Novel Model-based Methods for Performance Optimization of Multithreaded 2D Discrete Fourier Transform on Multicore Processors

Semyon Khokhriakov∗, Ravi Reddy†, and Alexey Lastovetsky†
School of Computer Science
University College Dublin
Dublin, Ireland
Email: ∗semen.khokhriakov@ucdconnect.ie, †{ravi.manumachu,alexey.lastovetsky}@ucd.ie

Abstract—Code modernization is a umbrella term used for porting and tuning codes to keep them up-to-date with the rapidly changing hardware landscape and to extract the best performance from the current hardware platforms. Roofline model is generally used to visually depict the cumulative performance gains from optimizations towards achieving the theoretical peak performance of a processor. However, this practice can be regressive with two typical symptoms. First, it can fall prey to Red Queen Principle where one spends several man-years putting extensive optimizations only in the long run to stay in the same place where one started. Second, it is very likely that an open source package with portable optimizations may exhibit better average performance overall than a heavily optimized vendor package.

In this paper, we expound this viewpoint using multithreaded fast Fourier transforms provided in three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT. Then, we propose a novel model-based parallel computing technique as a very effective and portable method for optimization of scientific multithreaded routines for performance, especially in the current multicore era where the processors have abundant number of cores. We present two optimization methods, PFFT-FPM and PFFT-FPM-PAD, based on this technique. They compute 2D-DFT of a complex signal matrix of size $N \times N$ using $p$ abstract processors. Both the algorithms take as inputs, discrete 3D functions of performance against problem size of the 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 1.9x and 6.8x respectively and the average and maximum speedups observed using Intel MKL FFT are 1.3x and 2x respectively. The average and maximum speedups observed for PFFT-FPM-PAD using FFTW-3.3.7 are 2x and 9.4x respectively and the average and maximum speedups observed using Intel MKL FFT are 1.4x and 5.9x respectively.

Index Terms—fast Fourier transform, multicore, data partitioning, load balancing, performance optimization, code tuning

I. INTRODUCTION

Code modernization is a perpetual endeavour of performance experts to port and tune their codes to keep up-to-date with the rapidly changing hardware platforms and to run efficiently on them. The roofline model is used to visually depict the trend of performance gains accrued from intra-node optimizations towards the theoretical peak performance of a processor. Using this model, the high optimized scientific applications such as Intel Math Kernel Library (Intel MKL) (BLAS, FFT) consistently demonstrate the close-to-peak performance or superior performance of their codes (such as BLAS) for new platforms. However, we show that this practice confined to the specialist domain of code optimization experts can, not only be time-consuming but also harmful in the long run with two typical symptoms. First, it can fall prey to Red Queen Principle where one spends several man-years putting extensive optimizations only in the long run to stay in the same place where one started. This is because hardware architectures are changing rapidly to fuel the progress towards unprecedented computational capabilities such as exascale computing. Architecture-specific optimizations may become obsolete for newer architectures. Second, it is very likely that an open source package with portable optimizations may exhibit better average performance overall than a heavily optimized vendor package. We exemplify this viewpoint using a case study.

We use three multithreaded FFT applications for comparison written using the packages FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT respectively. The performance profiles/speed functions for the applications are obtained on a modern Intel Haswell multicore server consisting of 2 sockets of 18 physical cores each (specification shown in Table I). 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, $\{128, 192, ..., 64000\}$. We will be referring frequently to width of performance variations in a performance profile. It is related to the difference of speed between two subsequent local minima ($s_1$) and maxima ($s_2$) and is defined below:

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

To make sure the experimental results are reliable, we
Based on these comparisons, we make the following important conclusions:

- **Extensive nodal optimization of FFT using highly architecture-specific techniques is harmful and futile in the long run since hardware platforms undergo drastic changes.** A supreme example is FFTW-2.1.5 versus FFTW-3.3.7. The FFT package, FFTW-2.1.5, was last updated in 1999. It outperforms FFTW-3.3.7, which undergoes constant revisions in terms of code optimizations.
- **An open source package may perform better than highly optimized vendor package since it employs portable optimizations.** A good example is FFTW-3.3.7 versus Intel MKL FFT. Intel MKL FFT is highly optimized for some specific problem sizes but exhibits poor performance for the rest. This can be seen from the width of its performance variations. Though the average performance of FFTW-3.3.7 is lesser than Intel MKL FFT, it outperforms Intel MKL FFT for many problem sizes and its variations are lesser.

There are three solution approaches that can be employed for the optimization of 2D-DFT computation by removal of performance variations. These approaches can be applied, in general, for optimization of data-parallel applications on modern multicore processors for performance.

- **Optimization through source code analysis and tuning:** This approach requires source code modification. It lacks portability if architecture-specific optimizations are used. It has other disadvantages, the most crucial being the disproportion between the time spent tuning the code and the continued long-term portable performance improvements.
- **Optimization using solutions to larger problem sizes with better performance:** This is a portable approach. However, there has to be a performance model, which given workload size \( N \) to solve will output the problem size \( N_i (> N) \) that is to be used for padding. 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 several identical multithreaded routines (abstract processors) and execute them in parallel. This is a highly portable approach and as we show in this paper, can demonstrate good portable performance.

We describe these approaches in the background section to follow.

In this paper, we propose a novel model-based parallel computing technique as a very effective and portable method for optimization of multithreaded routines for performance on multicore processors. We present two optimization methods, \( PFFT-FPM \) and \( PFFT-FPM-PAD \), based on this technique.

