/GPU platforms. The model predicts a kernel's speedup based on its static code structure. The input data to the scheduler are prediction and runtime input data size.

Khaleghzadeh et al. [59] propose a thread mapping technique that

minimizes remote communications and cache coherency costs of multi-threaded applications due to the maximization of data reuse. This technique includes components such as data sharing estimator, affine mapping finder and maximum speedup predictor. Using Phoenix benchmark suite they demonstrate improvement in performance by 25% compared to default Linux scheduler.

2.1.4 Data Partitioning

2.1.4.1 Load Balancing Algorithms

Load balancing is a widely used method for performance optimization of data-parallel applications on parallel platforms. There are different classifications of it: static or dynamic, and synchronous or asynchronous.

Static algorithms use a priori information about the parallel application and platform [9], [12]. They are particularly useful for applications where data locality is important because they do not require data redistribution. In its turn, static load balancing algorithms may be either deterministic or probabilistic. The first one uses information about the properties of nodes and the features of the processes. The second uses the information of the system such as the number of nodes, the processing capability of each node, the network topology, etc. The goal of static load balancing method is to reduce the overall execution time while minimizing the communication delays. In [60] presented a simple method that uses static load-balancing to balance parallel FDTD codes. FDTD is a collection of open source, Finite-Difference Time-Domain, demo programs. The author shows that the described method for partitioning in a single mesh dimension can be adapted also for 2D and 3D partitioning in an approximate way. Results show that this method gives significant improvements in running times and it can be comparable with optimization that is more expensive. The static algorithms however are unsuitable for non-dedicated platforms, where load changes with time.

Dynamic algorithms, on the other hand, balance the load by moving fine-grained tasks between processors during the execution [13], [14], [15]. They often use static partitioning for their initial step due to its provably

near-optimal communication cost, bounded small load imbalance, and lesser scheduling overhead. Dynamic load balancing can be categorized as centralized and distributed. In a centralized load balancing technique, global load information is stored at a central location, while in a distributed load balancing the information is distributed among the processors. In other words, in centralized load balancing, there is a centralized load balancer that decides when to distribute data based on global load information [61], [62]. In the distributed load balancing, at some point of computation, each processor find neighbours that are less loaded than itself and redistributes data between them [63], [64].

The synchronous algorithm means that for each processor to balance its load at time $t + 1$, a processor needs to have the load of its neighbor at time t [65]. In other words, there is time-synchronization between all processors. In an asynchronous algorithm, the time synchronization is absent [66].

The most advanced load balancing algorithms use functional performance models (FPMs), which are application-specific and assume the speed of a processor to be a continuous function of problem size satisfying some assumptions on its shape [8], [9]. These FPMs capture accurately the real-life behavior of applications executing on nodes consisting of uniprocessors (single-core CPUs).

Zhong et al. [42] extend the FPM based data partitioning to heterogeneous multicore and multi-GPU platforms. To show the efficiency of the partial FPM based data partitioning algorithm, they compare its execution time with different partitioning algorithms (homogeneous, CPM (constant performance model) and FPM (functional performance model)). The result is depicted in figure 2.2. The FPM based data partitioning outperforms homogeneous and CPM-based partitioning by up to 13% and 22% respectively.

2.1.4.2 Load Imbalancing Algorithms

Modern multicore CPUs contain tens of cores tightly integrated with multi-level cache hierarchy. This tight integration has resulted in the cores

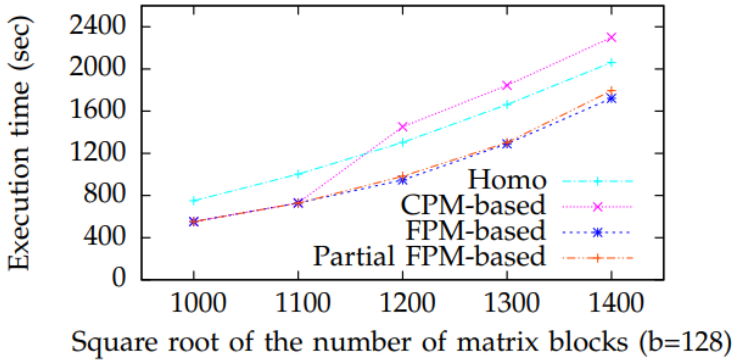


Figure 2.2: Execution time of the parallel matrix multiplication application with different data partitioning algorithms [42].

contending for various shared on-chip resources such as Last Level Cache and interconnect leading to resource contention and NUMA. Due to these newly introduced complexities, the performance and energy profiles of real-life scientific applications executing on these platforms are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions.

Lastovetsky et al. [16], [17] study these variations in performance profile for a real-life data-parallel scientific application, Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), on a Xeon Phi co-processor. MPDATA is one of the major parts of the dynamic core of the EULAG (Eulerian/semi-Lagrangian fluid solver) geophysical model [67]. They use functional performance models of the application to find partitioning that minimizes its computation time but not necessarily balances the load of processors. The processors are divided into two equal teams, called the left team and the right team. Then, they experimentally build the performance profiles of each team for a wide range of problem sizes separated by a minimum granularity Δx. After, the algorithm goes through a number of iterations. At each iteration, the workload of each processor from the right team is increased by Δx, while the workload of each processor from the left team is decreased by the same amount, Δx. It returns the distribution with the minimal calculated execution time. The distribution is not always even, i.e.

does not balance the workload. This is the first work where the load-imbalancing technique is applied to distribute the workload unevenly, minimizing the computation time of its parallel execution.

The research works [68], [69] propose model-based data partitioning methods that take as input discrete performance and dynamic energy functions with no shape assumptions, which accurately and realistically account for resource contention and NUMA inherent in modern multicore CPU platforms. Using a simulation of the execution of a data-parallel matrix multiplication application based on OpenBLAS DGEMM on a homogeneous cluster of multicore CPUs, they show that optimizing for performance alone results in average and maximum dynamic energy reductions of 24% and 68%, but optimizing for dynamic energy alone results in performance degradations of 95% and 100%. For a 2D fast Fourier transform application based on FFTW, the average and maximum dynamic energy reductions are 29% and 55% and the average and maximum performance degradations are both 100%. The methods target homogeneous HPC platforms.

Khaleghzadeh et al. [70] propose a solution method solving the bi-objective optimization problem on heterogeneous processors. The authors prove that for an arbitrary number of processors with linear execution time and dynamic energy functions, the globally Pareto-optimal front is linear and contains an infinite number of solutions out of which one solution is load balanced while the rest are load imbalanced. A data partitioning algorithm is presented that takes as an input discrete performance and dynamic energy functions with no shape assumptions. The research works [68]–[70] are theoretical demonstrating performance and energy improvements based on simulations of clusters of homogeneous and heterogeneous nodes.

2.2 Multicore CPUs: Energy Optimization

In this section, we introduce the terminology for power and energy in computing. This is followed by survey of research works on system-level energy optimization employing hardware parameters as decision variables. Then we focus on energy optimization of applications using application-level

parameters.

2.2.1 Terminology

There are two types of energy consumptions, static energy and dynamic energy [71], [72]. We define the static energy consumption as the energy consumption of the platform without the given application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumption of the platform during the given application execution. If P_S is the static power consumption of the platform, E_T is the total energy consumption of the platform during the execution of an application, which takes T_E seconds, then the dynamic energy E_D can be calculated as,

$$E_D = E_T - (P_S \times T_E) \tag{2.1}$$

In this thesis, we focus purely on minimization of the dynamic energy consumption for reasons below:

1. Static energy consumption is a constant (or a inherent property) of a platform that can not be optimized. It does not depend on the application configuration.

2. Although static energy consumption is a major concern in embedded systems, it is becoming less compared to the dynamic energy consumption due to advancements in hardware architecture design in HPC systems.

3. We target applications and platforms where dynamic energy consumption is the dominating energy dissipator.

4. Finally, we believe its inclusion can underestimate the true worth of an optimization technique that minimizes the dynamic energy consumption. We elucidate using two examples from published results.

- In our first example, consider a model that reports predicted and measured total energy consumption of a system to be 16500J and 18000J. It would report the prediction error to be 8.3%. If it is known that the static energy consumption of the system is 9000J, then the actual prediction error (based on dynamic energy consumptions only) would be 16.6% instead.
- In our second example, consider two different energy prediction models (M_A and M_B) with same prediction errors of 5% for an application execution on two different machines (A and B) with same total energy consumption of 10000J. One would consider both the models to be equally accurate. But supposing it is known that the dynamic energy proportions for the machines are 30% and 60%. Now, the true prediction errors (using dynamic energy consumptions only) for the models would be 16.6% and 8.3%. Therefore, the second model M_B should be considered more accurate than the first.

2.2.2 Energy Models of Computation

A big survey of energy models was done by O'brien et al. in [73]. They review the energy models for CPU, GPU along with Xeon Phis and FPGAs. As this thesis focuses only on multicore CPU, a simple linear energy model for CPU can be expressed as follows:

$$P = C_{base} + C_1 * U_{CPU},$$

where C_{base} is the idle state power consumption, C_1 is the difference in maximum power consumption between fully utilized processor (100%) and the idle, and U_i is total CPU utilization. This model implies that power consumption linearly increases when the CPU utilization increases. In [74] is also proposed linear model, however, they comprise one more parameter such as hard disk utilization.

Economou et al. [75] present a linear model that based on four parameters:

$$P = C_0 + C_1 * U_{CPU} + C_2 * U_{mem} + C_3 * U_{disk} + C_4 * U_{net},$$

where U_{CPU} - CPU utilization, U_{mem} - off-chip memory access count, U_{disk} - hard disk I/O rate, U_{net} - network I/O rate. The use hardware performance counters to obtain the utilization data of the memory.

An analogous work can be found in [76] that uses the PowerPack tool. This work focuses on profiling actual power consumption in a cluster of homogeneous nodes and measure the power consumption by a parallel application. This is done by isolating power by component and measuring the power consumed in the CPU, memory, disk and network interface components. A digital multi meter measures the voltage on each resistor taking 4 measurements per second. The data measured by the meters are then logged and processed.

The SimplePower framework is presented in [77]. The framework is focusing on providing information about the energy hotspots in the system and estimation the implications of applying architectural and software optimizations simultaneously on the overall energy consumption. They observed that the compiler optimizations gains the most significant energy reduction of the computing environment.

Gschwandtner et al. [78] propose in-band energy consumption models for the IBM POWER7 processor that are based on hardware counters and use linear regression. They study the effects of compiler and parallelism on the energy consumption. Furthermore, they also investigate energy consumption model for memory. For this, they use scope of performance counters such as memory reads and writes, cache misses, prefetching instructions, etc.

Zamani et al. [79] study the correlation between performance monitoring counters (PMCs) and power consumption from a stochastic perspective. They show the goodness of autoregressive moving average (ARMA) for trend's modelling in performance and power. For better accuracy, they use algorithms such as recursive least-squares (RLS) filter, Kalman filter (KF), or multivariate normal regression (MVNR). Furthermore, they attempt to predict

near-future power and PMC values with an average error of 8% for dynamic power consumption and an average error of less than 11.1% and 7% for dynamic runtime power and instructions per cycle respectively.

Shahid et al. [80] introduce a new parameter, additivity, which should be taken into account under selecting PMCs for energy predictive modelling. Additivity means that the value of a PMC for a serial execution of two applications should be equal to the sum of its values obtained for the individual execution of each application. Using linear regression, random forests and neural networks they demonstrate that this is an important parameter which has significant impact on the accuracy of energy predictive model.

2.2.3 System-level and Component-level Optimization

The techniques for system-level and component-level energy optimization can be classified into following three principal categories:

1. Dynamic voltage and frequency scaling (DVFS): Techniques based on scaling voltage and/or frequency levels of a computing system based on performance and power requirements.

2. Dynamic power management (DPM): Techniques that use different configurations-modes of operating system. For instance, the mode that puts unutilized system components at a low level of power consumption (by scaling the core frequency, voltage) or switches them off.

3. Component-level management (CPM): Techniques that deal with the minimizing energy consumption by the components of the system like main memory, caches, TLB and their reconfiguration.

2.2.3.1 Dynamic voltage and frequency scaling (DVFS)

Dynamic voltage and frequency scaling (DVFS) is the most dominant system-level approach in the area of energy efficiency in HPC. Modern multicore CPUs allow scaling of operating frequency of each core. The frequency f, however, has a relationship with core voltage V as shown in figure 2.3.

We can simply define that the dynamic power P_{dyn} of CMOS circuit is strongly dependent on the core voltage V and the clock frequency f:

$$P \infty f V^2$$

Based on the assumption that the number of clock cycles used in the computation is independent of the core frequency, the execution time is inversely proportional to the frequency [81]. Consequently, the total energy consumption of computations E is square proportional to the core voltage:

$$E \infty V^2$$

Although from this model we can see that the energy does not depend on the frequency, the reducing of core voltage that reduces overall energy consumption requires a reduction of the clock frequency (Figure 2.3) that summons longer execution time. Hence, one should always scale CPU voltage-frequency for depreciation energy consumption with minimal damage to the performance.

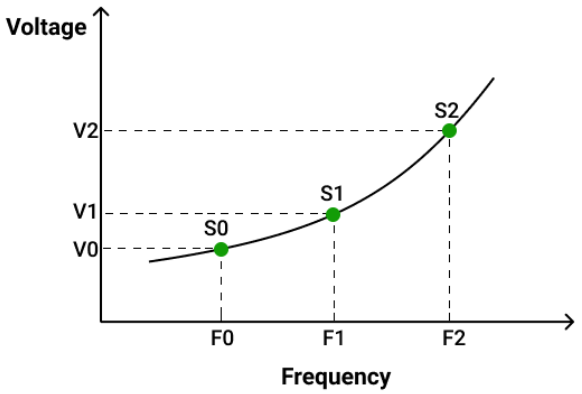


Figure 2.3: Relationship between core voltage and frequency.

Deng et al. [82] study the dynamic voltage and frequency scaling of both CPU and the memory system. They present *CoScale*, the method for effectively coordinating such kind of scaling under performance constraints. This method based on execution profiling of cores via performance counters,

models of core and memory performance, power consumption. *CoScale* goes through the set of possible frequency settings and finds those that efficiently minimizes the full-system energy consumption within the performance constrains.

Lai et al. [83] examine the latency of DVFS on a real many-core platform. The proposed latency-aware DVFS algorithm achieves profile-guided power management to avoid aggressive power state transitions.

Chen et al. [84] propose energy optimization technique which optimally combines DVFS and dynamic power management (DPM) in real-time multicore systems. This method is applicable in multicore systems where DVFS and DPM can be applied for each core independently and each core operates at several discrete voltage and frequency levels.

Datta et al. [85] present two CPU scheduling algorithms, Algorithm Cache Miss Priority CPU Scheduler (CM-PCS) and Algorithm Context Switch Priority CPU Scheduler (CS-PCS) for reducing energy consumption. These algorithms deal with a process's cache miss/reference ratio, number of context switches and CPU migrations, and system load. Thus, they match processes to cores better suited to execute those processes which lower the average task completion time.

Rizvandi et al. [86] propose an approach for energy reduction based on MVFS-DVFS algorithm wich find the best combination of frequencies. Furthermore, using a linear combination of more than one voltage-frequency variable, they proved that the optimal energy will be always achieved by using only one frequency if the working frequency of processor is assumed to be continues and that for real processors with a discrete set of working frequencies, the optimal energy is always achieved by using at most two frequencies.

Yang et al. [87] propose a technique for determination of the most energy efficient resource in a heterogeneous platform and the DVFS settings to apply for application execution given a performance requirement. The teqnique uses power/performance model based on hypotheses about the affect of frequency and voltage on the current and latency for each resource.

2.2.3.2 Dynamic Power Management

Each electronic system follows the standard for power management provided by the Advanced Configuration and Power Interface (ACPI) [88]. This standard states that operating systems can configure computer hardware components to perform power management, and perform status monitoring. For instance, they can putt unused or partially unexploited components to sleep, or set on the mode with minimum energy consumption. Dynamic power management (DPM) is a methodology that dynamically configures an electronic system to provide desired performance levels and services with a minimum number of active components in a way of energy efficiency [89].

Bircher et al, [90] investigate the power management of a multi-core processor such as the AMD Quad-Core Opteron and Phenom. Their optimization method considers the effects of the idle core frequency on the performance and power of the active cores. It adjusts the idle frequency of core in a way of the least detrimental effect on the active core performance.

Huang et al. [91] present method which adaptively controls the power mode of the system according to historical data about arrivals of tasks. It uses the earliest deadline first and fixed-priority preemptive scheduling.

Chung [92] propose a DPM scheme that manages power based on the output of adaptive learning trees algorithm that takes as an input the idle period information. The basic idea of this algorithm is the prediction of the future idle periods with high accuracy by observing idle periods in the recent past. This scheme is especially useful for multiple sleep state components.

Beloglazov et al. [93] study power management in data centers. They propose an energy efficient resource management system that reallocates VMs in run-time. They gain energy savings by the distribution of VMs according to virtual network topologies between VMs and thermal state of computing nodes and current resource utilization.

Lee et al. [94] propose a hibrid model for dynamic power management that combines moving average, time delay neural network and random walk model. The idea is to predict idle period based on the on central tendency of the past idle period time series. The main advantage of this model is that it combines

these techniques and as a result yields higher power savings in most cases compared to other models.

Niu et al. [71] propose a model for power management that balances dynamic and leak power consumption under processor activity. This model also uses earliest deadline first strategy. The benefit of this model is that it more precisely estimates the delay for the coming task instances.

Imes et al. [95] study resource allocation strategies for minimization of energy consumption. They demonstrate that different hardware platforms have fundamentally different resource allocation strategies. The investigations show that there are two classes of the systems: a race-to-idle and a never-idle to achieve energy consumption.

2.2.3.3 Component-level Management

In today's computing systems the main memory (DRAM) consumes as much as half of the total system power consumption due to the increasing demand for memory capacity and bandwidth. Hence, it is crucial to pay attention to this component under the energy optimization of the whole system.

Trajkovic et al. [96] propose a technique for reducing the energy consumption of SDRAM. They embedded to the memory controller two buffers: high speedfetch buffer and a write-combine buffer. This adjustment allows DRAM to read prefetching and combined write access to the main memory with reducing in time and energy consumption.

Song et al. [97] present an iso-energy-efficiency model to analyze, evaluate and predict energy-performance of data intensive parallel applications with various execution patterns. Their model helps users to scale system parameters (e.g. processor count, CPU power/frequency, workload size and network bandwidth) to balance energy and performance.

Ahn et al. [98] propose Multicore DIMM, a memory module where DRAM chips are grouped into multiple virtual memory devices. Each group has its own data path and receives separate commands. They demonstrate a simultaneous improvement in memory power, IPC, and system energy-delay product on 22%, 7.6%, and 18% respectively.

Labeck et al. [99] propose a technique that manages DRAM chips to reduce overall power consumption. The technique works by controlling virtual address to physical address mapping such that the physical pages of an application are clustered into a minimum number of DRAM chips and the unused chips are transitioned to low power modes. Then, it can turn off the chips that are in a low power mode. Furthermore, their technique also monitors the time period between accesses to a chip as a metric for measuring the frequency of reuse of a chip. When this time is greater than a certain threshold, the chip is transitioned to the low-power mode.

As DRAM power also depends on operating frequency and supply voltage, the works [100], [101], [102] aim to save DRAM memory power by dynamic voltage and frequency scaling.

The next system-level methods for energy optimization can be grouped into four solution's approaches: thread schedulers, cache partitioners, thermal management and symmetry-aware schedulers.

Thread schedulers

In CMP cores are not independent processors but rather share common resources among cores such as the last level cache (LLC). Shared resource contention can lead to severe and unpredictable performance and energy impact on the CMP system. Hence, thread schedulers attempt to map threads to certain cores by avoiding high contention for the shared resources leading to the improvements in energy efficiency and performance.

Banikazemi et al. [103] introduce the design and implementation of a Performance/power Aware Meta-scheduler called PAM. The scheduler uses performance and power data from performance counters and power monitoring hardware. Based on this information, PAM gives directions to the OS scheduler for remapping software threads to the hardware threads in a way that reduces power and energy consumption.

Merkel et al. [104] analyze energy-delay product scheduling by avoiding resource contention and utilising optimal CPU frequency. They distribute tasks between cores based on their similar characteristics, i.e. run all

memory-bound tasks on core 0 and all compute-bound tasks on core 1 of a dual-core processor.

Petrucci et al. [105] propose an approach for the optimal threads-to-core mapping that reduces energy consumption under the performance requirements and based on an integer linear programming model. The inputs to the model are IPC and LLC miss rates. The output is predicted at runtime the performance of threads. They use the Linux scheduler (CPU affinity) to place a thread on a specified core at runtime.

Qian et al. [106] study I/O threads scheduling in non-volatile memory express based NUMA systems. They found that energy-efficiency penalty is higher due to remote access of NVMe SSDs. To solve this challenge, they developed ESN scheduler that maps the I/O threads to the local or remote socket based on the number of concurrently running I/O threads on a socket.

Hankendi et al. [107] propose a multi-level technique for co-scheduling multiple workloads on a multi-core processor. The method is based on the analyzation of various co-scheduling policies such as cache misses, bus access and instruction-per-cycle. It determines the best co-scheduling policy that reduces energy consumption, depending on the characteristics of the workload sets.

Cache partitioners

The modern multicore architecture allocates some amount of memory (caches) to each core and even the common amount for all cores to reduce access data time and increase their performance. However, in some circumstances, some part of this memory is wasted or suffers from contentions among parallel running tasks that makes cores energy inefficient. The cache partitioning approaches aim to avoid these circumstances.

Wang et al. [108] propose an energy optimization technique that simultaneously uses dynamic reconfiguration of private caches and partitioning of the shared cache for multicore platforms. Dynamic programming with discretization in the energy values is used to find the optimal L1 configurations for each task and L2 partition factors for each core.

Chen et al. [109] present a framework that utilizes cache management for real-time MPSoCs. The framework supports dynamic way-based cache partitioning at hardware level, building time-triggered reconfigurable-cache MPSoCs for each task. It automatically determines timetriggered schedule and cache configuration for each task to improve the system performance while guarantee the realtime constraints.

Delaluz et al. [110] study the energy consumption of translation look-aside buffer (TLB), that used to translate virtual addresses to physical addresses. They especially focus on data TLB (dTLB) and propose solution employs dynamically resizing of it. The idea is to give the application the minimum dTLB size without damaging performance.

Hajimiri et al. [25] propose technique that based on dynamic cache reconfiguration and partitioning to improve performance and energy efficiency in multicore systems. They developed a genetic algorithm to find beneficial IL1/DL1 cache configurations as well as L1/L2 cache partition.

Sundararajan et al. [111] present Cooperative Partitioning, an approach to last-level cache partitioning for reducing both dynamic and static power. They designed "sets-ways" partitioning scheme where the data belonging to each core align along with way across all sets. An average dynamic and static energy savings of 35% and 25% compared to a fixed partitioning scheme.

Thermal management

Because of processor temperature has a significant impact on energy consumption, the thermal management is widely used in the challenge to reduce energy consumption in processors and overall system.

Liu et al. [112] present a thermal optimization framework based on a temperature-aware power model. The model is built on carefully planning DVFS at design time. The energy reduction is gained through optimization of system thermal profile, prevent run-time thermal emergencies and minimize cooling costs.

Huang et al. [113] present a framework for dynamic energy efficiency and temperature management (DEETM) that focuses on both energy efficiency

and temperature control. The framework employs different energy management techniques such as light sleep mode, reducing memory voltage, DVFS, etc.

Skadron et al. [114] propose HotSpot, a practical and computationally efficient approach to modelling thermal behaviour in architecture-level power/performance simulators. HotSpot helps to determine which the hottest microarchitectural units; understand the purpose of different thermal packages on architecture, performance, and temperature; understand the thermal behaviour of programs; and evaluate a number of techniques for regulating on-chip temperature.

Cohen et al. [115] study dynamic thermal management (DTM) strategies based on dynamic voltage scaling (DVS). They show that when the processor works below the limit temperature it is best to start with a high frequency and decrease it exponentially until the limit temperature is reached, and when it works close to the limit temperature, the best tactics is to stay there.

Ayoub et al. [116] propose a joint energy, thermal and cooling management technique (JETC) that dynamically optimizes the energy consumption of server cooling and memory. JETC maximizes the energy efficiency in the server by controlling the number of active memory modules. Furthermore, for the alleviation of the thermal coupling effect of the sockets and their thermal hot spots, JETC schedules the workload between the CPU sockets. Results show the average reduction in energy consumption of memory and cooling subsystems in 50.7%.

Asymmetry-aware schedulers

We incorporate in asymmetry-aware scheduling the schedulers that deal with asymmetric systems by its design feature or hardware faults [117]. For instance, adaptive scheduler for systems with asymmetric memory hierarchies is presented in [118]. The scheduler automatically maps threads to the system hierarchies. It uses AMS-Greedy and AMS-DP mapping algorithms, where the first one performs multiple rounds of cache partitioning and the second finds the optimal schedule given the performance model.

Saez et al. [119] propose a comprehensive scheduling for asymmetric multicore processors (CAMP). The scheduler based on the estimation of the speedup factor gained through the measuring of last-level cache (LLC) miss rate and utility factor. The method employs both efficiency specialization, that guarantees fast core utilization for CPU-intensive application and slow cores for memory-intensive application and thread-level parallelism, that ensures fast cores are used to accelerate sequential phases of parallel applications while leaving slow cores for energy-efficient execution of parallel phases.

Fan et al. [120] present a contention-aware scheduling for a commercial asymmetric multicore processors platform ARM big.LITTLE. The scheduler dynamically picking an appropriate application-to-core mapping in a real asymmetric system. It consists of two stages: an offline stage that builds a performance interference model, and an online stage that schedules a set of applications to the most appropriate core types based on both the speedup factor and the predicted performance interference model.

Li et al. [121] propose AMPS operating system scheduler, that efficiently supports both SMP and NUMA-style performance-asymmetric architectures. It employs three strategies: asymmetry-aware load balancing, fast-core first scheduling and NUMA-aware migration based on predictions of threads overheads. The thread's overhead is defined if the thread provokes a high number of LLC misses after the migration and a large portion of these LLC misses requires remote memory access.

Wang et al. [122] present a VM scheduling that based on the heterogeneity of the core's performance for optimization the overall system energy efficiency. To map VMs to heterogeneous cores they use energy-efficiency factors defined as the ratio performance to system power consumption. This solution can improve the system energy efficiency by 13.5% on average compared to the default Xen scheduler.

Bower et al. [123] study thread scheduling in dynamically heterogeneous multicore processors. In such systems, the scheduler must trace the dynamic knowledge of core status and thread demand. It should not map the thread to the core that has some faults or power wasting.

2.2.4 Application-level Optimization

Application-level optimization aims to optimize application for reducing energy consumption using application-level parameters such as data partitioning, algorithmic cost, communication cost etc.

For instance, Demmel et al. in their work [124] focus on saving energy at the algorithm level. On the example of matrix multiplication and the direct n-body problem, they show the existence of perfect strong scaling in energy. The energy remains constant with increasing the number of processors by some factor that leads to decreasing in runtime by the same factor.

Choi et al. [125] presented an energy-based analogue of the time based roofline model. The model describes algorithms in terms of operations, concurrency, memory traffic, and characterizes the machine based on a small number of simple cost parameters, such as the time and energy costs per operation or per word of communication.

Alessi et al. in their work [126] present OpenMPE that is an extension to OpenMP that deals with the power management of the system. The OpenMPE assists multi-objective goals and constraints and application adaptation, through several techniques such as DVFS, dynamic concurrency throttling (DCT), and application-level content adaptation.

Silva at al. [127] present a method that finds energy-optimal frequency and number of active cores to run single-node HPC applications. Their method employs an application-agnostic power model of the architecture and an architecture-aware performance model of the application. The power model is established by modelling of Complementary Metal-Oxide-Semiconductor (CMOS) logic in the function of the operating frequency. The performance model is based on the application parameters. Its decision parameters are the operating frequency, the number of active cores and the input size.

Wang et al. [128] study energy optimization in a single chip heterogeneous processor (SCHP). They demonstrate that the collective optimization of workload and power partitioning between the CPU and GPU outperforms the optimization of workload partitioning alone under a fixed

power budget allocation to the CPU and GPU on 13%. Furthermore, they designed an effective runtime algorithm for the determination of near-optimal or optimal combinations of workload and power budget partitioning.

2.3 Bi- and Multi-objective Optimization in HPC

This section classifies related literature into two categories. The first category lists a few works done on multi-objective optimization in HPC. The second category focuses on solution methods solving bi-objective optimization problem for performance and energy on HPC platforms.

2.3.1 Multi-Objective Optimization in HPC

ChiŞ et al. [129] introduce an automatic 4D methodology that simultaneously optimizes four objectives using multi-objective optimization tool called FADSE (Framework for Automatic Design Space Exploration). The targeted objectives are the followings: chip area - integration area of the chip must be as small as possible, performance, energy consumption and temperature of the chip. To find the best configuration which optimizes all objectives they varied both hardware (cache associativity and size, number of cores) and software (number of threads and schedulers) parameters.

Subramaniam et al. [130] propose multi-objective optimization techniques for power and energy modeling of the HPL benchmark. They use multi-variable regression with input parameters such as problem size, block size, process mapping, etc. They show that the minimum energy consumption is not always reached at the highest performance.

Sheikh et al. [131] present a dynamic method which simultaneously optimizes performance, energy, and temperature under dynamically varying task and system conditions. Based on the available information of execution times and the system model, they employ SPEA-II, multi-objective evolutionary algorithm (MOEA), to obtain an initial set of a set of Pareto optimal solutions. After, this set of solutions is evolved stage by stage to minimize the deviation of the objective parameters from the Pareto optimal

values. The method shows up to 8% improvement compared to the statically selected schedule. [132]–[134] focus on the energy reduction in data centers by scheduling workload.

2.3.2 Bi-objective Optimization for Performance and Energy

There are methods aiming to optimize several objectives of the system or the environment in computing settings such as cloud computing infrastructures, data centers, etc. The common feature of these methods is to model performance and energy consumption of applications based on heuristic models [135]–[140]. The methods consider various parameters that include number of nodes, computation and communication cost, processor utilization, and DVFS. The dominant decision variable in this category is DVFS.

Freeh et al. [141] study trade-off between energy consumption and performance in high-performance computing applications. Especially they investigate how memory and communication bottlenecks affect power consumption. They show that DVFS clusters can reduce energy consumption in programs that have a memory or communication bottleneck. Furthermore, they found that one can reduce energy consumption and execution time by increasing the number of nodes while reducing the frequency-voltage setting of each node.

Langer et al. [142] analyze bi-objective optimizaton for performance and energy in overprovisioned data centers where nodes are power capped to run below their Thermal Design Power (TDP). They show that there are configurations with a selected number of nodes and power cap that lead to significantly reduced energy consumption with negligible damage to performance.

The other methods predict the performance and energy consumption of applications based on application-level parameters. They can be classified into two categories: (i) Intra-node methods [143]–[148], and (ii) Inter-node and Intra-node methods [35], [149]–[155]. The parameters employed in these methods are memory costs, communication costs, DVFS, execution time,

computation costs, problem size, static power, the number of processors, etc. The key decision variables are DVFS, number of processors, and workload distribution.

In [156] Samee Ullah and Khan study the multi-objective problem of mapping independent tasks onto a set of computational grid machines that simultaneously minimizes the energy consumption and response time. They propose an algorithm based on goal programming that effectively converges to the compromised Pareto optimal solution. The solution is based on DVFS.

Reddy et al. [35] experimentally study the performance and energy profiles of real-life data-parallel applications on state-of-art multicore CPUs and demonstrate that there exist a complex (non-linear and non-convex) relationship between performance and problem size and energy and problem size. They propose an algorithm to solve bi-objective optimization for performance and energy of applications execution on homogeneous multicore platforms where it considers only one decision variable, the workload distribution.

Research works [26], [27], [35] propose model-based data partitioning methods that take as input discrete performance and dynamic energy functions with no shape assumptions. Using a simulation of the execution of a data-parallel matrix multiplication application based on OpenBLAS DGEMM on a homogeneous cluster of multicore CPUs, [26] show that optimizing for performance alone results in average and maximum dynamic energy reductions of 24% and 68%, but optimizing for dynamic energy alone results in performance degradations of 95% and 100%. For a 2D FFT application based on FFTW, the average and maximum dynamic energy reductions are 29% and 55% with 100% average and maximum performance degradations.

2.4 Summary

This section reviewed notable works on performance and energy optimization as well as the bi-objective optimization for performance and energy on modern multicore CPUs. Due to complexities inherent in multicore CPU platforms such as severe resource contention and non-uniform memory

access (NUMA), the optimization for performance and energy on these platforms faced serious challenges. Apart from source code modification that is a time- and labor-consuming process, the load balancing algorithms developed in a single-core era and based on FPMs also became not applicable in today's multicore era due to the newly introduced challenges. Other methods for performance and energy optimization mostly rely on performance data from hardware and software counters, scheduling and DVFS, that is not the focus of this work.

With these issues in mind, this thesis proposes single-objective as well as bi-objective optimization methods for performance and energy of data-parallel applications on modern multicore CPUs that are model-based and use application-level decision variables. In the first method, for one approach workload distribution is used as a decision variable for solving performance issue on modern multicore CPUs. Like [69] proposed for homogeneous multicore CPU platforms,[70] study the same approach for heterogeneous platforms. The works are theoretical and mainly focus on the design of data partitioning algorithms. The purpose of the work in this thesis is not to design new algorithms, but to use the existing algorithms and with their aid develop methods for performance and energy optimizations of data-parallel applications on a single multicore CPU. Another approach along with bi-objective optimization is based on the partitioning of the computation kernel into several groups of threads that are executed in parallel, where the workload is divided equally among them. The decision variables in these methods are the number of threadgroups and the number of threads in each threadgroup. Despite vast amount of literature studying the impact of thread management on the performance and energy [157]–[160], to the best of author's knowledge, there is no similar work that uses application-level decision variables such as the number of threadgroups and the number of threads in each threadgroup, for single- and bi-objective optimizations for performance and energy on a single multicore CPU. The following chapter introduces single-objective optimization methods for performance and energy.

Chapter 3

Novel Single-objective Optimization Methods for Performance and Energy On Modern Multicore CPUs

This chapter starts with overview of the challenges posed by inherent complexities in modern multicore CPUs to performance and energy optimization of data-parallel applications on such platforms. The influence of three-dimensional decision variable space on single-objective optimization of applications for performance and energy on multicore CPUs is studied. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups.

Then, the chapter proposes novel solution methods for optimization of two well-known highly optimized multithreaded scientific routines, matrix-matrix multiplication (DGEMM) and 2D fast Fourier transform (2D-FFT), for performance using only workload distribution as the decision variable.

The chapter is completed by proposing the first application-level method for single-objective optimization of multithreaded data-parallel applications for

performance and energy that uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application and the number of threads in each threadgroup, so that a given workload is partitioned equally between the threadgroups.

