Designing and Auto-Tuning Parallel 3-D FFT for Computation-Communication Overlap

Sukhyun Song
Department of Computer Science
University of Maryland, College Park
shsong@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland, College Park
hollings@cs.umd.edu

Abstract

This paper presents a method to design and auto-tune a new parallel 3-D FFT code using the non-blocking MPI all-to-all operation. We achieve high performance by optimizing computation-communication overlap. Our code performs fully asynchronous communication without any support from special hardware. We also improve cache performance through loop tiling. To cope with the complex tradeoff regarding our optimization techniques, we parameterize our code and auto-tune the parameters efficiently in a large parameter space. Experimental results from two systems confirm that our code achieves a speedup of up to 1.76× over the FFTW library.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–Distributed Programming, Parallel Programming

Keywords Performance; 3-D FFT; MPI; Overlap; Non-Blocking; All-to-All; Auto-Tuning

1. Introduction

The Fast Fourier Transform (FFT) is widely used in many fields of science and engineering. Uses include signal processing, image processing, and differential equation solving. More specifically, high-performance scientific applications have recently used three-dimensional FFT (3-D FFT) to run astrophysical N-body simulations [21] and blood flow simulations [25].

There have been many research efforts to achieve high performance of 3-D FFT in distributed-memory parallel systems. Most of the approaches require all-to-all communication among parallel computing processes. For example, in the FFTW [2, 17] and P3DFFT [7, 24] libraries, each process exchanges a 3-D array with other processes using `MPI_Alltoall`, then performs local 1-D FFT computations on a newly assigned array. Researchers tried to increase the parallel 3-D FFT performance through computation-communication overlap. One example is Bell et. al's UPC code [9], which divides an input array into multiple small blocks and overlaps the computation on each block with the communication for other blocks. In this way, they could hide communication latency behind computation.

As the MPI library is widely and successfully used in the parallel computing community, it is important to design a parallel 3-D FFT code based on MPI and achieve portability as well as high performance. However, none of the prior overlap approaches effectively used the non-blocking `MPI_Ialltoall` operation described in the MPI-3.0 standard [16]. Kandalla et. al [22] implemented their own `MPI_Ialltoall` and used it to overlap computation and communication between multiple independent input arrays. Kandalla et. al's approach is not effective in many scientific applications because scientific simulations [21, 25] normally perform successive 3-D FFT operations on a single array. Hoefler et. al [18] also followed the MPI-3.0 standard, and successfully overlapped computation and communication on a single 3-D FFT operation. However, Hoefler et. al's implementation does not optimize computation-communication overlap.

In this paper, we describe a new parallel 3-D FFT code with `MPI_Ialltoall` that overlaps computation and communication to achieve high performance. Our design approach is similar to prior work by Bell et. al [9] and Hoefler et. al [18]. We divide an input array into multiple small blocks to overlap computation on one block with communication on other blocks. However, there are several unique characteristics of our design. First, we *optimize computation-communication overlap*. We hide communication behind as much computation as possible. Second, we design a *portable* code. We use `MPI_Test` for fully asynchronous communication [19] rather than rely on special

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555249

hardware support or separate threads. Third, we also *optimize local computation*. We improve cache performance through loop tiling. Last, we *parameterize* our parallel 3-D FFT code. We can adjust the parameters and potentially cope with the complex trade-off regarding our optimization techniques.

To achieve the best performance of 3-D FFT, we *utilize auto-tuning*. Our new parallel 3-D FFT code contains many tunable parameters that can affect the performance. It is not feasible to hand-tune those parameters because the parameter space is very large (billions of possible configurations). Also, the optimal configuration may vary widely depending on system environment such as hardware, OS, and compiler. Thus, it will not work to tune in one system and reuse the optimal configuration for another. This paper also presents a method to find a good parameter configuration efficiently in the large parameter space. We integrate our parallel 3-D FFT code with the Active Harmony auto-tuning framework [28]. Our new contribution for auto-tuning is to introduce several techniques to help Active Harmony tune our FFT code fast and effectively.

The rest of the paper is organized as follows. We first provide background information in Section 2. Sections 3 and 4 describe how we design and auto-tune our parallel 3-D FFT code. Section 5 evaluates our approach with experiments. We describe the related work in Section 6. Finally, we conclude and discuss future work in Section 7.

2. Background and Assumptions

This section first reviews FFT computation. We then introduce a general method for parallel 3-D FFT. Last, we describe several basic assumptions underlying our design.

2.1 FFT

In mathematics, the Discrete Fourier Transform (DFT) converts a finite list of samples of a function into the list of coefficients of a finite combination of complex sinusoids. In other words, DFT converts a sampled function from the original time domain to the frequency domain. With an input array X of N complex numbers, a one-dimensional DFT produces an output array Y of N complex numbers. When $\omega_N = e^{-\frac{2\pi}{N}i}$, $Y[k]$ for $k = 0, 1, ..., N-1$ is defined as follows:

$$Y[k] = \sum_{j=0}^{N-1} X[j]{\omega_N}^{jk} \quad (1)$$

The Fast Fourier Transform (FFT) refers to an efficient algorithm to compute DFT. The Cooley-Tukey algorithm [10] is the most common FFT algorithm which takes $O(N \log N)$ for 1-D DFT on N complex numbers. FFTW [17] is a widely used software library that computes DFT using various FFT algorithms including the Cooley-Tukey. For the rest of this paper, we will use the term FFT to refer to the DFT computation as well as the algorithm for DFT.

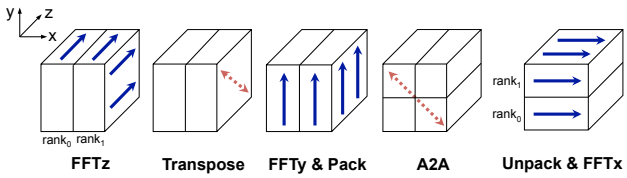


Figure 1. 1-D Decomposition Method for Parallel 3-D FFT

The d-dimensional FFT can be computed simply as the composition of a sequence of d sets of 1-D FFTs along each dimension. For example, 3-D FFT for N^3 complex numbers can be computed by three sets of N^2 1-D FFTs along each dimension.

2.2 1-D Decomposition Method for Parallel 3-D FFT

Our design for a parallel 3-D FFT that will be described in Section 3 basically follows the *1-D domain decomposition* method. The method is used by many parallel 3-D FFT codes [2, 9, 18]. We now describe the overall procedure of the 1-D domain decomposition method by defining seven steps in the procedure. Then in Section 3, we will present how we improve the procedure through computation-communication overlap.

