IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received September 22, 2018, accepted October 22, 2018, date of publication October 30, 2018, date of current version November 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2878271

Performance Optimization of Multithreaded 2D Fast Fourier Transform on Multicore Processors Using Load Imbalancing Parallel Computing Method

SEMYON KHOKHRIAKOV, RAVI REDDY MANUMACHU, AND ALEXEY LASTOVETSKY
School of Computer Science, University College Dublin, Dublin 4, D04 V1W8 Ireland

Corresponding author: Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

This work was supported by the Science Foundation Ireland under Grant 14/IA/2474.

ABSTRACT 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. Its performance on modern multicore platforms is therefore of paramount concern to the high-performance computing community. The inherent complexities in these platforms such as severe resource contention and non-uniform memory access, however, pose formidable challenges. We study the performance profiles of multithreaded 2D FFTs provided in three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel Math Kernel Library (Intel MKL) FFT, on a modern Intel Haswell multicore processor consisting of 36 cores. We show that all the three routines exhibit drastic performance variations, and hence, their average performances are considerably lower than their peak performances. The ratios of average-to-peak performance for the 2D FFT routines from the three packages are 40%, 30%, and 24%. We conclude that improving the average performance of 2D FFT on modern multicore processors by the removal of performance variations constitutes a tremendous research challenge. To address this challenge, we propose two novel optimization methods, PFFT-FPM and PFFT-FPM-PAD, specifically designed and implemented for 2D FFT. The methods employ model-based parallel computing using a load-imbalancing technique. They take as inputs, the discrete 3D functions of the performance of the processors against problem size, compute 2D DFT of a complex signal matrix of size $N \times N$ using p abstract processors, and output the transformed signal matrix. Based on our experiments on a modern Intel Haswell multicore server consisting of 36 physical cores, the average and maximum speedups observed for PFFT-FPM using FFTW-3.3.7 are 1.9× and 6.8×, and the average and maximum speedups observed using Intel MKL FFT are 1.3× and 2×. The average and maximum speedups observed for PFFT-FPM-PAD using FFTW-3.3.7 are 2× and 9.4×, and the average and maximum speedups observed using Intel MKL FFT are 1.4× and 5.9×.

INDEX TERMS Data partitioning, fast Fourier transform, load balancing, multicore, performance optimization.

I. INTRODUCTION

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 [1]–[5]. 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 (Intel MKL) [6] provides extensively optimized FFT routines for Intel processors, cuFFT [7] for Nvidia CUDA GPUs, and clFFT [8] 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

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

compute-bound applications ($O(N^3)$ for matrix-matrix multiplication MxM of two dense $N \times N$ matrices). Its arithmetic 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.

The performance of FFT, therefore, on modern multicore platforms is of paramount concern to the high performance computing community. To address the twin concerns of increasing performance and high energy efficiency, modern multicore platforms manifest tight integration of cores contending for shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intel's Quick Path Interconnect [9], AMD's Hyper Transport [10]), leading to severe resource contention and non-uniform memory access (NUMA). These inherent complexities however pose significant challenges to FFT achieving good performance on these platforms.

To elucidate the challenges, we use three multithreaded FFT applications for comparison written using the packages FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT. The packages, FFTW-2.1.5 and FFTW-3.3.7, are open-source. Hardware vendor libraries [6], [7] offer optimized implementations of the FFTW interface for their processors.

We obtain the performance profiles (speed functions) for the applications executing on a modern Intel Haswell multicore server consisting of 2 sockets of 18 physical cores each (specification shown in Table 1). All the FFT applications compute a 2D-DFT of complex signal matrix of size $N \times N$ using 36 threads. We do not use any special environment affinity variables during the execution of the application. 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, $\{128, 192, \ldots, 64000\}$. The FFTW-3.3.7 package is installed with multithreading, SSE/SSE2, AVX2, and FMA (fused multiply-add) optimizations enabled. For Intel MKL FFT, we do not use any special environment variables.

TABLE 1. Specification of the Intel Haswell server used to construct the performance profiles.

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

The performance profiles are shown for only one planner flag, *FFTW_ESTIMATE*. We have performed experiments with two other planner flags, {*FFTW_MEASURE*, *FFTW_PATIENT*}.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.

We 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:

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

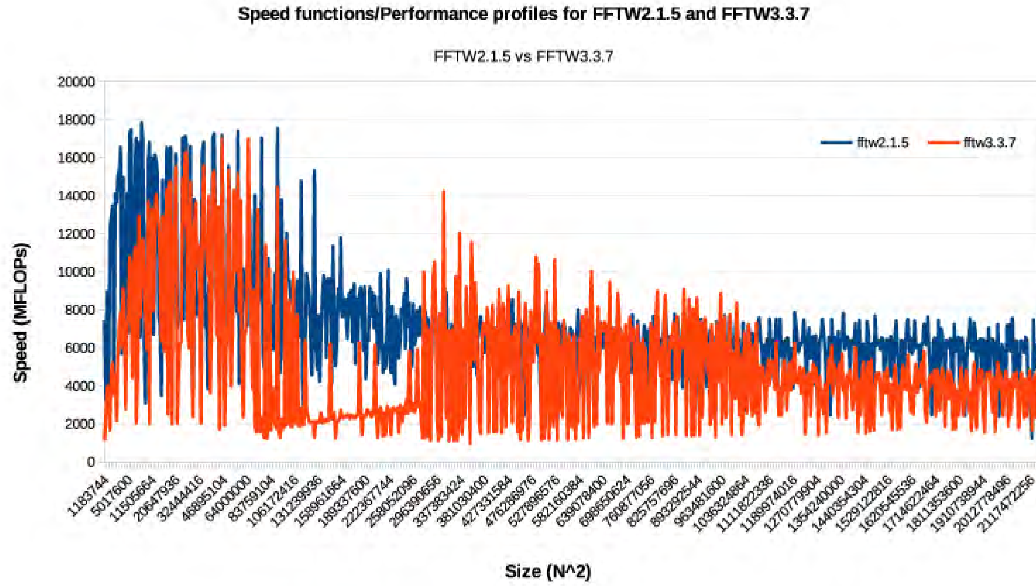
To make sure the experimental results are reliable, we follow a statistical methodology described in Appendix B, supplemental. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision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

Figure 1a, 1b show the performance profiles of FFTW 2.1.5 versus FFTW 3.3.7. Following are the key observations:

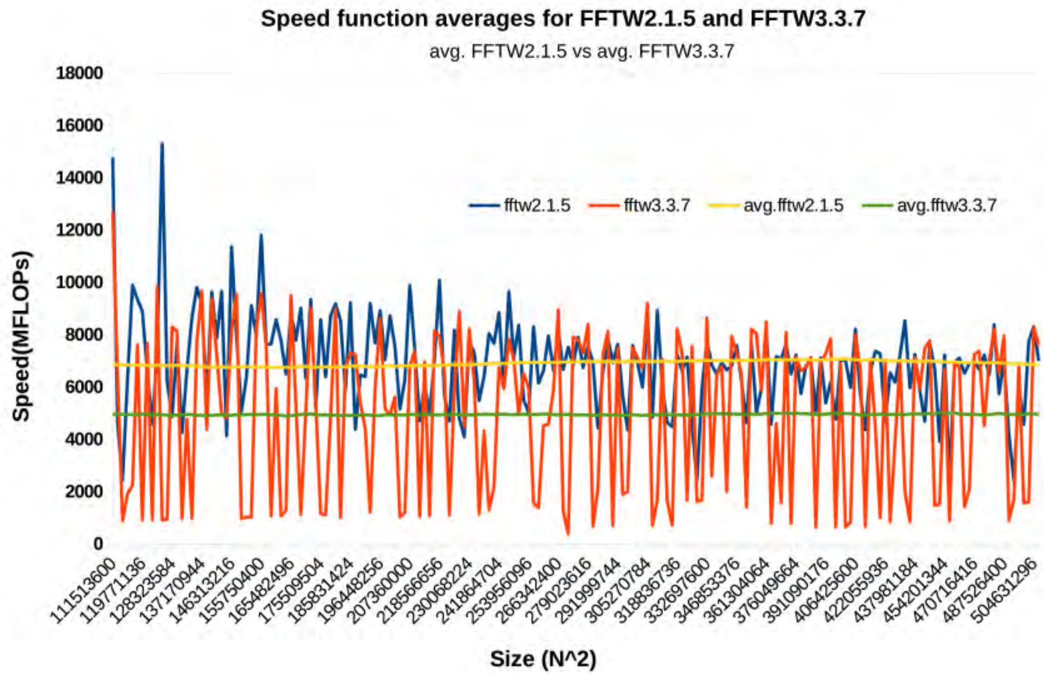
- The width of performance variations in FFTW-3.3.7 is considerably greater than that for FFTW-2.1.5.
- The peak performance of FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$) whereas that for FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$). The average performances of FFTW-2.1.5 and FFTW-3.3.7 are 7033 MFLOPs and 5065 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 2a, 2b present the performance comparisons between FFTW-2.1.5 and Intel MKL FFT. The most important observations are as follows:

- The peak performance of FFTW-2.1.5 is 17841 MFLOPs ($N = 2816$) whereas that for Intel MKL FFT is 39424 MFLOPs ($N = 1792$). The ratio of average to peak performances of FFTW-2.1.5 and Intel MKL FFT are 40% and 24%.



(a)

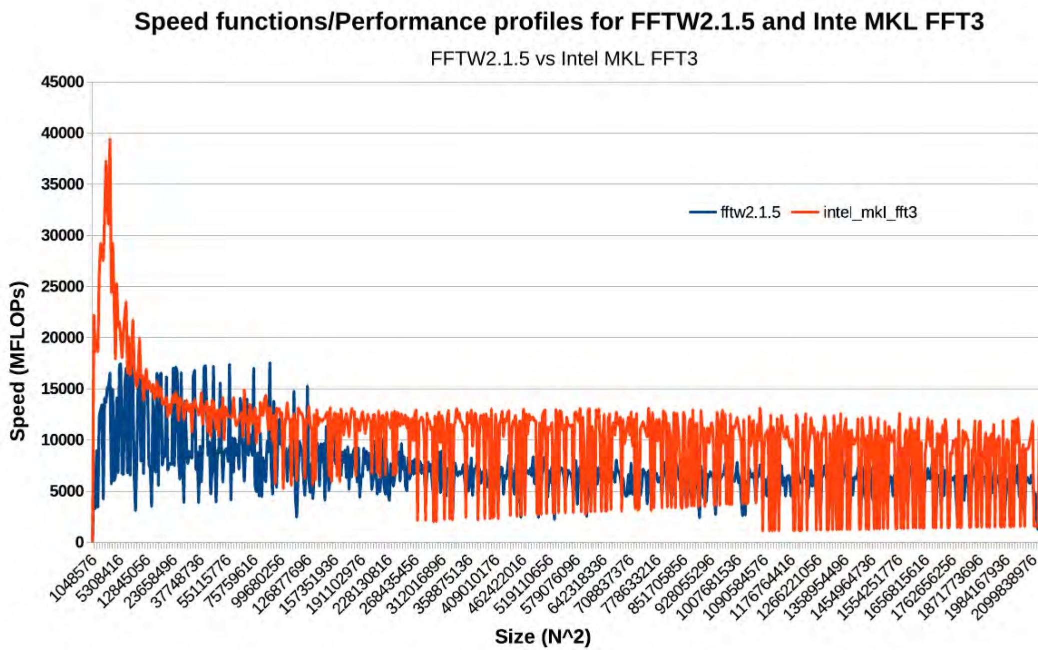


(b)