3.1 Performance and Energy Optimization on Modern Multicore CPUs: Challenges

In the era of uniprocessors prior to the advent of multicore CPUs, computer users came to expect performance doubling every 18 months due to Moore's law and Dennard scaling. However, achieving the high performance along with energy efficiency of applications on multicore CPUs is not as simple. There are complex challenges that need to be addressed to achieve maximize performance and minimize energy consumption on modern multicore CPUs. They are: a). Severe resource contention due to tight integration of tens of cores organized in multiple sockets with multi-level cache hierarchy and contending for shared on-chip resources such as last level cache (LLC), interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers, b). Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes, and c). Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM). These complexities pose serious challenges to model and algorithm designers to develop highly energy-efficient and prompt codes for HPC data-parallel applications on such platforms.

To elucidate the challenges, we first look at the performance and dynamic energy profiles of 2D-FFT using IMKL on a Intel Haswell server consisting of 36 physical cores (HCLServer3) and a Intel Skylake server consisting of 56 physical cores (HCLServer4). The specification of the servers is shown in table 3.1. The 2D-FFT application computes the 2D discrete Fourier transform of a complex signal matrix of size $N \times N$.

Table 3.1: Specifications of the Intel multicore CPUs, HCLServer01-04, with increasing number of sockets and an increasing number of cores per socket.

Technical Specifications	HCLServer3 (S3)	HCLServer4 (S4)
Processor	Intel Xeon CPU E5-2699	Intel Xeon Platinum 8180
Core(s) per socket	18	28
Socket(s)	2	2
L1d cache, L1i cache	32 KB, 32 KB	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 46080 KB	1024 KB, 39424 KB
Total main memory	256 GB	187 GB
Power meter	-	Yokogawa WT310

Figure 3.1 shows the performance profile of 2D-FFT on dual-socket S3. One can see there are large variations in the performance profile of the application. The variation is related to the difference of speed between two subsequent local minima (s_1) and maxima (s_2) and is defined as: $variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100$. The maximum variation can be up to 24%. The detailed study of performance profiles of the 2D-FFT application using three vendor packages, FFTW-2.1.5, FFTW-3.3.7 and IMKL FFT, can be found in the Appendix A, where also is show that the FFT routines in the packages demonstrate low average performance due to these variations.

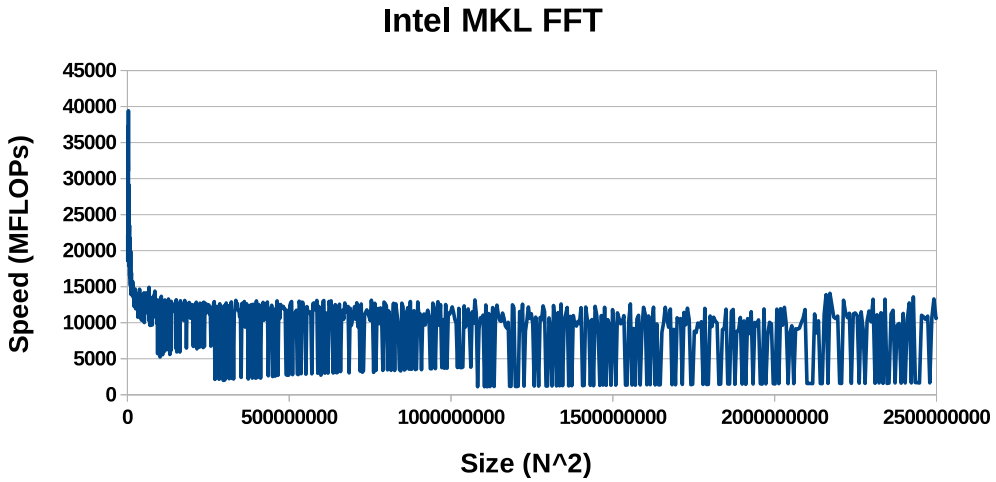


Figure 3.1: Speed function of IMKL FFT application executing with 36 cores on the Intel Haswell server.

Figure 3.2 depicts the dynamic energy consumption profile of 2D-FFT on S4. The dynamic energy consumption is measured with Yokogawa WT310

power meter. The maximum variation can be up to 73% that represents the maximum amount of energy savings possible.

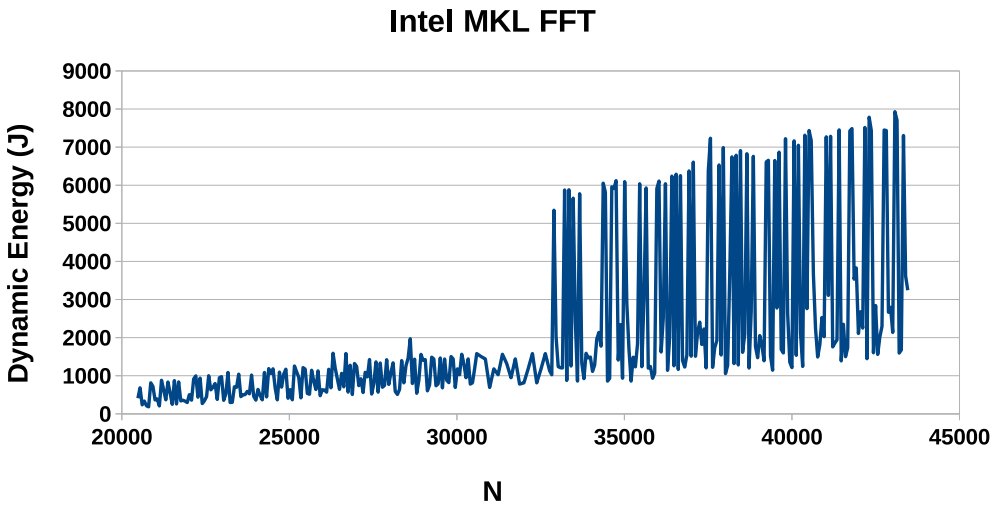


Figure 3.2: Dynamic Energy Consumption of IMKL FFT application executing with 56 cores on the Intel Xeon Platinum server.

To make sure the experimental results are reliable and not noise, a statistical methodology described in Appendix B is used. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%). Hence, these variations are consequences of tight integration which has resulted in the cores contending for various shared on-chip resources such as Last Level Cache (LLC) and interconnect (NUMA).

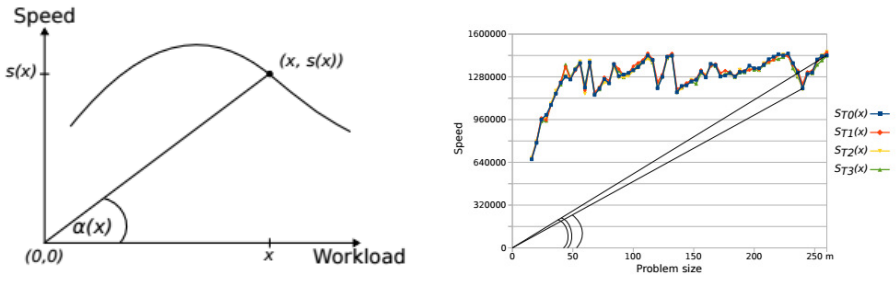
Next, it is described why such performance and energy profiles pose daunting challenges to state-of-the-art load balancing methods.

State-of-the-art load balancing methods use functional performance model (FPM) of the given application [42]–[44]. This model represents the processor speed by a function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application. These FPMs capture accurately the real-life behavior of applications executing on nodes consisting of uniprocessors (single-core CPUs). The assumptions require them to be

smooth enough in order to guarantee that optimal solutions minimizing the computation time are always load balanced. However, the profiles of real applications executed on multicore CPUs are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions (Figure 3.1, 3.2). As described in [161] the balanced configuration of an application will execute faster than any unbalanced configuration but assumes that the speed function $s_i(x)$ satisfies the condition:

$$\forall x > 0 : \frac{s_i(x)}{x} \geq \frac{s_i(x + \Delta x)}{x + \Delta x} \tag{3.1}$$

As can be seen from Figure 3.3 (left graph) angle $\alpha(x)$ between the straight line, connecting the point $(0, 0)$ and the point $(x, s(x))$ on the speed curve, and the x-axis will be inversely proportional to the execution time of the workload of size x by the processor. Indeed, the cotangent of this angle is directly proportional to the ratio $\frac{x}{s(x)}$ representing the execution time of the workload x. Note, that the graph represents the function of several processes in parallel like on the right graph. The behaviour of function of these processes is identical. Larger angles correspond to shorter execution times. We can see that if we give different workloads to processes then the parallel execution time will be represented by a smaller angle - longer execution time. Thus, the condition 3.1 means that the increase of the workload, x, will never result in the decrease of the execution time, or equivalently in the increase of the angle $\alpha(x)$.



(a) (b)

Figure 3.3: a). Intuition behind load balancing. b). Load imbalancing.

However, the performance and energy profiles of real-life scientific

applications on modern parallel platforms (Fig. 3.3 (right graph)) may significantly deviate from the conditions, which guarantee that load balancing will always optimize their computational performance. For instance, we can see, that the angle α, that represents the computation time, for the certain size m is less than the angle α for the size $m + \Delta m$. It means that the task of a smaller size is computed longer than the task of a bigger size that does not satisfy the condition 3.1 and disrupts the theory of load balancing. Hence, these variations are the main reason why traditional load balancing may not work on modern multicore CPUs.

Furthermore, reducing energy consumption is of paramount concern to the HPC community since its pervasiveness in data centers and cloud computing infrastructures. Energy in HPC is now an environment concern not only because of the maintenance cost of HPC systems but also of high carbon footprint which affects environmental sustainability as modern data centers already can rival cities in power consumption. This was not an issue in the past since until now we have followed Moore's Law enhancements in photolithography techniques which are proportional reductions in dynamic power consumption per transistor and consequent improvements in clock frequency at the same level of power dissipation. However, below 90 nm, the static power dissipation dominates over the dynamic power dissipation. This effect summons clock frequency freezing in order to stay within thermal power emission limits [24].

To address these challenges, this chapter proposes single-objective optimization methods for performance and energy that employ application-level parameters, functional performance model of applications and its analysis. Next, we look at the application-level parameters used in the methods and their importance.

Workload Distribution as a Decision Variable

The importance of workload distribution is depicted in the example above (Figure 3.3), where small uneven distribution can improve the whole outcome. Thus, the methods for performance optimisation proposed in this thesis use

the workload distribution as a decision variable. For finding the best workload distribution they use algorithms proposed in [162] and [43]. The algorithms obtain as input the performance profiles of application from each processor or device on which data-parallel application is executed and return the optimal workload distribution. Should be noticed that the optimal solutions returned by these new algorithms may not be balanced in terms of execution time.

Number of Threadgroups and Threads Per Group as Decision Variables

It is well known that vendors of highly optimized packages of scientific data-parallel applications (OpenBLAS DGEMM, IMKL DGEMM and FFT, FFTW) offer the possibility of varying threads to gain the maximum possible computation performance on multicore CPUs. Due to contention for shared resources, reducing or increasing the number of threads positively affects both performance and energy. The application also plays a role here. This thread management was studied in a vast amount of literature [157]–[160]. The main decision variables they use are the number of threads, the number of nodes, problem size, etc, however, no one performs the optimization of both performance and energy on a single multicore CPU as it is done in this work, using threadgroups as a decision parameter. It was experienced by the author that in some circumstances, it is not enough to vary only the number of threads for the improvement of energy efficiency and performance. The partitioning of execution kernel into several groups named threadgroups executed in parallel can yield more performance and less energy consumption in comparison with thread varying alone. As an example, using this approach, the speedup can be up to 80% with energy reduction of 35% for the application such as 2D fast Fourier transform. The author believes this observation makes a newly introduced decision variable, thredgroups, crucial in the optimization of data-parallel applications on modern multicore CPUs.

3.2 Performance Optimization Using Workload Distribution as a Decision Variable

This section presents novel optimization methods for performance optimization of the data-parallel applications on modern multicore CPUs using FPMs and workload distribution.

3.2.1 Load Imbalancing Using Uneven Workload Distribution

The main idea of workload distribution can be defined as follows: assume we want to design and implement a parallel version of the application of workload size $N \times N$, where $N = 24704$. It can be executed using, for example, two identical abstract processors named threadgroups in parallel. In this scenario, the workload is load balanced between threadgroups, i.e. each threadgroup has the same amount of data, $N \times M$, where $M = \frac{N}{2}$. However, this distribution is not guaranteed to be the best, hence to determine the best distribution, we build the profile of execution time against 2D problem size $N \times M$ in each threadgroup. For this, we keep one dimension constant, here N is constant, and increasing M in each threadgroup with a constant step of Δx, where $\Delta x \leq M \leq 24704$. In the end, we obtain performance profile of each threadgroup as depicted in figure 3.4. As we can see, if one threadgroup receives workload of size $N \times x_1$ and the other $N \times x_2$, where $x_1 + x_2 = 24704$ and $x_1 < x_2$, then the speed of this load imbalanced application is higher than that for load balanced with workload sizes $N \times \frac{N}{2}$. Hence, this section inroduces novel performance optimization methods specifically designed for 2D parallel FFT (PFFT-FPM and PFFT-FPM-PAD) and parallel matrix-matrix multiplication (PMM-FPM). The methods employ workload distribution as the decision variable and are based on model-based parallel computing technique using load-imbalancing data partitioning. The technique determines optimal solutions (workload distributions) that may not load-balance the application in terms of execution time. The methods take as

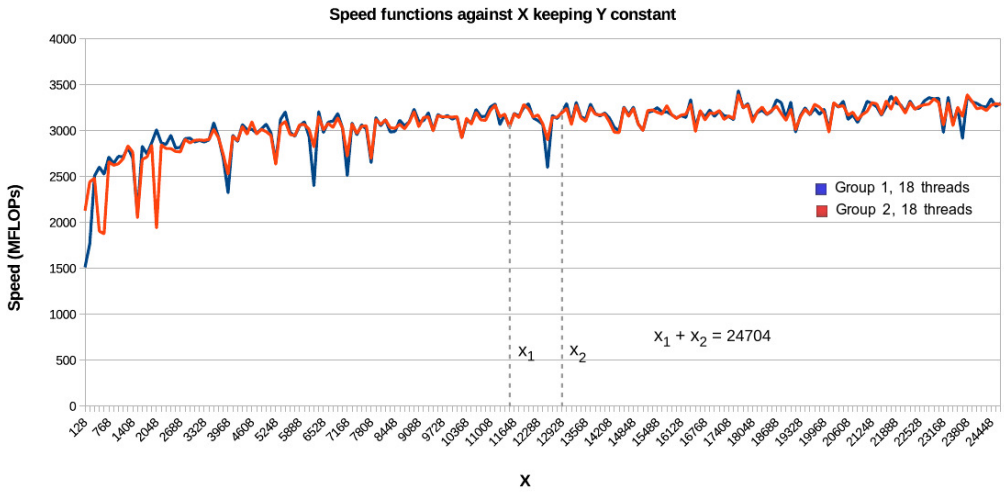


Figure 3.4: Each curve is the speed of one threadgroup.

inputs the discrete functions of the performance of the threadgroups against 2D problem size $N \times M$. FFT-FPM-PAD is different from FFT-FPM only that it uses padding determined from FPMs along with the optimal data distribution. Next, the application of these methods is shown in the example of real-life data-parallel applications.

3.2.2 PFFT-FPM Employing 2D Fast Fourier Transform

This section starts with description of the sequential 2D-FFT algorithm using the row-column decomposition method. Then it explains the parallel 2D-FFT algorithm based on the sequential 2D-FFT algorithm that uses load balancing technique. Finally, it proposes new single-objective optimization method for performance that uses load imbalancing and FPMs.

3.2.2.1 Sequential 2D-FFT Algorithm

The sequential algorithm computes the DFT on a two-dimensional point discrete signal $\mathcal{M}$ of size $N \times N$. $\mathcal{M}$ is the signal matrix where each element $\mathcal{M}[i][j]$ is a complex number. The definition of 2D-DFT of $\mathcal{M}$ is below:

$$\mathcal{M}[k][l] \; = \; \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \mathcal{M}[i][j] \times \omega_N^{ki} \times \omega_N^{lj}, \omega_N \; = \; e^{-\frac{2\pi}{N}}, 0 \; \leq \; k, l \; \leq \; N-1$$

The total number of complex multiplications required to compute the 2D-DFT is $\Theta(N^4)$. The row-column decomposition method reduces this complexity by computing the 2D-DFT using a series of 1D-DFTs, which are implemented using a fast 1D-FFT algorithm. The method consists of two phases called the row-transform phase and column-transform phase. Figure 3.5 depicts the method, which is mathematically summarized below:

$$\mathcal{M}[k][l] = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \mathcal{M}[i][j] \times \omega_N^{ki} \times \omega_N^{lj} = \sum_{i=0}^{N-1} \omega_N^{ki} \times (\sum_{j=0}^{N-1} \mathcal{M}[i][j] \times \omega_N^{lj})$$

$$= \sum_{i=0}^{N-1} \omega_N^{ki} \times (\tilde{\mathcal{M}}[i][l]) = \sum_{i=0}^{N-1} (\tilde{\mathcal{M}}[i][l]) \times \omega_N^{ki}, \omega_N = e^{-\frac{2\pi}{N}}, 0 \leq k, l \leq N-1$$

It computes a series of ordered 1D-FFTs on the N rows of x. That is, each row i (of length N) is transformed via a fast 1D-FFT to $\tilde{X}[i][l], \forall l \in [0, N-1]$. The total cost of this row-transform phase is $\Theta(N^2 \log_2 N)$. Then, it computes a series of ordered 1D-FFTs on the N columns of $\tilde{X}$. The column l of $\tilde{X}$ is transformed to $X[k][l], \forall k \in [0, N-1]$. The total cost of this column-transform phase is $\Theta(N^2 \log_2 N)$.

Therefore, by using the row-column decomposition method, the complexity of 2D-FFT is reduced from $\Theta(N^4)$ to $\Theta(N^2 \log_2 N)$.

3.2.2.2 PFFT-LB: Parallel 2D-FFT Algorithm Using Load Balancing

The parallel 2D-FFT algorithm is based on the sequential 2D-FFT row-column decomposition method and is executed using p threadgroups, $\{P_1, ..., P_p\}$. To aid clear exposition, we assume N is divisible by p. The rows of the complex matrix x are partitioned equally between the p threadgroups where each threadgroup gets $\frac{N}{p}$ rows. The other input to the algorithm is the signal matrix $\mathcal{M}$. The output from the algorithm is the transformed signal matrix $\mathcal{M}$. All the FFTs discussed in this work are considered in-place.

PFFT-LB consists of four steps:

Step 1. 1D-FFTs on rows: Threadgroups P_i executes sequential 1D-FFTs on rows $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 2. Matrix Transposition: Transpose the matrix $\mathcal{M}$.

Step 3. 1D-FFTs on rows: Threadgroups P_i executes sequential 1D-FFTs on rows $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 4. Matrix Transposition: Transpose the matrix $\mathcal{M}$.

The computational complexity of Steps 1 and 3 is $\Theta(\frac{N^2}{p} \log_2 N)$. The computational complexity of Steps 2 and 4 is $\Theta(\frac{N^2}{p})$. Therefore, the total computational complexity of PFFT-LB is $\Theta(\frac{N^2}{p} \log_2 N)$.

The algorithm is illustrated in the Figure 3.5.

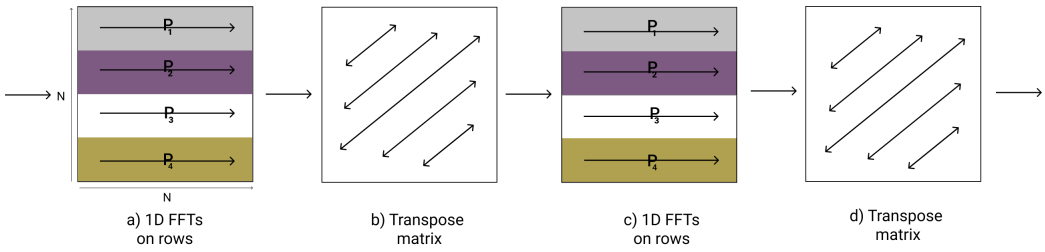


Figure 3.5: PFFT-LB performing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$ ($N = 16$) using four threadgroups. Each threadgroup gets four rows each. (a). Each threadgroup performs series of row 1D-FFTs locally indicated by solid arrows. (b). Matrix $\mathcal{M}$ is transposed. (c). Each threadgroup performs series of row 1D-FFTs locally indicated by solid arrows. (d). Matrix $\mathcal{M}$ is transposed again. It is the output of PFFT-LB.

3.2.2.3 PFFT-FPM: Performance Optimization Using FPMs and Load Imbalancing

This section describes novel optimization method called PFFT-FPM that takes functional performance models (FPMs) as input and that employs load imbalancing parallel computing technique.

PFFT-FPM is executed using p threadgroups, $\{P_1, ..., P_p\}$. The inputs to PFFT-FPM are the number of available threadgroups, p, the number of rows of the signal matrix, N, the speed functions of the threadgroups, $\mathcal{S}$, and the

user-input tolerance ϵ. The output from PFFT-FPM is the transformed signal matrix $\mathcal{M}$.

The discrete speed function of threadgroup P_i is given by $\mathcal{S}_i = \{s_i(x_1, y_1), ..., s_i(x_m, y_m)\})$ where $s_i(x, y)$ represents the speed of execution of x number of 1D-FFTs of length y by the threadgroup P_i. The speed is equal to $\frac{5.0 \times x \times y \times \log_2(y)}{t}$, where t is the time of execution of x number of 1D-FFTs of length y.

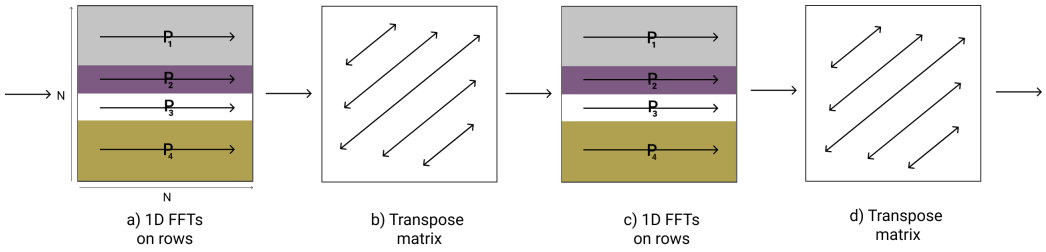


Figure 3.6: PFFT-FPM performing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$ ($N = 16$) using four threadgroups. Each threadgroup gets different number of rows given by the data distribution, $d = \{5, 3, 3, 5\}$. (a). Each threadgroup performs series of row (padded row) 1D-FFTs locally indicated by solid arrows. (b). Matrix $\mathcal{M}$ is transposed. (a). Each threadgroup performs series of row (padded row) 1D-FFTs locally indicated by solid arrows. (d). Matrix $\mathcal{M}$ is transposed again. It is the output of PFFT-FPM

It consists of following main steps:

Step 1. Partition rows:

1a. Plane intersection of speed functions: Speed functions $\mathcal{S}$ are sectioned by the plane $y = N$. A set of p curves on this plane are produced which represent the speed functions against variable x given parameter y is fixed.

1b. Are speed functions identical?: if $\exists(x_k, N), 1 \leq k \leq m, \quad (\frac{\max_{i=1}^{p} s_i(x_k,N) - \min_{i=1}^{p} s_i(x_k,N)}{\min_{i=1}^{p} s_i(x_k,N)} > \epsilon)$, go to Step 1d. Otherwise, go to Step 1c. If there exists a (x_k, N), the speed functions are not considered identical. To determine if the speed functions are identical, the difference between the maximum and minimum speeds for a point (x_k, N) is calculated and compared with tolerance ϵ.

1c. Partition rows using POPTA: Construct a speed function $S_{avg} =$

$\{s_{avg,i}(x)\}, \forall i \in [1, m]$, where $s_{avg,i}(x) = \frac{p}{\sum_{j=1}^{p} \frac{1}{s_j(x,N)}}$. Each speed $s_{avg,i}(x)$ in the function is the average of the speeds $\{s_1(x, N), \cdots, s_p(x, N)\}$. POPTA [68] is then invoked using this speed function as an input to obtain an optimal distribution of the rows, d.

1d. Partition rows using HPOPTA: HPOPTA [70] is invoked using the p speed curves as input to obtain an optimal distribution of the rows, d.

Step 2. 1D-FFTs on rows: Threadgroup P_i executes sequential 1D-FFTs on its rows given by $\{\sum_{k=1}^{i-1} d[i] + 1, \cdots, \sum_{k=1}^{i} d[i]\}$.

Step 3. Matrix Transposition: Transpose the matrix $\mathcal{M}$.

Step 4. 1D-FFTs on rows: Same as Step 2.

Step 5. Matrix Transposition: Same as Step 3.

The method is illustrated in the figure 3.6 for four threadgroups solving 2D-DFT of size $N \times N (N = 16)$.

The data partitioning algorithms POPTA and HPOPTA are described in detail in Lastovetsky et al. [68] and Khaleghzadeh et al. [70]. Briefly, POPTA determines the optimal data distribution for minimization of time for the most general performance profiles of data parallel applications executing on homogeneous multicore clusters. One of its inputs is a speed function of the processors involved in its execution since they are considered identical. HPOPTA is the extension of POPTA for heterogeneous clusters of multicore processors. The inputs to it are the S different speed functions of the p processors involved in its execution. Unlike load balancing algorithms, these algorithms output optimal solutions that may not load-balance an application in terms of execution time. The output from the data partitioning algorithms is the data distribution of the rows, $d = \{d_1, \cdots, d_p\}$.

Figures 3.7, 3.8 illustrate the data partitioning algorithm employed in PFFT-FPM for two threadgroups solving 2D-DFT of size $N \times N$ where $N = 24704$ using IMKL FFT on a Intel multicore server. The speed functions shown are segments of the full functions (given in Appendix C). Each threadgroup consists of 18 threads. Figure 3.7 shows a plane $y = N = 24704$ intersecting the two speed functions $\mathcal{S} = \{S_1, S_2\}$ producing two curves, one for each group showing speed versus x given $y = N = 24704$ constant (Figure 3.7). One can see that the two curves are not identical

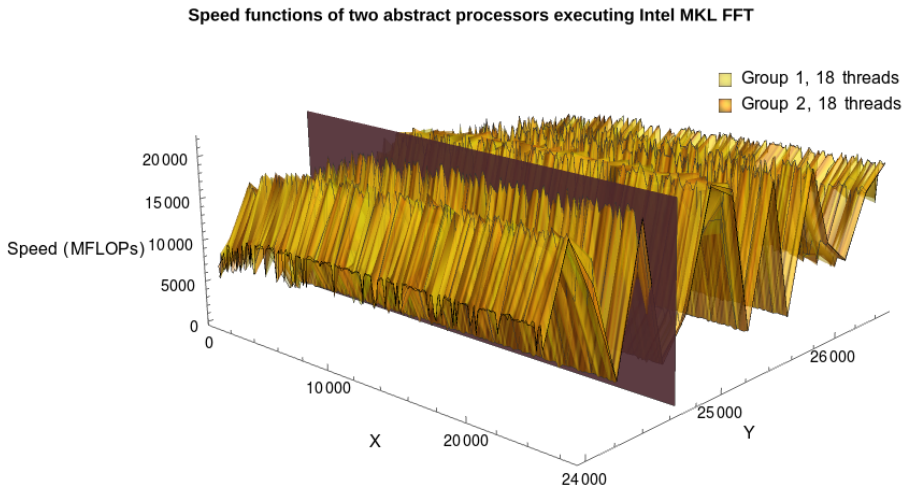


Figure 3.7: Speed functions of two threadgroups, each a group of 18 threads. Each group executes 2D-DFT of size $x \times y$ using IMKL FFT on a Intel multicore server consisting of two sockets of 18 cores each. The plane $y = N = 24704$ intersects the speed functions.

(heterogeneous). That is, there are points where the speeds differ from each other by more than 5% ($\epsilon = 0.05$). That means the speed functions are inputted to HPOPTA, which determines the optimal partitioning of rows, $(d[1], d[2]) = (11648, 13056)$, where each row is of length $N = 24704$.

3.2.2.4 Shared Memory Implementation of PFFT-FPM

This section describes shared memory implementations of PFFT-FPM.

The inputs to the implementation are the signal matrix $\mathcal{M}$ of size $N \times N$, the number of threadgroups p, the speed functions represented by a set $\mathcal{S}$ containing problem sizes and speeds, and number of threads in each threadgroup represented by t. The output is the transformed signal matrix $\mathcal{M}$ (considering that we are performing in-place FFT).

Algorithm 1 shows the pseudocode of the algorithm. The first step (Line 2) is to determine the partitioning of rows by invoking the routine PARTITION. The routine PARTITION can be found in Appendix C. This routine gives the array of data distribution, $d = \{d_1, \cdots, d_p\}$ based on user-input tolerance ϵ. If all the variations are less than or equal to ϵ, the data partition algorithm

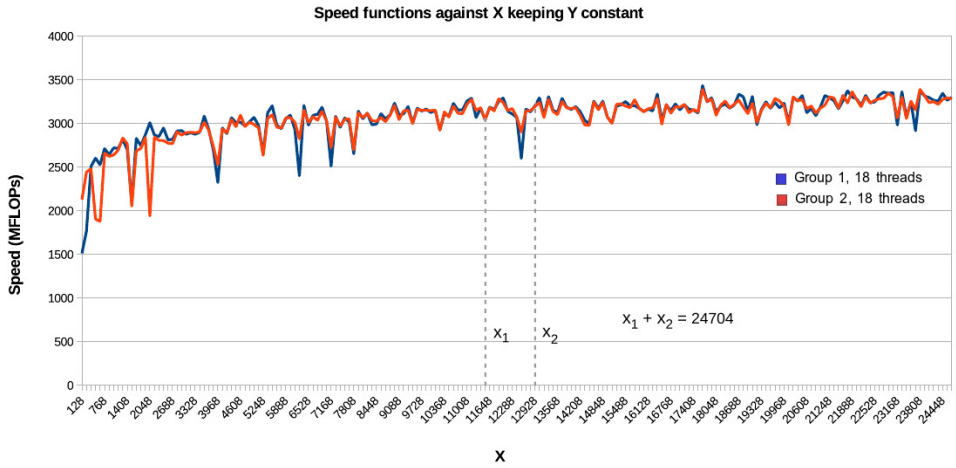


Figure 3.8: Each intersection produces two curves for the two threadgroups showing speed versus x keeping $y = N = 24704$. Application of HPOPTA *to determine optimal distribution of rows provides the partitioning, $(d[1] = x_1 = 11648, d[2] = x_2 = 13056)$.*

POPTA [68] determinates the data distribution. If all the variations are more than ϵ, HPOPTA [70] determinates the data distribution.

Then the routine PFFT_LIMB (Line 3) is invoked to execute the basic steps 1-4 of PFFT-LB. These are series of row 1D-FFTs computed in parallel (Algorithm 2, Lines 2-4), parallel transpose (Line 5), series of row 1D-FFTs computed in parallel (Lines 6-8), and parallel transpose (Line 9).

Each threadgroup performs the series of row 1D-FFTs locally using the routine *1D_ROW_FFTS_LOCAL*. The number of row 1D-FFTs performed by threadgroup P_i is given by first argument, d_i. Algorithm 3 illustrates the implementation of this routine using FFTW interface.

3.2.3 PFFT-FPM-PAD Employing 2D FFT

This section presents PFFT-FPM-PAD, an extension of PFFT-FPM where the partitions (problem sizes) are padded (extended) by lengths determined from the FPMs. The inputs and the outputs of this method are the same as those for PFFT-FPM. The data partitioning algorithms invoked in PFFT-FPM-PAD are the same as those employed in PFFT-FPM. But the series of 1D-FFTs are

Algorithm 1 Parallel algorithm computing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$ employing functional performance models (FPMs).

1: **procedure** PFFT-FPM($N, \mathcal{M}, p, \mathcal{S}, t$)
Input:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$
Number of threadgroups, $p \in \mathbb{Z}_{>0}$
Functional performance model (speed functions) represented by,
$\mathcal{S} = \{S_1, ..., S_p\}$,
$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$
User tolerance, $\epsilon \in \mathbb{R}_{>0}$
Output:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $d \leftarrow Partition(N, p, \mathcal{S}, \epsilon, d)$
3: $pfft\_limb(p, d, N, \mathcal{M})$
4: **return** $\mathcal{M}$
5: **end procedure**

performed locally on rows whose length is extended (padded with zeroes) by an extent determined from the FPM of the threadgroup. The determination of the length of padding is a local computation and is specific to the threadgroup. That is, the lengths can be different for different threadgroups. In some cases, there is no necessity for padding and therefore the length of the padding is zero.

PFFT-FPM-PAD consists of following main steps:

Step 1. Partition rows: This step is the same as that for the Algorithm PFFT-FPM.

Step 2. 1D-FFTs on padded rows: Threadgroup P_i executes sequential 1D-FFTs on its rows in $\mathcal{M}$ given by $d[i]$.

The length of each row N is padded to N_{padded}. It is determined as follows using the FPM, $\mathcal{S}_i = s_i(x, y)$:

$$N_{padded} = \underset{\mathcal{V} \in N y_m}{\arg\min} \left(\frac{d[i] \times \mathcal{V}}{s_i(d[i], \mathcal{V})} < \frac{d[i] \times N}{s_i(d[i], N)} \right)$$

The argument $\mathcal{V}$ ranges from problem size y_{N+1} to y_m in the speed function $s(x, y)$. The ratio $\frac{x \times y}{s_i(x,y)}$ gives the execution time of problem size $x \times y$. Essentially we select the point (problem size) $(d[i], y_{opt})$ in the range $\{(d[i], y_{N+1}), ..., (d[i], y_m)\}$ that has better execution time than the point

Algorithm 2 Parallel algorithm computing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$.

1: **procedure** PFFT_LIMB($p, d, N, \mathcal{M}$)
Input:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$
Number of threadgroups, $p \in \mathbb{Z}_{>0}$
Output:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $Par\ (proc \leftarrow 1, p)$
3: $1D\_ROW\_FFTS\_LOCAL(proc, d_{proc}, N, \mathcal{M})$
4: $End\ Par$
5: $Parallel\_Tranpose(\mathcal{M})$
6: $Par\ (proc \leftarrow 1, p)$
7: $1D\_ROW\_FFTS\_LOCAL(proc, d_{proc}, N, \mathcal{M})$
8: $End\ Par$
9: $Parallel\_Tranpose(\mathcal{M})$
10: **return** $\mathcal{M}$
11: **end procedure**

$(d[i], N)$. N_{padded} is set to the problem size y_{opt}. If no such point is found, the padding length is set to 0 and N_{padded} will be equal to N. The elements in the padded region $\mathcal{M}[*, c], \forall c \in [y_{N+1}, N_{padded}]$ are set to 0.

Step 3. Matrix Transposition: The matrix $\mathcal{M}$ (excluding the padded region) is transposed.

Step 4. 1D-FFTs on padded rows: The lengths of the paddings already determined in Step 2 are reused. Threadgroup P_i executes sequential 1D-FFTs on its padded rows.

Step 5. Matrix Transposition: Same as Step 3.