Figure 1 shows the overall procedure of parallel 3-D FFT with 1-D domain decomposition. We parallelize 3-D FFT with p processes rank_0, rank_1, ..., $\text{rank}_{(p-1)}$, and Figure 1 shows an example of $p = 2$. An input 3-D array is divided equally into p arrays along the x dimension and assigned to each process. Then each process executes the following steps on each divided 3-D array:

1. **FFTz**: Compute 1-D FFTs along the z dimension. (Assume that elements on the z dimension are adjacent in memory.)
2. **Transpose**: Change the memory layout for the next step, so that elements on the y dimension are adjacent in memory.
3. **FFTy**: Compute 1-D FFTs along the y dimension.
4. **Pack**: Pack the 3-D array data into a buffer in preparation for all-to-all communication.
5. **A2A**: Perform blocking all-to-all communication among all p processes.
6. **Unpack**: Unpack the received data into the 3-D array with the new memory layout such that elements on the x dimension are adjacent in memory. After this step, the entire input 3-D array is divided into p arrays along the y dimension.
7. **FFTx**: Compute 1-D FFTs along the x dimension.

An alternative way to compute a parallel 3-D FFT is to use a 2-D domain decomposition. The 2-D domain decomposition method decomposes an input array along two

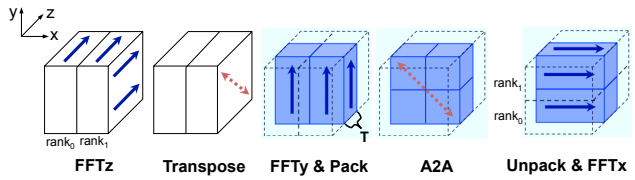


Figure 2. Our Approach for Parallel 3-D FFT

dimensions rather than one dimension. Accordingly, the method is more scalable than 1-D domain decomposition since we can use up to N^2 parallel computing processes for a 3-D FFT operation on N^3 complex numbers. However, 2-D decomposition requires two steps of all-to-all communication, which involves a highly complex communication pattern. So, depending on the system environment, 1-D decomposition can be a better choice than 2-D decomposition. In this paper, we focus on 1-D decomposition and compare the performance with other approaches that use a 1-D decomposition. We consider it as a future work to extend our method with the 2-D decomposition method.

2.3 Assumptions

First, this paper focuses on the *forward* transform that transforms X into Y in Equation 1. Our approach can be easily applied to transform Y backward into X.

Second, we only describe the *complex-to-complex* transform that takes an input array of complex numbers and produces an output array of complex numbers. There are special techniques [26] that can transform real numbers to complex numbers faster than the complex-to-complex transform. Our methods for computation-communication overlap is also applicable to the techniques for the real-to-complex transform.

Third, for simplicity, this paper only presents results for the case of $N_x \bmod p = 0$ and $N_y \bmod p = 0$, where an input 3-D array has N_x and N_y elements on the x and y dimensions, respectively, and p is the number of parallel processes. Note that our current code handles the general case whether N_x and N_y are divisible by p or not.

Last, we focus on the *in-place* transform and want the output to overwrite the input array. Our approach can be applied directly for the out-of-place transform where the output is written to a separate output array.

3. New Design of Parallel 3-D FFT

We improve the 1-D decomposition method by overlapping computation and communication. Our overlap strategy is similar to prior work by Bell et. al [9] and Hoefler et. al [18]. Each process divides an input array into small blocks and overlaps computation on one block with communication for other blocks. However, there are several unique contributions in our design. First, we overlap communication with as much computation as possible. For example, while Hoefler et. al's implementation [18] only overlaps the FFTy and Pack steps with A2A, we also make progress for A2A during Unpack and FFTx. Second, we design a portable code that requires no support of hardware or separate threads for asynchronous communication progression [19]. Third, we also optimize local computation by improving cache performance through loop tiling [29]. Last, our code contains ten tunable parameters so that we can cope with the complex trade-off regarding our optimization techniques.

3.1 Overall Procedure

Figure 2 shows the overall procedure of our parallel 3-D FFT. Algorithms 1-3 describe pseudocodes for each process. Assume that an entire input 3-D array has N_x, N_y, and N_z elements (complex numbers) on the x, y, and z dimensions, respectively. So each of p processes is assigned a partial 3-D array of $\frac{N_x}{p} \times N_y \times N_z$ elements. The memory layout for the divided 3-D array in each process starts with x-y-z in the row-major order. So the elements on the z dimension are adjacent in memory.

Following the original 1-D decomposition method, Algorithm 1 first computes 1-D FFTs along the z dimension (FFTz) and rearranges the memory layout to z-x-y (Transpose). To achieve high performance for the FFTz step, we utilize the highly optimized code for 1-D FFT from the FFTW library. For the Transpose step, the FFTW *guru* interface is used to execute a high-performance routine of memory rearrangement.

We then continue to the next steps to fulfill computation-communication overlap. Each process first divides the input 3-D array into multiple small blocks along the z dimension as shown in Figure 2. We call each divided block a *communication tile*. We define a tile size parameter $\boldsymbol{T}$ to handle the trade-off between the overlap efficacy and the messaging efficiency. With small T, we can overlap many computations with communication but there would be a large overhead of exchanging many small-sized messages. If T gets bigger, there will be less overlap but higher efficiency for communication. Thus we should find a good value of T to achieve high performance. Section 4 will discuss the method to autotune T and other parameters of our 3-D FFT code. Each tile contains T elements on the z dimension. So the number of elements in each communication tile is equal to $\frac{N_x}{p} \times N_y \times T$.

We also define a window size parameter $\boldsymbol{W}$ to specify the degree of communication parallelism. It is also important to adjust W properly to utilize as many concurrent communication connections as possible.

For each communication tile, Algorithm 1 repeats the FFTy, Pack, A2A, Unpack, and FFTx steps. Note that, unlike the original 1-D decomposition method, Algorithm 1 uses the non-blocking MPI all-to-all operation (`MPI_Ialltoall` and `MPI_Wait`) for the A2A step. Thus we can overlap computation (FFTy, Pack, Unpack, and FFTx) on one communication tile with communication (A2A) for other tiles. Algorithm 2 describes a pseudocode for the FFTy and Pack

Algorithm 1: Parallel 3-D FFT on Each Process

FFTz: 1-D FFTs along the z dimension
Transpose: Change the memory layout from x-y-z to z-x-y
Divide an input array into $k = \lceil N_z/T \rceil$ tiles of size T along the z dimension
for $i \leftarrow 0$ **to** $k + W - 1$ **do**
    if $i < k$ **then** FFTy and Pack on tile i
    if $i \geq W$ **then** `MPI_Wait` on tile $(i - W)$
    if $i < k$ **then** `MPI_Ialltoall` on tile i
    if $i \geq W$ **then** Unpack and FFTx on tile $(i - W)$

Algorithm 2: FFTy and Pack on Tile i

Divide tile i into sub-tiles of size $P_x \times N_y \times P_z$ along the x and z dimensions
foreach *sub-tile* **do**
    foreach *1-D array along the y dimension* **do**
        FFTy: Compute 1-D FFT
        Call `MPI_Test` on W previous tiles F_y times in total during this algorithm for tile i
    Pack: Pack the current sub-tile into a buffer
    Call `MPI_Test` on W previous tiles F_p times in total during this algorithm for tile i

Algorithm 3: Unpack and FFTx on Tile i

Divide tile i into sub-tiles of size $N_x \times U_y \times U_z$ along the y and z dimensions
foreach *sub-tile* **do**
    Unpack: Unpack the current sub-tile from a buffer into the input array with the z-y-x memory layout
    Call `MPI_Test` on W next tiles F_u times in total during this algorithm for tile i
    foreach *1-D array along the x dimension* **do**
        FFTx: Compute 1-D FFT
        Call `MPI_Test` on W next tiles F_x times in total during this algorithm for tile i

steps on one communication tile, and Algorithm 3 describes Unpack and FFTx. Details of Algorithms 2 and 3 will be described in Section 3.4. After the Unpack step, the data in each communication tile are rearranged in memory to the z-y-x order, so that we can execute the FFTx step on 1-D arrays along the x dimension. Like the FFTz step, we rely on FFTW's optimized code of 1-D FFT for the FFTy and FFTx steps.