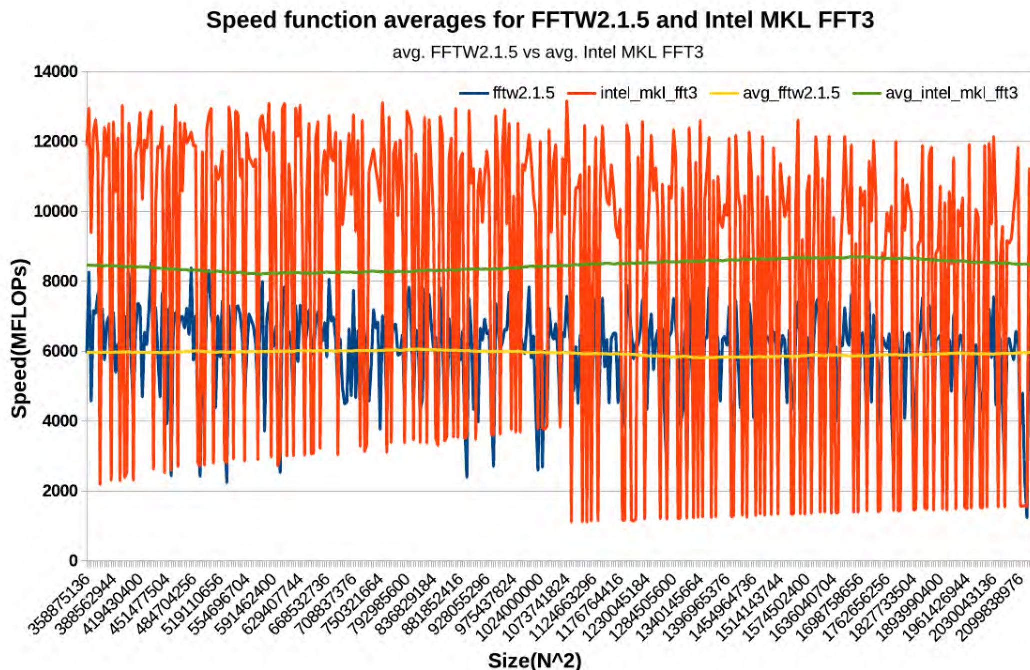
FIGURE 1. **(a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and FFTW-3.3.7. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each. (b). The average speeds of FFTW-2.1.5 vs FFTW-3.3.7.**

- The average performance of Intel MKL FFT is around 9572 MFLOPs versus 7033 MFLOPs for FFTW-2.1.5. So, on an average, Intel MKL FFT is 36% better than FFTW-2.1.5. Despite Intel MKL 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 Intel MKL FFT fill the picture. This is the reason why Intel MKL 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 Intel MKL FFT.

Figures 3a, 3b present the performance comparisons between FFTW-3.3.7 and Intel MKL FFT. The crucial observations are as follows:



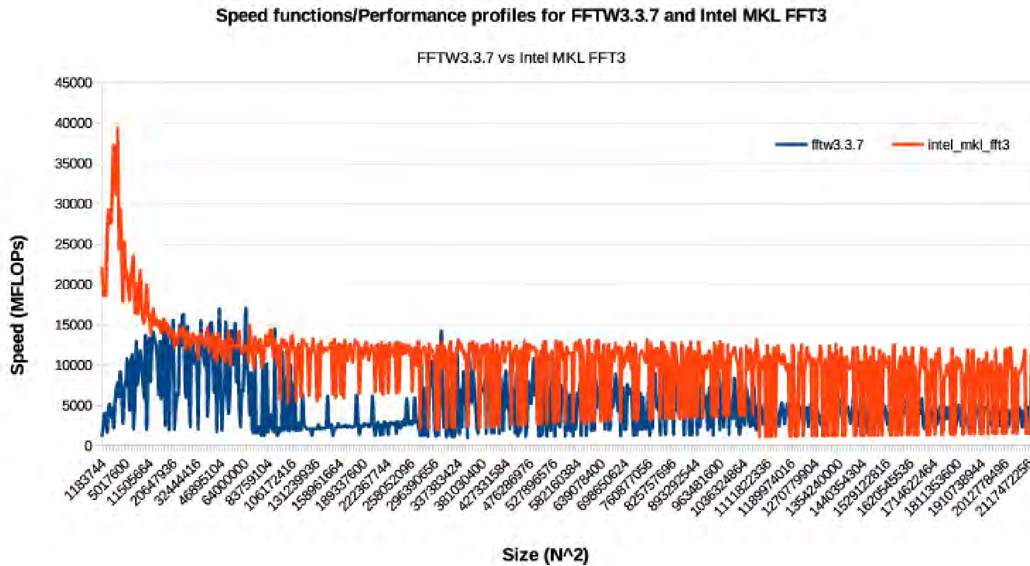
(a)



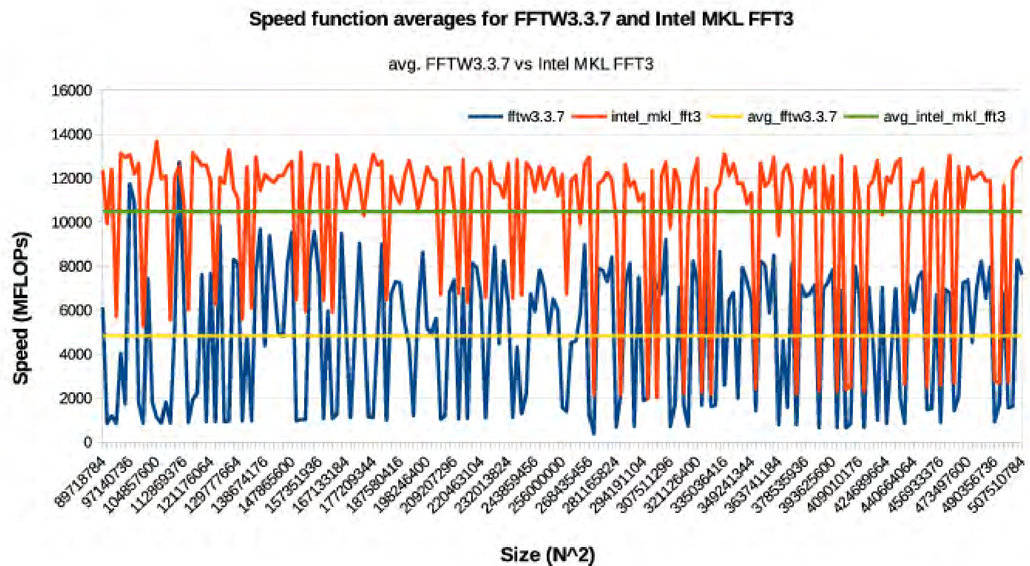
(b)

FIGURE 2. (a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and Intel MKL FFT. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each. (b). The average speeds of FFTW-2.1.5 and Intel MKL FFT.

- The peak performance of FFTW-3.3.7 is 16989 MFLOPs ($N = 8000$) whereas that for Intel MKL FFT is 39424 MFLOPs ($N = 1792$). The average performance of FFTW-3.3.7 is 5065 MFLOPs and Intel MKL FFT is 9572 MFLOPs. The ratio of average to peak performances of FFTW-3.3.7 and Intel MKL FFT are 30% and 24%.
- Intel MKL 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 Intel MKL FFT.



(a)



(b)

FIGURE 3. **(a). Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-3.3.7 and Intel MKL FFT. The executions of 2D-FFT applications employ 36 threads on a Intel multicore server consisting of two sockets of 18 cores each. (b). The average speeds of FFTW-3.3.7 and Intel MKL FFT.**

- The width of variations for Intel MKL FFT is noticeably greater than that for FFTW-3.3.7.

To remove the variations and therefore to improve the average performance of 2D-DFT computation, we regard three solution approaches:

- *Optimization through source code analysis and tuning*: It requires source code modification. It lacks portability if one employs architecture-specific optimizations.
- *Optimization using solutions to larger problem sizes with better performance*: This is a portable approach. A performance model is however required that given workload size N to solve will output the problem size $N_l (> N)$ that is then used for padding. The 2D DFT is computed for N_l. While programmatically extending 1D arrays logically is easy, it is not the case for 2D arrays such as matrices and multidimensional arrays.
- *Optimization using model-based parallel computing*: In the current era of multicores where processors have abundant number of cores, one can partition the workload between identical multithreaded routines (abstract

processors) and execute them in parallel. The starting step in this approach is to partition the workload evenly between the identical multithreaded routines by using load balanced distribution. Then, the workload is unevenly distributed using data partitioning algorithms that take as input realistic and accurate performance models of computation (output from the starting step) and that minimize the total execution time of the parallel execution of the 2D DFT computation. It is portable when the performance models of computation used in the data partitioning algorithms do not use architecture-specific parameters.

We describe these approaches in detail in Appendix C, supplemental.

We propose two novel optimization methods, *PFFT-FPM* and *PFFT-FPM-PAD*, specifically designed and implemented to remove the variations and therefore to improve the average performance of the 2D FFT on modern multicore processors. The methods employ model-based parallel computing using load imbalancing technique. Unlike load balancing methods, the methods provide optimal solutions (workload distributions) that may not load-balance the application in terms of execution time. They can be employed as nodal optimization techniques to construct a 2D FFT routine highly optimized for a dedicated target multicore platform.

The first method *PFFT-FPM* adopts the third solution approach and is a model-based parallel computing solution employing functional performance models (FPMs). The second method *PFFT-FPM-PAD* is an extension of the first. It combines the second and third approaches where the FPMs are used to determine the lengths of the paddings. Both methods take as inputs, discrete 3D functions of performance of the processors against problem size, compute 2D DFT of a complex signal matrix of size $N \times N$ using p abstract processors, and output the transformed signal matrix. We demonstrate tremendous speedups for both these methods over the basic versions offered in FFTW-3.3.7 and Intel MKL FFT.

Our main contributions are the following:

- We demonstrate the challenges posed by inherent complexities in modern multicore platforms such as severe resource contention and NUMA to 2D FFT achieving good performance on these platforms. To highlight the challenges, we study the performance profiles of multithreaded 2D FFT provided in three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT on a modern Intel Haswell multicore processor consisting of thirty-six cores. We show that all the three routines demonstrate drastic performance variations and that their average performances therefore are considerably lower than their peak performances.
- We propose two portable optimization methods specifically designed and implemented to remove the variations and to improve the average performance of 2D FFT on modern multicore processors. We report tremendous speedups of these methods over the basic FFT routines provided in the packages FFTW-3.3.7 and Intel MKL FFT. We show that using these methods improves the average performance of FFTW-3.3.7 over the unoptimized FFTW-2.1.5 by 42% and the average performance of Intel MKL FFT over the unoptimized FFTW-2.1.5 by 24% (over and above the 36% of unoptimized Intel MKL FFT).

We organize the rest of the paper as follows. Section 3 presents our two optimization methods. Section 4 contains the experimental results. Section 5 concludes the paper.

II. 2D-DFT: MODEL-BASED PARALLEL COMPUTING SOLUTIONS

In this section, we start with description of the sequential 2D-FFT algorithm using the row-column decomposition method. Next, we explain the parallel 2D-FFT algorithm based on the sequential 2D-FFT algorithm and that uses load balancing technique. Then, we present our two optimization methods, *PFFT-FPM* and *PFFT-FPM-PAD* that employ load imbalancing technique.