The first method adopts the third solution approach and is a model-based parallel computing solution employing functional performance models (FPMs). The second method is an exten-
Fig. 1. Performance profiles of 2D-FFT computing 2D-DFT of size $N \times N$ using FFTW-2.1.5 and FFTW-3.3.7 respectively. The 2D-FFT applications are executed using 36 threads on an Intel multicore server consisting of two sockets of 18 cores each.

Our main contributions can be summarized as follows:

- We describe the drawbacks inherent in the practice of extensive nodal optimization using highly architecture-specific optimizations. We use computation of 2D-DFT by multi-threaded FFT routines offered by three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT, for this purpose. We show that FFTW-2.1.5, which is obsolete and a decade older than FFTW-3.3.7 performs better than it for several problem sizes and has better average performance. We also show that a heavily optimized vendor package, Intel MKL FFT, has severe performance variations compared to FFTW-2.1.5 and FFTW-3.3.7 and several problem sizes where its performance is worse even though its average performance is a bit better.

- We propose two novel nodal optimization methods using model-based parallel computing to compute 2D-DFT on modern multicore servers and are therefore highly portable. 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 our optimization 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).

The rest of the paper is structured as follows. Section 3 presents our two model-based parallel computing solutions. Section 4 contains the experimental results. Section 5 concludes the paper.

II. Performance Optimization of Fast Fourier Transform on Multicore Processors: Solution Approaches

In this section, we describe three solution approaches for the optimization of 2D-DFT computation (by removal of performance variations). These approaches can be applied, in general, for optimization of data-parallel applications on modern multicore processors for performance. We discuss the advantages and disadvantages of each approach.

Optimization through source code analysis and tuning: This is typically the first approach adopted to improve the
If the code is highly tuned to a specific vendor architecture, its portability to other vendor architectures suffers. It is also debatable (as we show in this paper) if the performance improvements carry forward to different generations of the same architecture. Therefore, it lacks portable performance.

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 will require source code modification. Since the highly optimized packages such as FFTW are written with several 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 time consuming process.

- The most crucial disadvantage that we reiterate is the disproportion between the time spent tuning the code and the continued long-term portable performance improvements. Even if the code is available open source, tuning the code requires expertise and intricate knowledge of the hardware architecture. Therefore, code tuning is a highly specialized skill and is also usually time consuming. However, the performance improvements accrued long-term are not always promising. Figures [12] depict a striking example showing the performances of FFTW-2.1.5 and FFTW-3.3.7. FFTW-2.1.5 is last updated in 1999 whereas FFTW-3.3.7 is the latest release (September 2017) and contains numerous optimizations (SIMD, AVX, etc.). While for some special problem sizes, FFTW-3.3.7 is better than FFTW-2.1.5, there are many problem sizes where FFTW-2.1.5 outperforms FFTW-3.3.7. The
average performance of FFTW-2.1.5 is much better.

**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 the 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 flavours (restructuring arrays, aggregation) to minimize cache conflict misses [1], [2], [3], [4]. It requires no source code modification of the optimized package.

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

- There has to be a performance model, which given $N$ will provide the problem size $N_l$ that is 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, which is returned to the user. However, the drawback is the extra memory used for the work matrix.

**Optimization using model-based parallel computing:** Finally, we propose the third approach, which is based on parallel computing. In the current era of multicores where processors have abundant number of cores, one can partition the workload between several identical multithreaded routines (abstract processors) and execute them in parallel. This method can be a very effective nodal optimization technique especially when it employs realistic performance models of computation and efficient data partitioning algorithms that use the models as input.

Its advantages are:

- It is highly portable when the performance models of computation used in the data partitioning algorithms do not use architecture-specific parameters.
- No source code modification of the optimized package is required.
- Relatively less time-consuming programming effort involved, which is to distribute the workload between several identical multithreaded routines (abstract processors) and execute them in parallel.
- Speedups can be very good (as we show in this work) and are portable.
The disadvantages are:

- To distribute the data between the identical multithreaded routines (abstract processors), one can start with homogeneous distribution. But to squeeze out the maximum performance, realistic and accurate performance models and efficient data partitioning algorithms are necessary. It should be noted that the model must not be based on parameters, which are highly architecture-specific (for example: performance monitoring events (PMCs)). This would compromise the portability of this approach.

In this paper, we present two algorithms, \textit{PFFT-FPM} and \textit{PFFT-FPM-PAD}. The first algorithm adopts the third approach and is a model-based parallel computing solution employing functional performance models (FPMs). The second is an extension of the first algorithm. It combines the third approach with the second approach where the lengths of the paddings are determined from the FPMs.

### III. 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 novel model-based optimization methods. The first method \textit{PFFT-FPM} employs parallel computing technique and takes as input, discrete 3D functions of performance against problem size of the processors (FPMs). The second method \textit{PFFT-FPM-PAD} is an extension of \textit{PFFT-FPM} and employs padding, where the partitions (problem sizes) are padded by lengths determined from the FPMs.

#### A. Sequential 2D-FFT Algorithm

We first describe the sequential algorithm for computing the DFT on a two-dimensional point discrete signal \( M \) of size \( N \times N \). We call \( M \) the signal matrix where each element \( M[i][j] \) is a complex number. The 2D-DFT of \( M \) is defined by:

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

\[
\omega_N = e^{-\frac{2\pi}{N}}, 0 \leq k, l \leq N - 1
\]

The total number of complex multiplications required to compute the 2D-DFT is \( \Theta(N^4) \). This complexity can be
Fig. 5. Performance profiles of $2D$-FFT computing $2D$-DFT of size $N \times N$ using FFTW-3.3.7 and Intel MKL FFT respectively. The $2D$-FFT applications are executed using 36 threads on a Intel multicore server consisting of two sockets of 18 cores each.