3.2 Computation-Communication Overlap

Figure 3 shows how Algorithm 1 overlaps computation and communication between tiles over time. W is set to be two as an example in the figure. The long dashed arrow represents a single control flow of each process during the `for` loop in Algorithm 1. Note that there are at most W tiles with active communication being executed. While the process is working on tile i for the FFTy and Pack steps, the all-to-all communication (A2A) for two previous tiles $(i - 2)$ and $(i-1)$ takes place in the background. Also, when the process is working on tile i for Unpack and FFTx, the communication for two next tiles $(i + 1)$ and $(i + 2)$ goes on in the background. Likewise, during A2A on tile i, the previous two tiles are computed for Unpack and FFTx, and the next two tiles are computed for FFTy and Pack. A key characteristic of our design is that we optimize the performance by having all the computation steps (FFTy, Pack, Unpack and FFTx) overlapped with communication (A2A).

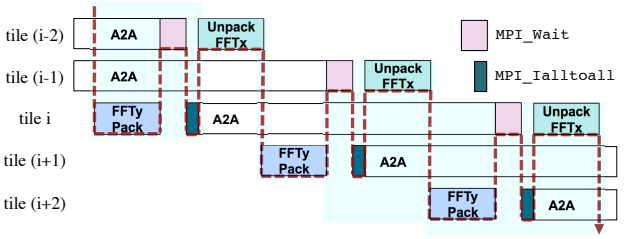


Figure 3. Computation-Communication Overlap between Communication Tiles over Time

3.3 Asynchronous Message Progression

In order to optimize the computation-communication overlap, *fully asynchronous communication* [19] is required for the A2A step. In other words, the non-blocking all-to-all communication should make progress while computation takes place. The first possible approach to ensure asynchronous message progression is to offload protocol processing to the communication hardware such as a programmable network interface card. Alternatively, we could maintain a thread on a separate CPU, so the thread can make progress for the all-to-all communication in the background. A third option is *manual progression*, which is to call `MPI_Test` periodically during the computation and let the MPI library make progress for the corresponding non-blocking all-to-all operation. We choose the manual progression method due to its greater portability. Accordingly, our code does not require any hardware support or separate threads for message progression.

It is important to determine the proper frequency of `MPI_Test` calls. Too high of frequency will incur unnecessary function call overhead, and too low of frequency will limit the progress of the all-to-all communication. To cope with the trade-off, we have four tunable parameters to adjust the frequency of `MPI_Test` calls. $\boldsymbol{F_y}$ defines the number of

`MPI_Test` calls during FFTy for one communication tile. $\boldsymbol{F_p}$ is the frequency parameter during the Pack step for one communication tile. Likewise, $\boldsymbol{F_u}$ is for Unpack, and $\boldsymbol{F_x}$ is for FFTx.

3.4 Loop Tiling for Pack and Unpack

Algorithm 2 describes the FFTy and Pack steps on each communication tile. To optimize the FFTy and Pack steps with respect to cache reuse, we tile the loop inside each communication tile. As shown in the left side of Figure 4, each communication tile is divided again into *sub-tiles* with $\boldsymbol{P_x}$ elements on the x dimension and $\boldsymbol{P_z}$ on the z dimension. So each sub-tile contains $P_x \times N_y \times P_z$ elements. Iterating over each sub-tile, we execute a 1-D FFT computation along the y dimension, then pack the result into a communication buffer. In this way, we can increase cache hit rate during Pack by reading the sub-tile information from the cache after FFTy. Note that, as described in Section 3.3, Algorithm 2 calls `MPI_Test` $(F_y + F_p)$ times for asynchronous message progression.

Similarly to FFTy and Pack, we optimize Unpack and FFTx by using the loop tiling technique shown in Algorithm 3. Each communication tile is divided into sub-tiles along the y and z dimensions as shown in the right side of Figure 4. The size of each sub-tile is determined by two parameters $\boldsymbol{U_y}$ and $\boldsymbol{U_z}$. Thus, we can increase cache hits during FFTx by reading the sub-tile information from the cache after Unpack. Algorithm 3 also calls `MPI_Test` $(F_u + F_x)$ times for asynchronous message progression.

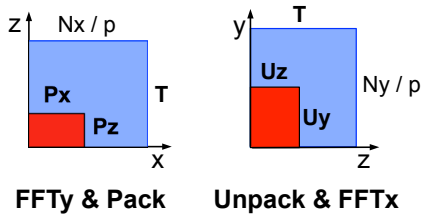


Figure 4. Loop Tiling for Improving Cache Performance

3.5 Improvement for the $N_x = N_y$ Case

As described in Section 3.1, the Transpose step changes the memory layout from x-y-z to z-x-y in preparation for FFTy. For the special case of an input array with $N_x = N_y$, we can improve the performance of Transpose by producing the new memory layout of x-z-y instead of z-x-y. The x-z-y rearrangement should be faster than the original z-x-y rearrangement as the former is simpler and has better cache reuse than the latter. Nonetheless, we cannot use the fast x-z-y rearrangement when $N_x \neq N_y$ because we use an in-place 3-D FFT and want the output to overwrite the input array. Suppose that Transpose produces the x-z-y layout for an input array with $N_x \neq N_y$ in Figure 2. Then it is impossible to match the memory area of a communication tile before A2A with the memory area after A2A because we divide the input array into communication tiles along the z dimension. On the other hand, for a special input with $N_x = N_y$, we can have the same memory area for A2A source and destination data in each communication tile, by generating the y-z-x layout (as opposed to z-y-x) after A2A and Unpack. Thus, for the case of $N_x = N_y$, Transpose rearranges the memory layout from x-y-z to x-z-y, so that we can improve the overall 3-D FFT performance.

4. Auto-Tuning Method

This section describes how we achieve high performance of parallel 3-D FFT by utilizing auto-tuning techniques. Our parallel 3-D FFT code contains ten parameters as described in Section 3. Table 1 summarizes these parameters. To the best of our knowledge, our work is the first that auto-tunes a complex parameter space to optimize computation-communication overlap in parallel 3-D FFT. We first describe how we optimize the code sections that are performed by FFTW. We then justify why we should auto-tune the ten parameters in Table 1. We also introduce an auto-tuning software framework that we use to tune our code, and describe the general tuning procedure. Last, we present our main contribution for auto-tuning. We describe several novel techniques to auto-tune the 3-D FFT code fast and effectively.