A. SEQUENTIAL 2D-FFT ALGORITHM

We describe here the sequential algorithm for computing the DFT on a two-dimensional point discrete signal $\mathcal{M}$ of size $N \times N$. We call $\mathcal{M}$ 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 4 depicts the method, which is mathematically summarized below:

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

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

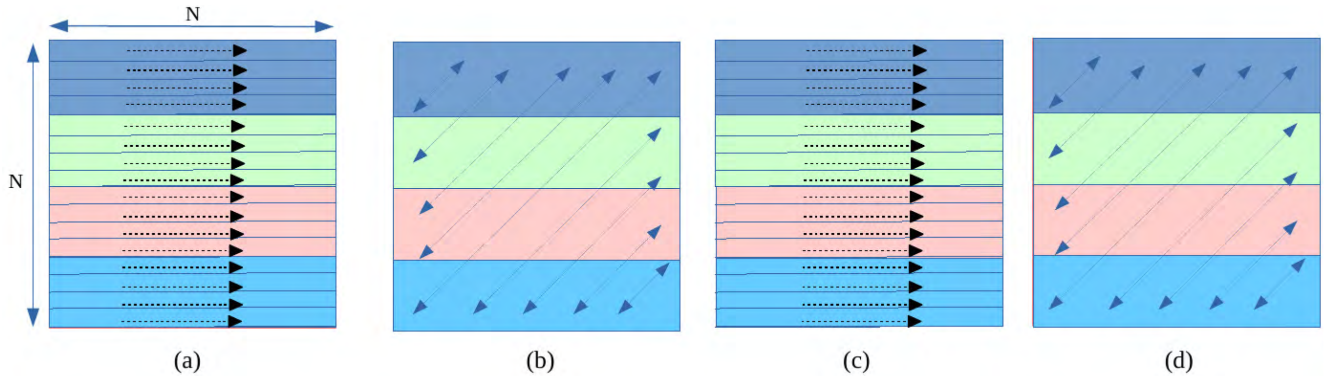


FIGURE 4. ***PFFT-LB*** **performing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$ ($N = 16$) using four identical processors. Each processor gets four rows each. (a). Each processor performs series of row 1D-FFTs locally indicated by dotted arrows. (b). Matrix $\mathcal{M}$ is transposed. (a). Each processor performs series of row 1D-FFTs locally indicated by dotted arrows. (d). Matrix $\mathcal{M}$ is transposed again. It is the output of *PFFT-LB*.**

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 $\Theta(N^2 \log_2 N)$.

B. 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 identical abstract processors, $\{P_1, \ldots, P_p\}$. To aid clear exposition, we assume N is divisible by p. The rows of the complex matrix x are partitioned equally between the p processors where each processor 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 that we discuss in this work are considered in-place.

PFFT-LB consists of four steps:

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

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

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

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 4.

C. PFFT-FPM: *PERFORMANCE OPTIMIZATION USING FPMS AND LOAD IMBALANCING*

We describe here our first novel optimization method called *PFFT-FPM* that takes 3D functional performance models (FPMs) as input and that employs load imbalancing parallel computing technique.

PFFT-FPM is executed using p identical abstract processors, $\{P_1, \ldots, P_p\}$. The inputs to *PFFT-FPM* are the number of available abstract processors, p, the number of rows of the signal matrix, N, the speed functions of the abstract processors, $\mathcal{S}$, and the user-input tolerance ϵ. The output from *PFFT-FPM* is the transformed signal matrix $\mathcal{M}$.

The discrete speed function of processor P_i is given by $\mathcal{S}_i = \{s_i(x_1, y_1), \ldots, s_i(x_m, y_m)\}\}$ where $s_i(x, y)$ represents the speed of execution of x number of 1D-FFTs of length y by the processor 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.

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?: $\exists (x_k, N), 1 \le k \le m, \ (\frac{\max_{i=1}^{p} s_i(x_k,N) - \min_{i=1}^{p} s_i(x_k,N)}{\min_{i=1}^{p} s_i(x_k,N)} > \epsilon)$, 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* [11] 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* [12] is invoked using the p speed curves as input to obtain an optimal distribution of the rows, d.

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

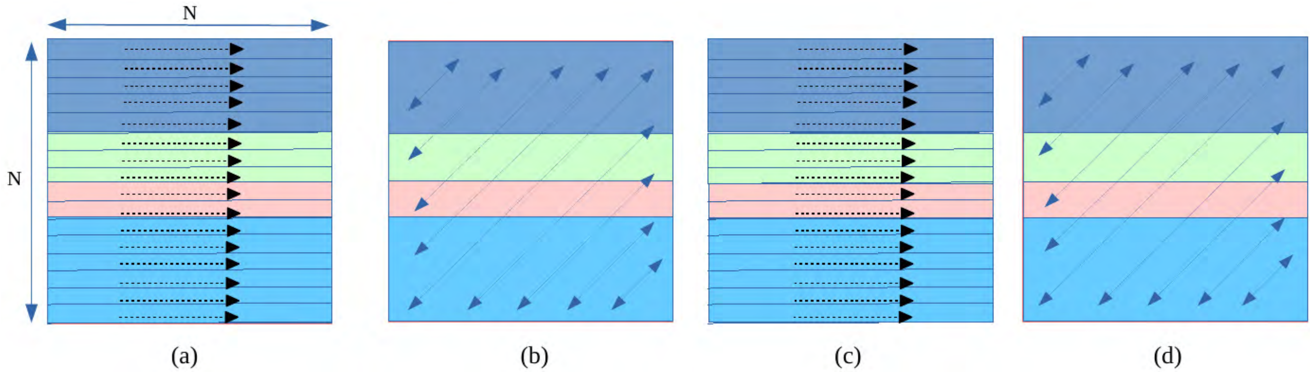


FIGURE 5. ***PFFT-FPM*** **performing 2D-DFT of signal matrix $\mathcal{M}$ of size $N \times N$ ($N = 16$) using four abstract processors. Each processor gets different number of rows given by the data distribution, $d = \{5, 3, 2, 6\}$. (a). Each processor performs series of row 1D-FFTs locally indicated by dotted arrows. (b). Matrix $\mathcal{M}$ is transposed. (a). Each processor performs series of row 1D-FFTs locally indicated by dotted arrows. (d). Matrix $\mathcal{M}$ is transposed again. It is the output of *PFFT-FPM*.**

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 5 for four abstract processors solving 2D-DFT of size $N \times N (N = 16)$.

The data partitioning algorithms *POPTA* and *HPOPTA* are described in detail in [11] and [12]. 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 p 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 6a, 6b illustrate the data partitioning algorithm employed in *PFFT-FPM* for two abstract processors solving 2D-DFT of size $N \times N$ where $N = 24704$ using Intel MKL FFT on a Intel multicore server. The speed functions shown are segments of the full functions (given in Appendix E, supplemental). Each abstract processor consists of 18 threads. Figure 6a 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 $y = N = 24704$. One can see that the two curves are not identical (heterogeneous). That is, there are points where the speeds differ from each other by more than 5% ($\epsilon = 0.05$). We input the speed functions to *HPOPTA*, which determines the optimal partitioning of rows, $(d[1], d[2]) = (11648, 13056)$, where each row is of length $N = 24704$.

D. PFFT-FPM-PAD*: PERFORMANCE OPTIMIZATION USING PADDING DETERMINED FROM FPMs*

In this section, we present *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 performed locally on rows whose length is extended (padded with zeroes) by an extent determined from the FPM of the processor. The determination of the length of padding is a local computation and is specific to an abstract processor. That is, the lengths can be different for different processors. 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: Processor 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 $\{(d[i], y_{N+1}), \ldots, (d[i], y_m)\}$ that has better execution time than the point $(d[i], N)$. 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. Processor 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 7a, 7b illustrate how they are determined from the FPMs for two abstract processors solving 2D-DFT

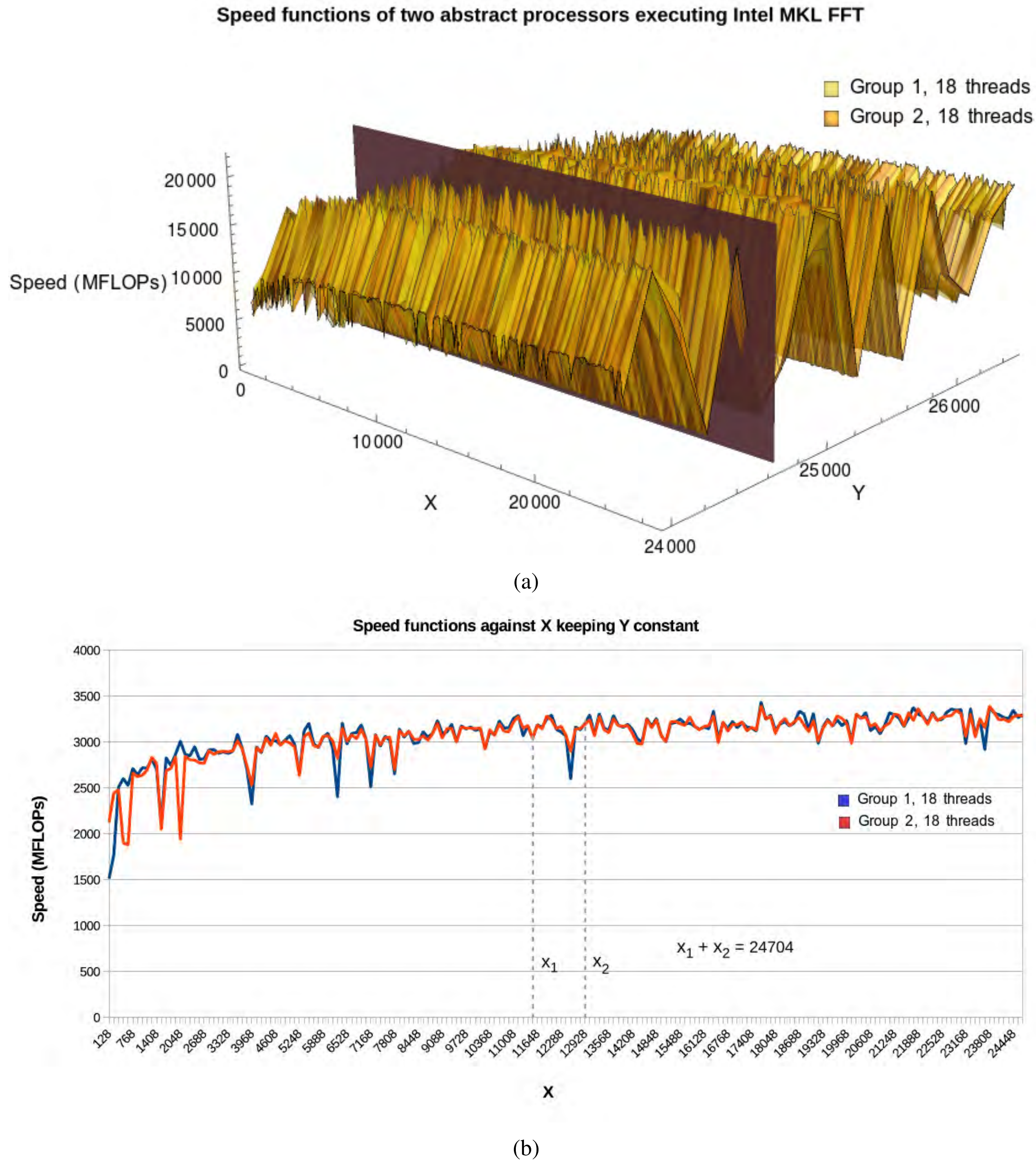


FIGURE 6. (a). Speed functions of two abstract processors, each a group of 18 threads. Each group executes 2D-DFT of size $x \times y$ using Intel MKL FFT on a Intel multicore server consisting of two sockets of 18 cores each. The plane $y = N = 24704$ intersects the speed functions. (b). Each intersection produces two curves for the two groups showing speed versus x keeping $y = N = 24704$. Application of *HPOPTA* to determine optimal distribution of rows provides the partitioning, ($d[1] = x_1 = 11648$, $d[2] = x_2 = 13056$).

of size $N \times N$ where $N = 24704$ using Intel MKL FFT on a Intel multicore server. The speed functions shown are segments of the full functions (given in Appendix E, supplemental). Each abstract processor consists of 18 threads. Figure 7a shows two planes $x_1 = 11648$ and $x_2 = 13056$ 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. The padded lengths $(N_{padded,1}, N_{padded,2})$ corresponding to x_1 and x_2 are determined from the curves and are equal to 24960.

III. SHARED MEMORY IMPLEMENTATIONS OF *PFFT-FPM* AND *PFFT-FPM-PAD*

In this section, we describe two shared memory implementations of *PFFT-FPM*, one using *Intel MKL FFT* and the other using *FFTW-3.3.7*.