All the steps of PFFT-FPM-PAD are the same as PFFT-FPM except the determination of the lengths of the paddings. Figures 3.9, 3.10 illustrate how they are determined from the FPMs for two threadgroups solving 2D-DFT of size $N \times N$ where $N = 24704$ using IMKL FFT on a Intel multicore server. The speed functions shown are segments of the full functions (given in Appendix C). Each threadgroup consists of 18 threads. Figure 3.9 shows two planes $x_1 = 11648$ and $x_2 = 13056$ (that is the optimal distribution) intersecting the two speed functions $\mathcal{S} = \{S_1, S_2\}$ producing two curves, one for each group showing speed versus y keeping x constant (Figure 3.10). The padded lengths $(N_{padded,1}, N_{padded,2})$ corresponding to x_1 and x_2 are

Algorithm 3 Series of x row 1D-FFTs using FFTW interface function fftw_plan_many_dft.

1: **procedure** 1D_ROW_FFTS_LOCAL($id, x, N, \mathcal{M}, flag$)
Input:
Threadgroup identifier, $id \in \mathbb{Z}_{>0}$
Problem size $x \in \mathbb{Z}_{>0}$
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$
Output:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $rank \leftarrow 1$; $howmany \leftarrow x$; $s \leftarrow N$;
3: $idist \leftarrow N$; $odist \leftarrow N$; $istride \leftarrow 1$;
4: $ostride \leftarrow 1$; $inembed \leftarrow s$; $onembed \leftarrow s$;
5: $plan \leftarrow fftw\_plan\_many\_dft(rank, s, howmany,$
$\mathcal{M}, inembed, istride, idist,$
$\mathcal{M}, onembed, ostride, odist,$
$FFTW\_FORWARD, FFTW\_ESTIMATE)$
6: $fftw\_execute(plan)$
7: $fftw\_destroy\_plan(plan)$
8: **return** $\mathcal{M}$
9: **end procedure**

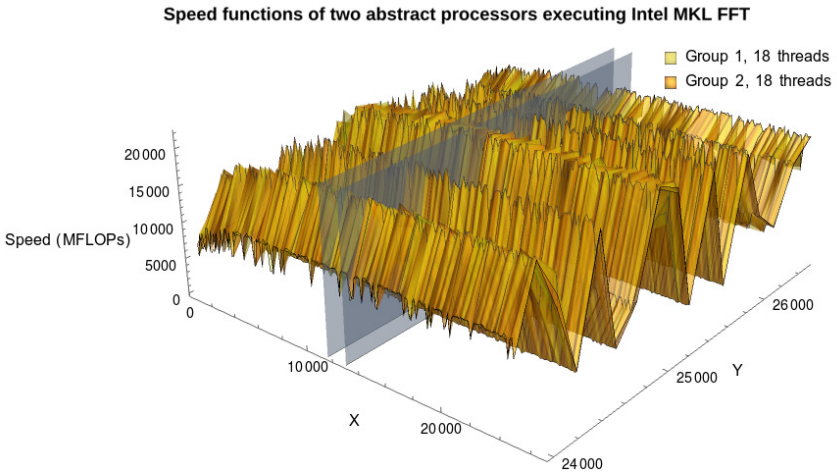


Figure 3.9: Speed function for threadgroup1 intersected by the plane $x_1 = 11648$. *Speed function for threadgroup2 intersected by the plane* $x_2 = 13056$.

determined from the curves and are equal to 24960.

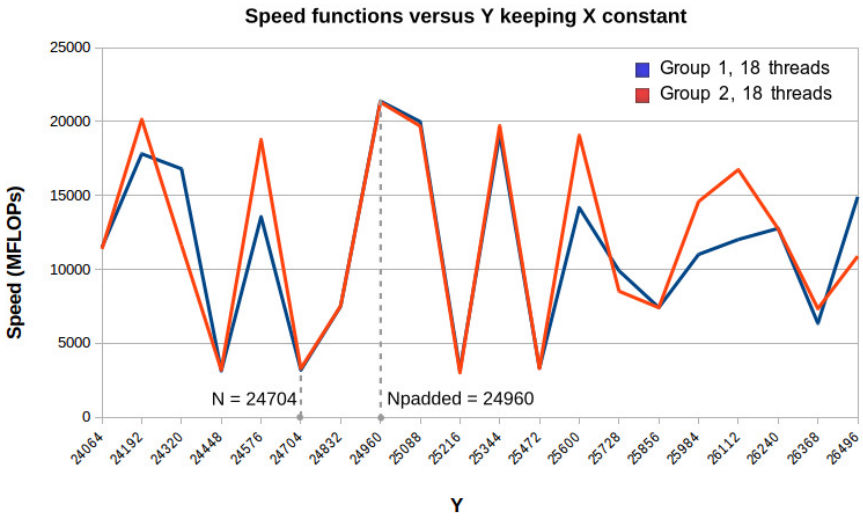


Figure 3.10: Each intersection produces a curve for the threadgroup showing speed versus y keeping x constant. The lengths of padding for the two threadgroups, $Npadded$, is the same and is equal to 24960.

3.2.3.1 Shared Memory Implementation of PFFT-FPM-PAD

The implementations of PFFT-FPM-PAD are similar to those for PFFT-FPM except that the routine *1D_ROW_FFTS_LOCAL_PADDED* determines the length of the padding from the FPMs using the function *Determine_Pad_Length* before executing the series of row 1D-FFTs.

Each threadgroup performs the series of padded row 1D-FFTs locally using the routine 1D_ROW_FFTS_LOCAL_PADDED. The number of row 1D-FFTs performed by threadgroup P_i is given by first argument, d_i. Algorithm 4 illustrates the implementation of this routine using FFTW interface.

The shared memory implementation of PFFT-FPM and PFFT-FPM-PAD that specifically designed for FFTW and IMKL FFT can be found in Appendix C.

Algorithm 4 Series of x row 1D-FFTs using FFTW interface function fftw_plan_many_dft. Each row is padded to N_{padded}.

1: **procedure** 1D_ROW_FFTS_LOCAL_PADDED($id, x, N, \mathcal{M}$)
Input:
Threadgroup identifier, $id \in \mathbb{Z}_{>0}$
Problem size $x \in \mathbb{Z}_{>0}$
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$
Functional performance model (speed functions) represented by,
$\mathcal{S} = \{S_1, ..., S_p\}$,
$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$
Output:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $N_{padded} \leftarrow Determine\_Pad\_Length(id, x, N, \mathcal{S})$
3: $rank \leftarrow 1$; $howmany \leftarrow x$; $s \leftarrow N_{padded}$;
4: $idist \leftarrow N_{padded}$; $odist \leftarrow N_{padded}$; $istride \leftarrow 1$;
5: $ostride \leftarrow 1$; $inembed \leftarrow s$; $onembed \leftarrow s$;
6: $plan \leftarrow fftw\_plan\_many\_dft(rank, s, howmany,$
$\mathcal{M}, inembed, istride, idist,$
$\mathcal{M}, onembed, ostride, odist,$
$FFTW\_FORWARD, FFTW\_ESTIMATE)$
7: $fftw\_execute(plan)$
8: $fftw\_destroy\_plan(plan)$
9: **return** $\mathcal{M}$
10: **end procedure**

3.2.4 PMM-FPM Employing Parallel Matrix Multiplication

This section starts with the description of load-balanced multithreaded algorithm (PMM-LB) computing the matrix product ($C = \alpha \times A \times B + \beta \times C$) of two dense square matrices A and B of size $N \times N$. Then, the PMM-FPM algorithm that uses imbalancing technique for performance optimization is proposed. The algorithms are executed using p threadgroups, $P_1, ..., P_p$. The input matrices A, B, and C are assumed to be stored in threadgroup P_1. To simplify the exposition of the algorithms, we assume N to be divisible by p.

3.2.4.1 PMM-LB Using Horizontal Decomposition and Load Balancing

The matrices A and C are partitioned horizontally such that each threadgroup is assigned $\frac{N}{p}$ of the rows of A and C as shown in the figure 3.11. In matrices A and C, each threadgroup is assigned a sub-matrix of size $\frac{N}{p} \times N$.

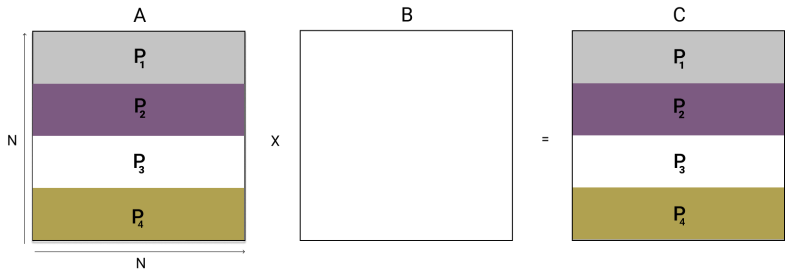


Figure 3.11: PMM-LB: Matrix B is replicated at all the threadgroups. Matrices A and C of size $(N \times N)$, $N = 16$, are horizontally partitioned among the threadgroups. Each threadgroup receives the same number of rows $\frac{N}{p} = 4$.

The main steps of the algorithm are:

Step 1. Replicate B: The matrix B is replicated to all the threadgroups.

Step 2. Scatter A and C: The horizontal partitions of A and C are scattered to the threadgroups as depicted in the Figure 3.11.

Step 3. Matrix Multiplication: Each threadgroup P_i computes its horizontal partition C_{P_i} using the matrix product, $C_{P_i} = \alpha \times A_{P_i} \times B + \beta \times C_{P_i}$.

Step 4. Gather C: All the computed horizontal partitions C_{P_i} are gathered at threadgroup P_1 into the result, C.

3.2.4.2 PMM-FPM Using Horizontal Decomposition, FPMs and Load Imbalancing

This section describes the novel optimization method called PMM-FPM that takes functional performance models (FPMs) as input and that employs load imbalancing parallel computing technique.

PMM-FPM is executed using p threadgroups, $\{P_1, ..., P_p\}$. The inputs to PMM-FPM are the number of threadgroups, p, the number N of rows of matrices A, B and C, the speed functions of the threadgroups, $\mathcal{S}$, and the user-input tolerance ϵ. The output of PMM-FPM is the resulting matrix C.

The discrete speed function of threadgroup P_i is given by $\mathcal{S}_i = \{s_i(x_1, N), ..., s_i(x_m, N)\})$ where $s_i(x, N)$ represents the speed computation of a block of size $x \times N$ of the resulting matrix C by threadgroup P_i. The speed is calculated as $\frac{2.0*N*x*N*e^{-6}}{t}$, where t is measured.

The method consists of the following main steps:

Step 1. Partition rows of matrices A and C:

1a. Plane intersection of speed functions: Speed functions $\mathcal{S}$ are sectioned by the plane $y = N$. A set of p curves on this plane are produced which represent the speed functions against variable x given parameter y is fixed.

1b. Are speed functions identical?: if $\exists(x_k, N), 1 \leq k \leq m, \quad (\frac{\max_{i=1}^{p} s_i(x_k,N) - \min_{i=1}^{p} s_i(x_k,N)}{\min_{i=1}^{p} s_i(x_k,N)} > \epsilon)$, go to Step 1d. Otherwise, go to Step 1c. If there exists a (x_k, N), the speed functions are not considered identical. To determine if the speed functions are identical, the difference between the maximum and minimum speeds for a point (x_k, N) is calculated and compared with tolerance ϵ.

1c. Partition rows using POPTA: Construct a speed function $S_{avg} = \{s_{avg,i}(x)\}, \forall i \in [1, m]$, where $s_{avg,i}(x) = \frac{p}{\sum_{j=1}^{p} \frac{1}{s_j(x,N)}}$. Each speed $s_{avg,i}(x)$ in the function is the average of the speeds $\{s_1(x, N), \cdots, s_p(x, N)\}$. POPTA [68] is then invoked using this speed function as an input to obtain an optimal distribution of the rows, d.

1d. Partition rows using HPOPTA: HPOPTA [70] is invoked using the p speed curves as input to obtain an optimal distribution of the rows, d.

Step 2. Replicate B: The matrix B is replicated to all the threadgroups.

Step 3. Scatter A and C: The horizontal partitions of A and C are scattered to the threadgroups as depicted in the Figure 3.12.

Step 4. Matrix Multiplication: Each threadgroup P_i computes its horizontal partition C_{P_i} using the matrix product, $C_{P_i} = \alpha \times A_{P_i} \times B + \beta \times C_{P_i}$.

Step 5. Gather C: All the computed horizontal partitions C_{P_i} are gathered at threadgroup P_1 into the result, C.

The method is illustrated in the figure 3.12 for four threadgroups computing $C_{P_i} = \alpha \times A_{P_i} \times B + \beta \times C_{P_i}$ of size $N \times N$.

Figures 3.13, 3.14 illustrate the data partitioning algorithm employed in PMM-FPM for four threadgroups solving $C = \alpha \times A \times B + \beta \times C$ of size $N \times N$ where $N = 18176$ consisting of 18 threads each. Figure 3.13 shows a plane $y = N = 18176$ intersecting the four speed functions $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$

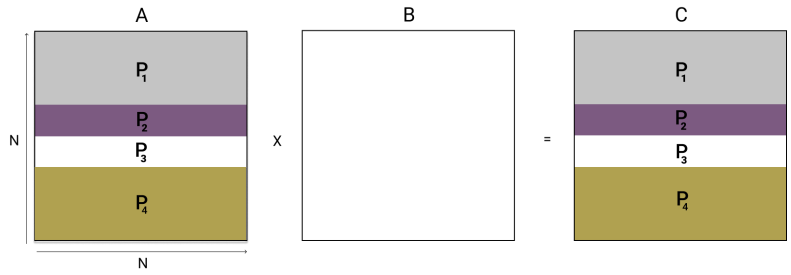


Figure 3.12: PMM-LB: Matrix B is replicated at all the threadgroups. Matrices A and C of size $(N \times N)$, $N = 16$, are horizontally partitioned among the threadgroups. Each threadgroups receives the different number of rows given by the data distribution, $d = \{5, 3, 3, 5\}$.

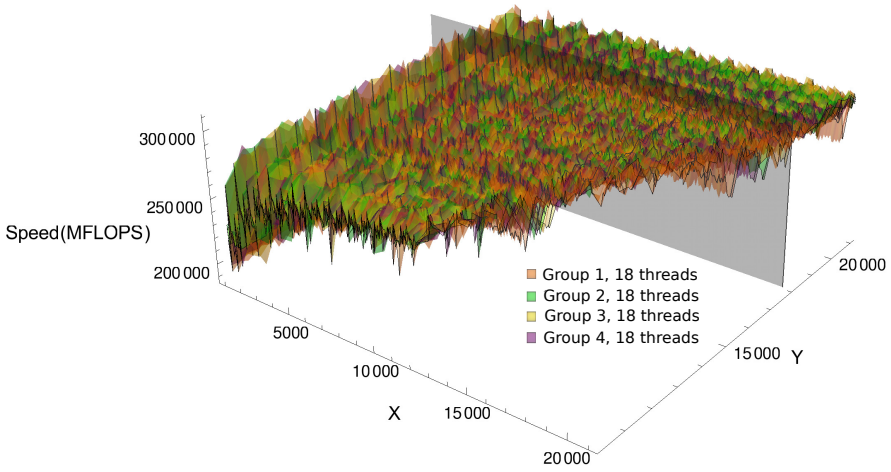


Figure 3.13: Speed functions of four threadgroups, each a group of 18 threads. Each group executes matrix product $C = \alpha \times A \times B + \beta \times C$ of size $N \times N$ of size $x \times y$ on a Intel multicore server consisting of two sockets of 18 cores each. The plane $y = N = 18176$ intersects the speed functions.

producing four curves, one for each threadgroups showing speed versus x given $y = N = 18176$ constant (Figure 3.14). One can see that the curves are almost identical (homogeneous). The speeds differ from each other by less than 5% ($\epsilon = 0.05$). That means we average these four functions and the resulting average function input to POPTA algorithm, which determines the optimal partitioning of rows, $(d[1], d[2], d[3], d[4]) = (4352, 4608, 4608, 4608)$, where each row is of length $N = 18176$.

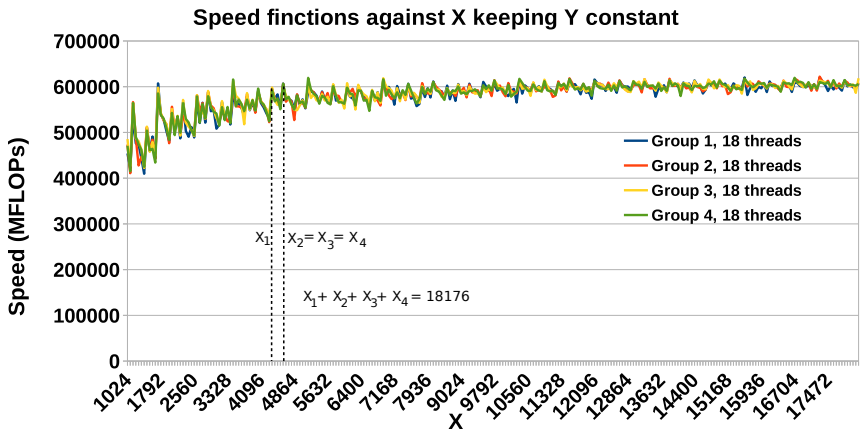


Figure 3.14: Each intersection produces four curves for the four threadgroups showing speed versus x keeping $y = N = 18176$ constant. Application of POPTA to determine optimal distribution of rows provides the partitioning, $(d[1] = x_1 = 4352, d[2] = x_2 = d[3] = x_3 = d[4] = x_4 = 4608)$.

3.2.4.3 Shared Memory Implementation of PMM-FPM

The algorithm is similar to that for FFT. The inputs to the implementation are matrices A, B and C of size $N \times N$, the number of threadgroups p, the speed functions represented by a set $\mathcal{S}$ containing problem sizes and speeds, and number of threads in each threadgroup represented by t. The output is the product of two dense matrices A and B - matrix C. The first step (Algorithm 5, Line 2) is to determine the partitioning of rows by invoking the routine PARTITION (Appendix C). This routine gives the array of data distribution, $d = \{d_1, \cdots, d_p\}$ based on user-input tolerance ϵ. If all the variations are less than or equal to ϵ, the data partition algorithm POPTA [68] determinates the data distribution. If all the variations are more than ϵ, HPOPTA [70] determinates the data distribution.

Algorithm 5 Parallel algorithm computing product ($C = \alpha \times A \times B + \beta \times C$) of two dense square matrices A and B of size $N \times N$ employing functional performance models (FPMs).

1: **procedure** PMM-FPM($N, A, B, C, p, \mathcal{S}, t$)

Input:

A, B and C are matrices of size $N \times N, N \in \mathbb{Z}_{>0}$

Number of threadgroups, $p \in \mathbb{Z}_{>0}$

Functional performance model (speed functions) represented by,

$\mathcal{S} = \{S_1, ..., S_p\}$,

$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$

User tolerance, $\epsilon \in \mathbb{R}_{>0}$

Number of threads, t

Output:

matrix C - the product of two dense square matrices A and B of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $d \leftarrow Partition(N, p, \mathcal{S}, \epsilon, d)$

3: $pmm\_limb(p, d, N, A, B, C)$

4: **return** C

5: **end procedure**

Then the routine PMM_LIMB (Line 3) is invoked to execute the basic steps 1-4 of PMM-LB. This is the replication of matrices (Algorithm 6, Lines 2-4), dgemm call (Line 5-7), product gathering (Lines 8-10).

Algorithm 6 Parallel algorithm computing product of two matrices A and B of size $N \times N$.

1: **procedure** PMM_LIMB(p, d, N, A, B, C)

Input:

A,B and C, matrices of size $N \times N, N \in \mathbb{Z}_{>0}$

Number of threadgroups, $p \in \mathbb{Z}_{>0}$

Output:

C, product matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $Par\ (proc \leftarrow 1, p)$

3: $\quad MEMCPY(proc, d_{proc}, N, A, C)$

4: $End\ Par$

5: $Par\ (proc \leftarrow 1, p)$

6: $\quad DGEMM(proc, d_{proc}, N, A, B, C)$

7: $End\ Par$

8: $Par\ (proc \leftarrow 1, p)$

9: $\quad MEMCPY(proc, d_{proc}, N, A, C)$

10: $End\ Par$

11: **return** C

12: **end procedure**

The shared memory implementation of PMM-FPM that specifically designed for OpenBLAS DGEMM and IMKL DGEMM can be found in Appendix C.

3.2.5 Experimental Analysis

This section presents experimental results that demonstrate the performance improvements provided by PFFT-FPM, PFFT-FPM-PAD and PMM-FPM. All experiments were performed on an Intel Haswell server containing of 36 physical cores (Table 3.2). To build each point in a function, a statistical methodology described in Appendix B was used. Briefly, for each data point in the speed functions, the procedure executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%). For this purpose, Student's t-test is used but assuming that the individual observations are independent and their population follows the normal distribution.

Technical Specifications	Intel Haswell Server
Processor	Intel Xeon CPU E5-2699 v3 @ 2.30GHz
OS	CentOS 7.1.1503
Microarchitecture	Haswell
Memory	256 GB
Core(s) per socket	18
Socket(s)	2
NUMA node(s)	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	46080 KB
NUMA node0 CPU(s)	0-17,36-53
NUMA node1 CPU(s)	18-35,54-71

Table 3.2: Specification of the Intel Haswell server used to construct the performance profiles.

3.2.5.1 PFFT-FPM and PFFT-FPM-PAD Using FFTW and IMKL FFT

For FFT, two packages, FFTW-3.3.7 and IMKL FFT, were used for the implementations of the methods. FFTW-2.1.5 is not used since the implementation of series of row 1D-FFTs is poor using fftw_threads

compared to the implementation of fftw_plan_many_dft in FFTW-3.3.7 and IMKL FFT.

The FFTW-3.3.7 package is installed with multithreading, SSE/SSE2, AVX2, and FMA (fused multiply-add) optimizations enabled. For IMKL FFT, there are not used any special environment variables. The experiments were performed with three planner flags, FFTW_ESTIMATE, FFTW_MEASURE, FFTW_PATIENT. The experimental results are shown for only one planner flag, FFTW_ESTIMATE. The execution times for these flags however are prohibitively larger compared to FFTW_ESTIMATE (Table 3.3) and severe variations are present. The long execution times are due to the lengthy times to create the plans because FFTW_MEASURE tries to find an optimized plan by computing several FFTs whereas FFTW_PATIENT considers a wider range of algorithms to find a more optimal plan.

N	FFTW_ESTIMATE (Sec)	FFTW_MEASURE (Sec)	PATIENT (Sec)
20160	3	31	5015
20480	16	41	2549
20672	6.5	3004	8228
21120	3.6	31	2746
21440	4	32	1367
21632	14.5	2937	9754

Table 3.3: Execution times in seconds for FFTW-3.3.7 on the Intel Haswell multicore server for three different planner flags.

For the implementations using FFTW-3.3.7, the combination consisting of 4 groups of 9 threads each is used, ($p = 4, t = 9$) since this pair performs the best among the following combinations: $\{(2, 18), (4, 9), (6, 6), (9, 4), (12, 3)\}$. For IMKL FFT, the combination consisting of 2 groups of 18 threads each is used, ($p = 2, t = 18$) since this is the best combination found experimentally among the following combinations: $\{(2, 18), (4, 9), (6, 6), (9, 4), (12, 3)\}$.

The full speed functions constructed for IMKL FFT and FFTW-3.3.7 are shown in the Appendix C. The set of problem sizes (x, y) used for the construction of speed functions is $\{(x, y) \mid 128 \leq x \leq y, 128 \leq y \leq 64000, x \bmod 128, y \bmod 128\} = \{128 \times 128, 128 \times 256, 256 \times 256, \cdots, 64000 \times 64000\}$. All threadgroups build a data point $((x, y), s_i(x, y))$ in their speed functions simultaneously. That is, all of them execute the same problem size $x \times y$ in parallel to determine the

speed $s_i(x, y)$ in their speed functions. For large problem sizes (for example: $\{(x, y) \mid 128 \leq x \leq 64000, y = 64000\}$), all the data points (x, y) can not be built due to main memory constraint. Therefore, the speed functions are constructed until permissible problem size.

The time to build the full speed functions can be expensive. This takes into account the fact that for each data point, statistical averaging is performed to determine its sample mean. One approach is to build partial speed functions [163],[164]. These are input to the data partitioning algorithm [68], which would return sub-optimal workload distributions (but better than load balanced solution) to be used in PFFT-FPM and PFFT-FPM-PAD. To build a partial speed function, data points in the neighbourhood of homogeneous distribution, $d_i = \frac{n}{p}, \forall i \in [1, p]$, are constructed until the allowed user-input execution time is exceeded.

To demonstrate the performance improvements of the solutions determined by PFFT-FPM and PFFT-FPM-PAD, the average and maximum speedups over to the basic FFT versions (that employ one group of 36 threads in their execution) are used. The speedup is determined as follows: Speedup $= \frac{t_{basic}}{t_{pfft-fpm-pad}}$, where where t_{basic} is the execution time obtained using the basic FFT version (IMKL FFT or FFTW-3.3.7) and $t_{pfft-fpm-pad}$ is the execution time obtained using PFFT-FPM-PAD.

FFTW

Figure 3.15 shows the execution times of PFFT-FPM and PFFT-FPM-PAD versus basic FFTW-3.3.7. One can see that for problem sizes $N > 33000$, while the speedups are still good (6x for FFTW-3.3.7), major variations still remain. However, for basic implementation these variations start from the size $N = 17000$.

Figure 3.16 shows the speedups of PFFT-FPM and PFFT-FPM-PAD over basic FFTW-3.3.7 which computes the 2D-DFT using one group consisting of 36 threads. Each data point in the speed functions involves a complex 2D-DFT of size $N \times N$. The average and maximum performance improvements for PFFT-FPM-PAD are 2.3x and 9.4x, and for PFFT-FPM are 2x and 6.8x.

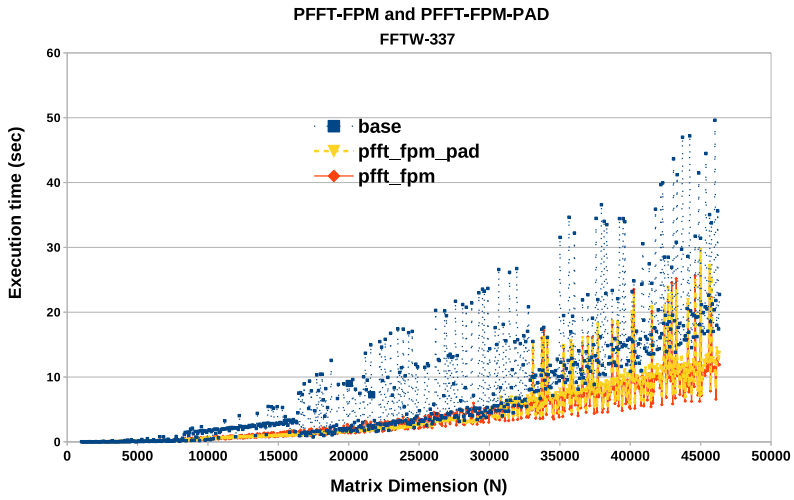


Figure 3.15: Execution times of PFFT-FPM and PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

Furthermore, we can see that for $N < 31000$, the PFFT-FPM-PAD outperforms PFFT-FPM, however, for $N > 31000$ their performance is practically the same.

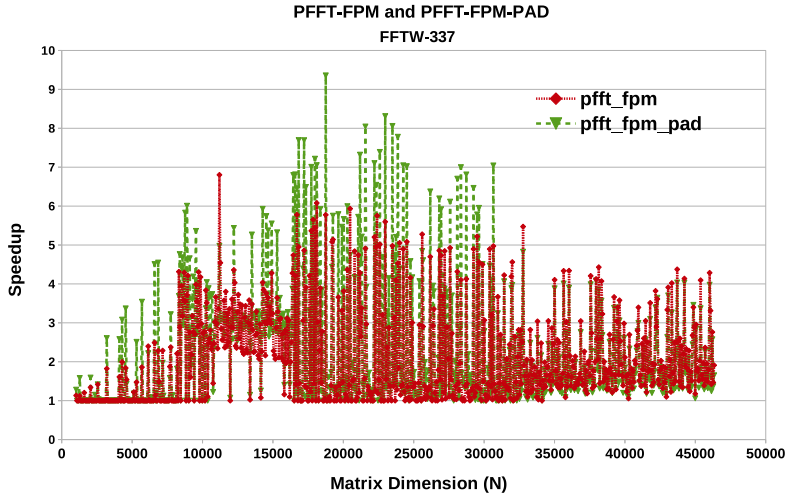


Figure 3.16: Speedup of PFFT-FPM and PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

IMKL FFT

Figure 3.17 shows the execution times of PFFT-FPM and PFFT-FPM-PAD versus basic IMKL FFT. One can see that for problem sizes $N > 33000$, while the speedups are still good (2x for IMKL FFT), the variations are still significant.

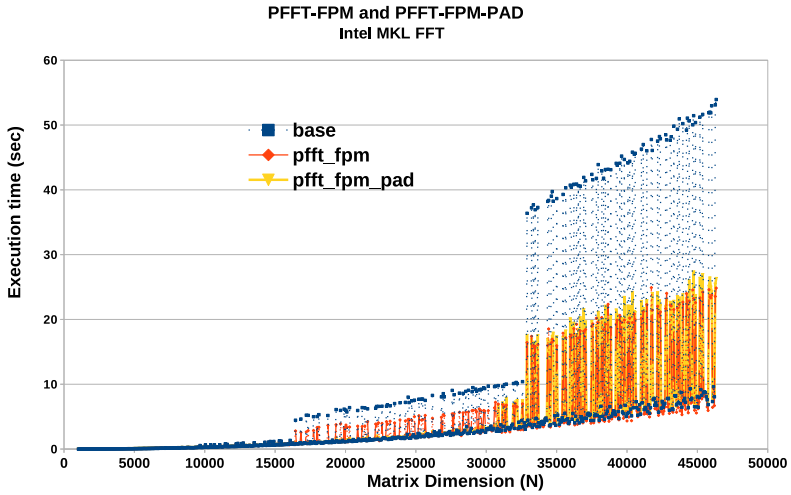


Figure 3.17: Execution times of PFFT-FPM and PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

Figure 3.18 compares the speedups PFFT-FPM and PFFT-FPM-PAD over basic IMKL FFT which computes the 2D-DFT using one group consisting of 36 threads. The average and maximum speedups for PFFT-FPM-PAD are 1.4x and 5.9x, and for PFFT-FPM are 1.1x and 2.4x. Also, it can be seen clearly that in the range of problem sizes ($10000 \leq N \leq 30144$) the PFFT-FPM-PAD outperforms PFFT-FPM, however, outside this range their performance is always the same.

3.2.5.2 PMM-FPM Using OpenBLAS DGEMM and IMKL DGEMM

For implementation of matrix-matrix multiplication, two packages are used, OpenBLAS DGEMM that is part of OpenBLAS-0.2.19 and IMKL DGEMM that is part of Intel Math Kernel Library v.2017. The default settings were used under installations and compilations of packages.

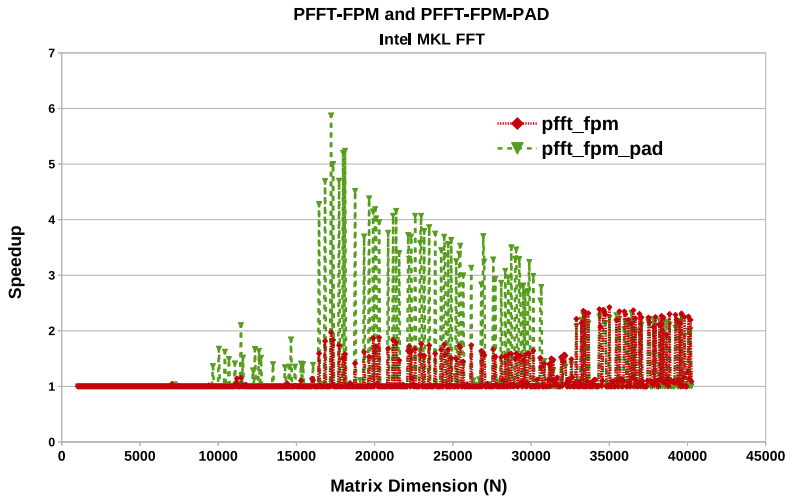


Figure 3.18: Speedups of PFFT-FPM and PFFT-FPM-PAD against the basic IMKL FFT executed using 36 threads.

For the implementations using OpenBLAS DGEMM, the combination consisting of four groups of 9 threads each is used, ($p = 4, t = 9$) since this pair performs the best among the following combinations: $\{(2, 18), (4, 9), (6, 6), (9, 4), (12, 3)\}$. For the IMKL DGEMM, the combination consisting of two groups of 18 threads each is used, ($p = 2, t = 18$) since this is the best combination found experimentally among the following combinations: $\{(2, 18), (4, 9), (6, 6), (9, 4), (12, 3)\}$.

The set of problem sizes (x, y) used for the construction of speed functions is $\{(x, y) \mid 10240 \leq x \leq y, 10240 \leq y \leq 20992, x \bmod 64, y \bmod 64\} = \{10240 \times 10240, 10240 \times 10304, 10304 \times 10304, \cdots, 20992 \times 20992\}$. All threadgroups build a data point $((x, y), s_i(x, y))$ in their speed functions simultaneously. That is, all of them execute the same problem size $x \times y$ in parallel to determine the speed $s_i(x, y)$ in their speed functions.

OpenBLAS DGEMM

Figure 3.19 shows the execution times of PMM-FPM versus basic OpenBLAS DGEMM. The average and maximum performance improvements are 17%

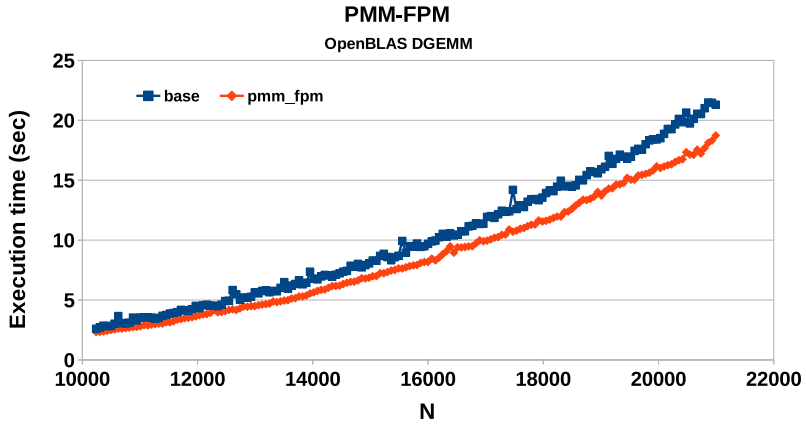


Figure 3.19: Execution times of PMM-FPM against the basic OpenBLAS DGEMM executed using 36 threads.

and 28.5%. Furthermore, the method removes noticeable drops in performance for sizes $N = 10624, N = 12608, N = 13952$ and $N = 17472$.

IMKL DGEMM

Figure 3.20 shows the execution times of PMM-FPM versus basic IMKL DGEMM. The average and maximum performance improvements are 11% and 27%. Our method removes noticeable drops in performance for sizes $N = 12288, N = 14336, N = 16384, N = 18432$ and $N = 20480$.