4.1 Tuning 1-D FFTs and Transpose

As described in Section 3.1, we rely on the FFTW library for all 1-D FFT computations and the Transpose step. Before we auto-tune the ten parameters in Table 1, we first optimize the FFTW code performance through the auto-tuning feature of FFTW. We choose to use the `FFTW_PATIENT` option for FFTW tuning among three options. Since we want to achieve the best performance of parallel 3-D FFT, we do not use the `FFTW_MEASURE` option. `FFTW_MEASURE` tunes the FFTW library slightly less than `FFTW_PATIENT`. We do not run `FFTW_EXHAUSTIVE` because it takes too much time to auto-tune the FFTW library. With a few empirical tests, we found the code tuned with `FFTW_PATIENT` had the similar performance to `FFTW_EXHAUSTIVE`. Further details about the auto-tuning feature of FFTW can be found in [2, 17].

4.2 Why Should We Auto-Tune the Ten Parameters?

After the FFTW tuning is finished, we continue to auto-tune the parameters defined by our parallel 3-D FFT code. Figure 5 gives evidence that the ten parameters in Table 1 really affect the code's performance, so it is necessary to tune the parameters to achieve the high performance. To gauge the impact of those parameters, we measured the execution time of our 3-D FFT code for 200 random parameter configurations using 16 processes and an array with 256^3 elements. We exclude the FFTz and Transpose steps as those steps have the fixed performance regardless of parameter values. Figure 5 shows the cumulative distribution of the execution time. The x-axis represents the execution time, and

Table 1. Ten Tunable Parameters of Our Parallel 3-D FFT Code

parameter	meaning
T	the number of elements on the z dimension in one communication tile (tile size)
W	the maximum number of communication tiles involved in concurrent all-to-all communication (window size)
P_x	the number of elements on the x dimension in one sub-tile during Pack
P_z	the number of elements on the z dimension in one sub-tile during Pack
U_y	the number of elements on the y dimension in one sub-tile during Unpack
U_z	the number of elements on the z dimension in one sub-tile during Unpack
F_y	the number of `MPI_Test` calls during FFTy for one communication tile
F_p	the number of `MPI_Test` calls during Pack for one communication tile
F_u	the number of `MPI_Test` calls during Unpack for one communication tile
F_x	the number of `MPI_Test` calls during FFTx for one communication tile

the y-axis shows the cumulative fraction of 200 parameter configurations. The code performance widely varies around from 0.16 to 0.48 second (nearly $3\times$) depending on parameter configurations.

We also claim that since the parameter space is very large, we need to auto-tune the parameters rather than adjusting them manually. It is hard to define the size of the parameter space by a single number because the range of possible parameter values is dependent on various factors such as the input array size, the number of processes, and other parameter values. So we can consider a conservative case where each parameter only has ten possible values. In spite of the conservative calculation, we have a large number (10^{10}) of possible parameter configurations. Since it is not feasible to investigate all the configurations manually and exhaustively, we must find a smart and fast way to auto-tune the parameters and determine a good configuration.

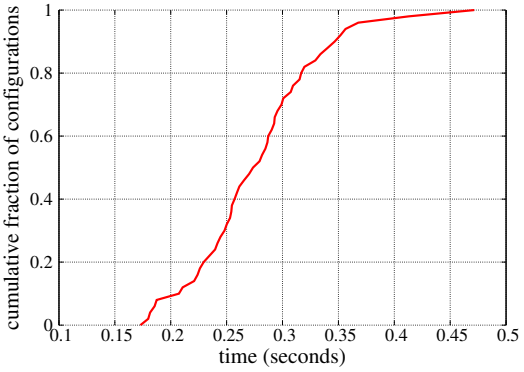


Figure 5. Cumulative Distribution of the 3-D FFT Execution Time for 200 Random Configurations (16 cores and 256^3 elements)

4.3 Active Harmony: An Auto-Tuning Framework

Active Harmony (AH) [11, 28] is a general software framework to auto-tune user-specified parameters for a tunable code. Figure 6 shows the overall procedure of how AH interacts with a tuning target (our parallel 3-D FFT code designed in Section 3). The AH server searches efficiently through a large parameter space and decides a parameter configuration to be tested on the tuning target. The AH client receives a parameter configuration from the server, executes the tuning target with the received configuration, and reports the performance back to the server. This procedure is repeated until the search converges.

Although AH supports several different search strategies, we use the Nelder-Mead (NM) method [23] to search for a good parameter configuration as it is a commonly used optimization technique in many fields of science. NM uses the concept of a *simplex*, which is a polytope of $(d+1)$ vertices in d dimensions. For example, a simplex is a triangle in two dimensions. The AH client measures the performance of the tuning target at each point (or parameter configuration) on a simplex. NM generates a new test point by extrapolating the performance values measured at points on the simplex. Then NM replaces one of the simplex points with the new point. For example, for $(d+1)$ simplex points, NM can sometimes replace the worst simplex point W with a new point R such that R is reflected from W through the centroid of the remaining d points. The search procedure finishes when all the points on a simplex are close to each other, and can be considered to be converged to a single point. Further details about NM are described in [23].

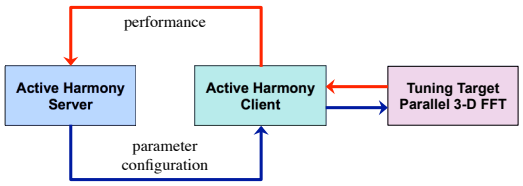


Figure 6. Auto-Tuning Procedure

4.4 Techniques for Effective and Fast Auto-Tuning

We now introduce several techniques to auto-tune the ten parameters in Table 1 effectively and fast. First, we *penalize an infeasible configuration*. The possible values of our parameters reside in a range that can be limited by other parameters. For example, the tile size T must be ≥ 1 and $\leq N_z$,

and the sub-tile size P_z must be ≥ 1 and $\leq T$. However, the Nelder-Mead method of AH is originally designed to work in a multi-dimensional orthotope (hyperrectangle) parameter space. [1] So it is possible that the AH server provides a test configuration that contains *out-of-range* values, for example, P_z that is $> T$. To cope with an infeasible parameter configuration, we modify the AH client in the following way. When the AH client receives an infeasible configuration, it reports the worst performance value (infinity) immediately back to the AH server without executing the tuning target code. Then the AH server and the NM strategy will suggest another configuration that might be in a feasible area in the parameter space.

The second technique is to *reuse the prior performance data for fast tuning*. NM was originally designed to optimize a continuous function. To support the discrete integer domain of parameters, the AH server determines the closest integer point to a simplex point in a continuous domain. So the AH server can sometimes provide the same configuration even though it has been already tested before. To save tuning time, we maintain the history of tested configurations and utilize it when the AH client receives the previously tested configuration again.