The inputs to the implementation are the signal matrix $\mathcal{M}$ of size $N \times N$, the number of abstract processors (groups) p, the speed functions represented by a set $\mathcal{S}$ containing problem sizes and speeds, and number of threads in each abstract processor (group) represented by t. The output is

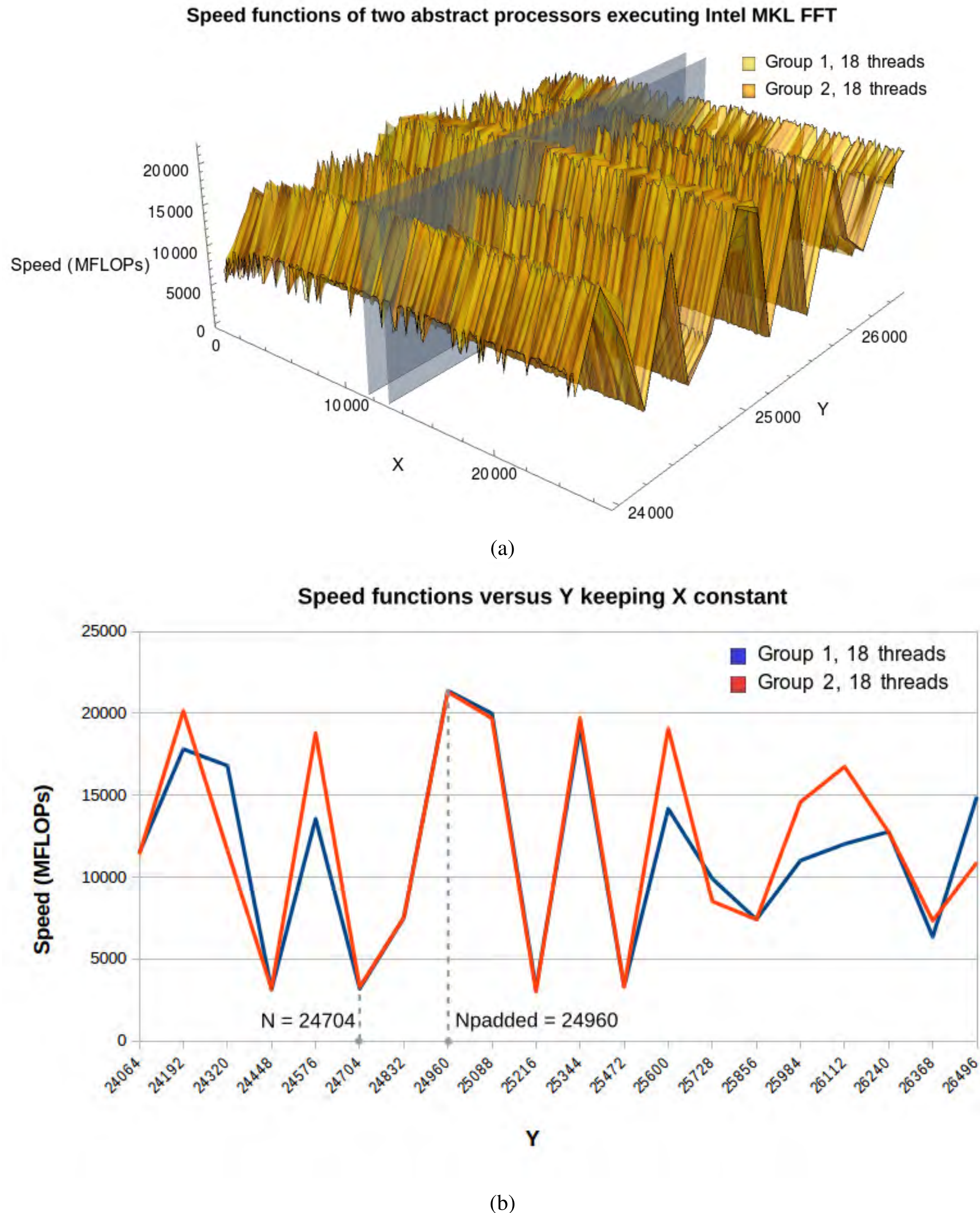


FIGURE 7. (a). Speed function for group1 intersected by the plane $x_1 = 11648$. Speed function for group2 intersected by the plane $x_2 = 13056$. (b). Each intersection produces a curve for the group showing speed versus y keeping x constant. The lengths of padding for the two groups, *Npadded*, is the same and is equal to 24960.

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 1) is to determine the partitioning of rows by invoking the routine *PARTITION*. The partitioning routine checks if the variation of the speeds for each data point is less than or equal to user-input tolerance ϵ (Algorithm 2, 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* [12] (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* [11] to determine the data partitioning of the rows (Line 9). The data distribution is output in the array, $d = \{d_1, \cdots, d_p\}$.

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

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 abstract processors, $p \in \mathbb{Z}_{>0}$
Functional performance model (speed functions) represented by,
$\mathcal{S} = \{S_1, \ldots, S_p\}$,
$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$
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**

Algorithm 2 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,
$\mathcal{S} = \{S_1, \ldots, S_p\}$,
$\mathcal{S}_i = \{(x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0}\}$
User tolerance, $\epsilon \in \mathbb{R}_{>0}$
Output:
Optimal partitioning of the rows of the signal matrix, $d = \{d_1, \ldots, d_p\}, d_i \in \mathbb{Z}_{>0}, \forall i \in [1, p]$

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**

Algorithm 3 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 abstract processors, $p \in \mathbb{Z}_{>0}$
Output:
$\mathcal{M}$, Signal matrix of size $N \times N, N \in \mathbb{Z}_{>0}$

2: **for** *proc* $\leftarrow 1, p$ **do**
3: 1D_ROW_FFTS_LOCAL($proc, d_{proc}, N, \mathcal{M}$)
4: **end for**
5: Parallel_Tranpose($\mathcal{M}$)
6: **for** *proc* $\leftarrow 1, p$ **do**
7: 1D_ROW_FFTS_LOCAL($proc, d_{proc}, N, \mathcal{M}$)
8: **end for**
9: Parallel_Tranpose($\mathcal{M}$)
10: **return** $\mathcal{M}$
11: **end procedure**

Algorithm 4 Intel MKL 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: *#pragmaompparallelsectionsnum_threads*(2)
5: *#pragmaompsection*
6: 1d_row_ffts_local($1, d_1, N, \mathcal{M}$)
7: *#pragmaompsection*
8: 1d_row_ffts_local($2, d_2, N, \mathcal{M}$)
9: Tranpose($\mathcal{M}$)
10: *#pragmaompparallelsectionsnum_threads*(2)
11: *#pragmaompsection*
12: 1d_row_ffts_local($1, d_1, N, \mathcal{M}$)
13: *#pragmaompsection*
14: 1d_row_ffts_local($2, d_2, N, \mathcal{M}$)
15: Tranpose($\mathcal{M}$)
16: fftw_cleanup_threads()
17: **return** $\mathcal{M}$
18: **end procedure**

(Line 5), series of row 1D-FFTs (Lines 6-8), and parallel transpose (Line 9).

Each processor performs the series of row 1D-FFTs locally using the routine *1D_ROW_FFTS_LOCAL*. The number of row 1D-FFTs performed by processor P_i is given by first argument, d_i. Algorithm 6 illustrates the implementation of this routine using FFTW interface.

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

Algorithm 5 FFTW Implementation of PFFT_LIMB Employing Two Groups ($p = 4$) of t Threads Each

```
1:  procedure PFFT_LIMB_FFTW(d, N, M)
 Input:
    M, Signal matrix of size N × N, N ∈ Z>0
    Workload distribution, d = {d1, d2, d3, d4}, di ∈ Z>0, ∀i ∈ [1, 4]
 Output:
    M, Signal matrix of size N × N, N ∈ Z>0

2:      fftw_init_threads()
3:      fftw_plan_with_nthreads(t)
4:      #pragmaompparallelsectionsnum_threads(4)
5:          #pragmaompsection
6:              1d_row_ffts_local(1, d1, N, M)
7:          #pragmaompsection
8:              1d_row_ffts_local(2, d2, N, M)
9:          #pragmaompsection
10:             1d_row_ffts_local(3, d3, N, M)
11:         #pragmaompsection
12:             1d_row_ffts_local(4, d4, N, M)
13:     Tranpose(M)
14:     #pragmaompparallelsectionsnum_threads(4)
15:         #pragmaompsection
16:             1d_row_ffts_local(1, d1, N, M)
17:         #pragmaompsection
18:             1d_row_ffts_local(2, d2, N, M)
19:         #pragmaompsection
20:             1d_row_ffts_local(3, d3, N, M)
21:         #pragmaompsection
22:             1d_row_ffts_local(4, d4, N, M)
23:     Tranpose(M)
24:     fftw_cleanup_threads()
25:     return M
26: end procedure
```

Determine_Pad_Length before executing the series of row 1D-FFTs.

A. SHARED MEMORY IMPLEMENTATIONS OF PFFT-FPM

We now describe the shared-memory implementations of the routine *PFFT_LIMB* for Intel MKL FFT and FFTW-3.3.7 on a Intel Haswell server containing 36 physical cores (Table 1).

The input parameters (p, t) used during the execution of *PFFT-FPM* and *PFFT-FPM-PAD* are obtained from the best load-balanced configuration observed experimentally.

1) INTEL MKL FFT

For the implementation using *Intel MKL FFT*, we use two groups of 18 threads each, $(p = 2, t = 18)$. From our experiments, this pair is the best amongst the following combinations: $\{(4, 9), (6, 6), (9, 4), (12, 3)\}$, experimentally.

The routine *PFFT_LIMB_INTEL_MKL* shows the implementation of *PFFT_LIMB* using the FFTW interface.

Algorithm 6 Series of x Row 1D-FFTs Using FFTW Interface Function fftw_Plan_Many_dft

```
1:  procedure 1D_ROW_FFTS_LOCAL(id, x, N, M, flag)
 Input:
    Processor identifier, id ∈ Z>0
    Problem size x ∈ Z>0
    M, Signal matrix of size N × N, N ∈ Z>0
 Output:
    M, Signal matrix of size N × N, N ∈ Z>0

2:      rank ← 1; howmany ← x; s ← N;
3:      idist ← N; odist ← N; istride ← 1;
4:      ostride ← 1; inembed ← s; onembed ← s;
5:      plan ← fftw_plan_many_dft(rank, s, howmany,
            M, inembed, istride, idist,
            M, onembed, ostride, odist,
            FFTW_FORWARD, FFTW_ESTIMATE)
6:      fftw_execute(plan)
7:      fftw_destroy_plan(plan)
8:      return M
9:  end procedure
```

Algorithm 7 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, M)
 Input:
    Processor identifier, id ∈ Z>0
    Problem size x ∈ Z>0
    M, Signal matrix of size N × N, N ∈ Z>0
    Functional performance model (speed functions) represented by,
    S = {S1, . . . , Sp},
    Si = {(xi[q][r], si[q][r]) | i ∈ [1, p], q, r ∈ [1, m], xi[q][r] ∈ Z>0, si[q][r] ∈ R>0}
 Output:
    M, Signal matrix of size N × N, N ∈ Z>0