Fig. 6. The average speeds of FFTW-3.3.7 and Intel MKL FFT respectively.
reduced very significantly by using row-column decomposition method where the 2D-DFT is computed 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. The method is depicted in Figure 4 and is mathematically summarized below:

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

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 \(\hat{X}[i][l]\), \(\forall l \in \{0, N - 1\}\). 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 \(\hat{X}\). The column \(l\) of \(\hat{X}\) 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 \((P_1, \ldots, P_p)\). 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 \(M\). The output from the algorithm is the transformed signal matrix \(\hat{M}\). All the FFTs that we discuss in this work are considered to be in-place.

**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:** The matrix \(M\) is transposed.

**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:** The matrix \(M\) is again transposed.

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 7.

C. PFFT-FPM: Performance Optimization Using FPMs and Load Imbalancing

We now describe our new model-based optimization method called PFFT-FPM that employs parallel computing technique and is based on functional performance models (FPMs).

**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, \(S\), and the user-input tolerance \(\epsilon\). The output from PFFT-FPM is the transformed signal matrix \(\hat{M}\). The discrete speed function of processor \(P_i\) is given by \(\hat{S}_i = \{s_i(x_1, y_1), \ldots, s_i(x_m, y_m)\}\) 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 calculated using the formula: \(2.5 \times x^g \times y^j\), 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 \(S\) 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?: \(\exists (x_k, N), 1 \leq k \leq m, \left(\max_{i=1}^{p} s_i(x_k, N) - \min_{i=1}^{p} s_i(x_k, N) > \epsilon\right)\), go to Step 1d. Otherwise, go to Step 1c. If there exists a \((x_k, N)\), the speed functions can not be considered identical.

1c. Partition rows using POPTA: Construct a speed function \(S_{avg} = \{s_{avg, i}(x), \forall i \in [1, m]\}\), where \(s_{avg, i}(x) = \frac{\sum_{i=1}^{p} s_i(x)}{p}\). POPTA [5] 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 [6] 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 \(\{\sum_{k=1}^{i-1} d[i] + 1, \ldots, \sum_{k=1}^{i} d[i]\}\).

**Step 3. Matrix Transposition:** The matrix \(M\) is transposed.

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

**Step 5. Matrix Transposition:** Same as Step 3.

The algorithm is illustrated in the Figure 8 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 Lastovetsky et al. [5] and Khaleghzadeh et al. [6] respectively. 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 to be 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, optimal solutions found by both these algorithms may not load-balance an application. The output from the data partitioning algorithms is the data distribution of the rows, \( d = \{d_1, \ldots, d_p\} \).

Figures III-C [III-C] 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 the experimental section V-B). Each abstract processor consists of 18 threads. Figure III-C shows a plane \( y = N = 24704 \) intersecting the two speed functions \( S = \{S_1, S_2\} \) 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 \( N = 24704 \).

In the following section IV, we present the pseudocode of PFFT-FPM and its shared-memory implementations for Intel MKL FFT and FFTW-3.3.7 respectively on a Intel Haswell server containing 36 physical cores (specification shown in Table I).

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 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. However, the series of 1D-FFTs are performed locally on rows whose length is extended (padded) by an extent determined from the FPM of the processor. It should be noted that 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:
Fig. 9. 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. Speed functions are intersected by the plane $y = N = 24704$.

Fig. 10. 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, $(d[1] = x_1 = 11648, d[2] = x_2 = 13056)$. 

$\text{Speed functions of two abstract processors executing Intel MKL FFT}$

$\text{Speed functions against X keeping Y constant}$
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 \( M \) given by \( d[i] \). The length of each row \( N \) is padded to \( N_{\text{padded}} \). It is determined as follows using the FPM, \( S_i = s_i(x, y) \):

\[
N_{\text{padded}} = \arg \min_{v \in [N_v, N]} \left( \frac{d[i] \times V}{s_i(d[i], V)} \right) < d[i] \times N \frac{d[i]}{s_i(d[i], N)}
\]

The ratio \( \frac{x \times y}{x \times y} \) gives the execution time of problem size \( x \times y \). Essentially we select the point in the range \( \{(d[i], y_{N+1}), \ldots, (d[i], y_m)\} \) that has minimal execution time and better execution time than the point \( (d[i], N) \). If no such point is found, the padding length is set to 0. The elements in the padded region \( M[x, c], \forall c \in [N + 1, V] \) are set to 0.

Step 3. Matrix Transposition: The matrix \( M \) (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 III-E and III-D illustrate how they are determined from the FPMs 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 the experimental section V-B). Each abstract processor consists of 18 threads. Figure III-D shows two planes \( x_1 = 11648 \) and \( x_2 = 13056 \) intersecting the two speed functions \( S = \{S_1, S_2\} \) producing two curves, one for each group showing speed versus \( y \) keeping \( x \) constant. The padded lengths \( (N_{\text{padded}, 1}, N_{\text{padded}, 2}) \) corresponding to \( x_1 \) and \( x_2 \) are determined from the curves and are equal to 24960.

In the following section IV we present the pseudocode of PFFT-FPM-PAD and its shared-memory implementations for Intel MKL FFT and FFTW-3.3.7 respectively on the Intel Haswell server containing 36 physical cores (specification is shown in Table I).