The third technique is also about improving the auto-tuning speed. In the auto-tuning procedure, we can *skip executing the code section that is independent of the ten parameters*. Since the performance of the FFTz and Transpose steps is fixed regardless of a parameter configuration, the AH client does not execute FFTz and Transpose during the auto-tuning procedure.

Fourth is *search space reduction*. Instead of searching a whole set of all possible values of a parameter, we reduce a search space to a log scale and consider power-of-two values for testing. The minimum and maximum values are additionally considered for testing whether the value is a power of two or not. So, we can also take into account the boundary values in the original parameter space after reducing the search space. For example, when $N_z = 24$, T can be 1, 2, 4, 8, 16, or 24. As an exception, the log-scale reduction is not applied to W because there are few possible values for W.

Finally, we carefully construct the initial simplex that the NM uses. The initial simplex can affect the tuning time and the quality of the tuning result. Thus it is important to guess a good initial simplex so that NM can find the global minimum point in a short time without falling into local minimum points. In this paper we determine an initial simplex in the following way, and demonstrate its performance with experiments in Section 5. We need to investigate further how to determine a good initial simplex and leave it as an open question. We construct an initial simplex by first defining a default point and determining the other ten points around the default point. We define the default point as follows. First, we set $T = N_z/16$ to guarantee some degree of computation-communication overlap. $W = 2$ is set to exploit some level of communication parallelism. Assuming that a cache size is equal to 256KB, we can fit 16K complex number elements in a cache. Assuming we use the cache to read/write a sub-tile for data-packing, we can have 8K complex numbers as a sub-tile size for Algorithm 2. Thus, we set $P_x = 8192/N_y$ and $P_z = 8192/N_y/P_x$. Similarly, it is set to be $U_y = 8192/N_x$ and $U_z = 8192/N_x/U_y$. We set $F_y = F_p = F_u = F_z = p/2$ where p is equal to the number of processes as `MPI_Ialltoall` requires more rounds of point-to-point communication as p increases.

[1] The developers of Active Harmony are currently implementing a constraint plugin that supports a non-hyperrectangle parameter space, but it was not ready when this paper was written.

5. Evaluation

We first show how fast our auto-tuned 3-D FFT is compared to two other approaches. Then we analyze where the improvement of our approach comes from. Last, we quantify why it is necessary to use an auto-tuning method for our 3-D FFT code.

5.1 Platforms and Comparison Models

We use two platforms for our experiments. The first platform, which is named **UMD-Cluster**, is a 64-node Linux cluster at the University of Maryland. Each node consists of two Intel Xeon 2.66GHz (SSE) cores. Each core has a 512KB L2 cache. We used one core per node in our experiments. A Myrinet 2000 interconnect is used to connect nodes. For a non-blocking all-to-all operation on UMD-Cluster, we use `NBC_Ialltoall` of the NBC library 1.1.1 [3, 20] on top of OpenMPI 1.4.1 [6]. The NBC library is designed compatible to the MPI-3.0 standard. FFTW 3.3.2 is used for 1-D FFT. We used `mpicxx` of OpenMPI 1.4.1 with the `-O2` option to compile all the libraries and codes.

The second platform, which is named **Hopper**, a Cray XE6 machine at NERSC [5]. Each node contains two twelve-core AMD MagnyCours 2.1GHz processors (153,216 cores total in the machine). Each core has its own L1 and L2 caches, with 64KB and 512KB, respectively. We used four cores per processor (eight cores per node) in our experiments. Nodes are connected via a Cray Gemini Network that forms a 3-D torus. For a non-blocking all-to-all operation on Hopper, we use `MPIX_Ialltoall` of the Cray Message Passing Toolkit 5.6.0 that is derived from the MPICH [4] implementation. FFTW 3.3.0.1 is used for 1-D FFT. We used the PGI C++ compiler with `-fast` option to compile all the codes except the underlying libraries. The FFTW and MPI libraries are already compiled and installed in the Hopper system.

We compare three different methods for parallel 3-D FFT. **FFTW** is the MPI-enabled FFTW library. We tune and optimize the parallel 3-D FFT computation of FFTW by using the `FFTW_PATIENT` option. The second approach is **NEW**,

Table 2. Parallel 3-D FFT Time (seconds)

(a) UMD-Cluster

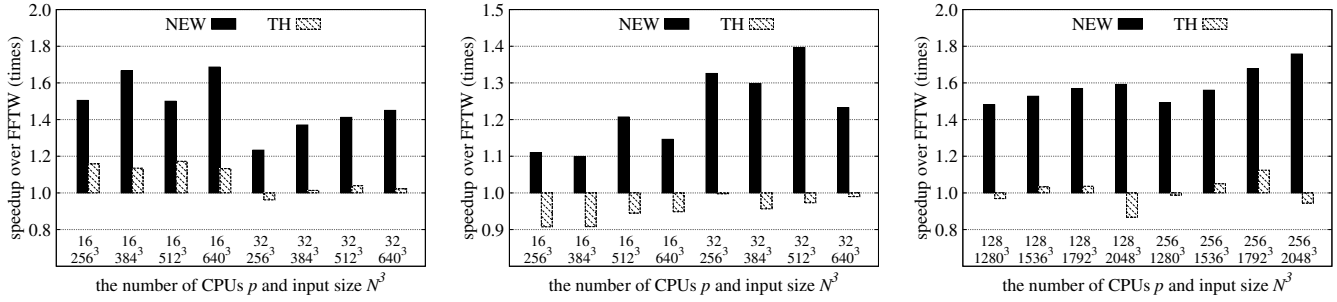
p	N^3	FFTW	NEW	TH
16	256^3	0.369	0.245	0.319
16	384^3	1.207	0.725	1.063
16	512^3	2.948	1.966	2.514
16	640^3	5.927	3.515	5.234
32	256^3	0.189	0.153	0.197
32	384^3	0.653	0.477	0.644
32	512^3	1.580	1.119	1.520
32	640^3	3.129	2.158	3.061

(b) Hopper

p	N^3	FFTW	NEW	TH
16	256^3	0.096	0.087	0.106
16	384^3	0.322	0.293	0.354
16	512^3	0.836	0.693	0.885
16	640^3	1.636	1.428	1.725
32	256^3	0.061	0.046	0.061
32	384^3	0.189	0.146	0.198
32	512^3	0.475	0.340	0.488
32	640^3	0.920	0.747	0.930