2:      Npadded ← Determine_Pad_Length(id, x, N, S)
3:      rank ← 1; howmany ← x; s ← Npadded;
4:      idist ← Npadded; odist ← Npadded; istride ← 1;
5:      ostride ← 1; inembed ← s; onembed ← s;
6:      plan ← fftw_plan_many_dft(rank, s, howmany,
            M, inembed, istride, idist,
            M, onembed, ostride, odist,
            FFTW_FORWARD, FFTW_ESTIMATE)
7:      fftw_execute(plan)
8:      fftw_destroy_plan(plan)
9:      return M
10: end procedure
```

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.

We present the transpose routine using blocking in the supplemental (Appendix D).

2) FFTW

For the implementation using *FFTW-3.3.7*, we use four groups of 9 threads each, $(p = 4, t = 9)$. From our experiments, this pair is the best amongst the following combinations: $\{(2, 18), (6, 6), (9, 4), (12, 3)\}$, experimentally.

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.

We present the transpose routine using blocking in the supplemental (Appendix D).

IV. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present our experimental results that demonstrate the performance improvements provided by *PFFT-FPM* and *PFFT-FPM-PAD*. Our experimental platform is a Intel Haswell server containing 36 physical cores (Table 1).

We use two packages, FFTW-3.3.7 and Intel MKL FFT, for the implementations of the methods. We could not optimize FFTW-2.1.5 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 Intel MKL FFT. We compare the speedups of optimized FFTW-3.3.7 and Intel MKL FFT with the unoptimized FFTW-2.1.5.

The FFTW-3.3.7 package is installed with multithreading, SSE/SSE2, AVX2, and FMA (fused multiply-add) optimizations enabled. For Intel MKL FFT, we do not use any special environment variables. We experiment 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* 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 our future work, we will present the speedups obtained by *PFFT-FPM* and *PFFT-FPM-PAD* using these planner flags in a technical report.

The input parameters (p, t), where p is the number of processes and t is the number of threads, used during the execution of *PFFT-FPM* and *PFFT-FPM-PAD* are obtained from the best load-balanced configuration observed experimentally. For the implementations using *FFTW-3.3.7*, we use four groups of 9 threads each, $(p = 4, t = 9)$ since this pair performs the best among the following combinations: $\{(2, 18), (6, 6), (9, 4), (12, 3)\}$. For the implementations using *Intel MKL FFT*, we use two groups of 18 threads each, $(p = 2, t = 18)$ since this is the best combination found experimentally among the following combinations: $\{(4, 9), (6, 6), (9, 4), (12, 3)\}$.

The full speed functions constructed for Intel MKL FFT and FFTW-3.3.7 are shown in the Appendix E, supplemental. To make sure the experimental results are reliable, we automated the construction of speed functions. Appendix B describes in detail the automation procedure. The inputs to the procedure are the FFT application and the application parameters $(p, t, \mathcal{M})$, and the set of problem sizes. The output is the set of discrete speed functions, $\mathcal{S} = \{S_1, \ldots S_p\}$, one for each abstract processor. The set of problem sizes (x, y) used for the construction of speed functions are $\{(x, y) \mid 128 \leq x \leq y, 128 \leq y \leq 64000, x \bmod 128, y \bmod 128\} = \{128 \times 128, 128 \times 256, 256 \times 256, \cdots, 64000 \times 64000\}$. All the abstract processors 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.

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, we use Student's t-test assuming that the individual observations are independent and their population follows the Normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test.

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 [13], [14]. These are input to the data partitioning algorithm [11], 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 neighborhood of homogeneous distribution, $d_i = \frac{n}{p}, \forall i \in [1, p]$, are constructed until the allowed user-input execution time is exceeded. We aim to research further into methods to reduce the construction times of speed functions in our future work.

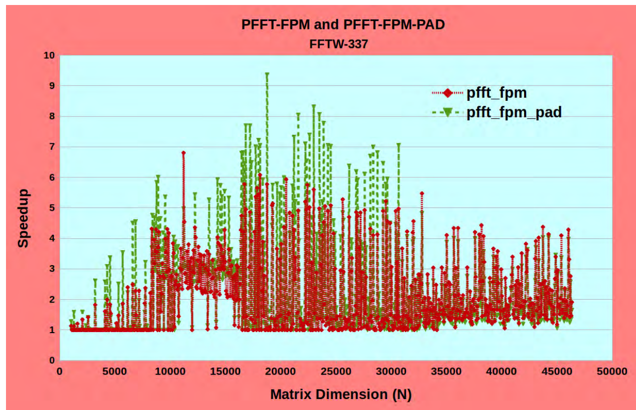


FIGURE 8. Speedup of *PFFT-FPM* and *PFFT-FPM-PAD* against the basic FFTW-3.3.7 executed using 36 threads.

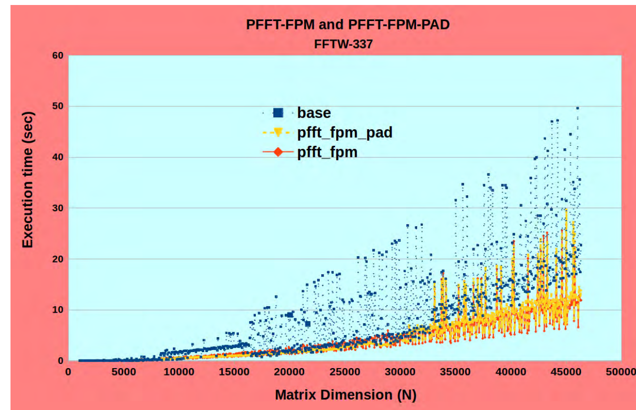


FIGURE 10. Execution times of *PFFT-FPM* and *PFFT-FPM-PAD* against the basic FFTW-3.3.7 executed using 36 threads.

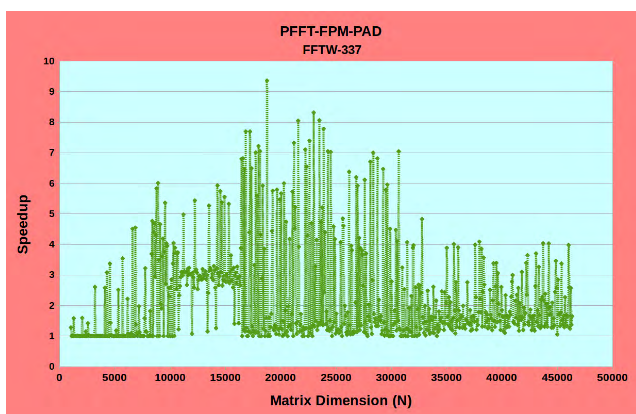


FIGURE 9. Speedup of *PFFT-FPM-PAD* against the basic FFTW-3.3.7 executed using 36 threads.

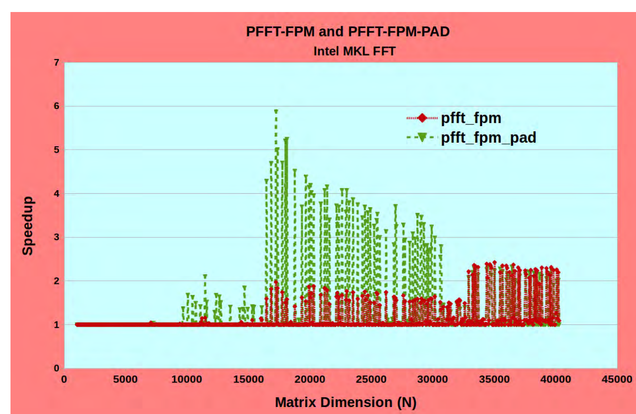


FIGURE 11. Speedups of *PFFT-FPM* and *PFFT-FPM-PAD* against the basic Intel MKL FFT executed using 36 threads.

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

A. PFFT-FPM *AND* PFFT-FPM-PAD *USING FFTW-3.3.7*

Figure 8 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 $N \times N$. Figure 9 shows the speedup of *PFFT-FPM-PAD*. The average and maximum performance improvements are 2x and 9.4x.

Figure 10 shows the execution times of *PFFT-FPM* and *PFFT-FPM-PAD* versus basic FFTW-3.3.7. In the supplemental (Appendix E), we show in separate plots the execution times of *PFFT-FPM* and *PFFT-FPM-PAD* versus basic FFTW-3.3.7. For problem sizes in the range ($N > 33000$), while the speedups are still good (6x for FFTW-3.3.7), major variations still remain.

B. PFFT-FPM *AND* PFFT-FPM-PAD *USING INTEL MKL FFT*

Figure 11 compares the speedups *PFFT-FPM* and *PFFT-FPM-PAD* over basic Intel MKL FFT which computes the 2D-DFT using one group consisting of 36 threads. Figure 12 shows the speedups of *PFFT-FPM-PAD*. The average and maximum speedups are 1.4x and 5.9x.

Figure 13 shows the execution times of *PFFT-FPM* and *PFFT-FPM-PAD* versus basic Intel MKL FFT. In the supplemental (Appendix E), we show in separate plots the execution times of *PFFT-FPM* and *PFFT-FPM-PAD* versus basic Intel MKL FFT . For problem sizes in the range ($N > 33000$), while the speedups are still good (2x for Intel MKL FFT), the variations are still significant.

C. *OPTIMIZED FFTW-3.3.7 AND INTEL MKL FFT VERSUS UNOPTIMIZED FFTW-2.1.5*

Finally, we compare how the optimized FFTW-3.3.7 and Intel MKL FFT using *PFFT-FPM-PAD* fares with respect to unoptimized FFTW-2.1.5.

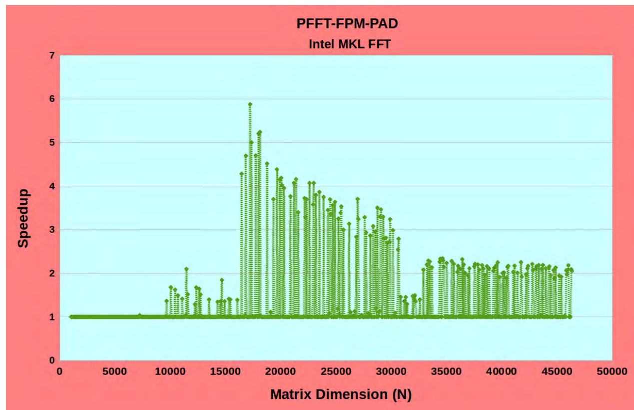


FIGURE 12. Speedup of *PFFT-FPM-PAD* against the basic Intel MKL FFT executed using 36 threads.

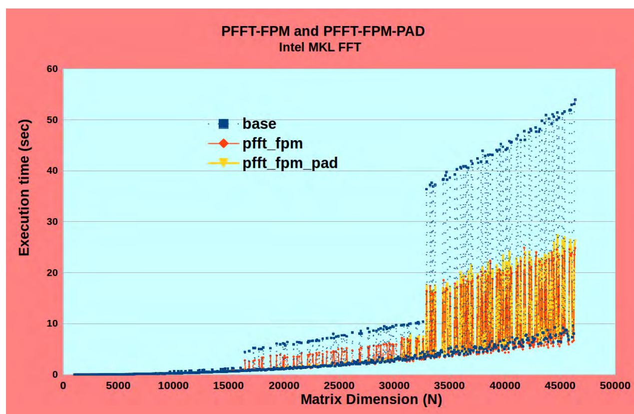


FIGURE 13. Execution times of *PFFT-FPM* and *PFFT-FPM-PAD* against the basic Intel MKL FFT executed using 36 threads.

Figure 14 shows the speedup of FFTW-3.3.7 using *PFFT-FPM-PAD* versus unoptimized FFTW-2.1.5. One can see that in the range of problem sizes ($N < 15000$), FFTW-2.1.5 performs better than FFTW-3.3.7. There are problem sizes in the range ($N > 30000$) again where it is better. The average performances of FFTW-3.3.7 and FFTW-2.1.5 are 7297 MFLOPs and 7033 MFLOPs. The average speedup of FFTW-3.3.7 over FFTW-2.1.5 is 1.2x. Our optimizations have improved the average performance of FFTW-3.3.7 over FFTW-2.1.5 by 42%.

Figure 15 shows the speedup of Intel MKL FFT using *PFFT-FPM-PAD* versus unoptimized FFTW-2.1.5. The average performances of Intel MKL FFT and FFTW-2.1.5 are 11170 MFLOPs and 7033 MFLOPs (Intel MKL FFT being 60% better). There are around 91 problem sizes (majority of them closer to the end of the figure) where FFTW-2.1.5 exhibits better performance than Intel MKL FFT. Our optimizations have improved the average performance of Intel MKL FFT over FFTW-2.1.5 by 24% (over and above the 36% of unoptimized Intel MKL FFT). The average speedup of FFTW-3.3.7 over FFTW-2.1.5 is 1.7x.

D. SUMMARY

We summarize the results below. The improvements to average to peak performance ratio is equal to the improvements for average performance since the peak performance has remained unchanged.

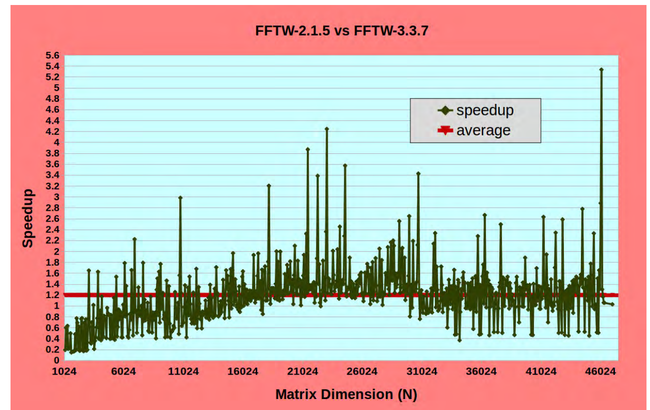


FIGURE 14. Speedup of optimized FFTW-3.3.7 (using *PFFT-FPM-PAD*) over unoptimized FFTW-2.1.5. The average speedup is 1.2x.

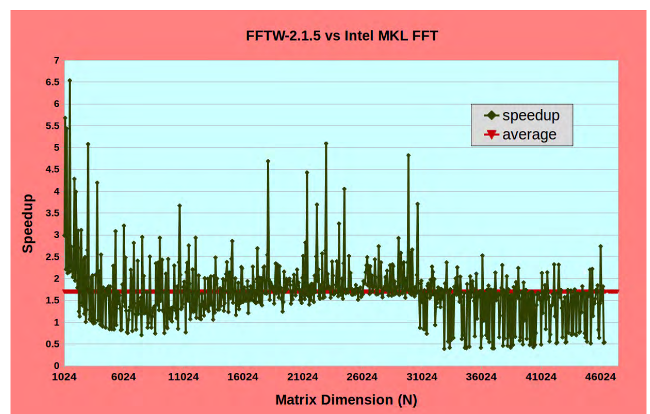


FIGURE 15. Speedup of optimized Intel MKL FFT (using *PFFT-FPM-PAD*) over unoptimized FFTW-2.1.5. The average speedup is 1.7x.

- For problem sizes in the range ($0 < N \leq 10000$), the speedups provided by *PFFT-FPM* and *PFFT-FPM-PAD* for Intel MKL FFT are not significant. This is because the variations (performance drops) are not remarkable.
- For problem sizes in the range ($10000 < N \leq 33000$), the speedups are good.
  For FFTW-3.3.7, the average and maximum speedups provided by *PFFT-FPM* are 2.7x and 6.8x and those provided by *PFFT-FPM-PAD* are 3x and 9.4x.
  For Intel MKL FFT, the average and maximum speedups provided by *PFFT-FPM* are 1.4x and 2x and those provided by *PFFT-FPM-PAD* are 2.7x and 5.9x. The variations (performance drops) are virtually removed.
- For problem sizes in the range ($N > 33000$), the speedups are good but major variations still remain. The variations are more severe for Intel MKL FFT. We aim to find solutions to remove them in our future work.

- The variations of performance are greater in the Y direction in the speed functions (Appendix E, supplemental). This is the reason why *PFFT-FPM-PAD* performs better than *PFFT-FPM* since it is able to exploit well the variations.
- The average speeds/performances of *PFFT-FPM* using FFTW-3.3.7 and Intel MKL FFT are 7041 MFLOPs and 10818 MFLOPs. So, Intel MKL FFT is on an average 54% better than FFTW-3.3.7. There are 135 problem sizes (out of 1000) where FFTW-3.3.7 performs better Intel MKL FFT.
- The average speeds/performances of *PFFT-FPM-PAD* using FFTW-3.3.7 and Intel MKL FFT are 7297 MFLOPs and 11170 MFLOPs. So, Intel MKL FFT is on an average 53% better than FFTW-3.3.7. There are 81 problem sizes (out of 1000) where FFTW-3.3.7 performs better than Intel MKL FFT.
- The optimized FFTW-3.3.7 and Intel MKL FFT using *PFFT-FPM-PAD* demonstrate average performance improvements of 42% and 24% over FFTW-2.1.5. There are problem sizes where FFTW-2.1.5 still performs better than FFTW-3.3.7 and Intel MKL FFT.

V. RELATED WORK

In this section, we review parallel solutions proposed for performance optimization of FFT on both homogeneous and heterogeneous platforms. We survey load-balancing algorithms employed for performance optimization of FFT and other scientific applications on modern multicore platforms. Finally, we present an overview of the latest efforts addressing the variations using load imbalancing algorithms on modern high performance computing platforms.

A. PARALLEL FFT SOLUTIONS FOR HOMOGENEOUS AND HETEROGENEOUS PLATFORMS

There are works that present parallel FFTs for distributed memory architectures. Averbuch and Gabber [15] present a parallel version of the CooleyTukey FFT algorithm for MIMD multiprocessors and demonstrate efficiency of 90% on a message-passing IBM SP2 computer.

Chen *et al.* [16] analyze the optimization challenges and opportunities of both 1D and 2D FFT including problem decomposition, load balancing, work distribution, and data-reuse together with the exploiting of the C64 architecture features on the IBM Cyclops-64 chip architecture.

Almeida and Moreno [17] consider parallelization of the bidimensional FFT-2D on heterogeneous system using master-slaves approaches.

Dmitruk *et al.* [18] use a 1D domain decomposition algorithm for performance improvement of 3D real FFT. They present techniques for reducing the cost of communications in the communication-intensive transpose operation of their algorithm.