IV. PSEUDOCODES 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 \( M \) of size \( N \times N \), the number of abstract processors (groups) \( p \), the speed functions represented by a set \( S \) respectively containing problem sizes and speeds, and number of threads in each abstract processor (group) represented by \( t \). The output from the algorithm is the transformed signal matrix \( M \) (considering that we are performing in-place FFT).

The pseudocode of the algorithm is illustrated in (Algorithm 1). 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 \( \epsilon \) (Algorithm 2, Line 3). If a point exists for which the variation exceeds \( \epsilon \), then the data partitioning algorithm HPOPTA [6] is invoked (Line 5) to determine the data partitioning of the rows. If all the variations are less than or equal to \( \epsilon \), the average of the speeds are calculated for each data point (Line 7). The averaged speed function is then input to POPTA [5] to determine the data partitioning of the rows (Line 9). The data distribution is output in the array, \( d = \{d_1, \ldots, d_p\} \).

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

1: procedure PFFT-FPM(\( M, p, S, t \))
2: \( d \leftarrow \) PARTITION(\( N, p, S, \epsilon, d \))
3: PFFT_LIMB(\( p, d, N, M \))
4: return \( M \)
5: end procedure

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 (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, \( d_i \). The implementation of this routine using FFTW interface is shown in Algorithm 6.

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.

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 respectively on a Intel Haswell server containing 36 physical cores (specification is shown in Table I).

The input parameters \( (p, t) \) to be used during the execution of PFFT-FPM and PFFT-FPM-PAD are obtained from the best load-balanced configuration observed experimentally.
Algorithm 2 Data partitioning of rows of signal matrix \( \mathcal{M} \) of size \( N \times N \) using the FPMs.

1: \textbf{procedure} \textsc{Partition}(\( N, p, \mathcal{S}, \epsilon, d \))

\textbf{Input:}
\( N \), Number of rows in the signal matrix, \( N \in \mathbb{Z}_{>0} \)
\( p \), Number of abstract processors, \( p \in \mathbb{Z}_{>0} \)
\( \mathcal{S} \), Functional performance model (speed functions) represented by,
\( \mathcal{S} = \{ S_1, \ldots, S_p \} \),
\( S_i = \{ (x_i[q][r], s_i[q][r]) \mid i \in [1, p], q, r \in [1, m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0} \} \)
\( \epsilon \), User tolerance, \( \epsilon \in \mathbb{R}_{>0} \)

\textbf{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: \textbf{for} \( \text{point} \leftarrow 1, m \) \textbf{do}
3: \hspace{1em} \text{rdiff} \leftarrow \max_{i=1}^{p} \min_{[\text{point}][N]} \left[ s_i[\text{point}][N] \right] - \min_{i=1}^{p} \min_{[\text{point}][N]} \left[ s_i[\text{point}][N] \right]
4: \hspace{1em} \textbf{if} (\text{rdiff} > \epsilon) \textbf{then}
5: \hspace{2em} \textbf{return} \textsc{HPOPTA}(\( N, p, \mathcal{S}, d \))
6: \hspace{1em} \textbf{end if}
7: \hspace{1em} \text{S}_{\text{avg}}[\text{point}] \leftarrow \sum_{i=1}^{p} \frac{d_i}{p}
8: \hspace{1em} \textbf{end for}
9: \textbf{return} \textsc{POPTA}(\( N, p, \text{S}_{\text{avg}}, d \))
10: \textbf{end procedure}

Algorithm 3 Parallel algorithm computing 2D-DFT of signal matrix \( \mathcal{M} \) of size \( N \times N \).

1: \textbf{procedure} \textsc{PFFT}\textsc{_LIMB}(\( p, d, N, \mathcal{M} \))

\textbf{Input:}
\( \mathcal{M} \), Signal matrix of size \( N \times N \), \( N \in \mathbb{Z}_{>0} \)
\( p \), Number of abstract processors, \( p \in \mathbb{Z}_{>0} \)

\textbf{Output:}
\( \mathcal{M} \), Signal matrix of size \( N \times N \), \( N \in \mathbb{Z}_{>0} \)

2: \textbf{for} \( \text{proc} \leftarrow 1, p \) \textbf{do}
3: \hspace{1em} \textsc{1D\_ROW\_FFTS\_LOCAL}(\( \text{proc}, d_{\text{proc}}, N, \mathcal{M} \))
4: \hspace{1em} \textbf{end for}
5: \textsc{PARALLEL\_TRANPOSE}(\( \mathcal{M} \))
6: \textbf{for} \( \text{proc} \leftarrow 1, p \) \textbf{do}
7: \hspace{1em} \textsc{1D\_ROW\_FFTS\_LOCAL}(\( \text{proc}, d_{\text{proc}}, N, \mathcal{M} \))
8: \hspace{1em} \textbf{end for}
9: \textsc{PARALLEL\_TRANPOSE}(\( \mathcal{M} \))
10: \textbf{return} \( \mathcal{M} \)
11: \textbf{end procedure}

1) Intel MKL FFT: For the implementation using Intel MKL FFT, we use two groups of 18 threads each, \( p = 2, t = 18 \). We experimentally found this pair to be the best among the following combinations: \{ (4, 9), (6, 6), (9, 4), (12, 3) \}, experimentally.

The routine \textsc{PFFT\_LIMB\_INTEL\_MKL} shows the implementation of \textsc{PFFT\_LIMB} using the FFTW interface. Lines
Fig. 12. Each intersection produces a curve for the group showing speed versus \(y\) keeping \(x\) constant. The lengths of padding for the two groups, \(N_{padded}\), is the same and is equal to 24960.