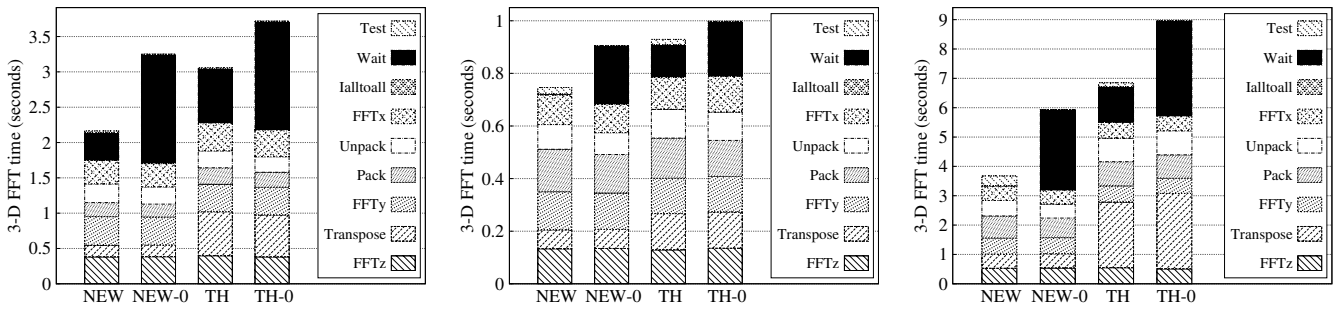
(c) Hopper (large scale)

p	N^3	FFTW	NEW	TH
128	1280^3	2.426	1.638	2.505
128	1536^3	4.722	3.092	4.573
128	1792^3	8.029	5.115	7.746
128	2048^3	11.269	7.079	12.994
256	1280^3	1.373	0.920	1.389
256	1536^3	2.574	1.650	2.452
256	1792^3	4.781	2.850	4.253
256	2048^3	6.467	3.679	6.850



(a) UMD-Cluster (b) Hopper (c) Hopper (large scale)

Figure 7. Parallel 3-D FFT Speedup over FFTW



(a) UMD-Cluster ($p = 32$ and $N^3 = 640^3$) (b) Hopper ($p = 32$ and $N^3 = 640^3$) (c) Hopper ($p = 256$ and $N^3 = 2048^3$)

Figure 8. Performance Breakdown

which is our method described in Section 3. Since NEW relies on the FFTW library for some computations, we auto-tune those computations with the `FFTW_PATIENT` option. The major part of NEW is auto-tuned by the Nelder-Mead strategy of Active Harmony as described in Section 4. Last is **TH**, which is Hoefler et. al's parallel 3-D FFT kernel [18] that also implements computation-communication overlap with the MPI-3.0 standard. For fair comparison, we slightly optimized the original code. We auto-tune 1-D FFT computations with the `FFTW_PATIENT` flag on. Also, we parameterize the code with three parameters of a communication tile size, a window size, and a frequency of `MPI_Test` calls, then auto-tune the three parameters with Active Harmony similarly to NEW. Thus, TH is the combination of a prior work for parallel 3-D FFT and our auto-tuning method described in Section 4. TH shows the higher performance than Hoefler et. al's original code.

5.2 Parallel 3-D FFT Performance

5.2.1 UMD-Cluster

Table 2(a) shows the 3-D FFT execution time of the three different approaches. p is the number of parallel computing processes. N is the number of elements on each dimension in a 3-D input array. So an input array contains N^3 complex numbers. To cope with the execution noise, we conducted five runs of auto-tuning each with five runs of 3-D FFT, and picked the best performance out of the 25 runs for each of the three algorithm being compared. For all different settings of

p and N, NEW is faster than FFTW and TH. Figure 7(a) shows the speedup of NEW and TH over FFTW. NEW has speedup over FFTW of 1.23× to 1.68×. On the other hand, TH, the other overlap approach, shows the maximum speedup of 1.17× compared to FFTW, and there is even a setting such that TH is worse than FFTW.

To better explain the effectiveness of NEW, we break down the performance of NEW and TH for the $p = 32$ and $N^3 = 640^3$ configuration, and show the result in Figure 8(a). Two extra variants of FFT are examined. **NEW-0** is a non-overlapped version of NEW where W and all the frequency parameters are set to be zero with all the other parameters equal to NEW. Also, lines 6 and 7 in Algorithm 1 are replaced with `MPI_Ialltoall` and `MPI_Wait` on tile i. Likewise, **TH-0** is a non-overlapped version of TH. The all-to-all communication time for this setting is around 1.6 seconds as marked with Wait in NEW-0, and the "overlappable" computation time (FFTy, Pack, Unpack, and FFTx) is 1.2 seconds. NEW reduces the Wait time down to 0.4 seconds, which means NEW nearly achieves the perfect computation-communication overlap. This high degree of overlap explains why NEW is faster than FFTW. Since FFTW does not exploit non-blocking communication, the performance should be similar to NEW-0. On the other hand, TH performs a low degree of overlap and results in a long 1.3 seconds for Wait. This is because TH does not overlap the Unpack and FFTx steps with communications while NEW uses all the computation steps for overlap as described in Section 3. Also, we can see that NEW optimizes computation better than TH. First, as NEW utilizes the highly-optimized matrix transpose of FFTW, NEW shows a large improvement for Transpose compared to TH in Figure 8(a). Second, the time NEW spent for Pack and FFTx is shorter than that of TH, which is the result of the loop tiling technique of NEW.

It is interesting that NEW shows better performance for $p = 16$ than $p = 32$ in Figure 7(a). It is not easy to find the exact reason for this because we measured the performance as the speedup over FFTW, and we are not aware of the details of the FFTW behavior. However, assuming the NEW-0 approach should be close to FFTW, we can find a partial reason. For the best overlap, the computation time should be ideally equal to the communication time. We found that, on UMD-Cluster, the computation-communication time is balanced better at $p = 16$ than $p = 32$. The reason for the worse computation-communication balance at $p = 32$ is the high complexity of the all-to-all operation at high p. So, for example, NEW-0 of Figure 7(a) shows the larger communication (Wait) time than the computation time (FFTy, Pack, Unpack, and FFTx).

5.2.2 Hopper

We conducted the same experiment on Hopper as what we did on UMD-Cluster. Table 2(b) shows the 3-D FFT execution time on Hopper. Figure 7(b) shows the speedup of NEW and TH over FFTW. Figure 8(b) shows the performance breakdown for the setting of $p = 32$ and $N^3 = 640^3$. Like on UMD-Cluster, NEW is faster than TH with better optimized overlap and computation. But on Hopper, the speedup of NEW over FFTW ranges from 1.10× to 1.40×, which is lower than the speedup on UMD-Cluster. This low speedup on Hopper comes from the relatively bad computation-communication time balance. For example, NEW-0 in Figure 8(b) shows a lower ratio of the Wait time than in Figure 8(a). This is because the Cray Gemini Network of Hopper is faster than the Myrinet 2000 on UMD-Cluster. Also, the intra-node communication between multiple cores in the same node on Hopper should be faster than the inter-node communication on UMD-Cluster. So the worse computation-communication time balance on Hopper limits the possibility of overlap and results in a relatively low speedup over FFTW. It is interesting that NEW shows worse performance for $p = 16$ than $p = 32$ in Figure 7(a). The reason for this difference on Hopper is the same as that on UMD-Cluster even though the result is opposite on UMD-Cluster. Due to the high complexity of the all-to-all operation, increasing p causes an increase in the ratio of the communication time to the computation time. So, the com-