3.2.6 Summary

This section proposed three novel methods for performance optimization of data-parallel applications namely matrix-matrix multiplication and 2D-FFT on modern multicore CPUs. The methods employ workload distribution as the decision variable and are beased on model-based parallel computing technique using load-imbalancing data partitioning. The experimental results obtained on a modern Intel Haswell multicore server consisting of two sockets of 18 physical cores each, demonstrated the superiority of the methods over the base implementations of those applications.

However, the methods have some limitations. First, they were not well

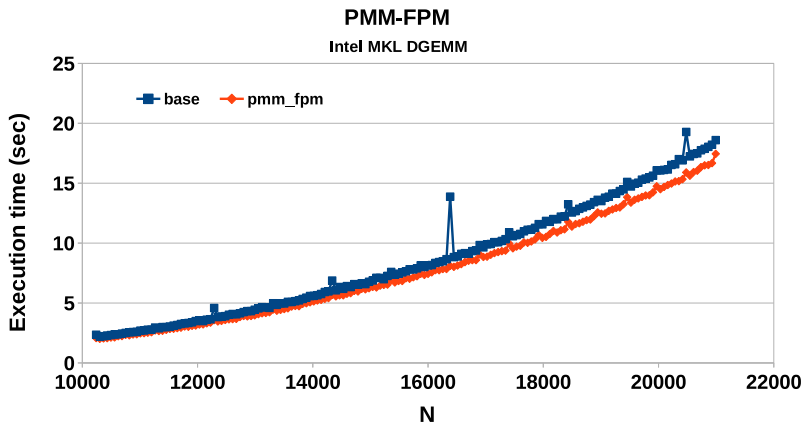


Figure 3.20: Execution times of PMM-FPM against the basic IMKL DGEMM executed using 36 threads.

studied on a single-socket multicore CPUs. This study is an object for future work. Second, they are not applicable for energy optimization on a single multicore CPUs as it is not straightforward to measure the energy profiles of each team of cores which is a subset of cores making one socket. Usually, on modern multicore CPUs, the energy is measured only on a socket level and a whole system. Hence, we cannot build the energy profiles of each team of cores involved in the implementation of the methods to find the best partitioning in term of energy consumption among them. This challenge is left for future work while now the following section proposes a method that uses the number of threadgroups and the number of threads in each threadgroup as decision variables and able to deal with the optimization of both performance and energy.

3.3 Performance and Energy Optimization Using Threadgroups and Threads per Group as Decision Variables

This section introduces the solution method, SOPPETG, for solving the single-objective optimization problem of a multithreaded data-parallel application on

multicore CPUs for performance and energy (SOPPE).

3.3.1 Solution Method Using Threadgroups and Threads as Decision Variables (SOPPETG)

SOPPETG uses two decision variables, the number of identical multithreaded kernels (threadgroups) and the number of threads in each threadgroup. A given workload is always partitioned equally between the threadgroups.

The inputs to the solution method are the workload size of the multi-threaded data-parallel application, n; the number of logical cores in the multicore CPU, l; the multithreaded kernel, $mtkernel$; the base power of the multicore CPU platform, P_b. The outputs are the optimal number of threadgroups, g_{opt} and the optimal number of threads per group, t_{opt}.

The main steps of SOPPETG are as follows:

Step 1. Parallel implementation configurable using (g,t): Design and implement a parallel version of the application that can be executed using g identical multithreaded kernels ($mtkernel$) in parallel. Each kernel is executed by a threadgroup containing t threads. The application should essentially allow its runtime configuration using number of threadgroups and number of threads per group with the workload equally partitioned between the threadgroups.

Step 2. Initialize g and t: All the application configurations, (g,t), where the product, $g \times t$, is less than or equal to the total number of logical cores (l) in the multicore platform are considered. $g \leftarrow 1$, $t \leftarrow 1$. Go to Step 3.

Step 3. Determine time and dynamic energy of (g,t): For each (g,t) combination, the starting execution time (t_i) is measured. The total energy consumption (e_i) is set to 0 J. This is followed by execution of the application, which is g identical multithreaded kernels, $mtkernel$, executed in parallel where each kernel employs t threads. The workload n is divided equally between the threadgroups g during the execution of the application. The ending execution time (t_f) and the total energy consumption (e_f) are measured. Go to Step 4.

Step 4. Determine the best configuration (g,t) for time or energy: The execution time and dynamic energy consumption of the application are

determined as follows: $s_o = t_f - t_i$, $e_o = e_f - e_i - P_b \times s_o$. The temporary variable S_{opt} stores the optimal solution, (s_o, e_o). Depends on the requirement, wich is the minimum execution time or dynamic energy consumption, the S_{opt} is updated. Go to Step 5.

Step 5. Test and Increment (g,t): If $t < l$, $t \leftarrow t + 1$, go to Step 3. Set $g \leftarrow g+1, t \leftarrow 1$. If $g \times t \leq l$, go to Step 3. Else return the optimal configuration (g_{opt}, t_{opt}).

In the following section, the application of SOPPETG to two data-parallel applications, matrix-matrix multiplication and 2D fast Fourier transform is illustrated. It is shown in particular how SOPPETG can reuse highly optimized scientific kernels with careful design and development of parallel versions of the application.

3.3.2 Parallel Matrix-Matrix Multiplication Using SOPPETG

This section presents the single-objective optimization of matrix-matrix multiplication using SOPPETG (PMMTG).

The PMMTG application computes the matrix product ($C = \alpha \times A \times B + \beta \times C$) of two dense square matrices A and B of size $N \times N$. The application is executed using p threadgroups, $\{P_1, ..., P_p\}$. To simplify the exposition of the algorithms, we assume N to be divisible by p.

There are three parallel algorithmic variants of PMMTG. In PMMTG-V, the matrices B and C are partitioned vertically such that each threadgroup is assigned $\frac{N}{p}$ of the columns of B and C as shown in the figure 3.21a. Each threadgroup P_i computes its vertical partition C_{P_i} using the matrix product, $C_{P_i} = \alpha \times A \times B_{P_i} + \beta \times C_{P_i}$. In PMMTG-H, the matrices A and C are partitioned horizontally such that each threadgroup is assigned $\frac{N}{p}$ of the rows of B and C as shown in the figure 3.21b. Each threadgroup P_i computes its horizontal partition C_{P_i} using the matrix product, $C_{P_i} = \alpha \times A_{P_i} \times B + \beta \times C_{P_i}$. In PMMTG-S, the p threadgroups $\{P_1, ..., P_p\}$ are arranged in a square grid $Q_{st}, s \in [1, \sqrt{p}], t \in [1, \sqrt{p}]$. The matrices A, B, and C are partitioned into equal squares among the threadgroups as shown in the figure 3.21c. In each matrix, each threadgroup $P_i (= Q_{st})$ is assigned a

sub-matrix of size $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$ and computes its square partition $C_{Q_{st}}$ using the matrix product, $C_{Q_{st}} = \alpha \times \sum_{k=1}^{\sqrt{p}} (A_{sk} \times B_{kt}) + \beta \times C_{Q_{st}}$. A_{sk} is the square block in matrix A located at (s, k). B_{kt} is the square block in matrix B located at (k, t). To exclude the decision variable such as form of partitioning, the performance of the best configuration for each form is compared in Appendix C. It is found that the difference between them is less than 3%, hence, all experiments are performed based on the horizontal form of partitioning.

The shared memory implementations of PMMTG-H based on OpenBLAS DGEMM and IMKL DGEMM are described in Appendix C.

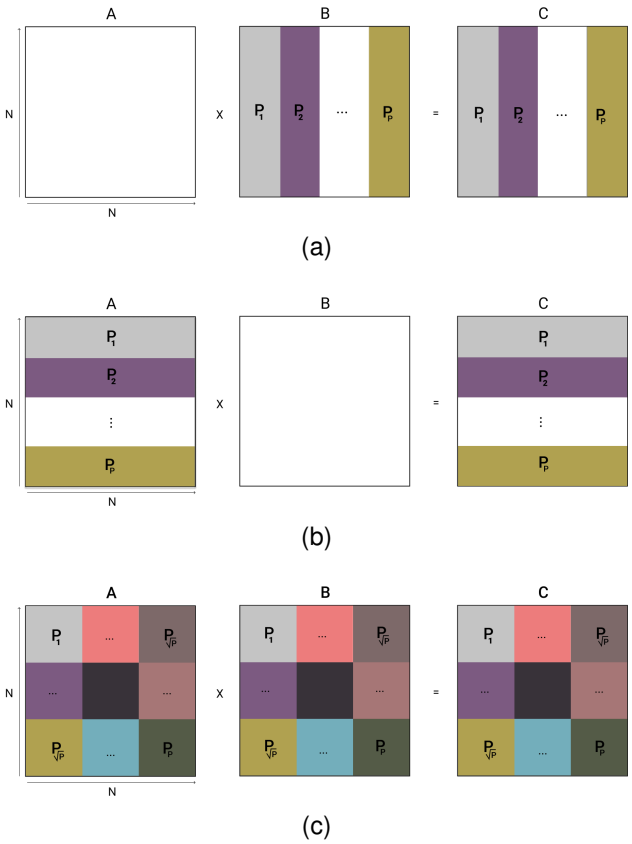


Figure 3.21: (a). PMMTG-V: Matrices B and C are vertically partitioned among the threadgroups. (b). PMMTG-H: Matrices A and C are horizontally partitioned among the threadgroups. (c). PMMTG-S: The p threadgroups are arranged in a square grid of size $\sqrt{p} \times \sqrt{p}$. All the matrices are partitioned into squares among the threadgroups.

3.3.3 2D Fast Fourier Transform Using SOPPETG

The PFFTTG application employing our solution method computes the 2D-DFT of the signal matrix of size $N \times N$ using p threadgroups, $\{P_1, ..., P_p\}$. It is based on the sequential 2D-FFT row-column decomposition method. There are two parallel algorithmic variants of PFFTTG, PFFTTG-H and PFFTTG-V. To simplify the exposition of the algorithms, we assume N to be divisible by p.

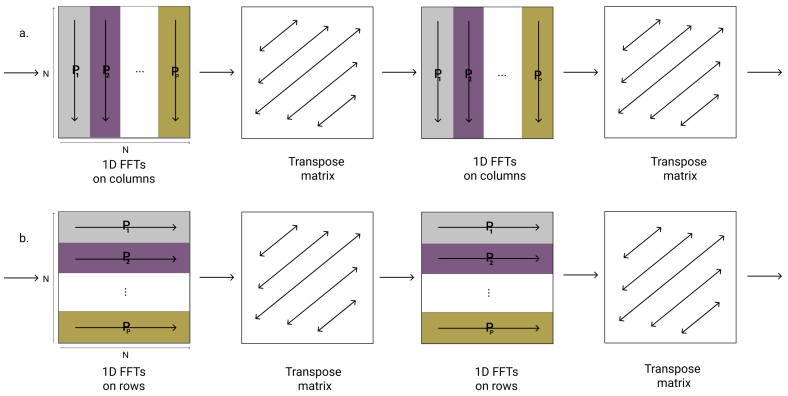


Figure 3.22: 2D-DFT of signal matrix M of size $N \times N$ using p threadgroups. a). PFFTTG-V using vertical decomposition of the signal matrix. b). PFFTTG-H using horizontal decomposition of the signal matrix.

PFFTTG-H: Using Horizontal Decomposition of Signal Matrix M

The parallel 2D-FFT algorithm, PFFTTG-H, consists of four steps:

Step 1. 1D-FFTs on rows: Threadgroup P_i executes sequential 1D-FFTs on rows $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 2. Matrix Transposition: Transpose the matrix M.

Step 3. 1D-FFTs on rows: Threadgroup P_i executes sequential 1D-FFTs on rows $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 4. Matrix Transposition: Transpose the matrix M.

The computational complexity of Steps 1 and 3 is $\Theta(\frac{N^2}{p} \log_2 N)$. The computational complexity of Steps 2 and 4 is $\Theta(\frac{N^2}{p})$. Therefore, the total computational complexity of PFFTTG-H is $\Theta(\frac{N^2}{p} \log_2 N)$.

The algorithm is illustrated in the Figure 3.22.

PFFTTG-V: Using Vertical Decomposition of Signal Matrix M

The parallel 2D-FFT algorithm, PFFTTG-V, consists of four steps:

Step 1. 1D-FFTs on columns: Threadgroup P_i executes sequential 1D-FFTs on columns $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 2. Matrix Transposition: Transpose the matrix M.

Step 3. 1D-FFTs on columns: Threadgroup P_i executes sequential 1D-FFTs on columns $(i-1) \times \frac{N}{p} + 1, ..., i \times \frac{N}{p}$.

Step 4. Matrix Transposition: Transpose the matrix M.

The computational complexity of Steps 1 and 3 is $\Theta(\frac{N^2}{p} \log_2 N)$. The computational complexity of Steps 2 and 4 is $\Theta(\frac{N^2}{p})$. Therefore, the total computational complexity of PFFTTG-V is $\Theta(\frac{N^2}{p} \log_2 N)$.

The algorithms are illustrated in the figure 3.22. Here, the performance of the vertical form of partitioning is considerably lower than that for the horizontal form of partitioning. Consequentially, all experiments for FFT performed with the horizontal form of partitioning. The shared memory implementations of PFFTTG-H based on FFTW and IMKL FFT are described in Appendix C.

3.3.4 Experimental Analysis for Performance

This section presents experimental results for performance for PMMTG and PFFTTG which were obtained on four multicore CPUs shown in the table 3.4.

To make sure the experimental results are reliable, a statistical methodology described in the Appendix B is used. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of

Table 3.4: Specifications of the Intel multicore CPUs, HCLServer01-04, ordered by increasing number of sockets and an increasing number of cores per socket.

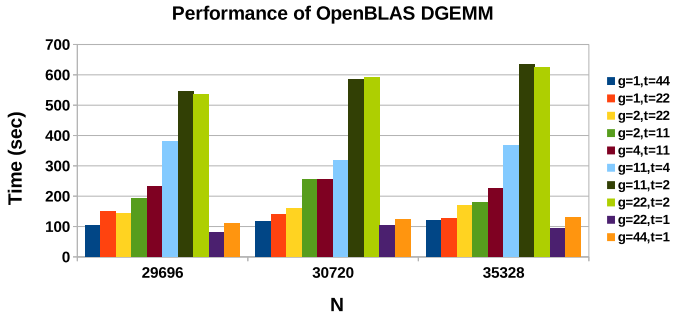
Technical Specifications	HCLServer1 (S1)	HCLServer2 (S2)	HCLServer3 (S3)	HCLServer4 (S4)
Processor	Intel Xeon Gold 6152	Intel Haswell E5-2670V3	Intel Xeon CPU E5-2699	Intel Xeon Platinum 8180
Core(s) per socket	22	12	18	28
Socket(s)	1	2	2	2
L1d cache, L1i cache	32 KB, 32 KB	32 KB, 32 KB	32 KB, 32 KB	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB	256 KB, 30976 KB	256 KB, 46080 KB	1024 KB, 39424 KB
Total main memory	96 GB	64 GB	256 GB	187 GB
Power meter	WattsUp Pro	WattsUp Pro	-	Yokogawa WT310

0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. The speed/performance/energy values shown in the graphical plots are the sample means.

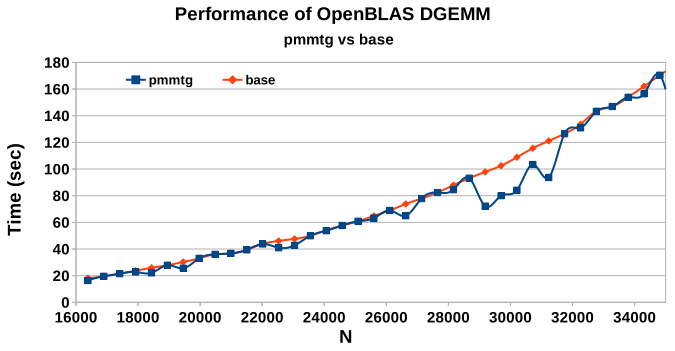
3.3.4.1 PMMTG Using OpenBLAS DGEMM and IMKL DGEMM

Optimization on Single-socket Multicore CPU

Figure 3.23a shows the execution time of different configurations (g,t) for PMMTG using OpenBLAS DGEMM on a single-socket multicore CPU (S1). It can be seen that the performance of application depends on the number of



(a)



(b)

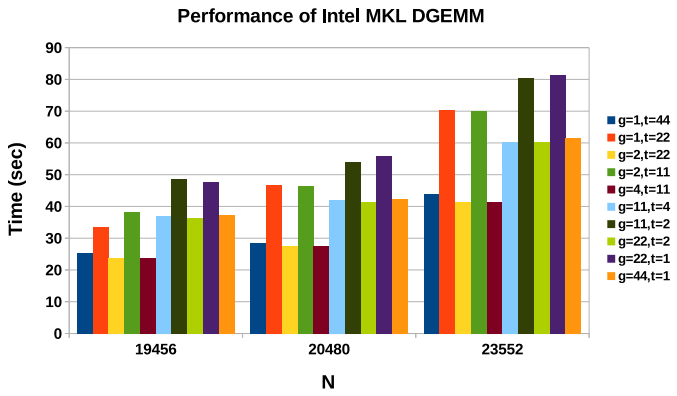
Figure 3.23: (a). Performance of PMMTG application employing OpenBLAS DGEMM with varying number of threadgroups on HCLServer1. (b). Execution time of PMMTG versus the best base configuration (g,t) employing OpenBLAS DGEMM on HCLServer1.

threadgroups and the number of threads in each threadgroup executing the application. The best configuration with minimum execution time is (g,t)=(22,1) and it is the same for all three sizes. Using this configuration we can improve performance over the best base configuration (g,t)=(1,44) by more than 20% for $N = 29696$ and $N = 35328$, and aroungd 10.5% for $N = 30720$. The worst performance configurations are (11,2) and (22,2) they degrade performance of the best base configuration 5-6 times.

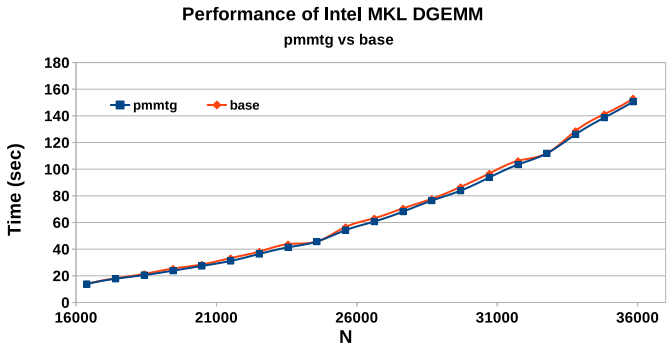
Figure 3.23b shows the execution time of PMMTG using OpenBLAS DGEMM versus its best base implementation for a range of sizes $16000 < N < 35000$. Each point in the pmmtg function represents the best configuration of (g,t) for a given problem size. All combinations are depicted as legends in figure 3.23a. The average and maximum performance improvements of PMMTG over the best base configuration are 7% and 26.3%. However, there are sizes where our method cannot optimize the best base implementation.

Figure 3.24a shows the execution time of different configurations (g,t) for PMMTG using IMKL DGEMM. One can observe that the best configuration with minimum execution time is (g,t)=(4,11) and it is the same for all three sizes. However, there is a configuration (g,t)=(2,22) where the execution time is almost the same as execution time of (4,11). They both outperform the best base configuration (1,44) by 5.5% for all three sizes. The worst performance configurations are (11,2) and (22,1). They damage performance 2.5 times for all three sizes.

Figure 3.24b shows the execution time of PMMTG using Intel MLK DGEMM versus its best base configuration for a range of sizes $16000 < N < 35000$. Each point in the pmmtg function represents the best configuration of (g,t) for the given problem size. All configurations are depicted as legends in figure 3.24a. The average and maximum performance improvements of PMMTG over the best base configuration are 4.1% and 6.5%.



(a)

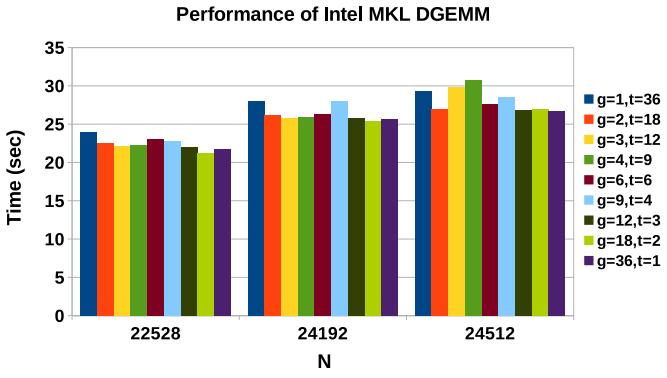


(b)

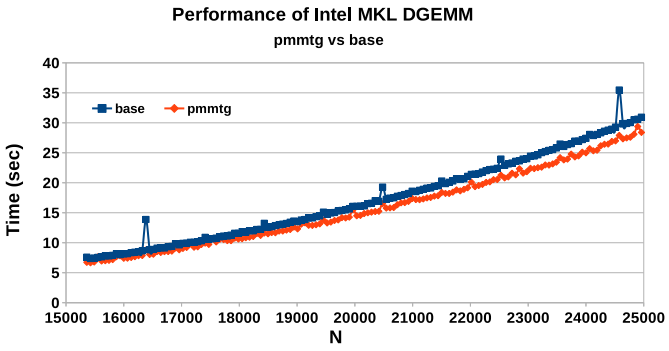
Figure 3.24: (a). Performance of PMMTG application employing IMKL DGEMM with varying number of threadgroups on HCLServer1. (b). Execution time of PMMTG versus the best base configuration (g,t) employing IMKL DGEMM on HCLServer1.

Optimization on Dual-socket Multicore CPU

Figure 3.25a shows the execution time of PMMTG between the different configurations employing IMKL DGEMM on HCLServer3. One can observe, that the execution time of application depends on the number of threadgroups and the number of threads in each threadgroup executing the application. There are three different configurations with the minimum execution time distinct from each other by less than 5%, (g,t)=(12,3),(18,2),(36,1). The performance improvements using these configurations over the best base



(a)



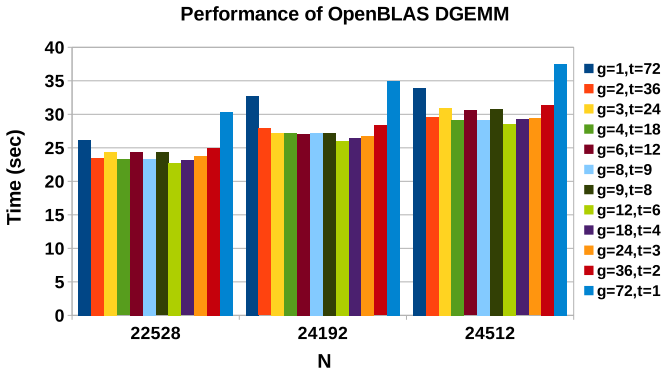
(b)

Figure 3.25: (a). Performance of PMMTG application employing IMKL DGEMM with varying number of threadgroups on HCLServer3. (b). Execution time of PMMTG versus the best base configuration (g,t) employing IMKL DGEMM on HCLServer3.

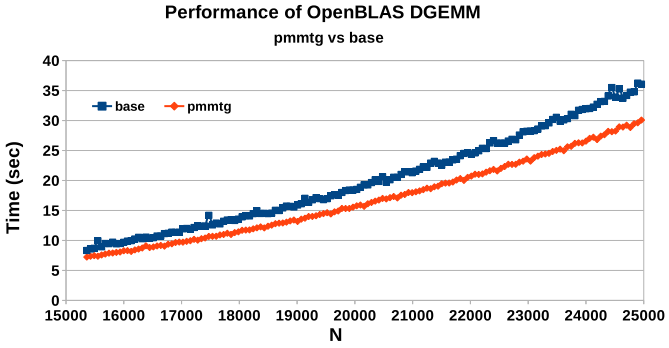
implementation can reach 10%.

Figure 3.25b shows the execution time of PMMTG versus its base implementation employing IMKL DGEMM on HCLServer3. The base implementation corresponds to application configuration employing one threadgroup with optimal number of threads. Each point in the pmmtg function represents the best configuration of (g,t) for a given problem size. It can be seen that PMMTG removes huge drops for $N = 16384, N = 20480$ and $N = 24576$ with performance improvements of 36.5%, 14.5% and 21.5% respectively. The average and maximum improvements of PMMGT over the

base version are 7% and 42.1%.



(a)



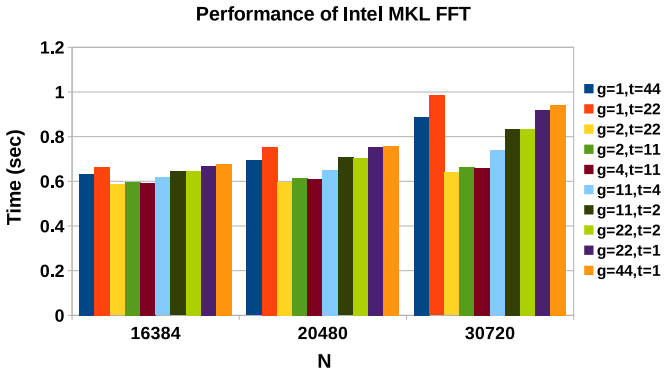
(b)

Figure 3.26: (a). Performance of PMMTG application employing OpenBLAS DGEMM with varying number of threadgroups on HCLServer3. (b). Execution time of PMMTG versus the best base configuration (g,t) employing OpenBLAS DGEMM on HCLServer3.

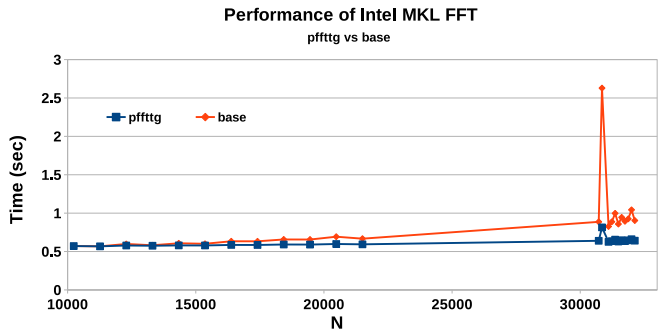
Figure 3.26a shows the performance of PMMTG for different configurations employing OpenBLAS DGEMM on HCLServer3. The best configuration with the minimum execution time is the same for all sizes and is (g,t)=(12,6). It outperforms the best base configuration by 13.3%, 20.7% and 15.9% for sizes $N = 22528, N = 24192$ and $N = 24512$ respectively.

Figure 3.26b shows the execution time of PMMGT versus its base version employing OpenBLAS DGEMM on HCLServer3. The base version corresponds to application configuration employing one threadgroup with

optimal number of threads. Each point in the pmmtg function represents the best configuration of (g,t) for a given problem size. The average and maximum performance improvements of PMMTG over the best base implementation are 19% and 31.7%. Furthermore, PMMTG removes notable drops in the performance profile of the base version.



(a)



(b)

Figure 3.27: (a). Performance of PFFTTG application employing IMKL FFT with varying number of threadgroups on HCLServer1. (b). Eexecution time of PFFTG versus the best base congregation (g,t) employing IMKL FFT on HCLServer1.

3.3.4.2 PFFTTG Using FFTW and IMKL FFT

Optimization on Single-socket Multicore CPU

Figure 3.27a shows the comparison of the execution time between the different configurations (g,t) of PFFTGT using IMKL FFT on a single-socket multicore CPU (S1). It can be seen that different configurations (g,t) have different execution times. The best configuration with minimum execution time for all three sizes is (g,t)=(2,22). Its performance improvements over the best base configuration are 7.9%, 15% and 27% for sizes $N = 16384, N = 20480$ and $N = 30720$ respectively.

Figure 3.27b shows the execution time of PFFTGT using IMKL FFT versus its best base configuration. Each point in the pfftg function represents the best configuration (g,t) for the given problem size. The average and maximum performance improvements of PFFTTG over the best base configuration are 7% and 13%. The best base configuration corresponds to application configuration employing one threadgroup with optimal number of threads.

Figure 3.28a shows the results for PFFTTG employing FFTW. The best combination with minimum execution time is (g, t)=(4,11) and is the same for problem sizes $m = n = 31936$ and $m = n = 32704$. Its improvements over the best base configuration (g, t)=(1,22) are 55% and 57% respectively. For problem size $m = n = 35648$ the base configuration (1,44) is the best and outperforms the closest configuration (g, t)=(2,22) by 5%.

Figure 3.28b depicts the performance profile of PFFTTG over the best base implementation employing FFTW on HCLServer1. Each point in both functions represents the best configuration of (g,t) for a given problem size. The average and maximum improvements of PFFTTG over the best base implementation are 25% and 51%.

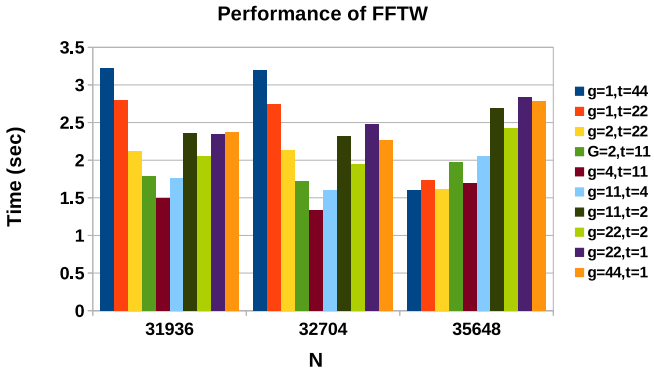
Optimization on Dual-socket Multicore CPU

All results in this section are represented by a surface in the 3D space represented by axes for performance or energy, number of threadgroups (g)

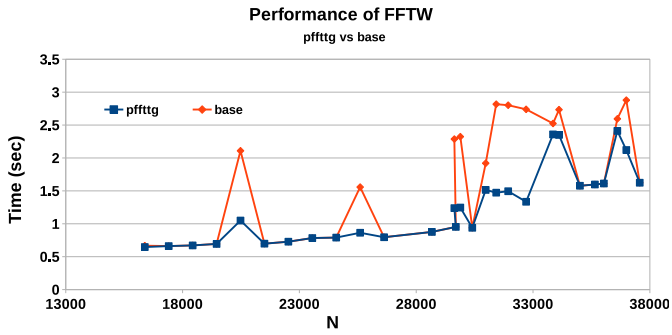
and the number of threads in each threadgroup, t. The location of the minimum in the surface is shown by a red dot.

Figure 3.29a presents the results of PFFTTG using FFTW3.3.7 on HCLServer3 for matrix dimension $m = n = 17728$. The minimum is centred around numbers of threadsgroups equal to {4,7,8}. The minimum is achieved for the combination, (g, t)=(4,16). The speedup is 80% in comparison with (g, t)=(1,72), which is the best combination for one group.

Figure 3.29b shows the comparison of PFFTTG versus the best base implementation employing FFTW. Each point in both functions represents the

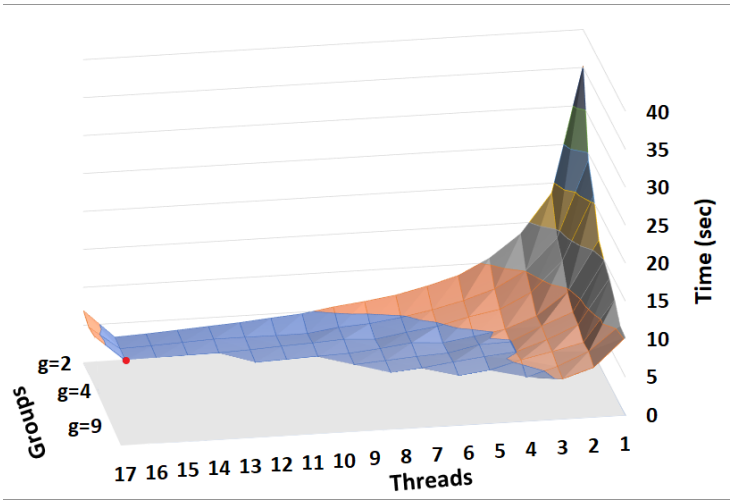


(a)

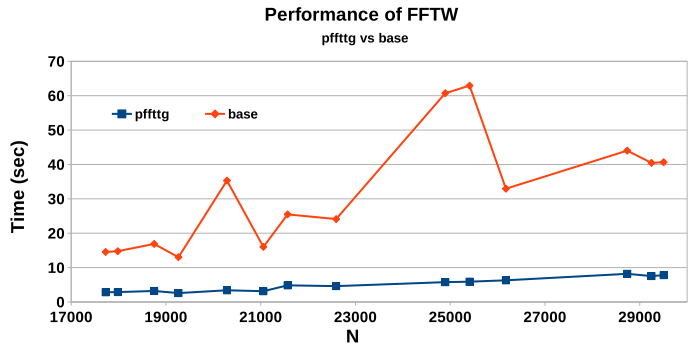


(b)

Figure 3.28: (a). Performance of PFFTTG application employing FFTW with varying number of threadgroups on HCLServer1. (b). Eexecution time of PFFTTG versus the best base congregation (g,t) employing FFTW on HCLServer1.



(a)



(b)

Figure 3.29: (a). Performance profile of FFTW PFFTTG application with varying number of threadgroups and number of threads per group on HCLServer3 (S3) for workload size, $m = n = 17728$. Red dot represents the minimum. (b). Performance profiles of FFTW PFFTTG application versus the best base implementation employing FFTW on HCLServer3 (S3).

best configuration (g,t) for a given problem size. The average and maximum performance improvements of PFFTTG over the best base version are 85% and 90%.

3.3.5 Experimental Analysis for Energy

This section presents experimental results for energy for matrix-matrix multiplication (PMMTG) and 2D fast Fourier transform (PFFTTG) employing proposed solution method. The statistical methodology of experiments is the same as for the previous section. The same multicore CPUs shown in the table 3.4 are used, where three platforms (HCLServer1, HCLServer2, HCLServer4) have a power meter installed between their input power sockets and the wall A/C outlets. HCLServer1 and HCLServer2 are connected with a *Watts Up Pro* power meter; HCLServer4 is connected with a *Yokogawa WT310* power meter.

The power meter captures the total power consumption of the server. It has data cable connected to one USB port of the server. A script written in Perl collects the data from the power meter using the serial USB interface. The execution of the script is non-intrusive and consumes insignificant power. *Watts Up Pro* power meters are periodically calibrated using the ANSI C12.20 revenue-grade power meter, Yokogawa WT310. The maximum sampling speed of *Watts Up Pro* power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts. The accuracy of Yokogawa WT310 is 0.1% and the sampling rate is 100k samples per second.

HCLWattsUp API [165] is used to gather the readings from the power meter to determine the dynamic energy consumption during the execution of PMMTG and PFFTTG applications. HCLWattsUp has no extra overhead and therefore does not influence the energy consumption of the application execution.