2) FFTW: For the implementation using FFTW-3.3.7, we use four groups of 9 threads each, \((p = 4, t = 9)\). We experimentally found this pair to be the best among the following combinations: \((2, 18), (6, 6), (9, 4), (12, 3)\), experimentally.

The routine \texttt{PFFT\_LIMB\_FFTW} shows the implementation of \texttt{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. It should be noted that only thread-safe routine in FFTW is \texttt{fftw\_execute}. All the other routines such an plan creation (\texttt{fftw\_plan\_many\_dft}) and plan destruction (\texttt{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. This is followed by fast transpose on Line 23.

The transpose routine using blocking is presented in the Appendix A.

V. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present our experimental results where we present the performance improvements provided by our two model-based optimization methods, \texttt{PFFT-FPM} and \texttt{PFFT-FPM-PAD}, respectively. Our experimental platform is an Intel Haswell server containing 36 physical cores. Its specification is shown in Table I.

We use two packages, FFTW-3.3.7 and Intel MKL FFT, for implementation of the algorithms. We could not optimize FFTW-2.1.5 since the implementation of series of row 1D-FFTs is quite poor using \texttt{fftw\_threads} compared to the implementation of \texttt{fftw\_plan\_many\_dft} in FFTW-3.3.7 and Intel MKL FFT. However, we will compare the speedups of optimized FFTW-3.3.7 and Intel MKL FFT with the unoptimized FFTW-2.1.5.

The input parameters \((p, t)\) to be used during the execution of \texttt{PFFT-FPM} and \texttt{PFFT-FPM-PAD} are obtained from the best load-balanced configuration observed experimentally. For the implementations using \texttt{FFTW-3.3.7}, we use four groups of 9 threads each, \((p = 4, t = 9)\) since
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, M)`

Input:
- $M$, 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:
- $M$, 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(1, d_1, N, M)`
7: `#pragma omp section`
8: `1D_ROW_FFTS_LOCAL(2, d_2, N, M)`
9: `TRANPOSE(M)`
10: `#pragma omp parallel sections num_threads(2)`
11: `#pragma omp section`
12: `1D_ROW_FFTS_LOCAL(1, d_1, N, M)`
13: `#pragma omp section`
14: `1D_ROW_FFTS_LOCAL(2, d_2, N, M)`
15: `TRANPOSE(M)`
16: `fftw_cleanup_threads()`
17: `return M`
18: `end procedure`

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, ($p = 2, t = 18$), which was found to be the best experimentally among the following combinations: $\{(4, 9), (6, 6), (9, 4), (12, 3)\}$.

A. 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.
- When an application is executed, it 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 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. We verify the validity of these assumptions by plotting the distributions of observations.
- The function `MeanUsingTtest`, shown in Algorithm 5, describes this step. For each data point, the function is invoked, which repeatedly executes the application `app` until one of the following three conditions is satisfied:
  1) The maximum number of repetitions (`maxReps`) have been exceeded (Line 3).
  2) The sample mean falls in the confidence interval (or the precision of measurement `eps` has been achieved) (Lines 15-17).
  3) The elapsed time of the repetitions of application execution has exceeded the maximum time allowed (`maxT` in seconds) (Lines 18-20).

So, for each data point, the function `MeanUsingTtest` is invoked and the sample mean `mean` is returned at the end of invocation. The function `Measure` measures the execution time using the HCL’s WattsUp library [7].
Algorithm 6 Series of $x$ row 1D-FFTs using FFTW interface function `fftw_plan_many_dft`.

1: procedure 1D_ROW_FFTS_LOCAL(id, $x$, $N$, $M$)
2: Input:
3: Processor identifier, $id \in \mathbb{Z}_{>0}$
4: Problem size $x \in \mathbb{Z}_{>0}$
5: $M$, Signal matrix of size $N \times N$, $N \in \mathbb{Z}_{>0}$
6: Output:
7: $M$, Signal matrix of size $N \times N$, $N \in \mathbb{Z}_{>0}$
8: 
9: rank ← 1; howmany ← $x$; $s ← N$
10: idist ← $N$; odist ← $N$; istride ← 1;
11: ostride ← 1; inembed ← $s$; onembed ← $s$
12: plan ← FFTW_PLAN_MANY_DFT(rank, $s$, howmany, $M$, inembed, istride, idist, $M$, onembed, ostride, odist, FFTW_FORWARD, FFTW_ESTIMATE)
13: FFTW_EXECUTE(plan)
14: FFTW_Destroy_plan(plan)
15: return $M$
16: 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, $x$, $N$, $M$)
2: Input:
3: Processor identifier, $id \in \mathbb{Z}_{>0}$
4: Problem size $x \in \mathbb{Z}_{>0}$
5: $M$, Signal matrix of size $N \times N$, $N \in \mathbb{Z}_{>0}$
6: Functional performance model (speed functions) represented by:
7: $S = \{S_1, \ldots, S_p\}$,
8: $S_i = \{ (x_i[q][r], s_i[q][r]) | i \in [1,p], q, r \in [1,m], x_i[q][r] \in \mathbb{Z}_{>0}, s_i[q][r] \in \mathbb{R}_{>0} \}$
9: Output:
10: $M$, Signal matrix of size $N \times N$, $N \in \mathbb{Z}_{>0}$
11: 
12: $N_{padded} ← \text{DETERMINE_PAD_LENGTH}(id, x, N, S)$
13: rank ← 1; howmany ← $x$; $s ← N_{padded}$
14: idist ← $N_{padded}$; odist ← $N_{padded}$; istride ← 1;
15: ostride ← 1; inembed ← $s$; onembed ← $s$
16: plan ← FFTW_PLAN_MANY_DFT(rank, $s$, howmany, $M$, inembed, istride, idist, $M$, onembed, ostride, odist, FFTW_FORWARD, FFTW_ESTIMATE)
17: FFTW_EXECUTE(plan)
18: FFTW_Destroy_plan(plan)
19: return $M$
20: end procedure