Ayala and Wang [19] propose a parallel FFT implementation based on 2D domain decomposition and they demonstrate scalability of their solution on extreme scale computers.

Jung *et al.* [20] introduce two schemes based on the volumetric decomposition for the optimization of hybrid (MPI + OpenMP) parallelization schemes of 3D FFT. In one scheme *1d_Alltoall*, they apply five 1D all-to-all communications among fewer processors and in another, two 1D all-to-all communication and one 2D communication *(2d_Alltoall)*. They state that both schemes show good performance and scalability in 3D FFT calculations.

Song and Hollingsworth [21] present a scalable method for parallel 3-D FFT that exploits computation-communication overlap. Their method employs non-blocking MPI collectives in the 2D decomposition method for parallel 3D FFT.

We now review research works that have proposed optimized FFT implementations for GPU platforms. Chen *et al.* [22] present optimized FFT implementations for GPU clusters. Gu *et al.* [23] propose out-of-card implementations for 1D, 2D, and 3D FFTs on GPUs. Wu and Jaja [24] present optimized multi-dimensional FFT implementations on CPUGPU heterogeneous platforms where the input signal matrix is too large to fit in the GPU global memory. Naik and Kusur [25] demonstrate good performance improvement of FFT on their heterogeneous cluster compared to a homogeneous cluster.

B. PARALLEL FFT LIBRARIES

The Fastest Fourier Transform in the West (FFTW) [26], [27] is a software library for computing discrete Fourier transforms (DFTs). It provides routines utilizing threads for parallel one- and multi-dimensional transforms of both real and complex data, and multi-dimensional transforms of real and complex data for parallel machines supporting MPI.

Pekurovsky [28] presents a library P3DFFT, which computes fast Fourier transforms (FFTs) in three dimensions by using two-dimensional domain decomposition. Li and Laizet [29] provide an to perform three-dimensional distributed FFTs using MPI. OpenFFT [30] is an open source parallel package for computing multi-dimensional Fast Fourier Transforms (3-D and 4-D FFTs) of both real and complex numbers of arbitrary input size.

The Intel Math Kernel library (Intel MKL) [6] provides an interface for computing a discrete Fourier transform in one, two, or three dimensions with support for mixed radices. It provides DFT routines for single-processor or shared-memory systems, and for distributed-memory architectures.

C. LOAD BALANCING ALGORITHMS FOR PERFORMANCE OPTIMIZATION ON MULTICORE PLATFORMS

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

Static algorithms use *a priori* information about the parallel application and platform [31], [32]. They are particularly useful for applications where data locality is important because they do not require data redistribution. These

algorithms however are unsuitable for non-dedicated platforms, where load changes with time.

Dynamic algorithms balance the load by moving fine-grained tasks between processors during the execution [33]–[35]. 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.

In the non-centralized load balancing algorithms, at some point of computation, each processor find neighbors that are less loaded than itself and redistributes data between them [36], [37]. In centralized algorithms, there is a centralized load balancer that decides when to distribute data based on global load information [38], [39].

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 [40]. In other words, there is time-synchronization between all processors. In an asynchronous algorithm, the time synchronization is absent [41].

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 [31], [42]. These FPMs capture accurately the real-life behavior of applications executing on nodes consisting of uniprocessors (single-core CPUs).

D. LOAD IMBALANCING ALGORITHMS FOR PERFORMANCE OPTIMIZATION ON HPC PLATFORMS

Lastovetsky *et al.* [43], [44] 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. 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. It does not propose a general partitioning algorithm for arbitrary p.

References [11], [12], and [45] are theoretical works that present novel data partitioning algorithms for minimization of time and energy of computations for the most general performance and energy profiles of data-parallel applications executing on homogeneous and heterogeneous multicore clusters.

We propose in this work novel performance optimization methods specifically designed and implemented for a real-life multithreaded application (2D-DFT) on multicore processors.

VI. CONCLUSION

Fast Fourier transform (FFT) computation is so fundamental that hardware vendors provide libraries offering optimized routines for it for their processors. Its performance on latest multicore platforms is therefore of paramount concern to the high performance computing community. The inherent complexities in these platforms such as severe resource contention and non-uniform memory access (NUMA) however pose formidable challenges.

We demonstrated the challenges by studying three highly optimized multithreaded 2D FFT packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT on a modern Intel Haswell multicore processor consisting of thirty-six cores. In summary, we showed that for the routines from the three packages, the average performance can be considerably lower than their peak performance due to drastic variations in their performance profiles. The percentage ratios of average to peak performance for FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT are 40%, 30% and 24%. Therefore, we conclude that improving the average performance of the FFT routines on modern multicore processors by removal of variations is an important research challenge.

We proposed two novel optimization methods, *PFFT-FPM* and *PFFT-FPM-PAD*, specifically designed and implemented to address the challenge. They employ parallel computing based on load imbalancing technique and are portable. The methods take as inputs, discrete 3D functions of performance against problem size of the processors, compute 2D-DFT of a complex signal matrix of size $N \times N$ using p abstract processors, and output the transformed signal matrix. They can be employed as nodal optimization techniques to construct a 2D FFT routine highly optimized for a dedicated target multicore platform.

We performed our experiments on a modern Intel Haswell multicore server consisting of two sockets of 18 physical cores each. The average and maximum speedups observed for *PFFT-FPM* using *FFTW-3.3.7* are 1.9x and 6.8x and the average and maximum speedups observed using *Intel MKL FFT* are 1.3x and 2x. The average and maximum speedups observed for *PFFT-FPM-PAD* using *FFTW-3.3.7* are 2x and 9.4x and the average and maximum speedups observed using *Intel MKL FFT* are 1.4x and 5.9x. We showed that *PFFT-FPM-PAD* improves the average performance of FFTW-3.3.7 over the unoptimized FFTW-2.1.5 by 42% and the average performance of Intel MKL FFT over the unoptimized FFTW-2.1.5 by 24% (over and above the 36% of unoptimized Intel MKL FFT). The improvements to average to peak performance ratio is equal to the improvements for average performance since the peak performance has remained unchanged.

The software implementations of the methods presented in this work are at [46].

In our future work, we would research into solution methods for removing the major variations that still remain for very large problem sizes. We plan to apply and extend our methods for fast computation of 3D DFT. We would also develop extensions of the methods for homogeneous and heterogeneous clusters of multicore nodes.

APPENDIX A SUPPLEMENTAL MATERIAL

The following materials supplement the main manuscript:

- Experimental methodology followed to construct the speed functions illustrated in the main manuscript.
- Three solution approaches for the optimization of 2D-DFT computation (by removal of performance variations).
- Helper routines used in the methods, *PFFT-FPM* and *PFFT-FPM-PAD*.
- Additional material to supplement the discussion in the experimental results.

APPENDIX B EXPERIMENTAL METHODOLOGY TO BUILD THE SPEED FUNCTIONS

We followed the methodology described below to make sure the experimental results are reliable:

- The server is fully reserved and dedicated to these experiments during their execution. We 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 behavior 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, we use Student's t-test assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. When we mention 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 8, 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 our experiments, we set the minimum and maximum number of repetitions, *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, we increase the number of repetitions and also the allowed maximum elapsed time. Therefore, we make sure that statistical confidence is achieved for all the data points that we use in our performance profiles.

Algorithm 8 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, $reps - 1$)) × 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
PERFORMANCE OPTIMIZATION OF FAST FOURIER TRANSFORM ON MULTICORE PROCESSORS: SOLUTION APPROACHES

In this section, we describe in detail the three solution approaches for the optimization of 2D-DFT computation (by removal of performance variations). We discuss the advantages and disadvantages of each approach.

A. OPTIMIZATION THROUGH SOURCE CODE ANALYSIS AND TUNING

This is typically the first approach adopted to improve the performance of an application. It has following disadvantages:

- If the code is finely tuned to a specific vendor architecture, its portability to other vendor architectures suffers.
- Most high quality codes are proprietary and therefore their sources are not available for inspection and tuning. For example: BLAS, FFT packages that are part of Intel MKL library.
- It requires source code modification. Since the highly optimized packages such as FFTW are written with many man-years of effort for different generations of hardware, any source code change may entail extensive testing to ensure old functionality is not broken. Therefore, it is a specialized skill practiced by code tuning experts and is time consuming.

B. OPTIMIZATION USING SOLUTIONS TO LARGER PROBLEM SIZES WITH BETTER PERFORMANCE

Supposing we are solving a problem where the size of the matrix is N. In this approach, the solution to a larger problem size ($N_l > N$), which has better execution time than N, is used as solution for N. The common approach is to pad the input matrix to increase its problem size from N to N_l and zero the contents of the extra padded areas. It is also a technique that is widely used in different flavors (restructuring arrays, aggregation) to minimize cache conflict misses [47]–[50]. It requires no source code modification of the optimized package.

While it is a portable approach, it also has some disadvantages.

- A performance model is necessary that given N as input will output the problem size N_l to be used for padding. In this work, we use functional performance models (FPMs) that will provide this information.
- While programmatically extending 1D arrays logically is easy, it is not the case for 2D arrays such as matrices and multidimensional arrays. One inexpensive technique is to locally copy the input signal matrix of size N to a work matrix of size N_l, compute 2D-DFT of the work matrix and copy the relevant content back to the signal matrix. One drawback however is the extra memory used for the work matrix.