Fans are significant contributors to energy consumption. On those three platforms, fans are controlled in two zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are 4 levels to control the speed of fans:

- *Standard*: BMC control of both fan zones, with CPU zone based on CPU temp (target speed 50%) and Peripheral zone based on PCH temp (target speed 50%)

- *Optimal*: BMC control of the CPU zone (target speed 30%), with Peripheral zone fixed at low speed (fixed 30%)
- *Heavy IO*: BMC control of CPU zone (target speed 50%), Peripheral zone fixed at 75%
- *Full*: all fans running at 100%

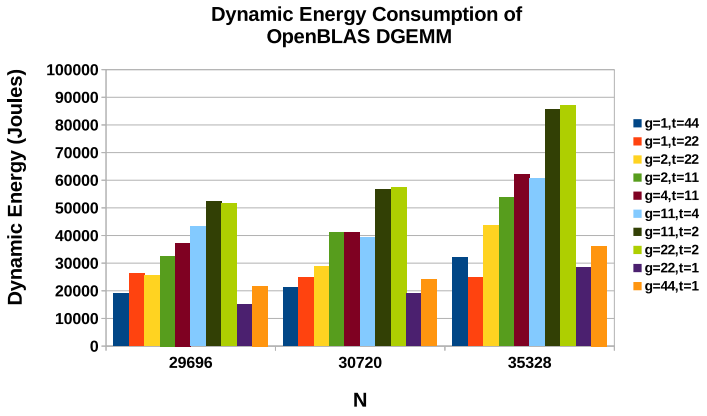
To rule out the contribution of fans in dynamic energy consumption, the fans set at full speed before executing the applications. When set at full speed, the fans run constantly at ~ 13400 rpm until they are set to a different speed level. In this way, energy consumption due to fans is included only in the static power consumption of the platform. The temperature of the platform and speeds of the fans (with *Full* setting) are monitored with the help of Intelligent Platform Management Interface (IPMI) sensors, both with and without the application run. An insignificant difference in the speeds of fans was found in both scenarios.

3.3.5.1 PMMTG Using OpenBLAS DGEMM and IMKL DGEMM

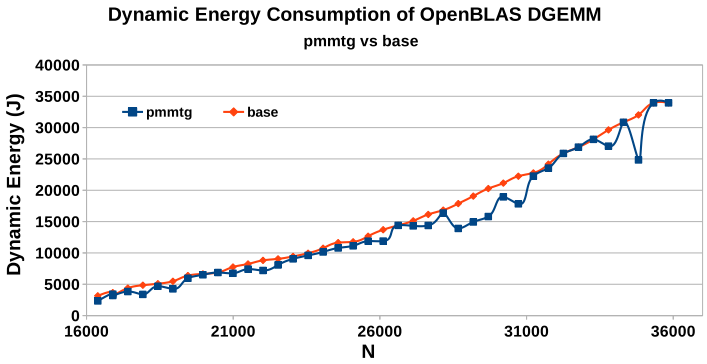
Optimization on Single-socket Multicore CPU

Figure 3.30a shows the comparison of dynamic energy consumption between the different configurations in PMMTG using OpenBLAS DGEMM on a single-socket CPU (S1). It can be seen that each configuration (g,t) differently consumes energy. The best configuration with the minimum energy consumption for sizes $N = 29696$ and $N = 30720$ is (g,t)=(22,1). It outperforms the best base configuration (g,t)=(1,44) by more than 21%. However, the best configuration for $N = 35328$ is the best base (g,t)=(1,22) which outperforms (1,44) by 22.5% and the closest (22,1) by 11%. The worst configurations (g,t) in terms of energy efficiency for all sizes are (11,2) and (22,2). They can increase energy consumption up to 3 times.

Figure 3.30b depicts the energy profiles of the best configuration (g,t) for a given problem size using PMMTG and the best base configuration (g,t) for the given problem size employing OpenBLAS DGEMM, for a range of sizes $16000 < N < 36000$. The average and maximum energy savings of PMMTG



(a)

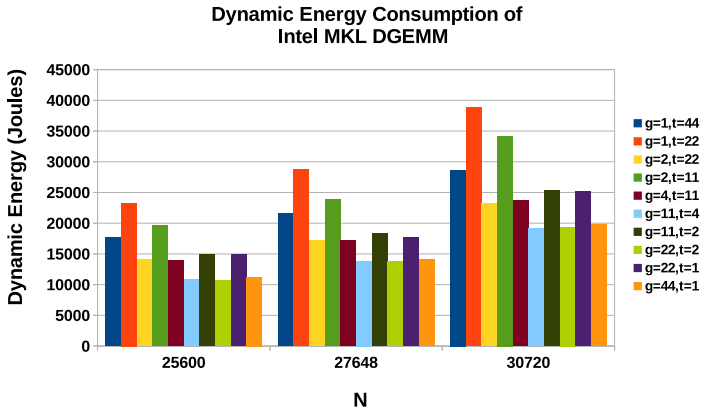


(b)

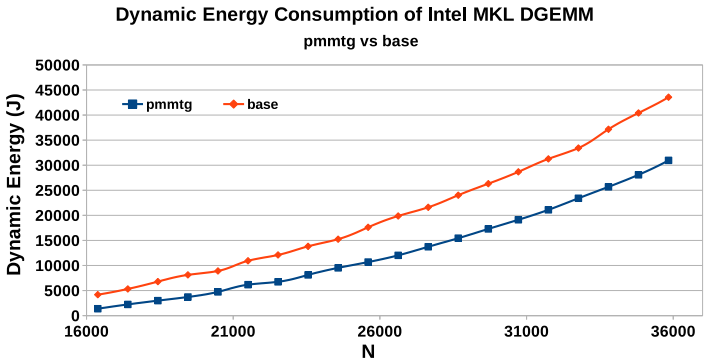
Figure 3.30: (a). Dynamic energy consumption of PMMTG application employing OpenBLAS DGEMM with varying number of threadgroups on HCLServer1. (b). Dynamic energy consumption of PMMTG versus the best base configuration (g,t) employing OpenBLAS DGEMM on HCLServer1.

over the best base implementation are 7.9% and 30%. However, there are sizes where our method cannot reduce energy consumption.

Figure 3.31a shows the comparison of dynamic energy consumption between different configurations of PMMTG using IMKL DGEMM. One can observe that there are three configurations in each problem size that consume the minimum energy and they are (g,t)=(11,4),(22,2),(44,1). The improvement in energy consumption using these configurations over the best base configuration (1,44) can reach 37%.



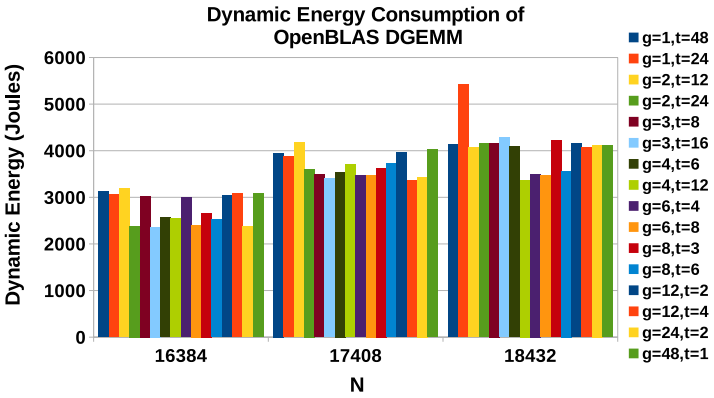
(a)



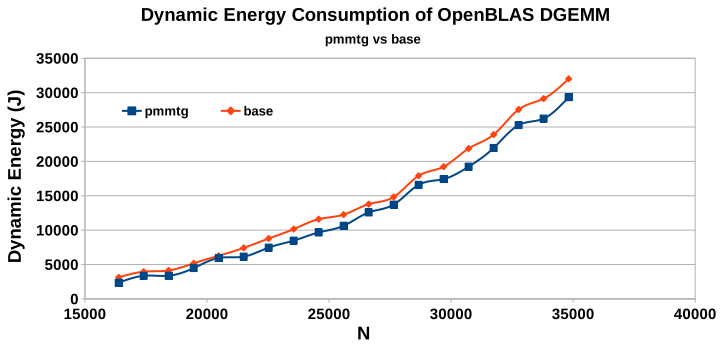
(b)

Figure 3.31: (a). Dynamic energy consumption of PMMTG application employing IMKL DGEMM with varying number of threadgroups on HCLServer1. (b). Dynamic energy consumption of PMMTG versus the best base configuration employing IMKL DGEMM on HCLServer1.

Figure 3.31b depicts the dynamic energy consumption of the best configurations (g,t) for PMMTG and the best base configuration (g,t) employing IMKL DGEMM for a given problem size for a range of sizes $16000 < N < 36000$. The average and maximum energy savings of PMMTG over the best base implementation are 35.7% and 67%.



(a)



(b)

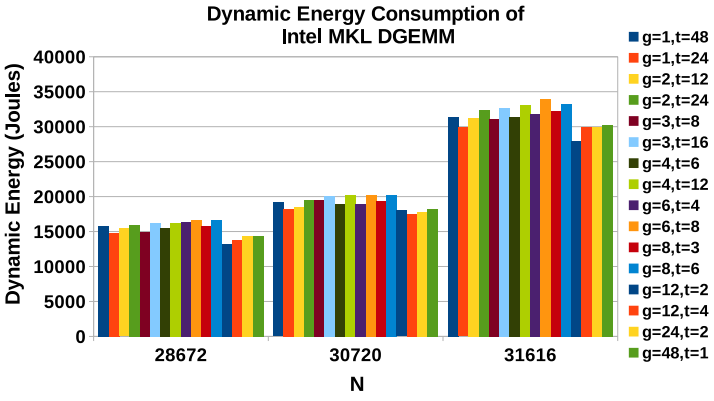
Figure 3.32: (a). Dynamic energy consumption of PMMTG application employing OpenBLAS DGEMM with varying number of threadgroups on HCLServer2. (b). Dynamic energy consumption of PMMTG versus the best base configuration employing OpenBLAS DGEMM on HCLServer2.

Optimization on Dual-socket Multicore CPU

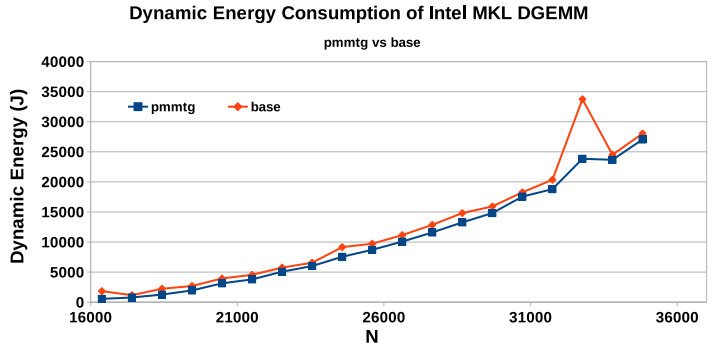
Figure 3.32a shows the dynamic energy consumption for different configurations (g,t) of the PMMTG application based on OpenBLAS DGEMM on HCLServer2. One can observe there are four configurations wich are almost equally minimizing dynamic energy consumption for workload size 16384, they are (g,t)=(2,24),(3,16),(6,8),(24,2). The energy savings for these configurations compared with the best base configuration (1,24) is around 21%. For the workload sizes 17408 and 18432, the best configurations are (12,4) and (4,12). The energy savings in comparison with the best base

configuration (1,24) for 17408 and (1,44) for 18432, are 15% and 18% respectively.

Figure 3.32b depicts the dynamic enegry consumption of PMMTG and the best base configuration employing OpenBLAS DGEMM for a range of sizes $15000 < N < 36000$. Each point in the pmmtg function represents the best configuration of (g,t) for the given problem size. All configurations depicted as a labels in figure 3.32a. The average and maximum energy savings of PMMTG over the best base implementation are 10% and 24.5%.



(a)



(b)

Figure 3.33: (a). Dynamic energy consumption of PMMTG application employing IMKL DGEMM DGEMM with varying number of threadgroups on HCLServer2. (b). Dynamic energy consumption of PMMTG versus the best base configuration employing IMKL DGEMM on HCLServer2.

Figure 3.33a depicts the energy consumption of each configuration (g,t) of

the PMMTG application based on IMKL DGEMM for three defferent problem sizes on HCLServer2. One can observe, the best configuration minimizing dynamic energy consumption for workload size 28672 involves 12 threadgroups with 2 threads in each. The energy savings for this configuration compared with the best base configuration (1,24) is 10.5%. For the workload sizes 30720 and 31616, the best configurations are (12,4) and (12,2). The energy savings in comparison with the best base configuration (1,24) are 4% and 7% respectively.

Figure 3.33b illustrates the dynamic enegy consumption of PMMTG versus the best base configuration employing IMKL DGEMM. Each point in both functions represents the best configuration (g,t) for the given problem size. The average and maximum energy savings of PMMTG over the best base implementation are 13% and 67%.

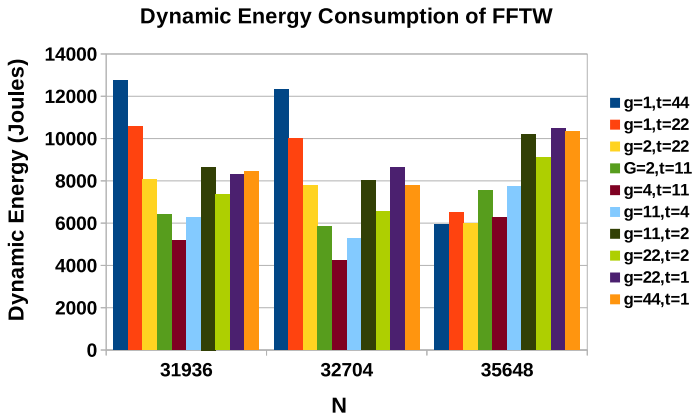
3.3.5.2 PFFTTG Using FFTW and IMKL FFT

Optimization on Single-socket Multicore CPU

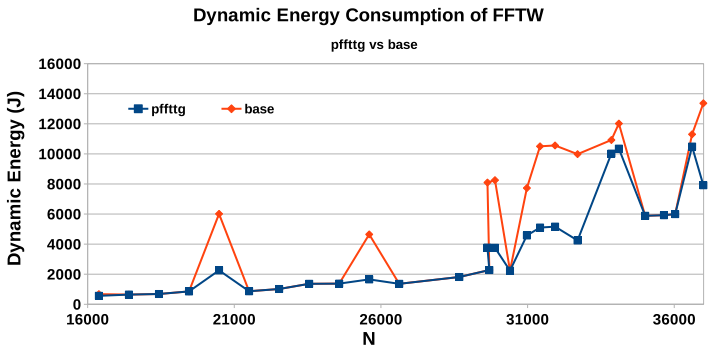
Figure 3.34a shows the dynamic energy comparision between different configurations of the PMMTG application for workload sizes 31936, 32704, and 35648 on a single-socket CPU (S1). It can be seen, the best configuration (g, t)=(4,11) is the same for sizes 31936 and 32704. The reductions in dynamic energy consumption using this configuration in comparison with the best base configuration (g, t)=(1,22) are 41% and 65% respectively. For size 35648 the best base configuration is the best and outperforms the closest configuration (g, t)=(2,22) by less than 5%.

Figure 3.34b depicts the dynamic energy consumption of the best configurations (g,t) for PFFTTG and the best base configuration (g,t) employing FFTW for a given problem size for a range of sizes $16000 < N < 37000$. The average and maximum energy savings of PFFTTG over the best base implementation are 30% and 63%.

For the IMKL FFT, PFFTTG could not optimize the best base configuration (g,t)=(1,44). Moreover, for IMKL FFT, the dynamic energy consumption increases with the increasing number of threadgroups.



(a)



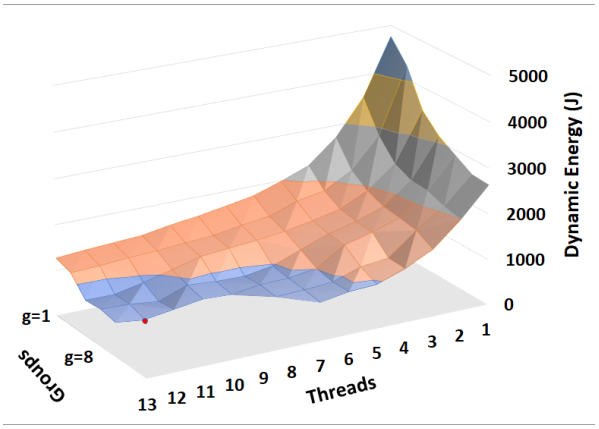
(b)

Figure 3.34: (a). Dynamic energy consumption of PFFTTG application employing FFTW with varying number of threadgroups on HCLServer1. (b). Dynamic energy consumption of PFFTTG versus the best base configuration employing FFTW on HCLServer1.

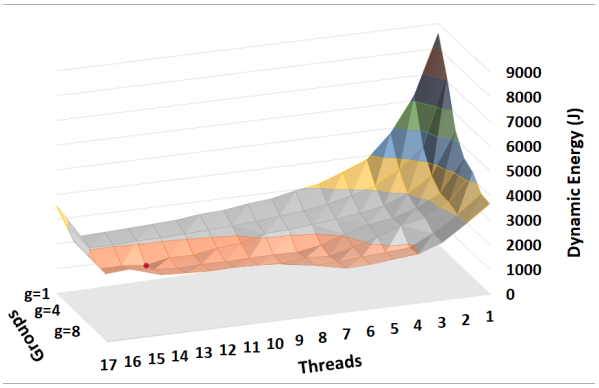
Optimization on Dual-socket Multicore CPU

Figures 3.35a, 3.35b show the results for PFFTTG employing FFTW3.3.7 on HCLServer4 for matrix sizes 30464 and 32192 respectively. The minimum for dynamic energy is achieved for {4,7,8} threadgroups with 14 threads in each threadgroup for workload size 32192 and 12 threads in each threadgroup for workload size 30464. The minimum for the workload size 30464 is achieved for the configuration (g, t)=(8,12). The dynamic energy consumption for this combination is 661 Joules. The energy saving is around 30% in comparison

with the best base configuration (g,t)=(1,45) whose dynamic energy consumption is 918 Joules. The minimum for the workload size 32192 is achieved for the configuration (g,t)=(4,14). The saving is around 35% in comparison with the best base configuration (g,t)=(1,16) where dynamic energy is 2197 Joules. Furthermore, the average and maximum energy savings for a range of sizes $30464 \leq N \leq 33280$ with picked 10 sizes are 23% and 43%.



(a)



(b)

Figure 3.35: (a). Energy profile of FFTW PFFTTG application with varying number of threadgroups and number of threads per group on HCLServer4 (S4) for workload size $m = n = 30464$. (b). Energy profile of FFTW PFFTTG application with varying number of threadgroups and number of threads per group on HCLServer4 (S4) for workload size $m = n = 32192$. Red dot represents the minimum.

3.3.6 Summary

This section proposed the methods for single-objective optimization for performance and energy on multicore CPUs using application-level decision variables such as the number of thredgroups and the number of threads in each threadgroup. Using two highly optimized scientific routines, matrix-matrix multiplication and 2D fast Fourier transform, it was demonstrated that the methods successfully optimize these objectives on both single-socket and dual-socket multicore CPUs.

3.4 Conclusion

This chapter overviewed the challenges posed by complexities such as thread contention for shared resources and NUMA in modern multicore CPUs to performance and energy optimization of data-parallel applications on such platforms. The influence of three-dimensional decision variable space on single-objective optimization of applications for performance and energy on multicore CPUs was studied. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups. At first, the methods for performance optimization which use only workload distribution as a decision variable were proposed. The methods employ model-based parallel computing technique using load-imbalancing data partitioning. Using the matrix-matrix multiplication and 2D fast Fourier transform, it was demonstrated that using workload distribution as a decision variable allow us to improve performance against the best base implementation. However, because of the complexity of energy measurement on modern multicore CPUs, these methods are not applicable to energy optimization. The chapter also proposed new methods for single-objective optimization for performance and energy on modern multicore CPUs using two application-level decision variables, the number of threadgroups and the number of threads per threadgroup. Using the same applications, matrix-matrix multiplication and 2D-FFT and four modern multicore servers (S1,S2,S3,S4), the experimental results demonstrated that these methods successfully optimize both performance and energy on both single-socket and dual-socket multicore CPUs. In future work, the author is looking to merge all these three decision variables into the one method.

Chapter 4

Bi-objective Optimization for Performance and Energy on Modern Multicore CPUs

Energy proportionality is the key design goal pursued by architects of modern multicore CPU platforms [28]. One of its implications is that optimization of an application for performance will also optimize it for energy. Modern multicore CPUs however have many inherent complexities, which are: a) Severe resource contention due to tight integration of tens of cores organized in multiple sockets with multi-level cache hierarchy and contending for shared on-chip resources such as last level lache (LLC), interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers; b) Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes; and c) Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM). This chapter shows that due to these complexities, energy proportionality does not hold true for multicore CPUs. This finding creates the opportunity for bi-objective optimization of applications for performance and energy.

Next, we review notable state-of-the-art methods solving the bi-objective

optimization problem of an application for performance and energy on multicore CPU platforms. System-level methods are introduced first since they dominated the landscape. This will be followed by recent research in application-level methods. Then we describe the proposed solution method solving the bi-objective optimization problem of an application for performance and energy on a single multicore CPU.

Solution methods solving the bi-objective optimization problem for performance and energy can be broadly classified into *system-level* and *application-level* categories. System-level methods aim to optimize performance and energy of the environment where the applications are executed. The methods employ application-agnostic models and hardware parameters as decision variables. They are principally deployed at operating system (OS) level and therefore require changes to the OS. They do not involve any changes to the application. The methods can be further divided into the following prominent groups:

1. Thread schedulers that are contention-aware and that exploit cooperative data sharing between threads [166], [167]. The goal of a scheduler is to find thread-to-core mappings to determine Pareto-optimal solutions for performance and energy. The schedulers operate at both user-level and OS-level with those at OS-level requiring changes to the OS. Thread-to-core mapping is the key decision variable. Performance monitoring counters such as LLC miss rate and LLC access rate are used for predicting the performance given a thread-to-core mapping.

2. Dynamic private cache (L1 and L2) reconfiguration and shared cache (L3) partitioning strategies [168], [169]. The proposed solutions in this category mitigate contention for shared on-chip resources such as last level cache by physically partitioning it and therefore require substantial changes to the hardware or OS [170].

3. Thermal management algorithms that place or migrate threads to not only alleviate thermal hotspots and temperature variations in a chip but

also reduce energy consumption during an application execution [171], [172]. Some key strategies are dynamic power management (DPM) where idle cores are switched off, Dynamic Voltage and Frequency Scaling (DVFS), which throttles the frequencies of the cores based on their utilization, sand migration of threads from hot cores to the colder cores.

4. Asymmetry-aware schedulers that exploit the asymmetry between sets of cores in a multicore platform to find thread-to-core mappings that provide Pareto-optimal solutions for performance and energy [173], [174]. Asymmetry can be explicit with fast and slow cores or implicit due to non-uniform frequency scaling between different cores or performance differences introduced by manufacturing variations. The key decision variables employed here are thread-to-core mapping and DVFS. Typical strategy is to map the most power-intensive threads to less power-hungry cores and then apply DVFS to the cores to ensure all threads complete at the same time whilst satisfying a power budget constraint.

In the second category, solution methods optimize applications rather than the executing environment. The methods use application-level decision variables and predictive models for performance and energy consumption of applications to solve the bi-objective optimization problem. The dominant decision variables include the number of threads, loop tile size, workload distribution, etc. Following the principle of energy proportionality, a dominant class of such solution methods aim to achieve optimal energy reduction by optimizing for performance alone. Definitive examples are scientific routines offered by vendor-specific software packages that are extensively optimized for performance. For example, Intel Math Kernel Library [175] provides extensively optimized multithreaded basic linear algebra subprograms (BLAS) and 1D, 2D, and 3D fast Fourier transform (FFT) routines for Intel processors. Open source packages such as [31], [32], [176] offer the same interface functions but contain portable optimizations and may exhibit better average performance than a heavily optimized vendor package [33], [34]. The

optimized routines in these software packages allow employment of one key decision variable, which is the number of threads. A given workload is load-balanced between the threads. *In this work, we show that the optimal number of threads (and consequently load-balanced workload distribution) maximizing the performance does not necessarily minimize the energy consumption of multicore CPUs.*

State-of-the-art research works on application-level optimization methods [26], [27], [35] demonstrate that due to the aforementioned design complexities of modern multicore CPU platforms, the functional relationships between performance and workload size and between dynamic energy and workload size for real-life data-parallel applications have complex (non-linear) properties and show that workload distribution has become an important decision variable that can no longer be ignored. Briefly, the total energy consumption during an application execution is the sum of dynamic and static energy consumptions. Static energy consumption is defined as the energy consumed by the platform without the application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumed by the platform during the application execution. The works [26], [27], [35] propose model-based data partitioning methods that take as input discrete performance and dynamic energy functions with no shape assumptions, which accurately and realistically account for resource contention and NUMA inherent in modern multicore CPU platforms. Using a simulation of the execution of a data-parallel matrix multiplication application based on OpenBLAS DGEMM on a homogeneous cluster of multicore CPUs, it is shown [26] that optimizing for performance alone results in average and maximum dynamic energy reductions of 24% and 68%, but optimizing for dynamic energy alone results in performance degradations of 95% and 100%. For a 2D fast Fourier transform application based on FFTW, the average and maximum dynamic energy reductions are 29% and 55% and the average and maximum performance degradations are both 100%. Research work [35] proposes a solution method to solve bi-objective optimization problem of an application for performance and energy on homogeneous clusters of modern multicore CPUs. This method is shown to determine a

diverse set of globally Pareto-optimal solutions whereas existing solution methods give only one solution when the problem size and number of processors are fixed. The methods [26], [27], [35] target homogeneous high performance computing (HPC) platforms. Khaleghzadeh et al. [36] propose a solution method solving the bi-objective optimization problem on heterogeneous processors. The authors prove that for an arbitrary number of processors with linear execution time and dynamic energy functions, the globally Pareto-optimal front is linear and contains an infinite number of solutions out of which one solution is load balanced while the rest are load imbalanced. A data partitioning algorithm is presented that takes as an input discrete performance and dynamic energy functions with no shape assumptions.

The research works [26], [27], [35], [36] are theoretical demonstrating performance and energy improvements based on simulations of clusters of homogeneous and heterogeneous nodes. Khokhriakov et al. [34] present two novel optimization methods to improve the average performance of the FFT routines on modern multicore CPUs. The methods employ workload distribution as the decision variable and are based on parallel computing employing threadgroups. They utilize load imbalancing data partitioning technique that determines optimal workload distributions between the threadgroups, which may not load-balance the application in terms of execution time. The inputs to the methods are discrete functions of performance against problem size of the threadgroups, and can be employed as nodal optimization techniques to construct a 2D FFT routine highly optimized for a dedicated target multicore CPU. The authors employ the methods to demonstrate significant performance improvements over the basic FFTW and IMKL FFT 2D routines on a modern Intel Haswell multicore CPU consisting of thirty-six physical cores.

The findings in [26], [27], [34]–[36] motivate the author of this thesis to study the influence of three-dimensional decision variable space on bi-objective optimization of applications for performance and energy on multicore CPUs. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an

application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups. The author focuses exclusively on the first two decision variables in this work. The number of possible workload distributions increases exponentially with increasing number of threadgroups employed in the execution of a data-parallel application and it would require employment of threadgroup-specific performance and energy models to reduce the complexity. It is a subject of our future work.

This chapter proposes and studies the first application-level method for bi-objective optimization of multithreaded data-parallel applications on a single multicore CPU for performance and energy. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application in parallel and the number of threads in each threadgroup. The workload distribution is not a decision variable. It is fixed so that a given workload is always partitioned equally between the threadgroups. The method allows full reuse of highly optimized scientific codes and does not require any changes to hardware or OS. As its first step, the method includes writing a data-parallel version of the base kernel that can be executed using a variable number of parallel threadgroups and solving the same problem as the base kernel which employs one threadgroup.

The following section reviews background of multi-objective optimization. Then, the first novel method, BOPPETG, for bi-objective optimization for performance and energy on a single multicore CPU is proposed.

4.1 Multi-Objective Optimization: Background

A multi-objective optimization (MOP) problem may be defined as follows [177],[178]:

$$\begin{aligned} & minimize \quad \{\mathcal{F}(x) = (f_1(x), ..., f_k(x))\} \\ & \text{Subject to} \quad x \in \mathcal{S} \end{aligned}$$

where there are $k(\geq 2)$ objective functions $f_i : \mathbb{R}^p \rightarrow \mathbb{R}$. The objective is to minimize all the objective functions simultaneously.

$\mathcal{F}(x) = (f_1(x), ..., f_k(x))^T$ denotes the vector of objective functions. The decision (variable) vectors $x = (x_1, ..., x_p)$ belong to the (non-empty) feasible region (set) $\mathcal{S}$, which is a subset of the decision variable space $\mathbb{R}^p$. We call the image of the feasible region represented by $\mathcal{Z}$ $(= f(\mathcal{S}))$, the feasible objective region. It is a subset of the objective space $\mathbb{R}^k$. The elements of $\mathcal{Z}$ are called objective (function) vectors or criterion vectors and denoted by $\mathcal{F}(x)$ or $z = (z_1, ..., z_k)^T$, where $z_i = f_i(x), \forall i \in [1, k]$ are objective (function) values or criterion values.

If there is no conflict between the objective functions, then a solution x^* can be found where every objective function attains its optimum [178].

$$\forall x \in \mathcal{S}, f_i(x^*) \leq f_i(x), \quad i = 1, ..., k$$

However, in real-life multi-objective optimization problems, the objective functions are at least partly conflicting. Because of this conflicting nature of objective functions, it is not possible to find a single solution that would be optimal for all the objectives simultaneously. In multi-objective optimization, there is no natural ordering in the objective space because it is only partially ordered. Therefore we must treat the concept of optimality differently from single-objective optimization problem. The generally used concept is *Pareto-optimality*.

Definition 1. *A decision vector $x^* \in \mathcal{S}$ is Pareto-optimal if there does not exist another decision vector $x \in \mathcal{S}$ such that $f_i(x) \leq f_i(x^*), \forall i = 1, ..., k$ and $f_j(x) < f_j(x^*)$ for at least one index j [177].*

An objective vector $z^* \in \mathcal{Z}$ is Pareto-optimal if there does not exist another objective vector $z \in \mathcal{Z}$ such that $z_i \leq z_i^*, \forall i = 1, ..., k$ and $z_j < z_j^*$ for at least one index j.

Definition 2. *A decision vector $x^* \in \mathcal{S}$ is weakly Pareto-optimal if there does not exist another decision vector $x \in \mathcal{S}$ such that $f_i(x) < f_i(x^*), \forall i = 1, ..., k$ [177].*

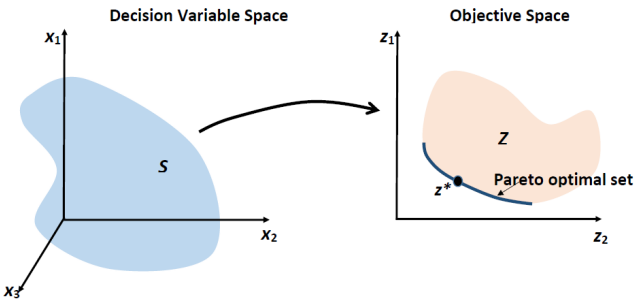


Figure 4.1: An example showing the set $\mathcal{S}$ of decision variable vectors, the set $\mathcal{Z}$ of objective vectors, and Pareto-optimal objective vectors shown by bold line. $\mathcal{S} \subset \mathbb{R}^3, \mathcal{Z} \subset \mathbb{R}^2$.

An objective vector $z^* \in \mathcal{Z}$ is Pareto-optimal if there does not exist any other vector for which all the component objective vector values are better.

Mathematically speaking, every Pareto-optimal point is an equally acceptable solution of the multi-objective optimization problem. Therefore, user preference relations (or preferences of decision maker) are provided as input to the solution process to select one or more points from the set of Pareto-optimal solutions [177].

In figure 4.1, a feasible region $\mathcal{S} \subset \mathbb{R}^3$ and its image, a feasible objective region $\mathcal{Z} \subset \mathbb{R}^2$, are shown. The thick blue line in the figure showing the objective space contains all the Pareto-optimal objective vectors. The vector z^* is one of them.

This thesis considers bi-objective optimization where performance and dynamic energy are the objectives.

4.2 Introduction in BOPPETG

This section describes solution method, BOPPETG, for solving the bi-objective optimization problem of a multithreaded data-parallel application on multicore CPUs for performance and energy (BOPPE). The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) and the number of threads in each threadgroup. A given workload is always partitioned equally between the threadgroups.

The bi-objective optimization problem (BOPPE) can be formulated as follows: Given a multithreaded data-parallel application of workload size n and a multicore CPU of l cores, the problem is to find a globally Pareto-optimal front of solutions optimizing execution time and dynamic energy consumption during the parallel execution of the workload. Each solution is an application configuration given by (threadgroups, threads per group).

The inputs to the solution method are the workload size of the multi-threaded data-parallel application, n; the number of cores in the multicore CPU, l; the multithreaded base kernel, $mtkernel$; the base power of the multicore CPU platform, P_b. The outputs are the globally Pareto-optimal front of objective solutions, $\mathcal{P}_{opt}$, and the optimal application configurations corresponding to these solutions, $\mathcal{C}_{opt}$. Each Pareto-optimal solution of objectives o is represented by the pair, (s_o, e_o), where s_o is the execution time and e_o is the dynamic energy. Associated with this solution is an array of application configurations, $\mathcal{A}(g_o, t_o)$, containing decision variable pairs, (g_o, t_o), where g_o represents the number of threadgroups each containing t_o threads.

The main steps of BOPPETG are as follows:

Step 1. Parallel implementation configurable using (g,t): Design and implement a parallel version of the base kernel $mtkernel$ and that can be executed using g identical multithreaded kernels in parallel. Each kernel is executed by a threadgroup containing t threads. The application should essentially allow its runtime configuration using number of threadgroups and number of threads per group with the workload equally partitioned between the threadgroups.

Step 2. Initialize g and t: All the application configurations, (g,t), where the product, $g \times t$, is less than or equal to the total number of cores (l) in the multicore platform are considered. $g \leftarrow 1$, $t \leftarrow 1$. Go to Step 3.

Step 3. Determine time and dynamic energy of the (g,t) configuration of the application: The data-parallel application composed in Step 1 is run using the (g,t) configuration where the workload n is divided equally between the g threadgroups during the execution of the application. The execution time

and dynamic energy consumption of the application are determined as follows: $s_o = t_f - t_i$, $e_o = e_f - P_b \times s_o$, where t_i and t_f are the starting and ending execution times and e_f is the total energy consumption during the execution of the application. Go to Step 4.

Step 4. Update Pareto-optimal front for (g,t): The solution (s_o, e_o) if Pareto-optimal is added to the globally Pareto-optimal set of objective solutions, $\{\mathcal{P}_{opt}\}$, and existing member solutions of the set that are inferior to it are removed. The optimal application configurations corresponding to the solution (s_o, e_o) are stored in $\mathcal{C}_{opt}$. Go to Step 5.

Step 5. Test and Increment (g,t): If $t < l$, $t \leftarrow t + 1$, go to Step 3. Set $g \leftarrow g + 1$, $t \leftarrow 1$. If $g \times t \leq l$, go to Step 3. Else return the globally Pareto-optimal front and optimal application configurations given by $\{\mathcal{P}_{opt}, \mathcal{C}_{opt}\}$ and quit.