The input minimum and maximum number of repetitions, minReps and maxReps, differ based on the problem size solved. For small problem sizes (32 $\leq n \leq 1024$), these values are set to 10000 and 100000 respectively. For medium problem sizes (1024 $< n \leq 5120$), these values are set to 100 and 1000. For large problem sizes ($n > 5120$), these values are set to 5 and 50. The values of maxT, el, and eps are respectively set to 3600, 0.95, and 0.025. If the precision of measurement is not achieved before the maximum number of repetitions have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

B. Full Speed Functions

The full speed functions constructed for Intel MKL FFT and FFTW-3.3.7 are shown in the Figures [13] and [14] respectively. The inputs to the experimental methodology are the FFT application and the application parameters ($p$, $t$, $M$), and the problem sizes. The output is the set of discrete speed functions, $S = \{S_1, \ldots, S_p\}$, one for each abstract processor. The set of problem sizes ($x, y$) used for the construction of speed functions are $(x, y) \in [128 \times y \leq 64000, y \mod 128, \{128 \times 128, 128 \times 256, 256 \times 256, \ldots, 4000 \times 4000\}$. 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. It should be noted that for large problem sizes (for example: $(x, y) \in [128 \leq x \leq 64000, y = 64000$), all the data points $(x, y)$ can not be built due to main memory constraint. Therefore, the speed functions are built until permissible problem size.

The time to build the full speed functions can be quite expensive. This takes into account the fact that for each data point, statistical averaging is performed to determine its sample mean. It took around 96 hours each to build the speed functions for Intel MKL FFT and FFTW-3.3.7. However, partial speed functions [8], [9] can be built and input to the data partitioning algorithm [5], which would return sub-optimal data 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, $d_i = \frac{2}{x}, \forall i \in [1, p]$, 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.

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 groups of 36 threads in their execution). For PFFT-FPM, the speedup is calculated as follows: $\text{Speedup} = \frac{t_{basic}}{t_{pfft-fpm}}$, 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.
Fig. 13. Full speed function of FFTW-3.3.7.

Fig. 14. Full speed function of Intel MKL FFT.
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 ← 0; stop ← 0; sum ← 0; etime ← 0
3: while (reps < maxReps) and (!stop) do
4:   st ← MEASURE(TIME)
5:   EXECUTE(app)
6:   et ← MEASURE(TIME)
7:   reps ← reps + 1
8:   etime ← etime + et − st
9:   ObjArray[reps] ← et − st
10:  sum ← sum + ObjArray[reps]
11:  if reps > minReps then
12:     clOut ← fabs(gsl_cdf_tdist_Pinv(cl, reps − 1)) \times \frac{gsl_stats_sd(ObjArray, 1, reps)}{\sqrt{reps}}
13:     if clOut \times \frac{reps}{sum} < eps then
14:       stop ← 1
15:     end if
16:     if etime > maxT then
17:       stop ← 1
18:     end if
19: end if
20: end while
21: repsOut ← reps; epsOut ← clOut \times \frac{reps}{sum}
22: etimeOut ← etime; mean ← \frac{sum}{reps}
23: end procedure

Fig. 15. Speedup of PFFT-FPM and PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

Fig. 16. Speedup of PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

obtained using PFFT-FPM. For PFFT-FPM-PAD, the speedup is calculated as follows: Speedup = \frac{t_{basic}}{t_{pfft-fpm-pad}} , where \( t_{pfft-fpm-pad} \) is the execution time obtained using PFFT-FPM-PAD.

C. PFFT-FPM and PFFT-FPM-PAD using FFTW-3.3.7

Figure [15] shows the speedups of PFFT-FPM and PFFT-FPM-PAD over basic FFTW-3.3.7 where the 2D-DFT is computed 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 [16] shows the speedup of PFFT-FPM-PAD for problem sizes where performance has been improved. The average and maximum performance improvements are 2x and 9.4x respectively.

Figure [17] shows the execution times of PFFT-FPM and PFFT-FPM-PAD versus basic FFTW-3.3.7. Figure [18] shows the execution times of PFFT-FPM only versus basic FFTW-3.3.7. Figure [19] shows the execution times of PFFT-FPM-PAD only versus basic FFTW-3.3.7.

For problem sizes in the range \( N > 33000 \), while the speedups are still quite good (6x for FFTW-3.3.7), major
Fig. 17. Execution times of PFFT-FPM and PFFT-FPM-PAD against the basic FFTW-3.3.7 executed using 36 threads.

Fig. 18. Execution times of PFFT-FPM against the basic FFTW-3.3.7.

Fig. 19. Execution times of PFFT-FPM-PAD against the basic FFTW-3.3.7.

Fig. 20. Speedups of PFFT-FPM and PFFT-FPM-PAD against the basic Intel MKL FFT executed using 36 threads.

Fig. 21. Speedup of PFFT-FPM-PAD against the basic Intel MKL FFT executed using 36 threads.

Fig. 22. Execution times of PFFT-FPM-PAD against the basic Intel MKL FFT executed using 36 threads.

variations still remain.