```
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);
        }
    }
}
```

FIGURE 16. Transpose of square matrix of size $n \times n$ using blocking.

C. OPTIMIZATION USING MODEL-BASED PARALLEL COMPUTING

Finally, we propose the third approach, which employs parallel computing. In the current era of multicores where processors have abundant number of cores, one can partition the workload between identical multithreaded routines (abstract processors) and execute them in parallel.

The starting step in this approach is to partition the workload evenly between the identical multithreaded routines by using load balanced distribution. Then, the workload is unevenly distributed using data partitioning algorithms that take as input realistic and accurate performance models of computation output from the starting step and that minimize the total execution time of the parallel execution of the 2D DFT computation. The models must not employ parameters, which are architecture-specific (For example: performance monitoring events (PMCs)). This would compromise the portability of this approach.

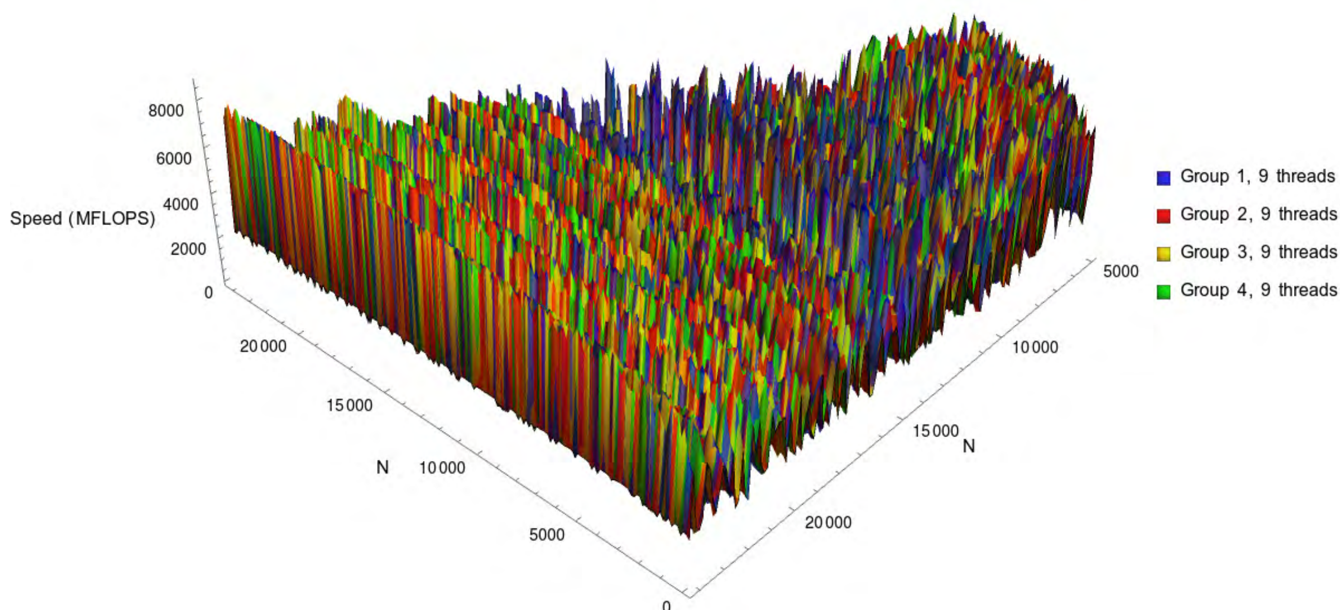


FIGURE 17. Full speed function of FFTW-3.3.7.

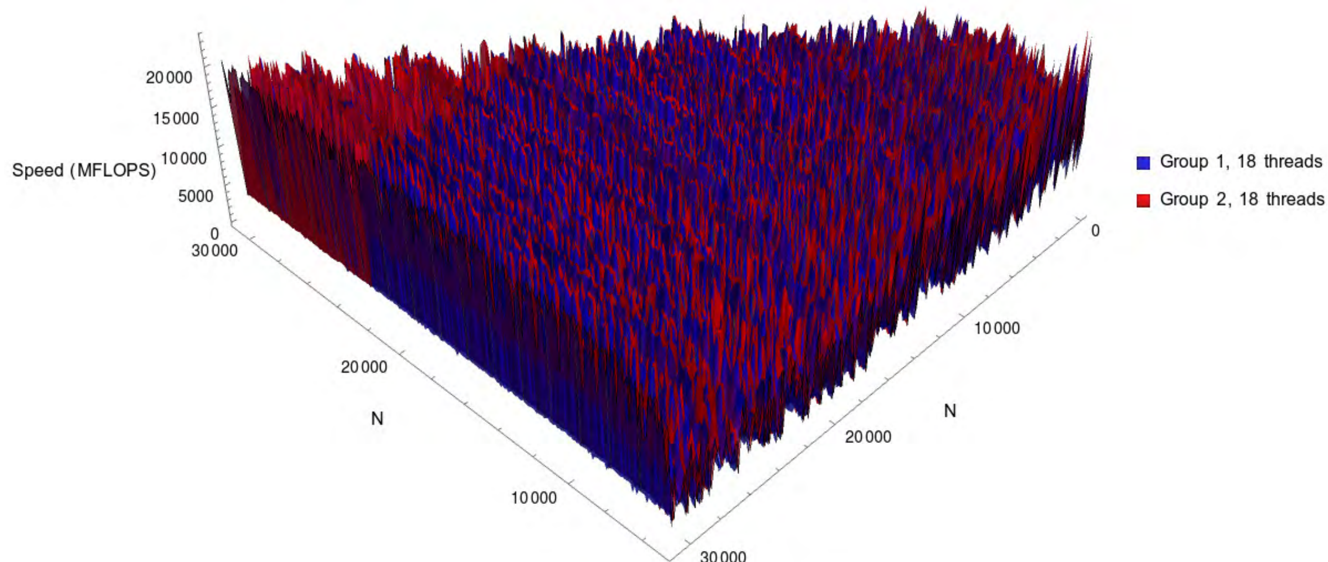


FIGURE 18. Full speed function of Intel MKL FFT.

Its advantages 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.

We propose in this work two methods, *PFFT-FPM* and *PFFT-FPM-PAD*. The first method adopts the third approach

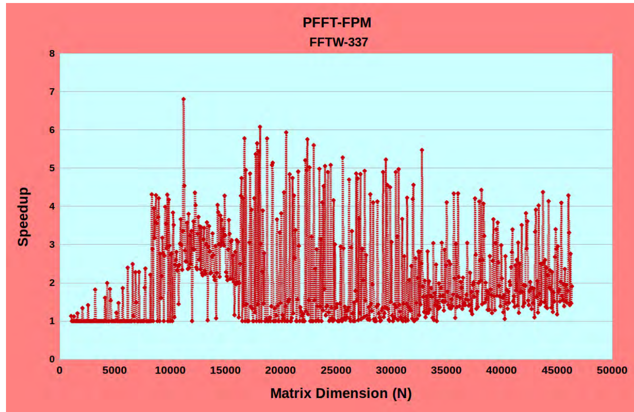


FIGURE 19. Speedup of *PFFT-FPM* against the basic FFTW-3.3.7.

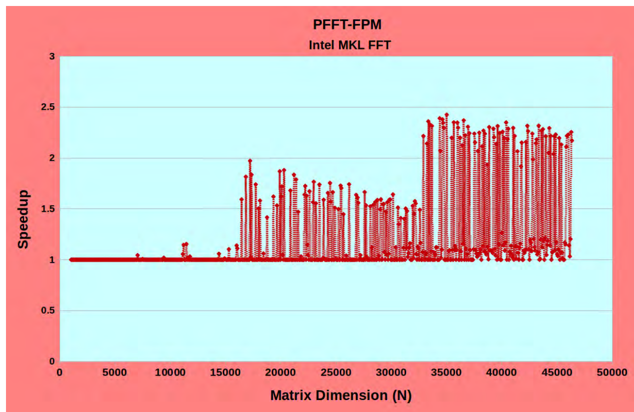


FIGURE 20. Speedup of *PFFT-FPM* against the basic Intel MKL FFT.

and is a model-based parallel computing solution employing functional performance models (FPMs). The second is an extension of the first method. It combines the third approach with the second approach where the lengths of the paddings are determined from the FPMs.

APPENDIX D
HELPER ROUTINES INVOKED IN *PFFT-FPM* AND *PFFT-FPM-PAD*

The following routine, *hcl_transpose_block*, performs in-place transpose of a complex 2D square matrix of size $n \times n$. We use a block size of 64 in our experiments as it is found to be optimal.

APPENDIX E
EXPERIMENTAL RESULTS: ADDITIONAL MATERIAL

D. FULL SPEED FUNCTIONS USING FFTW-3.3.7 AND INTEL MKL FFT

Figures 17 and 18 show the full speed functions of FFTW-3.3.7 and Intel MKL FFT.

E. EXECUTION TIMES OF PFFT-FPM *AND* PFFT-FPM-PAD

The Figure 19 shows the speedup of *PFFT-FPM*. The average and maximum speedups are 1.9x and 6.8x. Figure 20 shows the speedups of *PFFT-FPM*. The average and maximum speedups are 1.3x and 2.4x.

Figure 21 shows the execution times of *PFFT-FPM* only versus basic FFTW-3.3.7. Figure 22 shows the execution times of *PFFT-FPM-PAD* only versus basic FFTW-3.3.7.

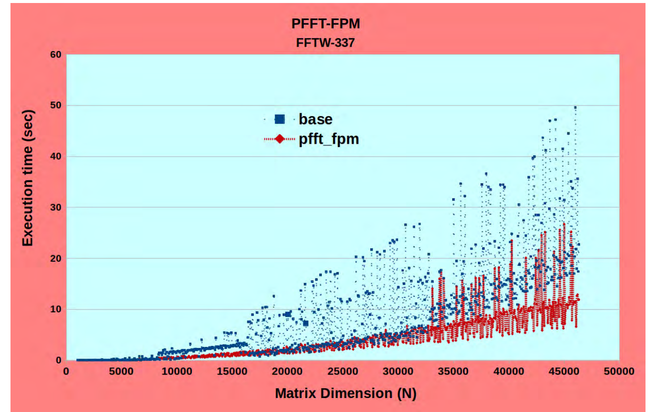


FIGURE 21. Execution times of *PFFT-FPM* against the basic FFTW-3.3.7.

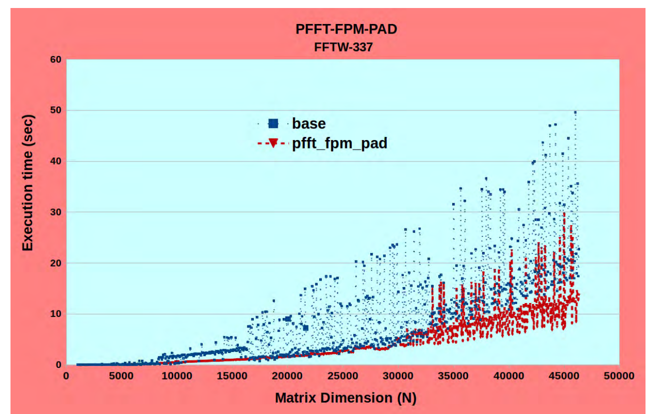


FIGURE 22. Execution times of *PFFT-FPM-PAD* against the basic FFTW-3.3.7.

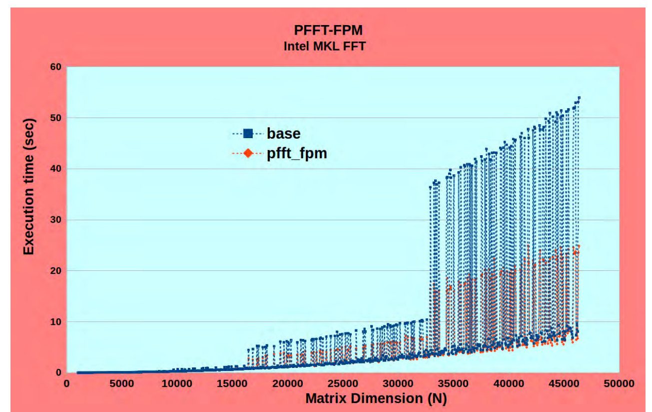


FIGURE 23. Execution times of *PFFT-FPM* against the basic Intel MKL FFT.

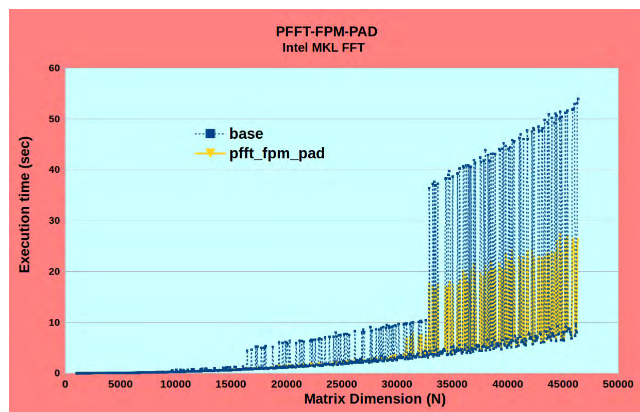


FIGURE 24. Execution times of *PFFT-FPM-PAD* against the basic Intel MKL FFT.

Figure 23 shows the execution times of *PFFT-FPM* only versus basic Intel MKL FFT. Figure 24 shows the execution times of *PFFT-FPM-PAD* only versus basic Intel MKL FFT.

REFERENCES

[1] W. Chu and B. Champagne, "A noise-robust FFT-based auditory spectrum with application in audio classification," *IEEE Trans. Audio, Speech, Language Process.*, vol. 16, no. 1, pp. 137–150, Jan. 2008.

[2] 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 Trans. Instrum. Meas.*, vol. 64, no. 11, pp. 3137–3146, Nov. 2015.

[3] 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. Ind. Electron.*, vol. 62, no. 12, pp. 7749–7761, Dec. 2015.

[4] M. Naoues, D. Noguet, L. Alaus, and Y. Louët, "A common operator for FFT and FEC decoding," *Microprocess. Microsyst.*, vol. 35, no. 8, pp. 708–715, 2011.

[5] J. P. Barbosa *et al.*, "A high performance hardware accelerator for dynamic texture segmentation," *J. Syst. Archit.*, vol. 61, no. 10, pp. 639–645, 2015.

[6] Intel Corporation. (2018). *Intel MKL FFT—Fast Fourier Transforms*. [Online]. Available: https://software.intel.com/en-us/mkl/features/fft

[7] cuFFT. (2018). *Optimized FFT Routines for NVIDIA Graphics Processors*. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

[8] clFFT. (2018). *Optimized FFT Routines for AMD Graphics Processors*. [Online]. Available: https://gpuopen.com/compute-product/clfft/

[9] (2008). *Intel Quickpath Interconnect*. [Online]. Available: https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect

[10] AMDHT. (2001). *Hypertransport*. [Online]. Available: https://en.wikipedia.org/wiki/HyperTransport

[11] 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 Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1119–1133, Apr. 2017.

[12] 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 Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2176–2190, Oct. 2018.

[13] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Process. Lett.*, vol. 21, pp. 195–217, Jun. 2011.

[14] A. Lastovetsky, R. Reddy, V. Rychkov, and D. Clarke. (2011). "Design and implementation of self-adaptable parallel algorithms for scientific computing on highly heterogeneous HPC platforms." [Online]. Available: https://arxiv.org/abs/1109.3074

[15] A. Averbuch and E. Gabber, "Portable parallel FFT for MIMD multiprocessors," *Concurrency, Pract. Exper.*, vol. 10, no. 8, pp. 583–605, 1998.

[16] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the fast fourier transform on a multi-core architecture," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Mar. 2007, pp. 1–8.

[17] F. Almeida and L. M. Moreno, "Parallel FFT-2D in heterogeneous systems," in *Proc. 23rd Multi-Conf. Appl. Inform. Int. Conf. Parallel Distrib. Comput. Netw. (IASTED)*, Innsbruck, Austria, Feb. 2005, pp. 551–558.