4.3 Experimental Results and Discussion

This section presents the experimental results for matrix-matrix multiplication (PMMTG) and 2D fast Fourier transform (PFFTTG) employing proposed solution method.

To make sure the experimental results are reliable, a statistical methodology described in Appendix B is used. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. The speed/performance/energy values shown in the graphical plots are the sample means.

Four multicore CPUs shown in the Table 4.1 are used for the experiments. Three platforms (HCLServer1, HCLServer2, HCLServer4) have a power meter installed between their input power sockets and the wall A/C outlets. HCLServer1 and HCLServer2 are connected with a *Watts Up Pro* power meter; HCLServer4 is connected with a *Yokogawa WT310* power meter. The power meter captures the total power consumption of the server. It has data

Table 4.1: Specifications of the Intel multicore CPUs, HCLServer01-04, ordered by increasing number of sockets and an increasing number of cores per socket.

Technical Specifications	**HCLServer1 (S1)**	**HCLServer2 (S2)**	**HCLServer3 (S3)**	**HCLServer4 (S4)**
Processor	Intel Xeon Gold 6152	Intel Haswell E5-2670V3	Intel Xeon CPU E5-2699	Intel Xeon Platinum 8180
Core(s) per socket	22	12	18	28
Socket(s)	1	2	2	2
L1d cache, L1i cache	32 KB, 32 KB	32 KB, 32 KB	32 KB, 32 KB	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB	256 KB, 30976 KB	256 KB, 46080 KB	1024 KB, 39424 KB
Total main memory	96 GB	64 GB	256 GB	187 GB
Power meter	WattsUp Pro	WattsUp Pro	-	Yokogawa WT310

cable connected to one USB port of the server. A script written in Perl collects the data from the power meter using the serial USB interface. The execution of the script is non-intrusive and consumes insignificant power. *Watts Up Pro* power meters are periodically calibrated using the ANSI C12.20 revenue-grade power meter, Yokogawa WT310. The maximum sampling speed of *Watts Up Pro* power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts. The accuracy of Yokogawa WT310 is 0.1% and the sampling rate is 100k samples per second.

HCLWattsUp API [165] id used to gather the readings from the power meter to determine the dynamic energy consumption during the execution of PMMTG and PFFTTG applications. HCLWattsUp has no extra overhead and therefore does not influence the energy consumption of the application execution.

Fans are significant contributors to energy consumption. The fans are controlled in two zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are 4 levels to control the speed of fans:

- *Standard*: BMC control of both fan zones, with CPU zone based on CPU temp (target speed 50%) and Peripheral zone based on PCH temp (target speed 50%)
- *Optimal*: BMC control of the CPU zone (target speed 30%), with Peripheral zone fixed at low speed (fixed 30%)
- *Heavy IO*: BMC control of CPU zone (target speed 50%), Peripheral zone fixed at 75%

- *Full*: all fans running at 100%

To rule out the contribution of fans in dynamic energy consumption, we set the fans at full speed before executing the applications. When set at full speed, the fans run constantly at ~ 13400 rpm until they are set to a different speed level. In this way, energy consumption due to fans is included only in the static power consumption of the platform. The temperature of the platform and speeds of the fans (with *Full* setting) are monitored with the help of Intelligent Platform Management Interface (IPMI) sensors, both with and without the application run. An insignificant difference in the speeds of fans is found in both scenarios.

Single Socket Multicore CPU

Figure 4.2a shows the globally Pareto-optimal front for PMMTG employing IMKL DGEMM on HCLServer1 for workload size 32768. Optimizing for dynamic energy consumption alone degrades performance by 27%, and optimizing for performance alone increases dynamic energy consumption by 30%. The average and maximum sizes of the Pareto-optimal fronts for IMKL DGEMM on S1 are (2.3,3).

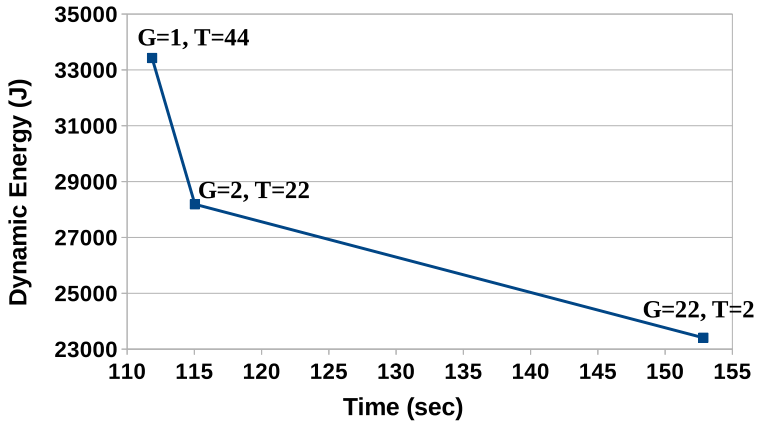
Figure 4.2b shows the globally Pareto-optimal front for PFFTTG based on IMKL FFT on HCLServer1 for workload size 31744. There are 2 globally Pareto-optimal solutions. Optimizing for dynamic energy consumption alone degrades performance by around 31%, and optimizing for performance alone increases dynamic energy consumption by 87%. We find the average and maximum sizes of the Pareto-optimal fronts for IMKL DGEMM on S1 are (2.6,3).

No bi-objective trade-offs were observed for FFTW and OpenBLAS applications. Two lines of this research will be investigated in future work. One is the influence of workload distribution; The other is the absence of bi-objective trade-offs for open-source packages such as FFTW and OpenBLAS using a dynamic energy predictive model.

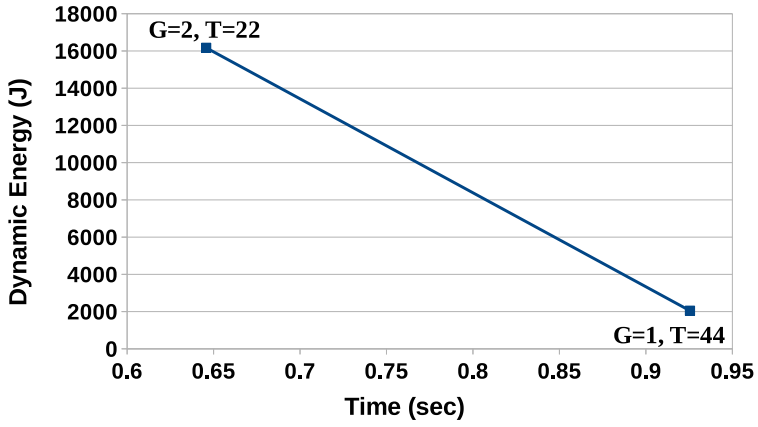
Dual-socket Multicore CPU

Figures 4.3a shows the globally Pareto-optimal fronts for PFFTTG FFTW on HCLServer4 for workload size $m = n = 30464$. The maximum number of globally Pareto-optimal solutions is 11. The optimization for dynamic energy consumption alone degrades performance by 49%, and optimizing for performance alone increases dynamic energy consumption by 35%.

Figure 4.3b shows the globally Pareto-optimal front for PFFTTG

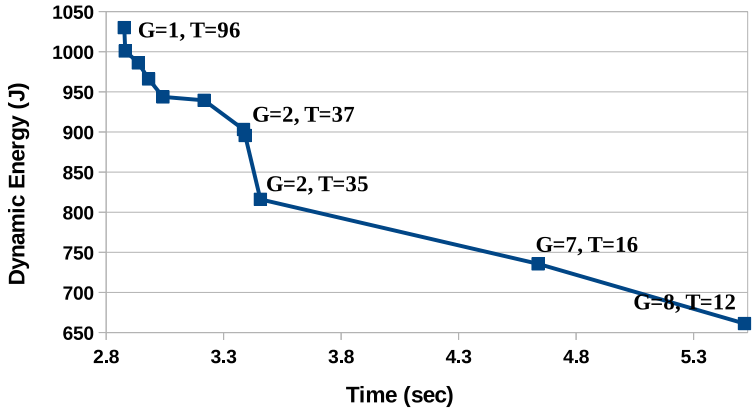


(a)

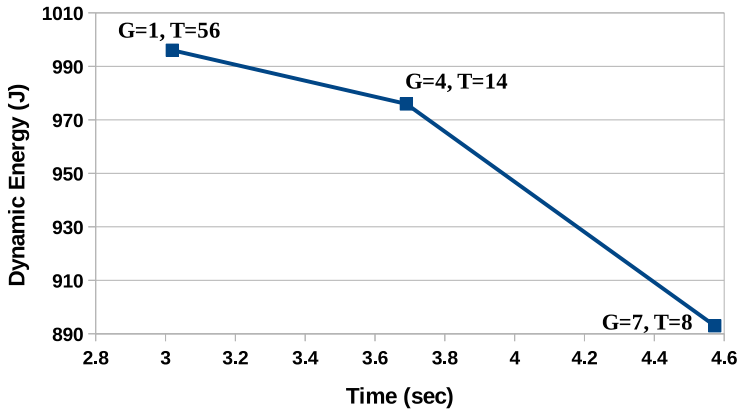


(b)

Figure 4.2: (a). Pareto frontier of IMKL DGEMM PMMTG application on HCLServer1 (S1) for workload size $N = 32768$. (b). Pareto frontier of IMKL FFT PFFTTG application on HCLServer1 (S1) for workload size $N = 31744$.



(a)

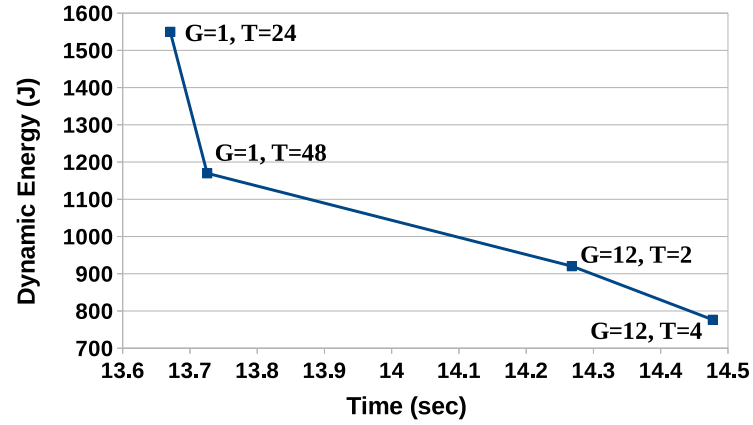


(b)

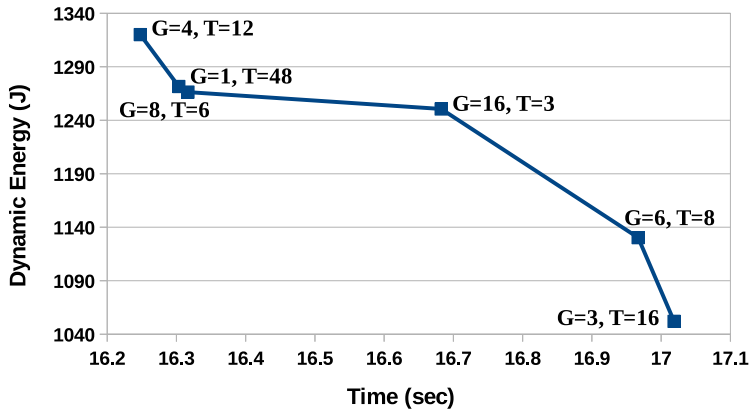
Figure 4.3: (a). Pareto frontier of FFTW PFFTTG application on HCLServer4 (S4) for workload size $m = n = 30464$. (b). Pareto frontier of PFFTTG application based on IMKL FFT on HCLServer4 (S4) for workload size $m = n = 22208$.

employing IMKL FFT on HCLServer2 for workload size 22208. Optimizing for dynamic energy consumption alone degrades performance by 33%, and optimizing for performance alone increases dynamic energy consumption by 10%. The average and maximum sizes of the Pareto-optimal fronts for FFTW and IMKL FFT on dual-socket CPUs are (3,11) and (2.7,3) respectively.

Figure 4.4a shows the globally Pareto-optimal front for PMMTG employing IMKL DGEMM on HCLServer2 for workload size 17408. Optimizing for



(a)



(b)

Figure 4.4: (a). Pareto frontier of IMKL DGEMM PMMTG application on HCLServer2 (S2) for workload size $m = n = 17408$. (b). Pareto frontier of PMMTG application employing OpenBLAS DGEMM on HCLServer2 (S2) for workload size $m = n = 17408$.

dynamic energy consumption alone degrades performance by 5.5%, and optimizing for performance alone increases dynamic energy consumption by 50.7%. The average and maximum sizes of the Pareto-optimal fronts for IMKL DGEMM on S2 are (1.8,4).

Figure 4.4b shows the globally Pareto-optimal front for PMMTG based on OpenBLAS DGEMM on HCLServer2 for workload size 17408. There are 6 globally Pareto-optimal solutions. Optimizing for dynamic energy consumption

alone degrades performance by around 5%, and optimizing for performance alone increases dynamic energy consumption by 20%. We find the average and maximum sizes of the Pareto-optimal fronts for OpenBLAS DGEMM on S2 to be 2.4 and 5 respectively.

4.4 Analysis Using Performance and Dynamic Energy Models

This section proposes dynamic energy model employing performance monitoring counters (PMCs) as predictor variables. This model along with the performance model employing execution time of the application is used to analyze the Pareto-optimal front determined by proposed solution method on multicore CPU platforms.

PMCs are special-purpose registers provided in modern microprocessors to store the counts of software and hardware activities. We will use the acronym PMCs to refer to software events, which are pure kernel-level counters such as *page-faults*, *context-switches*, etc. as well as micro-architectural events originating from the processor and its performance monitoring unit called the hardware events such as *cache-misses*, *branch-instructions*, etc. Software energy predictive models based on PMCs is one of the leading methods of measurement of energy consumption of an application.

The experimental platform, HCLServer02 (S2), and the application OpenBLAS-DGEMM are used for the analysis. Likwid tool [179] is used to obtain the PMCs. It offers 164 PMCs divided into 28 groups (L2CACHE, L3CACHE, NUMA, etc.) on this platform. The list of the groups is provided in the supplemental. All PMCs are collected for each workload size executed using different application configurations, (#threadgroups (g), #threads_per_group (t)). Each PMC value is the average for all the 24 physical cores. The data is analyzed to identify the major performance groups, which are highly correlated with the dynamic energy consumption. It is found that the highest correlation with dynamic energy consumption

Table 4.2: L1 dTLB PMC data for size 16384

Comb.(g, t)	DEnergy (J)	Time (sec)	L1 dTLB load (Cyc)	L1 dTLB store (Cyc)
(1,48)	824.2743	14.112	108.373	124.326
(4,12)	740.0211	14.177	113.515	105.363
(8,6)	729.1005	14.244	104.564	89.3753
(2,24)	802.6687	14.314	105.328	82.5185
(16,3)	750.6159	14.615	100.924	90.2733
(3,16)	631.3098	14.772	97.9180	76.1889
(6,8)	667.4856	14.818	96.8957	58.0210
(12,4)	528.0411	15.057	97.0492	52.8966
(24,2)	1352.141	15.875	100.106	82.7514
(48,1)	1719.012	18.685	111.902	85.9282

Table 4.3: L1 dTLB PMC data for size 17408

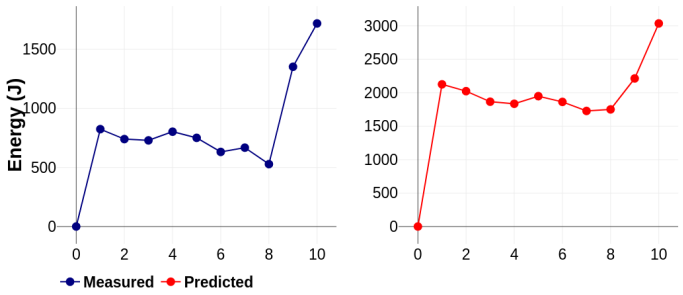
Comb.(g, t)	DEnergy (J)	Time (sec)	L1 dTLB load (Cyc)	L1 dTLB store (Cyc)
(4,12)	1320.0702	16.2478	105.961	122.191
(1,48)	1271.5506	16.3034	99.5398	63.7090
(8,6)	1266.3294	16.3166	95.7896	58.9096
(2,24)	1287.6882	16.4498	98.2180	74.6859
(16,3)	1250.5616	16.6824	95.2988	58.3551
(6,8)	1130.2412	16.9668	93.4336	47.9097
(3,16)	1052.0283	17.0187	90.5275	45.7483
(24,2)	1824.5795	18.0755	106.804	55.5686
(12,4)	1795.7680	20.5520	93.6595	46.5541
(48,1)	2164.1212	20.9868	96.6999	71.4943

between different combinations (g, t) is contained in the data provided by TLB_DATA performance group. This group provides data activity, such as load miss rate, store miss rate and walk page duration, in L1 data translation lookaside buffer (dTLB), a small specialized cache of recent page address translations. If a dTLB miss occurs, a system goes through the page table and retrieve the corresponding page table record from memory. This process named *page walk* and the duration of this walk has the highest correlation with dynamic energy consumption, based on our experiments.

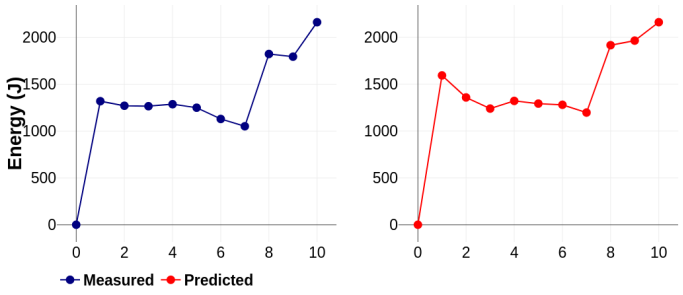
Non-negative multivariate regression is employed to construct our model of dynamic energy consumption based on the PMC data from dTLB:

$$E_{dynamic} = \beta_0 + \beta_1 \times T + \beta_2 \times L + \beta_3 \times S \tag{4.1}$$

where β_0 is the intercept, β_1 is the average CPU utilization, β_2 and β_3 are the regression coefficients for the PMC data. T is the execution time of the application, L is the time of page walk caused by load miss and S is the time of page walk caused by store miss in dTLB. The coefficients of the model



(a)



(b)

Figure 4.5: (a). Measured (left) and predicted (right) dynamic energy consumption of OpenBLAS DGEMM on HCLServer2 (S2) for workload size $m = n = 16384$*. (b). Measured (left) and predicted (right) dynamic energy consumption of OpenBLAS DGEMM on HCLServer2 (S2) for workload size* $m = n = 17408$*.*

($\{\beta_1, \beta_2, \beta_3\}$) are forced to be non-negative to avoid situations where large values for them can give rise to negative dynamic energy consumption prediction violating the fundamental energy conservation law of computing.

To test this model, two workload sizes 16384 and 17408 are used. The PMC data for these sizes used to train the model is shown in the tables 4.2 and 4.3 respectively. The rows of the tables are sorted in increasing order of time. The blue colour in the tables shows the rows that are in the Pareto-optimal front. The time of page walk (columns 4 and 5) is measured in cycles. As can be seen from the tables, the dynamic energy decreases as the number

of cycles decreases. There is however a trade-off between the execution time of application and the page walk time. If the execution time is too long, the small number of cycles anyway leads to high dynamic energy consumption.

There are two dynamic energy models constructed for workload sizes 16384 (Table 4.2) and 17408 (Table 4.3). The coefficients for the workload size 16384 are $\{\beta_0 = -3125.608, \beta_1 = 260.268, \beta_2 = 5.618, \beta_3 = 7.796\}$. The coefficients for 17408 are $\{\beta_0 = -4988.649, \beta_1 = 190.390, \beta_2 = 32.177, \beta_3 = 0.908\}$. Then model predicts the dynamic energy consumption and this predicted energy is compared with the dynamic energy measured using HCLWattsUp [165]. The figures 4.5a and 4.5b illustrate the comparision. The x axis represents the number of a row in tables 4.2, 4.3. It can be seen that the predicted dynamic energy demonstrates the same trend as the measured dynamic energy.

TLB activity has been the focus of research in [180]–[182] where the authors state that the address translation using the TLB consumes as much as 16% of the chip power on some processors. The authors propose different strategies to improve the reuse of TLB caches. In this work, the proposed solution method employing threadgroups (or grouping using multithreaded kernels) allows to fill the page table more evenly and reduce the duration of page walk along with dynamic energy consumption.

4.5 Conclusion

This chapter studied bi-objective optimization for performance and energy on multicore CPUs. The first application-level optimization method for bi-objective optimization of multithreaded data-parallel applications for performance and energy on a single multicore CPU was proposed. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) and the number of threads in each threadgroup. A given workload is partitioned equally between the threadgroups.

The method was demonstrated using four highly optimized multithreaded data-parallel applications, 2D fast Fourier transform based on FFTW and Intel MKL, and dense matrix-matrix multiplication written using Openblas DGEMM

and Intel MKL, on four modern multicore CPUs one of which is a single socket multicore CPU and the other three dual-socket with increasing number of physical cores per socket. The experimental results demonstrated in particular that optimizing for performance alone results in significant increase in dynamic energy consumption whereas optimizing for dynamic energy alone results in considerable performance degradation and that the method determined a good number of globally Pareto-optimal solutions.

Finally, a predictive dynamic energy model was proposed to explain the Pareto-optimal solutions determined by our bi-objective optimization solution method proposed in this chapter.

Chapter 5

Conclusion

Modern multicore CPUs have several inherent complexities, which are: a) Severe resource contention due to tight integration of tens of cores organized in multiple sockets with multi-level cache hierarchy and contending for shared on-chip resources such as last level lache (LLC), interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), and DRAM controllers; b) Non-uniform memory access (NUMA) where the time for memory access between a core and main memory is not uniform and where main memory is distributed between locality domains or groups called NUMA nodes; and c) Dynamic power management (DPM) of multiple power domains (CPU sockets, DRAM). These complexities pose serious challenges to single-objective optimization methods for performance and energy and bi-objective optimization methods for performance and energy due to the highly variational performance and energy profiles of data-parallel applications. They also provide the opportunity for bi-objective optimization of applications for performance and energy. The fundamentals of the dominant methods and algorithms currently used for performance and energy optimization were developed in the time when single-core CPUs dominated the computing landscape and cannot be applied to the modern profiles as they are extremely distinguished from these in a single-core era.

This thesis studied the influence of three-dimensional decision variable space on single-objective optimization as well as on bi-objective optimization

of data-parallel applications for performance and energy on modern multicore CPUs. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups. Using the highly optimized multithreaded data-parallel applications, 2D fast Fourier transform based on FFTW and IMKL, and dense matrix-matrix multiplication written using Openblas DGEMM and IMKL, on four modern multicore CPUs one of which is a single-socket multicore CPU and the other three dual-socket with an increasing number of physical cores per socket, the importance of these variables along with significant improvements by proposed solution methods for single- and bi-objective optimizations for performance and energy employing these variables were demonstrated.

Chapter 3 presented the methods for performance optimization only, using the workload distribution as a decision variable. Based on the experiments it was demonstrated that the solution optimizing for performance is a load imbalancing solution where the workload distribution between the processors is not load balanced in terms of execution time. Then, the methods for optimization of both performance and energy using two decision variables, the number of threadgroups and the number of threads in each threadgroup were proposed. The workload is always distributed equally between the threadgroups. The efficacy of the methods was demonstrated on both single-socket CPU as well as on dual-socket CPUs.

Chapter 4 studied bi-objective optimization for performance and energy using the same two decision variables, the number of threadgroups and the number of threads in each threadgroup on a single multicore CPU. Based on the experiments, it was illustrated that solution method provided good bi-objective trade-offs for performance and energy (Pareto-optimal solutions) on both single- and dual-socket CPUs. Furthermore, a predictive dynamic energy model employing performance monitoring counters (PMCs) as predictor variables was designed. It was used for the explanation of Pareto-optimal solutions determined by our solution method. The model employs PMCs from L1 dTLB, such as load and store miss duration, due to a

strong positive correlation between dynamic energy and these PMCs.

5.1 Future Work

The following future research directions are outlined:

- In the thesis author studied the influence of three-dimensional decision variable space on bi-objective optimization of applications for performance and energy on multicore CPUs. The three decision variables are: a). The number of identical multithreaded kernels (threadgroups) involved in the parallel execution of an application; b). The number of threads in each threadgroup; and c). The workload distribution between the threadgroups. Author proposed solution methods for optimization of multi-threaded data-parallel applications for performance using uneven workload distribution but where the number of threadgroups and number of threads per group are fixed. Author specifically studied load-imbalanced workload distribution for two and four threadgroups. Next, author proposed and studied the first application-level method for bi-objective optimization of multithreaded data-parallel applications for performance and energy. The method uses two decision variables, the number of identical multithreaded kernels (threadgroups) executing the application and the number of threads in each threadgroup, so that a given workload is partitioned equally between the threadgroups. A future line of research is a thorough exploration of the three-dimensional decision variable space where all the decision variables are varied. The number of possible workload distributions increases exponentially with increasing number of threadgroups employed in the execution of a data-parallel application and it would require employment of threadgroup-specific performance and energy models to reduce the complexity.

- By comparing and visualizing the patterns of the interplays between execution time and dynamic energy for different runtime configurations (number of threadgroups, number of threads per group, workload size),

a practical optimization guide for energy proportionality and bi-objective optimization on multicore CPUs will be explored that will serve as a suitable alternative to roofline models for performance and energy.

- The cost of building the five dimensional discrete graph with performance and dynamic energy as two objectives and the three decision variables can be quite expensive and prohibitive for employment in dynamic schedulers and self-adaptable data-parallel applications. Heuristic approaches to reduce the cost can be explored.
- This thesis proposed a predictive dynamic energy model employing performance monitoring counters (PMCs) as predictor variables, which is used to explain the Pareto-optimal solutions determined by designed method. The model can be made more comprehensive by studying thoroughly the influence of the three dimensional decision variable space on single-socket and dual-socket CPUs for all the four data-parallel applications.
- Extending the proposed solution methods to performance and energy optimization of applications especially in the current popular fields of artificial intelligence including deep learning, and data analytics.

Bibliography

[1] Intel Corporation, *Top500 list*, July, 2019. [Online]. Available: https://www.top500.org/lists/2019/06/ (cit. on p. 1).

[2] Scientific interest group (GIS), *Grid5000:home*, 2019. [Online]. Available: https://www.grid5000.fr/w/Grid5000:Home (cit. on p. 1).

[3] G. E. Moore, "Gramming more components onto integrated circuits," *Electronics*, vol. 38, p. 8, 1965 (cit. on p. 1).

[4] Dennard, *Dennard scaling*, 1974. [Online]. Available: https://en.wikipedia.org/wiki/Dennard_scaling (cit. on p. 1).

[5] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ACM, 2006, pp. 67–72 (cit. on p. 2).

[6] QPI. (2008). Intel quickpath interconnect, [Online]. Available: https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect (cit. on p. 2).

[7] AMDHT. (2001). Hypertransport, [Online]. Available: https://en.wikipedia.org/wiki/HyperTransport (cit. on p. 2).

[8] A. L. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, IEEE, 2004, p. 104 (cit. on pp. 6, 26).

[9] A. Lastovetsky and R. Reddy, “Data partitioning with a functional performance model of heterogeneous processors,” *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007 (cit. on pp. 6, 21, 25, 26).

[10] A. Lastovetsky and J. Twamley, “Towards a realistic performance model for networks of heterogeneous computers,” in *High Performance Computational Science and Engineering*, Springer, 2005, pp. 39–57 (cit. on p. 6).

[11] A. Lastovetsky and R. Reddy, “Data partitioning with a functional performance model of heterogeneous processors,” *The International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007 (cit. on p. 6).

[12] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, “An efficient, model-based CPU-GPU heterogeneous FFT library,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, IEEE, 2008, pp. 1–10 (cit. on pp. 6, 21, 25).

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: A programming model for heterogeneous multi-core systems,” in *ACM SIGOPS operating systems review*, ACM, vol. 42, 2008, pp. 287–296 (cit. on pp. 6, 21, 25).

[14] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. Van de Geijn, “Solving dense linear systems on platforms with multiple hardware accelerators,” in *ACM Sigplan Notices*, ACM, vol. 44, 2009, pp. 121–130 (cit. on pp. 6, 21, 25).

[15] C. Augonnet, S. Thibault, and R. Namyst, “Automatic calibration of performance models on heterogeneous multicore architectures,” in *European Conference on Parallel Processing*, Springer, 2009, pp. 56–65 (cit. on pp. 6, 21, 25).

[16] A. L. Lastovetsky, L. Szustak, and R. Wyrzykowski, “Model-based optimization of MPDATA on Intel Xeon Phi through load imbalancing,” *CoRR*, vol. abs/1507.01265, 2015 (cit. on pp. 7, 8, 27).

[17] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, (cit. on pp. 7, 8, 27).

[18] A. Lastovetsky and R. R. Manumachu, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017 (cit. on pp. 7, 8).

[19] S. WiLLiAmS, A. WAteRmAn, and D. PAtteRSon, "The roofline model offers insight on how to improve the performance of software and hardware.," *communicAtionS of the Acm*, vol. 52, no. 4, 2009 (cit. on pp. 7, 21).

[20] S. Ghose and J. Tse, "Cs 5220: Project 1 tuning the matrix multiply algorithm," (cit. on pp. 8, 22).

[21] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, p. 4 (cit. on pp. 8, 22).

[22] J. Huang and R. A. Van de Geijn, "Blislab: A sandbox for optimizing gemm," *arXiv preprint arXiv:1609.00076*, 2016 (cit. on pp. 8, 22).

[23] A. Shahid, M. Fahad, R. Reddy, and A. Lastovetsky, "Additivity: A selection criterion for performance events for reliable energy predictive modeling," *Supercomputing Frontiers and Innovations*, vol. 4, no. 4, pp. 50–65, 2017 (cit. on p. 8).

[24] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2008, pp. 1–8 (cit. on pp. 10, 52).

[25] H. Hajimiri, P. Mishra, and S. Bhunia, "Dynamic cache tuning for efficient memory based computing in multicore architectures," in *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, IEEE, 2013, pp. 49–54 (cit. on pp. 11, 39).

[26] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017 (cit. on pp. 11, 14, 45, 108, 109).

[27] R. Reddy Manumachu and A. L. Lastovetsky, "Design of self-adaptable data parallel applications on multicore clusters automatically optimized for performance and energy through load distribution," *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, e4958, (cit. on pp. 11, 14, 45, 108, 109).

[28] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, no. 12, pp. 33–37, 2007 (cit. on pp. 12, 105).

[29] Intel Corporation, *Intel MKL FFT - fast fourier transforms*, 2018. [Online]. Available: `https://software.intel.com/en-us/mkl/features/fft` (cit. on pp. 13, 155, 156).

[30] OpenBLAS, *OpenBLAS: An optimized BLAS library*, 2016. [Online]. Available: `http://www.openblas.net/` (cit. on pp. 13, 21).

[31] FFTW, *Fastest fourier transform in the west*, 2018. [Online]. Available: `http://www.fftw.org/` (cit. on pp. 13, 22, 107).

[32] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky., *ZZGemmOOC: Multi-GPU out-of-core routines for dense matrix multiplization*, 2019. [Online]. Available: `https://git.ucd.ie/hcl/zzgemmooc.git` (cit. on pp. 13, 107).

[33] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "Out-of-core implementation for accelerator kernels on heterogeneous clouds," *The Journal of Supercomputing*, vol. 74, no. 2, pp. 551–568, 2018 (cit. on pp. 13, 107).

[34] S. Khokhriakov, R. R. Manumachu, and A. Lastovetsky, "Performance optimization of multithreaded 2d fast fourier transform on multicore processors using load imbalancing parallel computing method," *IEEE Access*, vol. 6, pp. 64 202–64 224, 2018 (cit. on pp. 13, 107, 109).

[35] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 160–177, 2018 (cit. on pp. 14, 44, 45, 108, 109).

[36] H. Khaleghzadeh, M. Fahad, A. Shahid, R. Reddy, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous hpc platforms for performance and energy through workload distribution," *CoRR*, vol. abs/1907.04080, 2019. arXiv: 1907.04080. [Online]. Available: http://arxiv.org/abs/1907.04080 (cit. on pp. 14, 109).

[37] N. Ding, S. Xu, Z. Song, B. Zhang, J. Li, and Z. Zheng, "Using hardware counter-based performance model to diagnose scaling issues of hpc applications," *Neural Computing and Applications*, vol. 31, no. 5, pp. 1563–1575, 2019 (cit. on p. 21).

[38] J.-P. Lehr, "Counting performance: Hardware performance counter and compiler instrumentation," *Informatik 2016*, 2016 (cit. on p. 21).

[39] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ACM, 2018, pp. 457–468 (cit. on p. 21).

[40] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, IEEE, 2008, pp. 59–67 (cit. on p. 21).

[41] D. Dauwe, E. Jonardi, R. D. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel, "Hpc node performance and energy modeling with the co-location of applications," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4771–4809, 2016 (cit. on p. 21).

[42] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-gpu platforms using functional performance models," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2506–2518, 2014 (cit. on pp. 21, 26, 27, 50).

[43] A. Lastovetsky and R. R. Manumachu, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2016 (cit. on pp. 21, 50, 53).

[44] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of eulag kernel on intel xeon phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2016 (cit. on pp. 21, 50).

[45] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008 (cit. on p. 21).

[46] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, no. 99, pp. 1–16, 2018 (cit. on p. 22).

[47] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005 (cit. on p. 22).

[48] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby, "A balanced approach to application performance

tuning," in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2009, pp. 111–125 (cit. on p. 22).

[49] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms," *Journal of Parallel and Distributed Computing*, vol. 69, no. 9, pp. 762–777, 2009 (cit. on p. 22).

[50] PeXL, *Maqao (modular assembly quality analyzer and optimizer)*, 2004. [Online]. Available: `http://www.maqao.org` (cit. on p. 22).

[51] M. Hashimoto, M. Terai, T. Maeda, and K. Minami, "Cca/ebt: Code comprehension assistance tool for evidence-based performance tuning," 2018 (cit. on p. 22).

[52] M. Rajagopalan, B. T. Lewis, and T. A. Anderson, "Thread scheduling for multi-core platforms.," in *HotOS*, 2007 (cit. on p. 23).

[53] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*, IEEE, 2005, pp. 340–351 (cit. on p. 23).

[54] P. Radojkovic, V. Cakarevic, J. Verdu, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread assignment of multithreaded network applications in multicore/multithreaded processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2513–2525, 2013 (cit. on p. 23).

[55] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 362–373 (cit. on p. 24).

[56] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems," in *IEEE International Symposium on High-Performance Comp Architecture*, IEEE, 2012, pp. 1–12 (cit. on p. 24).

[57] M. De Vuyst, R. Kumar, and D. M. Tullsen, “Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors,” in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2006, 10–pp (cit. on p. 24).

[58] Y. Wen, Z. Wang, and M. F. O’boyle, “Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms,” in *2014 21st International Conference on High Performance Computing (HiPC)*, IEEE, 2014, pp. 1–10 (cit. on p. 24).