Table 3. Parameter Values Found via Auto-Tuning

(a) UMD-Cluster

p	N^3	T	W	P_x	P_z	U_y	U_z	F_y	F_p	F_u	F_x
16	256^3	32	3	8	2	16	4	32	8	8	16
16	384^3	16	2	16	1	16	2	16	16	8	16
16	512^3	64	3	16	2	16	2	32	16	32	32
16	640^3	32	3	16	1	16	2	16	16	16	16
32	256^3	64	3	8	8	8	4	64	8	16	64
32	384^3	32	2	12	2	8	2	32	8	8	16
32	512^3	32	2	16	4	16	4	64	8	8	16
32	640^3	32	2	8	1	8	1	16	16	16	16

(b) Hopper

p	N^3	T	W	P_x	P_z	U_y	U_z	F_y	F_p	F_u	F_x
16	256^3	32	3	16	2	8	2	16	16	16	32
16	384^3	32	3	24	1	24	2	16	16	16	16
16	512^3	64	3	32	1	16	2	64	64	64	64
16	640^3	64	3	16	2	16	2	64	32	64	32
32	256^3	64	2	8	4	8	4	64	16	16	64
32	384^3	64	3	12	2	8	2	128	32	64	128
32	512^3	128	3	16	2	8	4	128	64	32	64
32	640^3	64	3	16	2	16	2	64	64	64	64

(c) Hopper (large scale)

p	N^3	T	W	P_x	P_z	U_y	U_z	F_y	F_p	F_u	F_x
128	1280^3	256	4	10	2	8	2	512	128	256	512
128	1536^3	128	3	12	1	8	2	1024	128	128	1024
128	1792^3	128	4	14	1	8	2	256	128	128	512
128	2048^3	128	4	16	1	8	2	512	128	128	512
256	1280^3	256	4	5	4	2	8	1280	64	64	1024
256	1536^3	256	3	6	2	4	2	1024	128	256	1024
256	1792^3	256	3	7	2	4	2	512	128	256	1024
256	2048^3	512	3	8	2	4	2	2048	256	512	2048

munication ratio at $p = 16$ is lower than that at $p = 32$. Since the communication ratio is already low at $p = 32$ because of the fast communication on Hopper, the lower communication ratio at $p = 16$ would results in a worse computation-communication balance.

We did the similar experiments on Hopper for larger scale settings with more cores and bigger input sizes. As seen in Table 2(c), Figure 7(c), and Figure 8(c), the trend is similar to the previous experiments on Hopper. We can see our approach NEW is still faster than FFTW and TH. The speedup of NEW over FFTW ranges from 1.48× to 1.76×.

5.3 Auto-Tuning

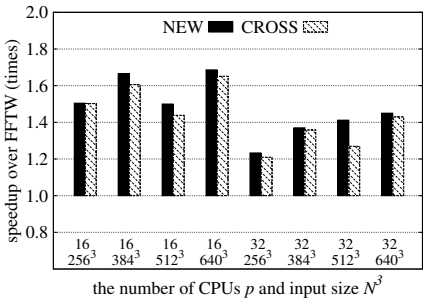
As described in Section 4, the performance of our 3-D FFT widely varies depending on configurations. The varying performance in a large parameter space justifies why we need to auto-tune the 3-D FFT code.

5.3.1 Different Tuning Results on Different Systems

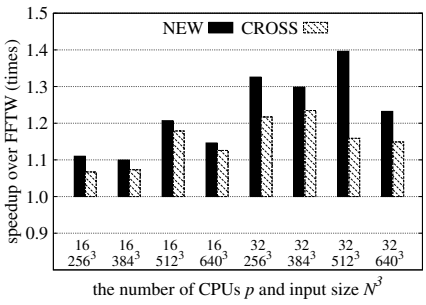
We found that the best parameter values in one system setting differ from those in another setting. Table 3 contains the auto-tuned parameter configurations of NEW that are used to create Table 2. The auto-tuned parameter configuration varies depending on system setting such as the underlying platform, input size, and the number of CPUs. The next question is how good is an auto-tuning result of the Nelder-Mead method, compared to random search. The tuning result for the setting of $p = 16$ and $N^3 = 256^3$ on UMD-Cluster ranks in the first percentile in the distribution of 200 random configurations in Figure 5. Although the Nelder-Mead method did not find the optimal configuration, its deterministic strategy works faster than the random search. For example, the Nelder-Mead method found the first percentile configuration after testing 35 configurations. However, the probability to find the point within 35 random configurations is only $1 - (1 - 0.01)^{35} \approx 30\%$.

5.3.2 Cross-Platform Test

We found that the tuning result from one platform does not work well for another platform. We executed the 3-D FFT code on UMD-Cluster with the tuning result from Hopper as in Table 3(b). The performance of this cross-platform test is named **CROSS** in Figure 9(a). NEW means the performance of the tuning result on UMD-Cluster, which should be the same as NEW in Figure 7(a). For all the settings, NEW is faster than CROSS. Specifically, NEW is around 10% faster than CROSS for $p = 32$ and $N^3 = 512^3$ on UMD-Cluster. Likewise, we have executed the 3-D FFT code on Hopper with the tuning result of the UMD-Cluster as in Table 3(a). The difference between NEW and CROSS in Figure 9(b) is more significant than that in Figure 9(a). NEW is around 20% faster than CROSS on Hopper when $p = 32$ and $N^3 = 512^3$. The best configuration for the UMD-Cluster is not the best on Hopper, and vice versa. Thus, it is necessary



(a) UMD-Cluster



(b) Hopper

Figure 9. Cross-Platform Test

to auto-tune our 3-D FFT code for each different platform to achieve the best of its performance.

5.3.3 Auto-Tuning Time

Table 4 contains the time spent for auto-tuning to achieve the performance in Table 2. Although we focus on the 3-D FFT performance, it is not desirable to spend unacceptably long time on auto-tuning. NEW shows a comparable tuning speed to FFTW as NEW finds a good configuration faster for most of the cases. It takes less time for Active Harmony to tune TH than NEW because TH only has three parameters while NEW has ten. Fewer dimensions mean a small search space, and it is natural to find a good configuration quickly in a small search space.

6. Related Work

This section introduces several studies related to parallel 3-D FFT and describes how our approach is different from those studies.