D. PFFT-FPM and PFFT-FPM-PAD using Intel MKL FFT

Figure 20 compares the speedups PFFT-FPM and PFFT-FPM-PAD over basic Intel MKL FFT where the 2D-DFT is computed using one group consisting of 36 threads. Figure 21 shows the speedups of PFFT-FPM-PAD for problem sizes where performance has been improved. The average and maximum speedups are 1.4x and 5.9x respectively.

Figure 22 shows the execution times of PFFT-FPM and PFFT-FPM-PAD versus 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.

For problem sizes in the range ($N > 33000$), while the speedups are still quite good (2x for Intel MKL FFT), the variations are still quite significant.
Fig. 22. Execution times of PFFT-FPM and PFFT-FPM-PAD against the basic Intel MKL FFT executed using 36 threads.

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

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

Fig. 25. Speedup of optimized FFTW-3.3.7 (using PFFT-FPM-PAD) over unoptimized FFTW-2.1.5. The average speedup is 1.2x.

E. 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 25 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 few 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 respectively. The average speedup of FFTW-3.3.7 over FFTW-2.1.5 is 1.2x. Most importantly, our optimizations have improved the average performance of FFTW-3.3.7 over FFTW-2.1.5 by 42%.

Figure 26 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 respectively (Intel MKL FFT being 60% better). However, 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. Most importantly, 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.

F. Summary

We summarize the results below:

- 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 tremendous. For FFTW-3.3.7, the average and maximum speedups provided by PFFT-FPM are 2.7x and 6.8x respectively 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 respectively and those provided by PFFT-FPM-PAD are 2.7x and 5.9x respectively. The variations (performance drops) have been virtually completely removed.

- For problem sizes in the range \((N > 33000)\), the speedups are good but not as excellent and 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 average speeds/performances of PFFT-FPM using FFTW-3.3.7 and Intel MKL FFT are 7041 MFLOPs and 10818 MFLOPs respectively. So, Intel MKL FFT is on an average 54% better than FFTW-3.3.7. However, there are 135 problem sizes (out of 700) where FFTW-3.3.7 outperforms 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 respectively. There are 81 problem sizes (out of 700) where FFTW-3.3.7 outperforms Intel MKL FFT. So, Intel MKL FFT is on an average 53% better than FFTW-3.3.7.

- The optimized FFTW-3.3.7 and Intel MKL FFT using PFFT-FPM-PAD demonstrate average performance improvements of 42% and 24% respectively over FFTW-2.1.5. However, there are problem sizes where FFTW-2.1.5 still performs better than FFTW-3.3.7 and Intel MKL FFT. This will be the subject of our future research.

### VI. 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 scientific applications on modern multicore platforms. Finally, we present an overview of the latest efforts addressing the variations using load imbalancing algorithms on modern multicore platforms.

#### A. Parallel FFT solutions for homogeneous and heterogeneous platforms

There are several works that present parallel FFTs for distributed memory architectures. Averbuch et al. [10] present a parallel version of the Cooley-Tukey FFT algorithm for MIMD multiprocessors and demonstrate efficiency of 90% on a message-passing IBM SP2 computer.

Dmitruk et al. [11] 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.

We review few research works that have proposed optimized FFT implementations for GPU platforms. Chen et al. [12] present optimized FFT implementations for GPU clusters. Gu et al. [13] propose out-of-card implementations for 1D, 2D, and 3D FFTs on GPUs. Wu et al. [14] present optimized multi-dimensional FFT implementations on CPU/GPU heterogeneous platforms where the input signal matrix is too large to fit in the GPU global memory. Naik et al. [15] 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) [16], [17] 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 et al. [18] present a library P3DFFT, which computes fast Fourier transforms (FFTs) in three dimensions by using two-dimensional domain decomposition. Li et al. [19] provides an to perform three-dimensional distributed FFTs using MPI. OpenFFT [20] 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) [21] 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 several 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 [22], [23]. They are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms are may be 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 [24], [25], [26]. They often use static partitioning for their initial step due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead.

In the non-centralized load balancing algorithms, at some point of computation, each processor finds neighbors that are less loaded than itself and redistributes data between them [27], [28]. In centralized algorithms, there is a centralized load balancer that decides when to distribute data based on global load information [29], [30].

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 $t$ [31]. In other words, there is time-synchronization between all processors. In an asynchronous algorithm, the time synchronization is absent [32].

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 [33], [22]. These FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

D. Load imbalancing algorithms for performance optimization on multicore platforms

Lastovetsky et al. [34], [35] 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. However, no general partitioning algorithm is proposed in this work.

Lastovetsky et al. [5], Reddy et al. [36], and Khaleghzadeh et al. [6] are theoretical works that present novel data partitioning algorithms for minimizing time and energy of computations for the most general performance and energy profiles of data-parallel applications executing on homogeneous and heterogeneous multicore clusters.

In this paper, we present novel model-based methods for performance optimization of a real-life multithreaded application (2D-DFT) on multicore processors.

VII. Conclusion

Code modernization experts are engaged in a perpetual battle to keep their codes up-to-date with the ever-changing hardware landscape by porting and tuning them to extract the utmost performance from the current hardware platforms. They commonly use roofline model to gauge the performance gains accrued from incremental optimizations towards achieving the theoretical peak performance of a processor. However, since hardware platforms are changing at a rapid pace, this practice of incremental nodal optimization using architecture-specific techniques can be retrogressive with two typical symptoms. First, it can fall prey to Red Queen Principle where one spends several man-years putting extensive optimizations only in the long run to stay in the same place where one started. Second, it is very likely that an open source package with portable optimizations may exhibit better performance for some problem sizes and better average performance overall than a heavily optimized vendor package.