[59] H. Khaleghzadeh, H. Deldari, R. Reddy, and A. Lastovetsky, “Hierarchical multicore thread mapping via estimation of remote communication,” *The Journal of Supercomputing*, vol. 74, no. 3, pp. 1321–1340, 2018 (cit. on p. 24).

[60] O. Franek, “A simple method for static load balancing of parallel fdtd codes,” in *Electromagnetics in Advanced Applications (ICEAA), 2016 International Conference on*, IEEE, 2016, pp. 587–590 (cit. on p. 25).

[61] R. L. Cariño and I. Banicescu, “Dynamic load balancing with adaptive factoring methods in scientific applications,” *The Journal of Supercomputing*, vol. 44, no. 1, pp. 41–63, 2008 (cit. on p. 26).

[62] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García, “Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE,” *J. Supercomput.*, vol. 58, no. 2, Nov. 2011 (cit. on p. 26).

[63] G. Cybenko, “Dynamic load balancing for distributed memory multiprocessors,” *Journal of parallel and distributed computing*, vol. 7, no. 2, pp. 279–301, 1989 (cit. on p. 26).

[64] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, “Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms,” *IEEE transactions on parallel and distributed systems*, vol. 16, no. 4, pp. 289–299, 2005 (cit. on p. 26).

[65] J. Bahi, R. Couturier, and F. Vernier, “Synchronous distributed load balancing on dynamic networks,” *Journal of Parallel and Distributed Computing*, vol. 65, no. 11, pp. 1397–1405, 2005 (cit. on p. 26).

[66] F. Liu, Y. Chen, and W. S. Wong, “An asynchronous load balancing scheme for multi-server systems,” in *Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), IEEE Annual*, IEEE, 2016, pp. 1–7 (cit. on p. 26).

[67] P. K. Smolarkiewicz, “Multidimensional positive definite advection transport algorithm: An overview,” *International Journal for Numerical Methods in Fluids*, vol. 50, no. 10, pp. 1123–1144, 2006 (cit. on p. 27).

[68] A. Lastovetsky and R. Reddy, “New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017 (cit. on pp. 28, 59, 61, 68, 70, 74, 166).

[69] R. Reddy and A. Lastovetsky, “Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy,” *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 160–177, 2017 (cit. on pp. 28, 46).

[70] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, “A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018. DOI: 10.1109/TPDS.2018.2827055 (cit. on pp. 28, 46, 59, 61, 68, 70, 166).

[71] L. Niu and G. Quan, “Reducing both dynamic and leakage energy consumption for hard real-time systems,” in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM, 2004, pp. 140–148 (cit. on pp. 29, 36).

[72] S. Mittal, “A survey of techniques for improving energy efficiency in embedded computing systems,” *arXiv preprint arXiv:1401.0765*, 2014 (cit. on p. 29).

[73] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 37, 2017 (cit. on p. 30).

[74] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr, and R. Bianchini, "Energy conservation in heterogeneous server clusters," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, 2005, pp. 186–195 (cit. on p. 30).

[75] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-system power analysis and modeling for server environments," International Symposium on Computer Architecture-IEEE, 2006 (cit. on p. 31).

[76] X. Feng, R. Ge, and K. W. Cameron, "Power and energy profiling of scientific applications on distributed systems," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005, pp. 34–34 (cit. on p. 31).

[77] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using simplepower," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 95–106, 2000 (cit. on p. 31).

[78] P. Gschwandtner, M. Knobloch, B. Mohr, D. Pleiter, and T. Fahringer, "Modeling cpu energy consumption of hpc applications on the ibm power7," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, 2014, pp. 536–543 (cit. on p. 31).

[79] R. Zamani and A. Afsahi, "Adaptive estimation and prediction of power and performance in high performance computing," *Computer Science-Research and Development*, vol. 25, no. 3-4, pp. 177–186, 2010 (cit. on p. 31).

[80] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky, “Improving the accuracy of energy predictive models for multicore cpus using additivity of performance monitoring counters,” in *International Conference on Parallel Computing Technologies*, Springer, 2019, pp. 51–66 (cit. on p. 32).

[81] D. C. Snowdon, S. Ruocco, and G. Heiser, “Power management and dynamic voltage scaling: Myths and facts,” in *Proceedings of the 2005 workshop on power aware real-time computing*, vol. 12, 2005, pp. 1–7 (cit. on p. 33).

[82] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, “Coscale: Coordinating cpu and memory system dvfs in server systems,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2012, pp. 143–154 (cit. on p. 33).

[83] Z. Lai, K. T. Lam, C.-L. Wang, J. Su, Y. Yan, and W. Zhu, “Latency-aware dynamic voltage and frequency scaling on many-core architectures for data-intensive applications,” in *2013 International Conference on Cloud Computing and Big Data*, IEEE, 2013, pp. 78–83 (cit. on p. 34).

[84] G. Chen, K. Huang, and A. Knoll, “Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 111, 2014 (cit. on p. 34).

[85] A. K. Datta and R. Patel, “Cpu scheduling for power/energy management on multicore processors using cache miss and context switch data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1190–1199, 2013 (cit. on p. 34).

[86] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya, “Some observations on optimal frequency selection in dvfs-based energy consumption minimization,” *Journal of Parallel and Distributed Computing*, vol. 71, no. 8, pp. 1154–1164, 2011 (cit. on p. 34).

[87] S. Yang, R. A. Shafik, G. V. Merrett, E. Stott, J. M. Levine, J. Davis, and B. M. Al-Hashimi, “Adaptive energy minimization of embedded heterogeneous systems using regression-based learning,” in *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, IEEE, 2015, pp. 103–110 (cit. on p. 34).

[88] F. P. Miller, A. F. Vandome, and J. McBrewster, “Advanced configuration and power interface: Open standard, operating system, power management, cross-platform, intel corporation, microsoft, toshiba,... sleep mode, hibernate (os feature), synonym,” 2009 (cit. on p. 35).

[89] L. Benini, A. Bogliolo, and G. De Micheli, “A survey of design techniques for system-level dynamic power management,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 8, no. 3, pp. 299–316, 2000 (cit. on p. 35).

[90] W. L. Bircher and L. K. John, “Analysis of dynamic power management on multi-core processors,” in *Proceedings of the 22nd annual international conference on Supercomputing*, ACM, 2008, pp. 327–338 (cit. on p. 35).

[91] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo, “Adaptive power management for real-time event streams,” in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, IEEE Press, 2010, pp. 7–12 (cit. on p. 35).

[92] E.-Y. Chung, L. Benini, and G. De Micheli, “Dynamic power management using adaptive learning tree,” in *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, IEEE Press, 1999, pp. 274–279 (cit. on p. 35).

[93] A. Beloglazov and R. Buyya, “Energy efficient resource management in virtualized cloud data centers,” in *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, IEEE Computer Society, 2010, pp. 826–831 (cit. on p. 35).

[94] W.-K. Lee, S.-W. Lee, and W.-O. Siew, “Hybrid model for dynamic power management,” *IEEE Transactions on Consumer Electronics*, vol. 55, no. 2, pp. 656–664, 2009 (cit. on p. 35).

[95] C. Imes and H. Hoffmann, “Minimizing energy under performance constraints on embedded platforms: Resource allocation heuristics for homogeneous and single-isa heterogeneous multi-cores,” *ACM SIGBED Review*, vol. 11, no. 4, pp. 49–54, 2015 (cit. on p. 36).

[96] J. Trajkovic, A. V. Veidenbaum, and A. Kejariwal, “Improving sdram access energy efficiency for low-power embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 24, 2008 (cit. on p. 36).

[97] S. Song, C.-Y. Su, R. Ge, A. Vishnu, and K. W. Cameron, “Iso-energy-efficiency: An approach to power-constrained parallel computation,” in *2011 IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2011, pp. 128–139 (cit. on p. 36).

[98] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, “Multicore dimm: An energy efficient memory module with independently controlled drams,” *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 5–8, 2008 (cit. on p. 36).

[99] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, “Power aware page allocation,” *ACM Sigplan Notices*, vol. 35, no. 11, pp. 105–116, 2000 (cit. on p. 37).

[100] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, “Memscale: Active low-power modes for main memory,” in *ACM SIGARCH Computer Architecture News*, ACM, vol. 39, 2011, pp. 225–238 (cit. on p. 37).

[101] J. Lin, H. Zheng, Z. Zhu, E. Gorbatov, H. David, and Z. Zhang, “Software thermal management of dram memory for multicore systems,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 337–348, 2008 (cit. on p. 37).

[102] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ACM, 2011, pp. 31–40 (cit. on p. 37).

[103] M. Banikazemi, D. Poff, and B. Abali, "Pam: A novel performance/power aware meta-scheduler for multi-core systems," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE, 2008, pp. 1–12 (cit. on p. 37).

[104] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European conference on Computer systems*, ACM, 2010, pp. 153–166 (cit. on p. 37).

[105] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 1, p. 15, 2015 (cit. on p. 38).

[106] J. Qian, H. Jiang, W. Srisa-An, S. Seth, S. Skelton, and J. Moore, "Energy-efficient i/o thread schedulers for nvme ssds on numa," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, IEEE, 2017, pp. 569–578 (cit. on p. 38).

[107] C. Hankendi and A. K. Coskun, "Reducing the energy cost of computing through efficient co-scheduling of parallel workloads," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2012, pp. 994–999 (cit. on p. 38).

[108] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2011, pp. 948–953 (cit. on p. 38).

[109] G. Chen, B. Hu, K. Huang, A. Knoll, D. Liu, and T. Stefanov, "Automatic cache partitioning and time-triggered scheduling for real-time mpsocs,"

in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, IEEE, 2014, pp. 1–8 (cit. on p. 39).

[110] V. Delaluz, M Kandemir, A. Sivasubramaniam, M. J. Irwin, and N. Vijaykrishnan, "Reducing dtlb energy through dynamic resizing," in *Proceedings 21st International Conference on Computer Design*, IEEE, 2003, pp. 358–363 (cit. on p. 39).

[111] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative partitioning: Energy-efficient cache partitioning for high-performance cmps," in *IEEE International Symposium on High-Performance Comp Architecture*, IEEE, 2012, pp. 1–12 (cit. on p. 39).

[112] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang, "Thermal vs energy optimization for dvfs-enabled processors in embedded systems," in *8th International Symposium on Quality Electronic Design (ISQED'07)*, IEEE, 2007, pp. 204–209 (cit. on p. 39).

[113] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "The design of deetm: A framework for dynamic energy efficiency and temperature management," *Journal of Instruction-Level Parallelism*, vol. 3, pp. 1–31, 2002 (cit. on p. 39).

[114] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, IEEE, 2003, pp. 2–13 (cit. on p. 40).

[115] A. Cohen, F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy, "On estimating optimal performance of cpu dynamic thermal management," *IEEE Computer Architecture Letters*, vol. 2, no. 1, pp. 6–6, 2003 (cit. on p. 40).

[116] R. Ayoub, R. Nath, and T. Rosing, "Jetc: Joint energy thermal and cooling management for memory and cpu subsystems in servers," in *IEEE International Symposium on High-Performance Comp Architecture*, IEEE, 2012, pp. 1–12 (cit. on p. 40).

[117] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of energy-cognizant scheduling techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1447–1464, 2012 (cit. on p. 40).

[118] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive scheduling for systems with asymmetric memory hierarchies," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 641–654 (cit. on p. 40).

[119] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*, ACM, 2010, pp. 139–152 (cit. on p. 41).

[120] X. Fan, Y. Sui, and J. Xue, "Contention-aware scheduling for asymmetric multicore processors," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2015, pp. 742–751 (cit. on p. 41).

[121] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, IEEE, 2007, pp. 1–11 (cit. on p. 41).

[122] Y. Wang, X. Wang, and Y. Chen, "Energy-efficient virtual machine scheduling in performance-asymmetric multi-core architectures," in *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*, IEEE, 2012, pp. 288–294 (cit. on p. 41).

[123] F. A. Bower, D. J. Sorin, and L. P. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," *IEEE micro*, vol. 28, no. 3, pp. 17–25, 2008 (cit. on p. 41).

[124] J. Demmel, A. Gearhart, B. Lipshitz, and O. Schwartz, "Perfect strong scaling using no additional energy," in *2013 IEEE 27th International*

Symposium on Parallel and Distributed Processing, IEEE, 2013, pp. 649–660 (cit. on p. 42).

[125] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, “A roofline model of energy,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, 2013, pp. 661–672 (cit. on p. 42).

[126] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. S. Nikolopoulos, “Application-level energy awareness for openmp,” in *International Workshop on OpenMP*, Springer, 2015, pp. 219–232 (cit. on p. 42).

[127] V. R. Silva, A. Furtunato, K. Georgiou, K. Eder, and S. Xavier-de Souza, “Energy-optimal configurations for single-node hpc applications,” *arXiv preprint arXiv:1805.00998*, 2018 (cit. on p. 42).

[128] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, “Workload and power budget partitioning for single-chip heterogeneous processors,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 401–410 (cit. on p. 42).

[129] R. CHIŞ, A. Florea, C. Buduleci, and L. VINŢAN, “Multi-objective optimization for an enhanced multi-core sniper simulator,” 2018 (cit. on p. 43).

[130] B. Subramaniam and W.-c. Feng, “Statistical power and performance modeling for optimizing the energy efficiency of scientific computing,” in *Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, IEEE Computer Society, 2010, pp. 139–146 (cit. on p. 43).

[131] H. F. Sheikh and I. Ahmad, “Dynamic task graph scheduling on multicore processors for performance, energy, and temperature optimization,” in *2013 International Green Computing Conference Proceedings*, IEEE, 2013, pp. 1–6 (cit. on p. 43).

[132] H. Lei, R. Wang, T. Zhang, Y. Liu, and Y. Zha, "A multi-objective co-evolutionary algorithm for energy-efficient scheduling on a green data center," *Computers & Operations Research*, vol. 75, pp. 103–117, 2016 (cit. on p. 44).

[133] N. K. Sharma and G. R. M. Reddy, "Multi-objective energy efficient virtual machines allocation at the cloud data center," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 158–171, 2016 (cit. on p. 44).

[134] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng, "Energy-saving virtual machine placement in cloud data centers," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, IEEE, 2013, pp. 618–624 (cit. on p. 44).

[135] M. Mezmaz, N. Melab, Y. Kessaci, Y. Lee, E.-G. Talbi, A. Zomaya, and D. Tuyttens, "A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 11, pp. 1497 –1508, 2011 (cit. on p. 44).

[136] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12, IEEE Computer Society, 2012, pp. 300–309 (cit. on p. 44).

[137] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755 –768, 2012, Special Section: Energy efficiency in large-scale distributed systems (cit. on p. 44).

[138] Y. Kessaci, N. Melab, and E.-G. Talbi, "A pareto-based metaheuristic for scheduling hpc applications on a geographically distributed cloud federation," *Cluster Computing*, vol. 16, no. 3, pp. 451–468, Sep. 2013 (cit. on p. 44).

[139] J. J. Durillo, V. Nae, and R. Prodan, “Multi-objective energy-efficient workflow scheduling using list-based heuristics,” *Future Generation Computer Systems*, vol. 36, pp. 221 –236, 2014 (cit. on p. 44).

[140] J. Kołodziej, S. U. Khan, L. Wang, and A. Y. Zomaya, “Energy efficient genetic-based schedulers in computational grids,” *Concurr. Comput. : Pract. Exper.*, vol. 27, no. 4, pp. 809–829, Mar. 2015 (cit. on p. 44).

[141] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, “Analyzing the energy-time trade-off in high-performance computing applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 835–848, 2007 (cit. on p. 44).

[142] A. Langer, H. Dokania, L. V. Kalé, and U. S. Palekar, “Analyzing energy-time tradeoff in power overprovisioned hpc data centers,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IEEE, 2015, pp. 849–854 (cit. on p. 44).

[143] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, “Analyzing the energy-time trade-off in high-performance computing applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 6, Jun. 2007 (cit. on p. 44).

[144] I. Ahmad, S. Ranka, and S. U. Khan, “Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–6 (cit. on p. 44).

[145] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, “A roofline model of energy,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, 2013, pp. 661–672 (cit. on p. 44).

[146] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, “Algorithmic time, energy, and power on candidate HPC compute building blocks,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, IEEE, 2014, pp. 447–457 (cit. on p. 44).

[147] P. Balaprakash, A. Tiwari, and S. M. Wild, “Multi objective optimization of HPC kernels for performance, power, and energy,” in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*, A. S. Jarvis, A. S. Wright, and D. S. Hammond, Eds. Springer International Publishing, 2014, pp. 239–260 (cit. on p. 44).

[148] M. A. Aba, L. Zaourar, and A. Munier, “Approximation algorithm for scheduling a chain of tasks on heterogeneous systems,” in *European Conference on Parallel Processing*, Springer, 2017, pp. 353–365 (cit. on p. 44).

[149] B. Subramaniam and W. C. Feng, “Statistical power and performance modeling for optimizing the energy efficiency of scientific computing,” in *2010 IEEE/ACM Int’l Conference on Int’l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010 (cit. on p. 44).

[150] S. Song, C. Y. Su, R. Ge, A. Vishnu, and K. W. Cameron, “Iso-energy-efficiency: An approach to power-constrained parallel computation,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 128–139 (cit. on p. 44).

[151] J. Demmel, A. Gearhart, B. Lipshitz, and O. Schwartz, “Perfect strong scaling using no additional energy,” in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013 (cit. on p. 44).

[152] M. Drozdowski, J. M. Marszalkowski, and J. Marszalkowski, “Energy trade-offs analysis using equal-energy maps,” *Future Generation Computer Systems*, vol. 36, pp. 311–321, 2014 (cit. on p. 44).

[153] J. M. Marszalkowski, M. Drozdowski, and J. Marszalkowski, “Time and energy performance of parallel systems with hierarchical memory,” *Journal of Grid Computing*, vol. 14, no. 1, pp. 153–170, 2016 (cit. on p. 44).

[154] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. Chong, "Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1633–1646, 2016 (cit. on p. 44).

[155] E. Gabaldon, J. L. Lerida, F. Guirado, and J. Planes, "Blacklist muti-objective genetic algorithm for energy saving in heterogeneous environments," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 354–369, 2017 (cit. on p. 44).

[156] S. U. Khan, "A goal programming approach for the joint optimization of energy consumption and response time in computational grids," in *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, IEEE, 2009, pp. 410–417 (cit. on p. 45).

[157] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *ACM SIGPLAN Notices*, ACM, vol. 49, 2014, pp. 345–360 (cit. on pp. 46, 53).

[158] Y. Guo, "A scalable locality-aware adaptive work-stealing scheduler for multi-core task parallelism," PhD thesis, 2011 (cit. on pp. 46, 53).

[159] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 297–314, 2012 (cit. on pp. 46, 53).

[160] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 513–528, 2014 (cit. on pp. 46, 53).

[161] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of eulag kernel on intel xeon phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2017 (cit. on p. 51).

[162] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018 (cit. on p. 53).

[163] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, pp. 195–217, 2011 (cit. on p. 74).

[164] A. Lastovetsky, R. Reddy, V. Rychkov, and D. Clarke, "Design and implementation of self-adaptable parallel algorithms for scientific computing on highly heterogeneous HPC platforms," *arXiv preprint arXiv:1109.3074*, 2011 (cit. on p. 74).

[165] HCL, *HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter*, 2016. [Online]. Available: `http://git.ucd.ie/hcl/hclwattsup` (cit. on pp. 94, 115, 123).

[166] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, Jan. 2015 (cit. on p. 106).

[167] Y. G. Kim, M. Kim, and S. W. Chung, "Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1878–1889, 2017 (cit. on p. 106).

[168] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 948–953 (cit. on p. 106).

[169] G. Chen, K. Huang, J. Huang, and A. Knoll, "Cache partitioning and scheduling for energy optimization of real-time mpsocs," in *2013 IEEE*

24th International Conference on Application-Specific Systems, Architectures and Processors, 2013, pp. 35–41 (cit. on p. 106).

[170] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012 (cit. on p. 106).

[171] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin, "Dynamic thermal management through task scheduling," in *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, 2008, pp. 191–201 (cit. on p. 107).

[172] R. Z. Ayoub and T. S. Rosing, "Predict and act: Dynamic thermal management for multi-core processors," in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '09, ACM, 2009, pp. 99–104 (cit. on p. 107).

[173] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–11 (cit. on p. 107).

[174] E. Humenay, D. Tarjan, and K. Skadron, "Impact of process variations on multicore performance symmetry," in *2007 Design, Automation Test in Europe Conference Exhibition*, 2007, pp. 1–6 (cit. on p. 107).

[175] Intel® Math Kernel Library (Intel® MKL), *Intel MKL FFT - fast fourier transforms*, 2019. [Online]. Available: `https://software.intel.com/en-us/mkl` (cit. on p. 107).

[176] Z. Xianyi, *Openblas, an optimized blas library*, 2019. [Online]. Available: `http://www.netlib.org/blas/` (cit. on p. 107).

[177] K. Miettinen, *Nonlinear multiobjective optimization*. Kluwer, 1999 (cit. on pp. 110–112).

[178] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74 (cit. on pp. 110, 111).

[179] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *2010 39th International Conference on Parallel Processing Workshops*, IEEE, 2010, pp. 207–216 (cit. on p. 120).

[180] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam, “Reducing data tlb power via compiler-directed address generation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 312–324, 2007 (cit. on p. 123).

[181] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, “Energy-efficient address translation,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 631–643 (cit. on p. 123).

[182] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant memory mappings for fast access to large memories,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15, ACM, 2015, pp. 66–78 (cit. on p. 123).

[183] W. Chu and B. Champagne, “A noise-robust FFT-based auditory spectrum with application in audio classification,” *IEEE Transactions on audio, speech, and language processing*, vol. 16, no. 1, pp. 137–150, 2008 (cit. on p. 155).

[184] A. Sapena-Bañó, M. Pineda-Sanchez, R. Puche-Panadero, J. Martinez-Roman, and D. Matić, “Fault diagnosis of rotating electrical machines in transient regime using a single stator current’s FFT,” *IEEE Transactions on Instrumentation and Measurement*, vol. 64, no. 11, pp. 3137–3146, 2015 (cit. on p. 155).

[185] M. Kang, J. Kim, L. M. Wills, and J.-M. Kim, “Time-varying and multiresolution envelope analysis and discriminative feature analysis for bearing fault diagnosis.,” *IEEE Trans. Industrial Electronics*, vol. 62, no. 12, pp. 7749–7761, 2015 (cit. on p. 155).

[186] M. Naoues, D. Noguet, L. Alaus, and Y. Louët, "A common operator for FFT and FEC decoding," *Microprocessors and Microsystems*, vol. 35, no. 8, pp. 708–715, 2011 (cit. on p. 155).

[187] J. P. Barbosa, A. P. Ferreira, R. C. Rocha, E. S. Albuquerque, J. R. Reis, D. S. Albuquerque, and E. N. Barros, "A high performance hardware accelerator for dynamic texture segmentation," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 639–645, 2015 (cit. on p. 155).

[188] cuFFT, *Optimized FFT routines for Nvidia graphics processors*, 2018. [Online]. Available: `https://docs.nvidia.com/cuda/cufft/index.html` (cit. on pp. 155, 156).

[189] clFFT, *Optimized FFT routines for AMD graphics processors*, 2018. [Online]. Available: `https://gpuopen.com/compute-product/clfft/` (cit. on p. 155).

Appendix A

Analysis of Performance Profiles of Data-parallel Applications on Modern Multicore CPUs

Experimental Set-Up

All performance profiles (speed functions) for the applications were obtained on a modern Intel Haswell multicore server consisting of 2 sockets of 18 physical cores each (specification shown in Table A.1). All applications using 36 threads. There are not used any special environment affinity variables during the execution of the applications.

Technical Specifications	Intel Haswell Server
Processor	Intel Xeon CPU E5-2699 v3 @ 2.30GHz
OS	CentOS 7.1.1503
Microarchitecture	Haswell
Memory	256 GB
Core(s) per socket	18
Socket(s)	2
NUMA node(s)	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	46080 KB
NUMA node0 CPU(s)	0-17,36-53
NUMA node1 CPU(s)	18-35,54-71

Table A.1: Specification of the Intel Haswell server used to construct the performance profiles.

We also will be referring frequently to width of performance variations in a performance profile. It is the difference of speeds between two subsequent local minima (s_1) and maxima (s_2) as shown below:

$$variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100 \tag{A.1}$$

To make sure the experimental results are reliable, a statistical methodology described in Appendix B is used. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%). For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. The speed/performance values shown in the graphical plots throughout this work are the sample means.

The total number of problem sizes $N \times N$ experimented is around 1000 with N ranging from 128 to 64000 with a step size of 64, $\{128, 192, ..., 64000\}$.

2D Fast Fourier Transform

Fast Fourier transform (FFT) is a key routine employed in application domains such as molecular dynamics, computational fluid dynamics, signal processing, image processing, and condition monitoring systems [183]–[187]. It is so fundamental that hardware vendors provide libraries containing 1D, 2D, and 3D FFT routines highly optimized for their processors. For example, Intel Math Kernel library (IMKL) [29] provides extensively optimized FFT routines for Intel processors, cuFFT [188] for Nvidia CUDA GPUs, and clFFT [189] for AMD processors.

The theoretical computational complexity and arithmetic intensity of 2D FFT lie between those for highly memory-bound and highly compute-bound routines. For a 2D FFT of complex input and output, its computational complexity is $O(N^2 \times log_2 N)$, which lies between those for highly memory-bound applications ($O(N^2)$ for matrix-vector multiplication MxV of a dense matrix $N \times N$) and highly compute-bound applications ($O(N^3)$ for matrix-matrix multiplication of two dense $N \times N$ matrices). Its arithmetic

N	FFTW_ESTIMATE (Sec)	FFTW_MEASURE (Sec)	PATIENT (Sec)
20160	3	31	5015
20480	16	41	2549
20672	6.5	3004	8228
21120	3.6	31	2746
21440	4	32	1367
21632	14.5	2937	9754

Table A.2: Execution times in seconds for FFTW-3.3.7 on the Intel Haswell multicore server for three different planner flags.

intensity (I_A) ($I_A = \frac{\#flops}{\#\text{memory accesses}} = O(log_2 N)$) lies between those for highly memory-bound applications (I_A for MxV is 1) and highly compute-bound applications (I_A for MxM is N). Code tuning techniques such as multithreading, Fused Multiply-Add (FMA), SIMD acceleration using specialized instruction sets such as SSE2, AltiVec, etc. are used to optimize it for different processor architectures.

In the experiments are used three multithreaded FFT applications for comparison written using the packages FFTW-2.1.5, FFTW-3.3.7, and IMKL FFT. The packages, FFTW-2.1.5 and FFTW-3.3.7, are open-source. Hardware vendor libraries ([29], [188]) offer optimized implementations of the FFTW interface for their processors.

The performance profiles are shown for only one planner flag, FFTW_ESTIMATE. The experiments with two other planner flags were also performed, {FFTW_MEASURE, FFTW_PATIENT} (Table A.2). The execution times for these flags however are prohibitively larger compared to FFTW_ESTIMATE and severe variations are present. The long execution times are due to the lengthy times to create the plans because FFTW_MEASURE tries to find an optimized plan by computing several FFTs whereas FFTW_PATIENT considers a wider range of algorithms to find a more optimal plan.

In the graphs showing speed functions, the speed of execution of a 2D-DFT of complex signal matrix of size $N \times N$ is equal to $\frac{5.0*N^2*\log_2(N^2)}{t}$, where t is the time of execution of the 2D-DFT.

Figure A.1, A.2 show the performance profiles of FFTW 2.1.5 versus FFTW 3.3.7. Following are the key observations:

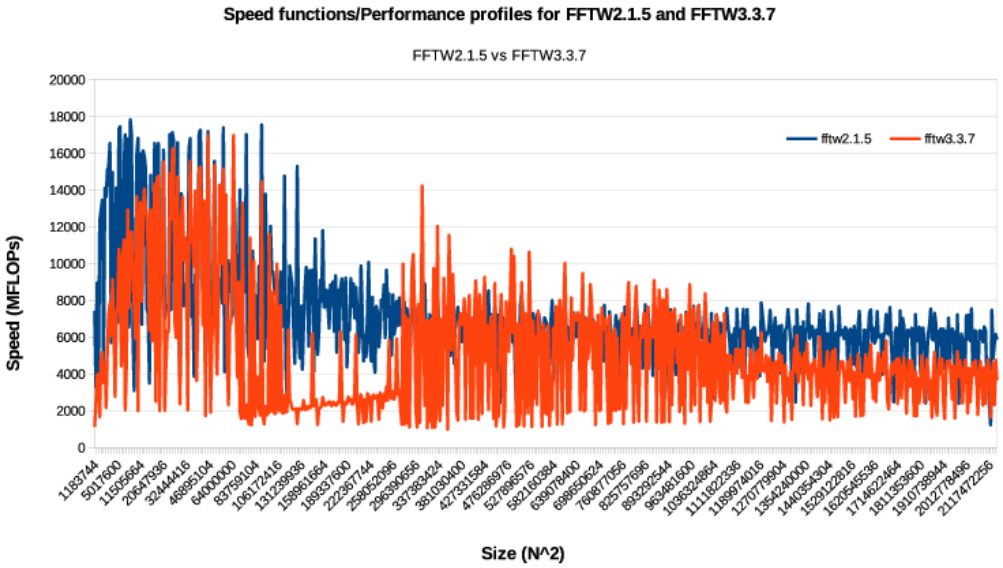


Figure A.1: Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and FFTW-3.3.7. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each.

- The width of performance variations in FFTW-3.3.7 is considerably greater than that for FFTW-2.1.5.
- The peak performance of FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$) whereas that for FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$). The average performances of FFTW-2.1.5 and FFTW-3.3.7 are 7033 MFLOPs and 5065 MFLOPs. The ratio of average to peak performances of FFTW-2.1.5 and FFTW-3.3.7 are 40% and 30%.
- FFTW-2.1.5 is better than FFTW-3.3.7 by around 38% (on an average). There are 529 problem sizes (out of 1000) where the performance of FFTW-2.1.5 is better than FFTW-3.3.7.

Figures A.3, A.4 present the performance comparisons between FFTW-2.1.5 and IMKL FFT. The most important observations are as follows:

- The peak performance of FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$) whereas that for IMKL FFT is 39424 MFLOPs ($N = 1792$). The ratio of

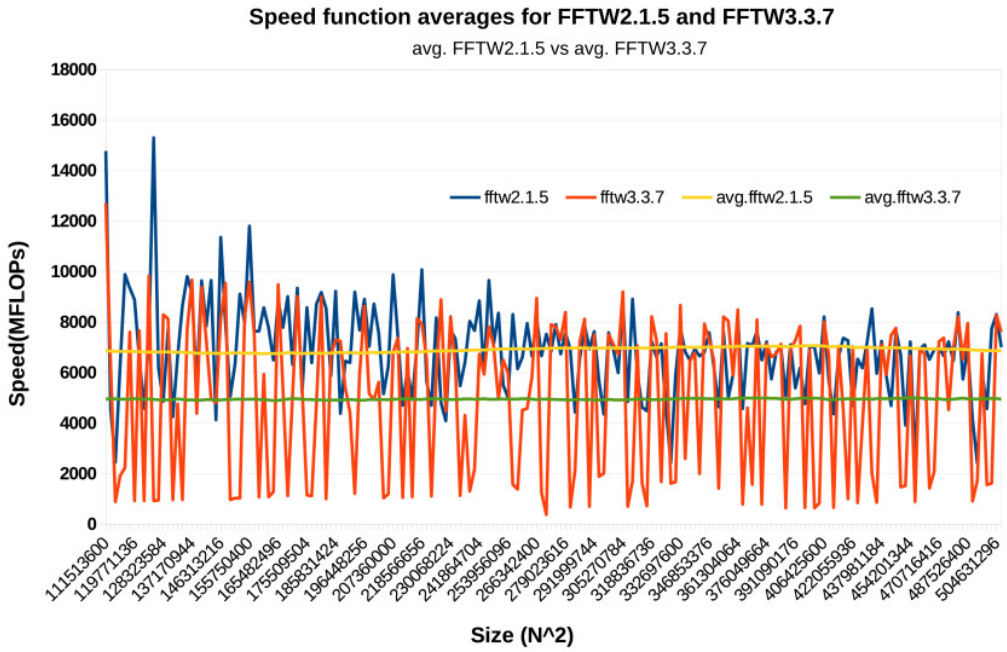


Figure A.2: The average speeds of FFTW-2.1.5 vs FFTW-3.3.7.

average to peak performances of FFTW-2.1.5 and IMKL FFT are 40% and 24%.

- The average performance of IMKL FFT is around 9572 MFLOPs versus 7033 MFLOPs for FFTW-2.1.5. So, on an average, IMKL FFT is 36% better than FFTW-2.1.5. Despite IMKL FFT demonstrating better average performance than FFTW-2.1.5, its width of variations is considerably greater than that for FFTW-2.1.5. The variations of IMKL FFT fill the picture. This is the reason why IMKL FFT demonstrates comparatively poorer average performance despite its higher peak performance.

- There are 162 problem sizes (out of 1000) where FFTW-2.1.5 is better than IMKL FFT.

Figures A.5, A.6 present the performance comparisons between FFTW-3.3.7 and IMKL FFT. The crucial observations are as follows:

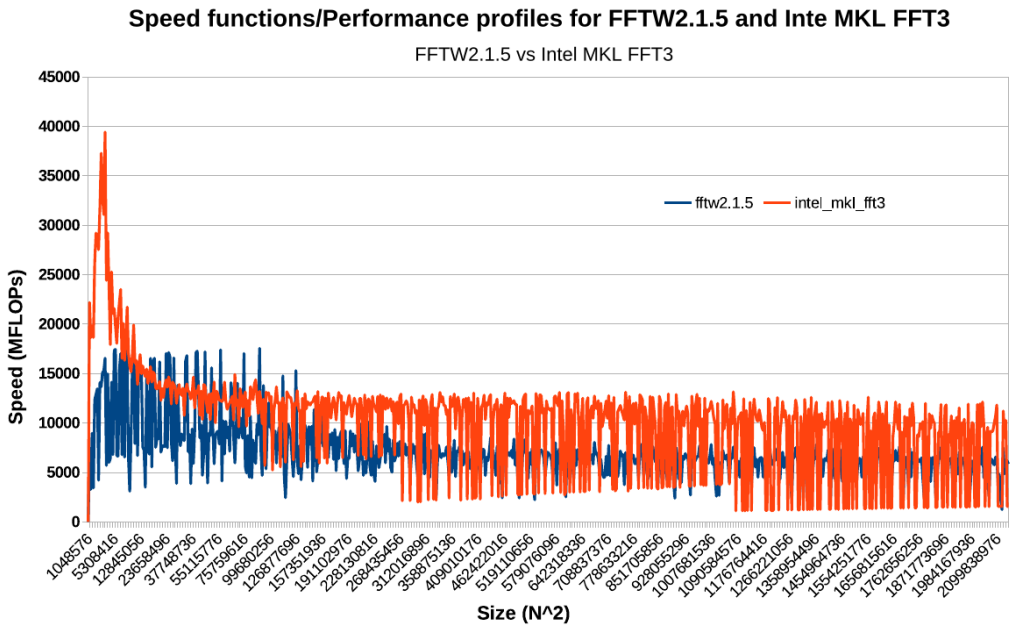


Figure A.3: Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and IMKL FFT. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each.