FFTW [17] is the most popular library for FFT computations, and its MPI-enabled version supports parallel 3-D FFT. FFTW exploits no computation-communication overlap, which results in a relatively poor performance as shown in Section 5. P3DFFT [24], Ayala et. al [8], Takahashi [27], and Eleftheriou et. al [14] achieved high scalability with the high-dimensional domain decomposition technique. However, they did not include any computation-communication overlap. On the other hand, our approach increases the 3-

Table 4. Auto-Tuning Time (seconds)

(a) UMD-Cluster

p	N^3	FFTW	NEW	TH
16	256^3	22.569	16.443	5.732
16	384^3	60.859	27.178	13.279
16	512^3	87.568	123.993	30.916
16	640^3	202.134	197.916	71.724
32	256^3	14.388	11.385	3.768
32	384^3	44.795	28.489	7.834
32	512^3	67.426	45.308	25.124
32	640^3	174.081	73.263	52.897

(b) Hopper

p	N^3	FFTW	NEW	TH
16	256^3	11.413	9.091	2.221
16	384^3	37.786	17.342	17.984
16	512^3	69.912	43.718	27.020
16	640^3	249.358	87.573	22.857
32	256^3	6.614	6.467	1.382
32	384^3	23.317	155.975	10.425
32	512^3	41.969	165.527	6.666
32	640^3	188.474	38.279	15.027

(c) Hopper (large scale)

p	N^3	FFTW	NEW	TH
128	1280^3	461.240	140.986	34.474
128	1536^3	460.229	198.068	60.475
128	1792^3	484.678	335.273	83.986
128	2048^3	562.398	396.553	120.555
256	1280^3	400.582	80.085	17.172
256	1536^3	401.474	109.250	34.568
256	1792^3	414.020	144.743	46.684
256	2048^3	465.411	224.744	75.616

D FFT performance through computation-communication overlap.

Kandalla et. al [22] overlap the computation on one input array with the communication for other input arrays. This "inter-array" overlap is useful when there are many independent input arrays for 3-D FFT. However, scientific simulations [21, 25] normally need successive 3-D FFT computations over time on a single input array. In this case, our "intra-array" method is effective as we optimize computation-communication overlap inside each 3-D FFT operation. Also, Kandalla et. al's approach requires hardware support for asynchronous communication while our approach does not. 2DECOMP&FFT [1] follows Kandalla et. al's overlap method, so it naturally has the limitation of the inter-array overlap. Also, 2DECOMP&FFT is not optimized for asynchronous communication as they do not auto-tune the frequency of `MPI_Test` calls unlike our approach. Bell et. al [9] overlap computation and communication on a single input array. However, unlike our portable MPI-based approach, their code is written in UPC and requires hardware support for asynchronous communication. Also, the overlap may not be optimized because they use a fixed communication tile size. Fang et. al [15] lack portability as they use a specialized API for communication with special hardware support. Also, there is no auto-tuning of overlap-related parameters. Doi et. al [12] utilize multiple threads in a shared-memory parallel environment, and overlap computation and communication between different computing cores. On the other hand, we overlap the computation of one core with the communication of the same core. Hoefler et. al's approach [18] is the closest to our work, but it does not maximize computation-communication overlap, as shown with TH in Section 5.

Dotsenko et. al [13] auto-tuned the 3-D FFT computations on top of special GPU processors while our approach is more generalized based on MPI and CPU and focuses on computation-communication overlap.

7. Conclusions and Future Work

This paper has presented a novel method to optimize parallel 3-D FFT for computation-communication overlap. We first designed a portable parallel 3-D FFT code that uses the non-blocking MPI all-to-all operation and requires no hardware support for asynchronous communication. We then described a method to auto-tune the parameters of our 3-D FFT code in a large parameter space and optimize computation-communication overlap. With extensive experiments on two systems, we showed that our approach for parallel 3-D FFT maximized computation-communication overlap and performed faster than two existing approaches.

We are currently extending this work in several ways. First, we are improving the auto-tuning method. The quality and speed of the Nelder-Mead heuristic is dependent on how an initial simplex is defined. Although our definition of the initial simplex was successful, it is worth investigating if there exist other more effective initial simplex construction techniques. Also, we plan to try optimization strategies other than Nelder-Mead. Second, we intend to apply our overlap method to the 2-D domain decomposition technique. If successful, we could achieve high scalability with many computing cores as well as the high performance with the maximized computation-communication overlap. Finally, we plan to overlap additional computation and communication between multiple independent input arrays. The communication time can dominate the 3-D FFT performance at large scale where ran on many cores. So it would be helpful to overlap more communication time with computation. We are planning to find a method to achieve both intra-array and inter-array computation-communication overlap.

Acknowledgments

Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award numbers ER25763 and ER26054.

We thank Torsten Hoefler for providing us with his parallel 3-D FFT code, which is used to evaluate our approach in Section 5. We are also grateful to Alan Sussman for reviewing this paper and providing helpful comments.

References

[1] 2decomp&fft. http://www.2decomp.org/.

[2] Fastest fourier transform in the west. http://www.fftw.org/.

[3] Libnbc - nonblocking mpi collective operations. http://htor.inf.ethz.ch/research/nbcoll/libnbc/.

[4] Mpich. http://www.mpich.org/.

[5] National energy research scientific computing center. http://www.nersc.gov/.

[6] Open mpi: Open source high performance computing. http://www.open-mpi.org/.

[7] Parallel three-dimensional fast fourier transforms. http://www.sdsc.edu/us/resources/p3dfft/.

[8] O. Ayala and L.-P. Wang. Parallel implementation and scalability analysis of 3d fast fourier transform using 2d domain decomposition. *Parallel Computing*, 39(1), Jan. 2013.

[9] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2006.

[10] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90), 1965.

[11] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Press, 2002.

[12] J. Doi and Y. Negishi. Overlapping methods of all-to-all communication and fft algorithms for torus-connected massively parallel supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2010.

[13] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*. ACM, 2011.

[14] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain. Scalable framework for 3d ffts on the blue gene/l supercomputer: implementation and early performance measurements. *IBM J. Res. Dev.*, 49(2), Mar. 2005.

[15] B. Fang, Y. Deng, and G. J. Martyna. Performance of the 3d fft on the 6d network torus qcdoc parallel supercomputer. *Computer Physics Communications*, 176(8), 2007.

[16] M. P. I. Forum. Mpi: A message-passing interface standard version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[17] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] T. Hoefler, P. Gottschling, and A. Lumsdaine. Brief announcement: Leveraging non-blocking collective communication in high-performance applications. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, 2008.

[19] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER)*, 2008.

[20] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Proceedings of the 2007 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Press, 2007.

[21] T. Ishiyama, K. Nitadori, and J. Makino. 4.45 pflops astrophysical n-body simulation on k computer: the gravitational trillion-body problem. In *Proceedings of the 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM, Nov. 2012.

[22] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda. High-performance and scalable non-blocking all-to-all with collective offload on infiniband clusters: a study with parallel 3d fft. *Computer Science*, 26(3-4), June 2011.

[23] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4), 1965.

[24] D. Pekurovsky. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM Journal on Scientific Computing*, 34(4), Aug. 2012.

[25] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2010.

[26] H. Sorensen, D. Jones, M. Heideman, and C. Burrus. Real-valued fast fourier transform algorithms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(6):849–863, 1987.

[27] D. Takahashi. An implementation of parallel 3-d fft with 2-d decomposition on a massively parallel cluster of multi-core processors. In *Parallel Processing and Applied Mathematics*, volume 6067 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[28] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of the 25th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2011.

[29] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 655–664. ACM Press, 1989.