In this paper, we expounded this insight using multithreaded Fast Fourier transforms provided in three highly optimized packages, FFTW-2.1.5, FFTW-3.3.7, and Intel MKL FFT. Then, we proposed two novel model-based optimization methods, PFFT-FPM and PFFT-FPM-PAD, that employ parallel computing based on advanced functional performance models and are therefore highly portable. They compute 2D-DFT of a complex signal matrix of size $N \times N$ using $p$ abstract processors. Both the algorithms take as inputs, discrete 3D functions of performance against problem size of the processors and output the transformed signal matrix.

We performed our experiments on a modern Intel Haswell multicore server consisting of two processors 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 respectively and the average and maximum speedups observed using Intel MKL FFT are 1.3x and 2x respectively. The average and maximum speedups observed for PFFT-FPM-PAD using FFTW-3.3.7 are 2x and 9.4x respectively and the average and maximum speedups observed using Intel MKL FFT are 1.4x and 5.9x respectively. We showed that using our optimization 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).

The software implementations of the algorithms presented in this paper can be found at [37].

In our future work, we plan to extend our algorithms for fast computation of 3D-DFT. We would also develop extensions of them for homogeneous and heterogeneous clusters of multicore nodes.

ACKNOWLEDGMENT

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

REFERENCES

[1] K. Ishizaka, M. Obata, and H. Kasahara, “Cache optimization for coarse grain task parallel processing using inter-array padding,” in International Workshop on Languages and Compilers for Parallel Computing, Springer, 2003, pp. 64–76.

[2] 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, Nov. 2007.

[3] C. Hong, W. Bao, A. Cohen, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, “Effective padding of multidimensional arrays to avoid cache conflict misses,” in Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’16. ACM, 2016, pp. 129–144.
M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: A new model-based method and algorithms for performance and energy optimization of data-parallel applications on heterogeneous multicore clusters,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 4, pp. 1119–1133, 2017.

H. Khaleghzadeh, R. Reddy, 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, 2018.

R. Reddy and A. L. Lastovetsky, “HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter,” 2016. [Online]. Available: [http://git.ucd.ie/hcl/hclwattsup]

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, 06/2011.

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:1109.5074, 2011.

A. Averbuch and E. Gabber, “Portable parallel FFT for SIMD multiprocessors,” Concurrency: Practice and Experience, vol. 10, no. 8, 1998.

P. Dmitruk, L.-P. Wang, W. Mathaues, R. Zhang, and D. Seckel, “Scalable parallel FFT for spectral simulations on a Beowulf cluster,” Parallel Computing, vol. 27, no. 14, 2001.

Y. Chen, X. Cui, and H. Mei, “Large-scale FFT on GPU clusters,” in Proceedings of the 24th ACM International Conference on Supercomputing, ser. ICS '11. ACM, 2010.

L. Gu, J. Siegel, and X. Li, “Using GPUs to compute large out-of-card FFTs,” in Proceedings of the International Conference on Supercomputing, ser. ICS 11. ACM, 2011.

J. Wu and J. JaJa, “Optimized FFT computations on heterogeneous platforms with application to the Poisson equation,” Journal of Parallel and Distributed Computing, vol. 74, no. 8, 2014.

V. H. Naik and C. S. Kusur, “Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment,” in Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on. IEEE, 2015, pp. 1–5.

FFTW, “Fastest fourier transform in the west,” 2018. [Online]. Available: [http://www.fftw.org]

FFTW, “Parallel FFTW,” 2018. [Online]. Available: [http://www.fftw.org/doc/fftw-3.html]

D. Pekurovsky, “P3DFFT: A framework for parallel computations of fourier transforms in three dimensions,” SIAM Journal on Scientific Computing, vol. 34, no. 4, pp. C192–C209, 2012.

N. Li and S. Laizet, “2DECOMP and FFT - a highly scalable 2D decomposition library and FFT interface,” in Cray User Group 2010 conference, 2010, pp. 1–13.

T. V. T. Duy and T. Ozaki, “A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs,” Computer Physics Communications, vol. 185, no. 1, pp. 153 – 164, 2014.

I. Corporation, “Intel MKL FFT - fast fourier transforms,” 2018. [Online]. Available: [https://software.intel.com/en-us/mkl/features/fft]

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.

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.

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, vol. 42, no. 2. ACM, 2008, pp. 287–296.

G. Quintana-Ottli, F. D. Igual, E. S. Quintana-Ottli, and R. A. Van de Geijn, “Solving dense linear systems on platforms with multiple hardware accelerators,” in ACM Sigplan Notices, vol. 44, no. 4. ACM, 2009, pp. 121–130.
```c
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, \text{block size}); p++) {
        for (q = 0; q < \min(n-j, \text{block size}); q++) {
            double tmpr = X1[\text{p+n+q}][0];
            double tmpi = X1[\text{p+n+q}][1];
            X1[\text{p+n+q}][0] = X2[\text{q+n+p}][0];
            X1[\text{p+n+q}][1] = X2[\text{q+n+p}][1];
            X2[\text{q+n+p}][0] = tmpr;
            X2[\text{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[\text{start + i*n + j}],
                &X[\text{start + j*n + i}],
                i, j, n, block_size);
        }
    }
}
```

Fig. 27. Transpose of square matrix of size \( n \times n \) using blocking.