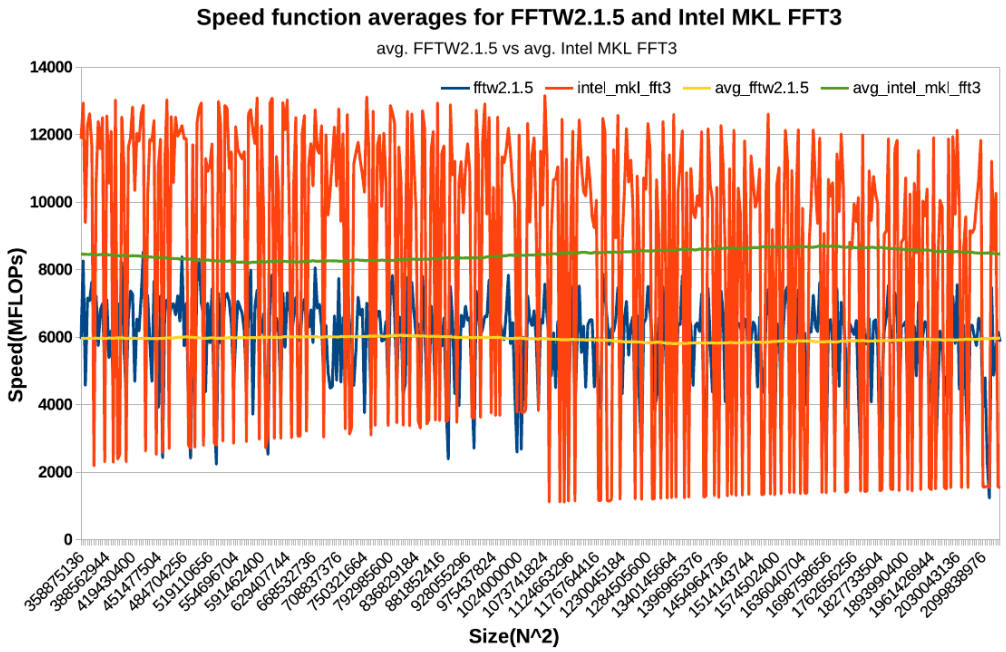


Figure A.4: The average speeds of FFTW-2.1.5 and IMKL FFT.

- The peak performance of FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$) whereas that for IMKL FFT is 39424 MFLOPs ($N = 1792$). The average performance of FFTW-3.3.7 is 5065 MFLOPs and IMKL FFT is 9572 MFLOPs. The ratio of average to peak performances of FFTW-3.3.7 and IMKL FFT are 30% and 24%.

- IMKL FFT, on an average, is 89% faster than FFTW-3.3.7. There are 199 problem sizes (out of 1000) where FFTW-3.3.7 performs better than IMKL FFT.

- The width of variations for IMKL FFT is noticeably greater than that for FFTW-3.3.7.

To summarize, the performance of FFT on multicore CPUs is highly unstable. All these new complexities (NUMA, tight integration of cores contending) cause huge variations in performance profiles. These variations make the average performance of application very low in comparison with its peak. Furthermore, despite IMKL FFT is known as a highly optimized

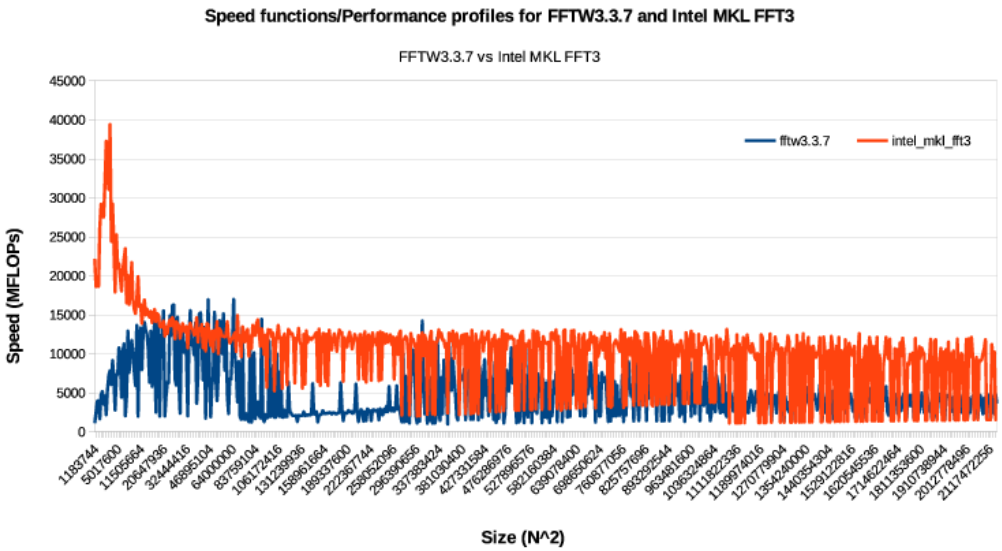


Figure A.5: Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-3.3.7 and IMKL FFT. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each.

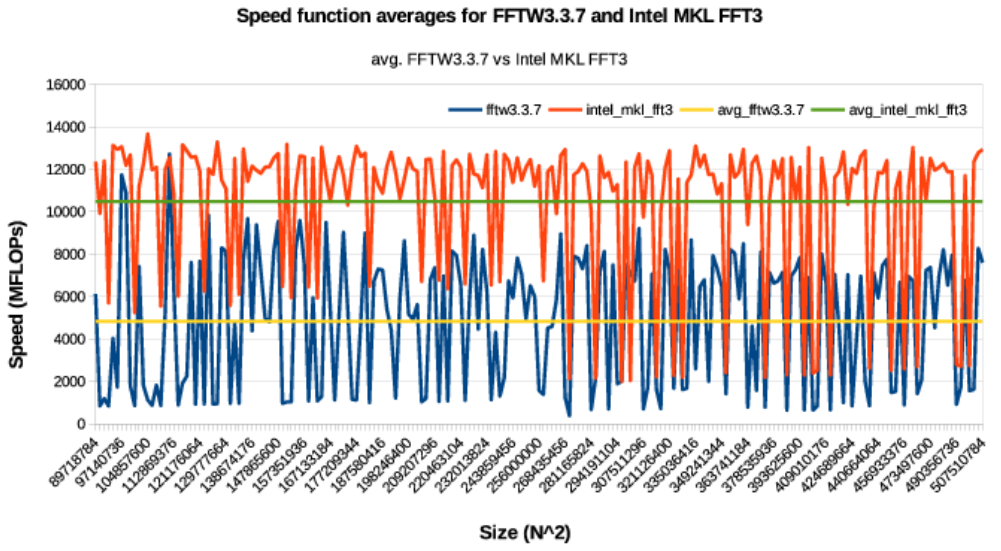


Figure A.6: The average speeds of FFTW-3.3.7 and IMKL FFT.

package for Intel platforms, its width of variations in performance profiles is still huge.

Appendix B

Methodology for Reliable Experimental Results

The methodology described below is used to make sure the experimental results are reliable:

- The server is fully reserved and dedicated to these experiments during their execution. It also made certain that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behaviour of the server.

- An application during its execution is bound to the physical cores using the *numactl* tool.

- To obtain a data point in the speed function, the application is repeatedly executed until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%). For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. The validity of these assumptions is verified using Pearson's chi-squared test. When it is mentioned a single number such as floating-point performance (in MFLOPs or GFLOPs), we imply the sample mean determined using the

Student's t-test.

The function $MeanUsingTtest$, shown in Algorithm 7, determines the sample mean for a data point. For each data point, the function repeatedly executes the application app until one of the following three conditions is satisfied:

1. The maximum number of repetitions ($maxReps$) is exceeded (Line 3).
2. The sample mean falls in the confidence interval (or the precision of measurement eps is achieved) (Lines 13-15).
3. The elapsed time of the repetitions of application execution has exceeded the maximum time allowed ($maxT$ in seconds) (Lines 16-18).

So, for each data point, the function $MeanUsingTtest$ returns the sample mean $mean$. The function $Measure$ measures the execution time using *gettimeofday* function.

- In the experiments, the minimum and maximum number of repetitions are setted, $minReps$ and $maxReps$, to 10 and 100000. The values of $maxT$, cl, and eps are 3600, 0.95, and 0.025. If the precision of measurement is not achieved before the completion of maximum number of repeats, the number of repetitions and also the allowed maximum elapsed time are increased. Therefore, it is sure that statistical confidence is achieved for all the data points that are used in performance profiles.

Algorithm 7 Function determining the mean of an experimental run using Student's t-test.

1: **procedure** MeanUsingTtest($app, minReps, maxReps,$
$maxT, cl, accuracy,$
$repsOut, clOut, etimeOut, epsOut, mean$)

Input:

The application to execute, app
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$

Output:

The number of experimental runs actually made, $repsOut \in \mathbb{Z}_{>0}$
The confidence level achieved, $clOut \in \mathbb{R}_{>0}$
The accuracy achieved, $epsOut \in \mathbb{R}_{>0}$
The elapsed time, $etimeOut \in \mathbb{R}_{>0}$
The mean, $mean \in \mathbb{R}_{>0}$

2: $reps \leftarrow 0$; $stop \leftarrow 0$; $sum \leftarrow 0$; $etime \leftarrow 0$
3: **while** ($reps < maxReps$) and (!$stop$) **do**
4: $st \leftarrow measure(TIME)$
5: $Execute(app)$
6: $et \leftarrow measure(TIME)$
7: $reps \leftarrow reps + 1$
8: $etime \leftarrow etime + et - st$
9: $ObjArray[reps] \leftarrow et - st$
10: $sum \leftarrow sum + ObjArray[reps]$
11: **if** $reps > minReps$ **then**
12: $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv(cl, $reps - 1$))
$\times$ gsl_stats_sd($ObjArray$, 1, $reps$)
/ sqrt($reps$)
13: **if** $clOut \times \frac{reps}{sum} < eps$ **then**
14: $stop \leftarrow 1$
15: **end if**
16: **if** $etime > maxT$ **then**
17: $stop \leftarrow 1$
18: **end if**
19: **end if**
20: **end while**
21: $repsOut \leftarrow reps$; $epsOut \leftarrow clOut \times \frac{reps}{sum}$
22: $etimeOut \leftarrow etime$; $mean \leftarrow \frac{sum}{reps}$
23: **end procedure**

Appendix C

HCLLIMB: Software for Bi-objective Optimization of DGEMM and FFT on Modern Multicore CPUs for Performance and Energy

Data Partitioning Algorithm Using FPMs

The routine *PARTITION* routine checks if the variation of the speeds for each data point is less than or equal to user-input tolerance ϵ (Algorithm 8, Line 3). If a point exists for which the variation exceeds ϵ, then we determine the distribution of the rows using the data partitioning algorithm *HPOPTA* [70] (Line 5). If all the variations are less than or equal to ϵ, we determine the average of the speeds for each data point (Line 7. The averaged speed function is then input to *POPTA* [68] to determine the data partitioning of the rows (Line 9). The data distribution is output in the array, $d = \{d_1, \cdots, d_p\}$.

Algorithm 8 Data partitioning of rows of signal matrix $\mathcal{M}$ of size $N \times N$ using the FPMs.

1: **procedure** Partition($N, p, \mathcal{S}, \epsilon, d$)

Input:

N, Number of rows in the signal matrix, $N \in \mathbb{Z}_{>0}$

Number of abstract processors, $p \in \mathbb{Z}_{>0}$

Functional performance model (speed functions) represented by,

$\mathcal{S} = \{S_1, ..., S_p\}$,

$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$

User tolerance, $\epsilon \in \mathbb{R}_{>0}$

Output:

Optimal partitioning of the rows of the signal matrix, $d = \{d_1, ..., d_p\}, d_i \in \mathbb{Z}_{>0}, \forall i \in [1, p]$

2: **for** $point \leftarrow 1, m$ **do**
3: $rdiff \leftarrow \frac{\max_{i=1}^{p} s_i[point][N] - \min_{i=1}^{p} s_i[point][N]}{\min_{i=1}^{p} s_i[point][N]}$
4: **if** ($rdiff > \epsilon$) **then**
5: **return** $HPOPTA(N, p, S, d)$
6: **end if**
7: $S_{avg}[point] \leftarrow \frac{p}{\sum_{i=1}^{p} \frac{1}{s_i[point][N]}}$
8: **end for**
9: **return** $POPTA(N, p, S_{avg}, d)$
10: **end procedure**

Transpose of square matrix of size $n \times n$ using blocking.

```
void hcl_transpose_scalar_block(
    fftw_complex* X1,
    fftw_complex* X2,
    const int i, const int j,
    const int n,
    const int block_size)
{
    int p, q;

    for (p = 0; p < min(n-i,block_size); p++) {
        for (q = 0; q < min(n-j,block_size); q++) {
            int index1 = i*n+j + p*n+q;
            int index2 = j*n+i + q*n+p;

            if (index1 >= index2)
            continue;

            double tmpr = X1[p*n+q][0];
            double tmpi = X1[p*n+q][1];
            X1[p*n+q][0] = X2[q*n+p][0];
            X1[p*n+q][1] = X2[q*n+p][1];
            X2[q*n+p][0] = tmpr;
            X2[q*n+p][1] = tmpi;
        }
    }
}

void hcl_transpose_block(
    fftw_complex* X,
    const int start, const int end,
    const int n,
    const unsigned int nt,
    const int block_size)
{
    int i, j;

#pragma omp parallel for shared(X) private(i, j) num_threads(nt)
    for (i = 0; i < end; i += block_size) {
        for (j = 0; j < end; j += block_size) {
            hcl_transpose_scalar_block(
                &X[start + i*n + j],
                &X[start + j*n + i],
                i, j, n, block_size);
        }
    }
}
```

Shared Memory Implementation of PFFT_LIMB Employing IMKL FFT

For the implementation using IMKL FFT, two groups of 18 threads each are used, ($p = 2, t = 18$).

The routine PFFT_LIMB_INTEL_MKL shows the implementation of PFFT_LIMB using the FFTW interface. Lines 2-3 sets the number of threads to use during the execution of a 1D-FFT. Lines 4-8 show the execution of row 1D-FFTs by the two abstract processors (groups of 18 threads each) in parallel. Line 9 contains the fast transpose of the signal matrix. Lines 10-14 show the execution of row 1D-FFTs by the two abstract processors (groups of 18 threads each) in parallel. Line 15 contains invocation of the fast transpose.

The transpose routine using blocking can be found above.

Algorithm 9 IMKL implementation of PFFT_LIMB using FFTW interface employing two groups ($p = 2$) of t threads each.

1: **procedure** PFFT_LIMB_INTEL_MKL($id, d, N, \mathcal{M}$)

Input:

$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

Workload distribution, $d = \{d_1, d_2\}, d_1, d_2 \in \mathbb{Z}_{>0}$

Output:

$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $fftw\_init\_threads()$
3: $fftw\_plan\_with\_nthreads(t)$
4: **#pragma omp parallel sections num_threads(2)**
5: **#pragma omp section**
6: $1d\_row\_ffts\_local\_padded(1, d_1, N, \mathcal{M})$
7: **#pragma omp section**
8: $1d\_row\_ffts\_local\_padded(2, d_2, N, \mathcal{M})$
9: $Tranpose(\mathcal{M})$
10: **#pragma omp parallel sections num_threads(2)**
11: **#pragma omp section**
12: $1d\_row\_ffts\_local\_padded(1, d_1, N, \mathcal{M})$
13: **#pragma omp section**
14: $1d\_row\_ffts\_local\_padded(2, d_2, N, \mathcal{M})$
15: $Tranpose(\mathcal{M})$
16: $fftw\_cleanup\_threads()$
17: **return** $\mathcal{M}$
18: **end procedure**

Shared Memory Implementation of PFFT_LIMB Employing FFTW

For the implementation using FFTW-3.3.7, four groups of 9 threads each are used, ($p = 4, t = 9$).

The routine PFFT_LIMB_FFTW shows the implementation of PFFT_LIMB. Lines 2-3 sets the number of threads to use during the execution of a 1D-FFT. Lines 4-12 show the execution of row 1D-FFTs by the four abstract processors (groups of 9 threads each) in parallel. The only thread-safe routine in FFTW is fftw_execute. All the other routines such an plan creation (fftw_plan_many_dft) and plan destruction (fftw_destroy_plan) must be called from one thread at a time. Line 13 contains the fast transpose of the signal matrix. Lines 14-22 show the execution of row 1D-FFTs by the four abstract processors (groups of 9 threads each) in parallel. Line 15 contains invocation of the fast transpose.

The transpose routine using blocking can be found above.

Algorithm 10 FFTW implementation of PFFT_LIMB employing two groups ($p = 4$) of t threads each.

1: **procedure** PFFT_LIMB_FFTW($d, N, \mathcal{M}$)

Input:

$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

Workload distribution, $d = \{d_1, d_2, d_3, d_4\}, d_i \in \mathbb{Z}_{>0}, \forall i \in [1, 4]$

Output:

$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: $fftw\_init\_threads()$
3: $fftw\_plan\_with\_nthreads(t)$
4: **#pragma omp parallel sections num_threads(4)**
5: **#pragma omp section**
6: $1d\_row\_ffts\_local\_padded(1, d_1, N, \mathcal{M})$
7: **#pragma omp section**
8: $1d\_row\_ffts\_local\_padded(2, d_2, N, \mathcal{M})$
9: **#pragma omp section**
10: $1d\_row\_ffts\_local\_padded(3, d_3, N, \mathcal{M})$
11: **#pragma omp section**
12: $1d\_row\_ffts\_local\_padded(4, d_4, N, \mathcal{M})$
13: $Tranpose(\mathcal{M})$
14: **#pragma omp parallel sections num_threads(4)**
15: **#pragma omp section**
16: $1d\_row\_ffts\_local\_padded(1, d_1, N, \mathcal{M})$
17: **#pragma omp section**
18: $1d\_row\_ffts\_local\_padded(2, d_2, N, \mathcal{M})$
19: **#pragma omp section**
20: $1d\_row\_ffts\_local\_padded(3, d_3, N, \mathcal{M})$
21: **#pragma omp section**
22: $1d\_row\_ffts\_local\_padded(4, d_4, N, \mathcal{M})$
23: $Tranpose(\mathcal{M})$
24: $fftw\_cleanup\_threads()$
25: **return** $\mathcal{M}$
26: **end procedure**

Shared Memory Implementation of PMM_LIMB Employing OpenBLAS DGEMM

The implementation of OpenBLAS DGEMM using four groups of 18 threads each $(p = 4, t = 18)$.

The routine PMM_OPEN_BLAS shows the implementation of PMM_LIMB using OpenBLAS DGEMM. On lines 2-6 it calls routine memcpy. We checked that implementation with memcpy is faster than that through sending addresses of matrixes to dgemm routine. On lines 7-15 the routine dgemm is called on each processor. Lines 16-20 call memcpy to return the result. Line 21 returns product matrix C.

Algorithm 11 OpenBLAS implementation of PMM_LIMB employing four groups ($p = 4$) of 18 threads ($t = 18$) each.

1: **procedure** PMM_OPEN_BLAS(id, d, N, A, B, C)

Input:

A, B and C are matrices of size $N \times N, N \in \mathbb{Z}_{>0}$

Workload distribution, $d = \{d_1, d_2, d_3, d_4\}, d_1, d_2, d_3, d_4 \in \mathbb{Z}_{>0}$

Output:

matrix C - the product of two dense square matrices A and B of size $N \times N, N \in \mathbb{Z}_{>0}$

2: **#pragma omp parallel for num_threads(72)**
3: $MEMCPY(d_1, N, A\_1, A, C\_1, C)$
4: $MEMCPY(d_2, N, A\_2, A, C\_2, C)$
5: $MEMCPY(d_3, N, A\_3, A, C\_3, C)$
6: $MEMCPY(d_4, N, A\_4, A, C\_4, C)$
7: **#pragma omp parallel sections num_threads(4)**
8: **#pragma omp section**
9: $DGEMM(1, d_1, N, A\_1, B, C\_1)$
10: **#pragma omp section**
11: $DGEMM(2, d_2, N, A\_2, B, C\_2)$
12: **#pragma omp section**
13: $DGEMM(3, d_3, N, A\_3, B, C\_3)$
14: **#pragma omp section**
15: $DGEMM(4, d_4, N, A\_4, B, C\_4)$
16: **#pragma omp parallel for num_threads(72)**
17: $MEMCPY(d_1, N, A, A\_1, C, C\_1)$
18: $MEMCPY(d_2, N, A, A\_2, C, C\_2)$
19: $MEMCPY(d_3, N, A, A\_3, C, C\_3)$
20: $MEMCPY(d_4, N, A, A\_4, C, C\_4)$
21: **return** C
22: **end procedure**

Shared Memory Implementation of PMM_LIMB Employing IMKL DGEMM

The implementation of IMKL DGEMM using two groups of 18 threads each $(p = 2, t = 18)$.

The routine PMM_INTEL_MKL shows the implementation of PMM_LIMB using OpenBLAS DGEMM. On lines 2-4 it calls routine memcpy. We checked that implementation with memcpy is faster than that through sending addresses of matrixes to dgemm routine. On lines 5-9 the routine dgemm is called on each processor. Lines 10-12 call memcpy to return the result. Line 21 returns product matrix C.

Algorithm 12 Intel MLK implementation of PMM_LIMB employing two groups ($p = 2$) of 18 threads ($t = 18$) each.

1: **procedure** PMM_INTEL_MKL(id, d, N, A, B, C)

Input:

A, B and C are matrices of size $N \times N, N \in \mathbb{Z}_{>0}$

Workload distribution, $d = \{d_1, d_2\}, d_1, d_2 \in \mathbb{Z}_{>0}$

Output:

matrix C - the product of two dense square matrices A and B of size $N \times N, N \in \mathbb{Z}_{>0}$

2: **#pragma omp parallel for num_threads(72)**
3: $MEMCPY(d_1, N, A\_1, A, C\_1, C)$
4: $MEMCPY(d_2, N, A\_2, A, C\_2, C)$
5: **#pragma omp parallel sections num_threads(2)**
6: **#pragma omp section**
7: $DGEMM(1, d_1, N, A\_1, B, C\_1)$
8: **#pragma omp section**
9: $DGEMM(2, d_2, N, A\_2, B, C\_2)$
10: **#pragma omp parallel for num_threads(72)**
11: $MEMCPY(d_1, N, A, A\_1, C, C\_1)$
12: $MEMCPY(d_2, N, A, A\_2, C, C\_2)$
13: **return** C
14: **end procedure**

Implementation of PMMTG-H Based on OpenBLAS DGEMM

Here is described an OpenBLAS implementation of PMMTG-H.

The inputs to an implementation are: a). Matrices A, B, and C of sizes $N \times N$; b). Constants α and β; c) The number of threadgroups, $\{P_1, \cdots, P_p\}$; d). The number of threads in each threadgroup represented by t. The output matrix, C, contains the matrix product.

The vertical partitions of A and C, $\{A_{P_i}, C_{P_i}\}$, $i \in [1, p]$, assigned to the threadgroups, $\{P_1, ..., P_p\}$, are initialized in Lines 24-34. Then p pthreads representing the p threadgroups are created, each a multithreaded OpenBLAS DGEMM kernel executing t OpenMP threads (Lines 36-43).The p threadgroups compute the matrix-matrix product (Lines 1-20). The result is gathered in the matrix C (Lines 45-56).

The implementations using IMKL differ from those using OpenBLAS. In IMKL, the matrix-matrix computation by a threadgroup is performed using an OpenMP parallel region with t threads whereas the same is done in OpenBLAS using a pthread.

```
void *dgemm(void *input){
   int i = *(int*)input;
   openblas_set_num_threads(t);
   goto_set_num_threads(t);
   omp_set_num_threads(t);
   if (i == 1){
       cblas_dgemm(CblasRowMajor, CblasNoTrans,
           CblasNoTrans, N/p, N, N, alpha, A1, N,
           B, N, beta, C1, N);
   }
    ...
   if (i == p){
       cblas_dgemm(CblasRowMajor, CblasNoTrans,
           CblasNoTrans, N/p, N, N, alpha, Ap, N,
           B, N, beta, Cp, N);
   }
}

int main() {
   int row;
#pragma omp parallel for num_threads(p*t)
   for (row = 0; row < N/p; row++) {
      memcpy(&A1[row*N], &A[row*N], N*sizeof(double));
      ...
      memcpy(&Ap[row*N], &A[(p-1)*N*(N/p)+row*N],
             N*sizeof(double));
      memcpy(&C1[row*N], &C[row*N], N*sizeof(double));
      ...
      memcpy(&Cp[row*N], &C[(p-1)*N*(N/p)+row*N],
             N*sizeof(double));
   }

   pthread_t t1, ..., tp;
   int i1 = 1, ..., ip = p;
   pthread_create(&t1, NULL, dgemm, &i1);
   ...
   pthread_create(&tp, NULL, dgemm, &ip);
   pthread_join(tp, NULL);
   ...
   pthread_join(t1, NULL);

#pragma omp parallel for num_threads(p*t)
   for (row = 0; row < N/p; row++){
      memcpy(&A[row*N], &A1[row*N], N*sizeof(double));
      ...
      memcpy(&A[(p-1)*N*(N/p)+row*N], &Ap[row*N],
             N*sizeof(double));
      memcpy(&C[row*N], &C1[row*N], N*sizeof(double));
      ...
      memcpy(&C[(p-1)*N*(N/p)+row*N], &Cp[row*N],
             N*sizeof(double));
   }
}
```

Implementation of PFFTTG-H Based on FFTW

Here is described FFTW implementation of PFFTTG-H.

The inputs to an implementation are: a). Signal matrix M of size $N \times N$; b). The number of threadgroups, p, $\{P_1, \cdots, P_p\}$; c). The number of threads in each threadgroup represented by t. The output is the transformed signal matrix M (considering that we are performing in-place FFT).

Lines 17-18 show the initialization of FFTW multithreaded runtime. Lines 19-25 show the creation of p FFT plans, each plan executed by a threadgroup of t threads. Lines 1-11 illustrate the creation of a plan using fftw_dft_plan_many routine. Lines 26-39 show the execution and destruction of the plans (1D-FFTs on rows) by the threadgroups. This is followed by transpose of the signal matrix (Line 40). Lines 41-46 contain the creation of p FFT plans (1D-FFTs on rows) followed by their execution by the threadgroups. Finally, the signal matrix is transposed again (Line 61). The FFTW runtime is then destroyed (Line 62).

The implementations based on IMKL differ from those employing FFTW. In FFTW, only plan execution (fftw_plan_many_dft) and plan destruction (fftw_destroy_plan) are thread-safe and can be called in an OpenMP parallel region.

```
fftw_plan fftw1d_init_plan(const int sign, const int m,
    const int n, fftw_complex* X, fftw_complex* Y){
        int rank = 1, howmany = m;
        int s[] = {n}, idist = n;
        int odist = n, istride = 1;
        int ostride = 1, *inembed = s, *onembed = s;
        return fftw_plan_many_dft(rank, s, howmany,
               X, inembed, istride, idist, Y, onembed,
               ostride, odist, sign, FFTW_ESTIMATE);
}

int fftw2d(const int sign, const int p, const int N,
    const unsigned int t, const unsigned int blockSize,
    fftw_complex* X){
        fftw_init_threads();
        fftw_plan_with_nthreads(t);
        fftw_plan plan1, plan2, ..., planp;
        plan1 = fftw1d_init_plan(sign, N/p, N, X, X);
        plan2 = fftw1d_init_plan(sign, N/p, N,
               &X[(N/p)*N], &X[(N/p)*N]);
        ...
        planp = fftw1d_init_plan(sign, N-(p-1)*(N/p), N,
               &X[(p-1)*(N/p)*N], &X[(p-1)*(N/p)*N]);

    #pragma omp parallel sections num_threads(p){
        #pragma omp section{
            fftw_execute(plan1);
            fftw_destroy_plan(plan1);
        }
        ...
        #pragma omp section{
            fftw_execute(plan12);
            fftw_destroy_plan(plan12);
        }
    }
        hcl_transpose_block(X, 0, N, N, t, blockSize);
        plan1 = fftw1d_init_plan(sign, N/p, N, X, X);
        plan2 = fftw1d_init_plan(sign, N/p, N,
               &X[(N/p)*N], &X[(N/p)*N]);
        ...
        planp = fftw1d_init_plan(sign, N-(p-1)*(N/p), N,
               &X[(p-1)*(N/p)*N], &X[(p-1)*(N/p)*N]);

    #pragma omp parallel sections num_threads(p){
        #pragma omp section{
            fftw_execute(plan1);
            fftw_destroy_plan(plan1);
        }
        ...
        #pragma omp section{
            fftw_execute(plan12);
            fftw_destroy_plan(plan12);
        }
}
    hcl_transpose_block(X, 0, N, N, nt, blockSize);
    fftw_cleanup_threads();
}
```

Best Form of Partitioning: OpenBLAS and IMKL DGEMM

Figures C.1 and C.2 depict the best form of partitioning between horizontal, vertical and square for OpenBLAS DGEMM and IMKL DGEMM respectively. The configurations of (g,t) for each form of partitioning were chosen using the average speedup of each configuration (g,t) over the base implementation. The difference between forms of partitioning is less than 3%.

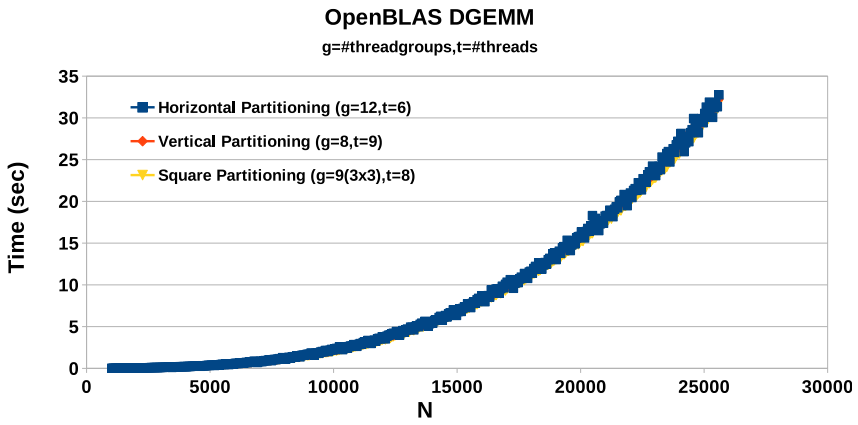


Figure C.1: Best form of partitioning for OpenBALS DGEMM

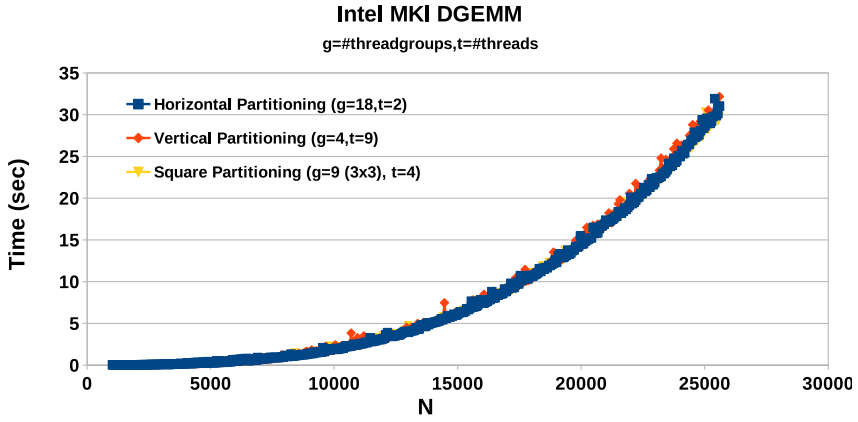


Figure C.2: Best form of partitioning for IMKL DGEMM

Full speed functions using FFTW-3.3.7 and IMKL FFT

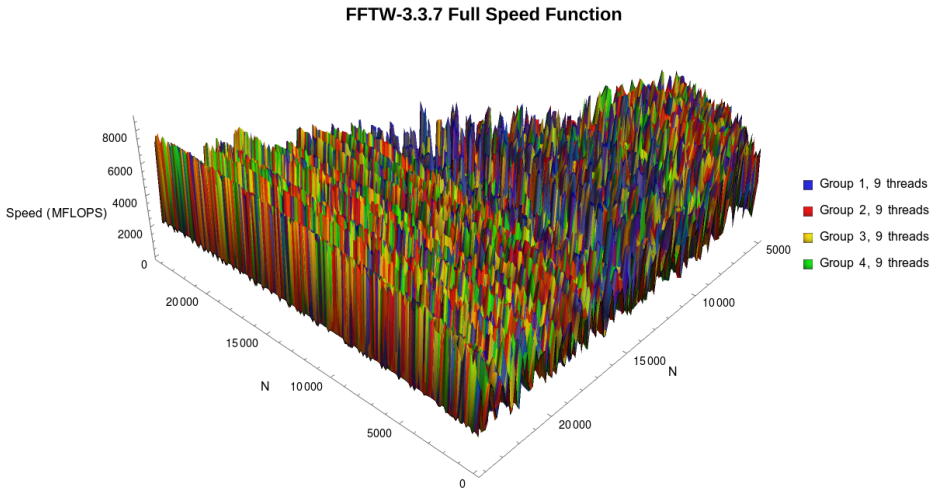


Figure C.3: Full speed function of FFTW-3.3.7.

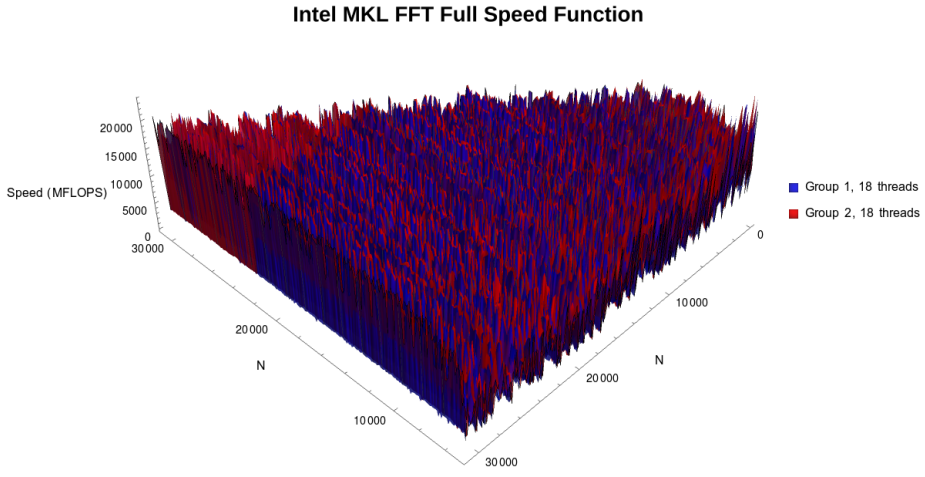


Figure C.4: Full speed function of IMKL FFT.