[18] P. Dmitruk, L.-P. Wang, W. Matthaeus, R. Zhang, and D. Seckel, "Scalable parallel FFT for spectral simulations on a Beowulf cluster," *Parallel Comput.*, vol. 27, no. 14, pp. 1921–1936, 2001.

[19] O. Ayala and L.-P. Wang, "Parallel implementation and scalability analysis of 3D fast fourier transform using 2D domain decomposition," *Parallel Comput.*, vol. 39, no. 1, pp. 58–77, 2013.

[20] J. Jung, C. Kobayashi, T. Imamura, and Y. Sugita, "Parallel implementation of 3D FFT with volumetric decomposition schemes for efficient molecular dynamics simulations," *Comput. Phys. Commun.*, vol. 200, pp. 57–65, Mar. 2016.

[21] S. Song and J. K. Hollingsworth, "Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT," *J. Comput. Sci.*, vol. 14, pp. 38–50, May 2016.

[22] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proc. 24th ACM Int. Conf. Supercomput. (ICS)*, 2010, pp. 315–324.

[23] L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 255–264.

[24] J. Wu and J. JaJa, "Optimized FFT computations on heterogeneous platforms with application to the Poisson equation," *J. Parallel Distrib. Comput.*, vol. 74, no. 8, pp. 2745–2756, 2014.

[25] V. H. Naik and C. S. Kusur, "Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment," in *Proc. Nat. Conf. Parallel Comput. Technol. (PARCOMPTECH)*, Feb. 2015, pp. 1–5.

[26] FFTW. (2018). *Fastest Fourier Transform in the West*. [Online]. Available: http://www.fftw.org/

[27] PFFTW. (2018). *Parallel FFTW*. [Online]. Available: http://www.fftw.org/fftw2_doc/fftw_4.html

[28] D. Pekurovsky, "P3DFFT: A framework for parallel computations of fourier transforms in three dimensions," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C192–C209, 2012.

[29] N. Li and S. Laizet, "2DECOMP and FFT—A highly scalable 2D decomposition library and FFT interface," in *Proc. Cray User Group Conf.*, 2010, pp. 1–13.

[30] T. V. T. Duy and T. Ozaki, "A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs," *Comput. Phys. Commun.*, vol. 185, no. 1, pp. 153–164, 2014.

[31] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 1, pp. 76–90, 2007.

[32] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2008, pp. 1–10.

[33] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A programming model for heterogeneous multi-core systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 287–296, 2008.

[34] G. Quintana-Ortı, F. D. Igual, E. S. Quintana-Ortí, and R. A. Van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 121–130, 2009.

[35] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *Proc. Eur. Conf. Parallel Process.* Springer, 2009, pp. 56–65.

[36] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[37] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 4, pp. 289–299, Apr. 2005.

[38] R. L. Cariño and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *J. Supercomput.*, vol. 44, no. 1, pp. 41–63, 2008.

[39] 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, pp. 151–159, Nov. 2011.

[40] J. Bahi, R. Couturier, and F. Vernier, "Synchronous distributed load balancing on dynamic networks," *J. Parallel Distrib. Comput.*, vol. 65, no. 11, pp. 1397–1405, 2005.

[41] F. Liu, Y. Chen, and W. S. Wong, "An asynchronous load balancing scheme for multi-server systems," in *Proc. IEEE Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, Oct. 2016, pp. 1–7.
[42] A. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Proc. 18th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Santa Fe, NM, USA, Apr. 2004, p. 104.
[43] A. L. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of MPDATA on Intel Xeon Phi through load imbalancing," *CoRR*, Jul. 2015.
[44] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 787–797, Mar. 2017.
[45] R. Reddy and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 160–177, Feb. 2017.
[46] S. Khokhriakov and R. Reddy. (2018). *HCLFFT: Software Library for Performance Optimization of 2D Discrete Fourier Transform on Multicore Processors*. [Online]. Available: https://git.ucd.ie/manumachu/hclfft.git
[47] K. Ishizaka, M. Obata, and H. Kasahara, "Cache optimization for coarse grain task parallel processing using inter-array padding," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.* Springer, 2003, pp. 64–76.
[48] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, "Forma: A framework for safe automatic array reshaping," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, p. 2, Nov. 2007.
[49] C. Hong *et al.*, "Effective padding of multidimensional arrays to avoid cache conflict misses," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2016, pp. 129–144.
[50] P. Jiang and G. Agrawal, "Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation," in *Proc. Int. Conf. Supercomput. (ICS)*, 2017, pp. 24:1–24:11.

SEMYON KHOKHRIAKOV received the bachelor's and master's degrees in physics from Udmurt State University, Russia, in 2011 and 2013, respectively. He is currently a Ph.D. Researcher with the Heterogeneous Computing Laboratory, School of Computer Science, University College Dublin. His main research interests include performance and energy consumption optimization on homogeneous and heterogeneous platforms, parallel/distributed computing, models, and algorithms.

RAVI REDDY MANUMACHU received the B.Tech. degree from IIT Madras in 1997 and the Ph.D. degree from the School of Computer Science and Informatics, University College Dublin, in 2005. His main research interests include high-performance heterogeneous computing, parallel computational fluid dynamics and finite-element analysis, complexity theory, and cryptography.

ALEXEY LASTOVETSKY received the Ph.D. degree from the Moscow Aviation Institute in 1986 and the D.Sc. degree from the Russian Academy of Sciences in 1997. He has authored over a 100 technical papers in refereed journals, edited books, and international conferences. He has authored the monographs *Parallel Computing on Heterogeneous Networks* (Wiley, 2003) and *High Performance Heterogeneous Computing* (Wiley, 2009). His main research interests include algorithms, models, and programming tools for high-performance heterogeneous computing.