Optimization of Fast Fourier Transforms on the Blue Gene/L Supercomputer

Yogish Sabharwal* Saurabh K. Garg† Rahul Garg*
John A. Gunnels‡ Ramendra K. Sahoo‡

Abstract

Fast Fourier Transform is a class of efficient algorithms used to compute Discrete Fourier Transforms, widely used in many scientific and technical applications. In this paper, we analyze the bottlenecks in the parallel FFT algorithm and describe optimizations carried out for the algorithm on the Blue Gene/L Supercomputer. There were three avenues for improving the performance of the algorithm – single-node FFT performance, Alltoall collective performance and overlap of computation and communication. Performance at all these levels has been optimized using the double-hummer intrinsics of the Blue Gene/L CPU, careful ordering and synchronization of messages in Alltoall communications and suitable interleaving of message exchanges with computations. Using these optimizations, we obtained a performance of 2875 Gflops for the HPC Challenge FFT benchmark on the 64 racks Blue Gene/L system – which is the best reported FFT performance on any system built so far. We give a brief overview of the Alltoall optimizations, describe our computation-communication overlap strategy and present results for strong scaling and weak scaling of parallel FFT on Blue Gene/L. We also discuss the fundamental limits to scaling of the parallel transpose algorithm for computing FFT.

1 Introduction

The Discrete Fourier Transform (DFT) plays an important role in many scientific and technical applications, including time-series and waveform analysis, solutions to linear partial differential equations, convolution, digital signal processing, and image filtering [17, 14, 16]. Cooley and Tukey designed an algorithm [5] to compute the DFT of an n point series in $O(n \log n)$ operations, significantly improving over previously known methods for computing DFTs. Many algorithms have since been proposed to compute DFTs with similar efficiency. This class of efficient algorithms is generally referred to as fast Fourier transform (FFT) algorithms. One of the most widely used FFT implementations in both academia and the

*IBM India Research Laboratory, Plot-4, Block C, Vasant Kunj Insitutional Area, New Delhi, India ({ysabharwal,grahul}@in.ibm.com)

†Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Australia (sgarg@csse.unimelb.edu.au)

‡IBM T. J. Watson Research Center, 1101 Kitchawan Rd, Rt. 134, Yorktown Heights, NY 10598, USA ({gunnels,rsahoo}@us.ibm.com)

industry is FFTW [8]. It provides excellent performance on a variety of machines – even competitive with or faster than equivalent libraries supplied by vendors.

Many parallel algorithms have been proposed and designed for computation of FFTs on parallel computers (see [18, 21, 1] and references therein). There are two basic approaches to parallelizing FFT algorithms based on the on the underlying interconnection network topology – the binary-exchange algorithms and the transpose based algorithms. The former is suited for systems where the bisection bandwidth of the interconnect scales linearly with the number of nodes, whereas, the latter is more suitable for systems where the bisection bandwidth scales sub-linearly. For more details the reader is referred to [13].

Implementation of the transpose algorithm [13] carries out two basic operations - parallel computation of FFT on individual nodes with (almost) perfectly balanced load and perfectly balanced Alltoall communications where every pair of nodes exchanges the same amount of data. On a system where bisection bandwidth does not scale linearly with the number of processors, the Alltoall communication ends up taking most of the time. Moreover, as the number of processors is increased (while keeping the same problem size), the amount of data exchanged between every pair of nodes decreases quadratically. As a result, the header overheads start dominating in Alltoall communications. This leads to an absolute limit for strong as well as weak scaling of this algorithm.

There were three avenues for performance optimization of the transpose-based parallel FFT algorithm single node performance, performance of Alltoall communication and overlap of communication and communication. The single node performance was optimized using a cache efficient decomposition of FFT (using the Cooley-Tukey algorithm), Blue Gene/L specific dual floating point intrinsics and several hints to the compiler in the C code. The Alltoall performance was optimized by organizing the communication into several short phases such that each node exchanges data with exactly four other nodes in each phase. The phases were then bundled into groups which were separated using an efficient hardware-based barrier. Curiously, inserting a barrier between groups of phases improves performance by as much as 35% on the 64 racks Blue Gene/L system. This is primarily due to congestion avoidance. The computation and communication was overlapped by carefully dividing the data into smaller subsets. This results in hiding (to a large extent) the latency of the faster of the two operations behind the other.

With these optimizations, we obtained a peak performance of 2875 Gflops for the HPC Challenge FFT benchmark on the 64 racks Blue Gene/L system – which is the best reported FFT performance on any system built so far.

In another work [7], the authors studied the performance of parallel 3D-FFT on the Blue Gene/L Supercomputer. They compared the performance of the algorithm using product MPI as well as the underlying lower layer communication primitives directly. This work did not explore the optimization of All2all performance and computation-communication overlap. Moreover, its focus was on parallel 3D-FFT algorithms. Our techniques for optimization of All2all performance, computation-communication overlap can be applied to the parallel 3D-FFT algorithms as well.

The rest of this paper is organized as follows. We discuss fast Fourier transforms and parallel algorithms for FFT in Section 2. This is followed, in Section 3 by an analysis of the bottlenecks for the transpose based parallel FFT algorithms. The Blue Gene/L specific optimizations for single CPU performance, Alltoall communication and for overlap of

computation and communication are described in Section 4. We present results for strong scaling and weak scaling of optimized parallel FFT on Blue Gene/L in Section 5. Finally, we conclude in Section 6.

2 Fast Fourier Transforms

The Discrete Fourier Transform (DFT) for a sequence of n complex numbers $x_0, ..., x_{n-1}$ is another sequence of n complex numbers $y_o, ..., y_{n-1}$, where

$$y_k = \sum_{j=0}^{n-1} x_j \omega^{jk} \; ; \; k = 0, .., n-1 \tag{1}$$

and ω is the primitive nth root of unity in the complex plane, i.e., $\omega = e^{2\pi\sqrt{-1}/n}$.

Cooley and Tukey [5] presented an efficient algorithm that recursively breaks down the computations of a DFT of any composite size into computation of smaller DFTs. More formally, let n be a composite and n_1, n_2 be its factors such that $n = n_1 n_2$. Rewriting the indices in equation 1 as $k = k_1 \cdot n_2 + k_2$ where $0 \leq k_1 < n_1, 0 \leq k_2 < n_2$ and $j = j_2 \cdot n_1 + j_1$ where $0 \leq j_1 < n_1, 0 < j_2 \leq n_2$, we get

$$\begin{aligned} y_k = y_{(n_2 k_1 + k_2)} &= \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x_{(n_1 j_2 + j_1)} \cdot \omega_{n_1 n_2}^{(n_1 j_2 + j_1).(n_2 k_1 + k_2)} \\ &= \sum_{j_1=0}^{n_1-1} \left[\left(\sum_{j_2=0}^{n_2-1} x_{(n_1 j_2 + j_1)} \cdot \omega_{n_2}^{j_2 k_2} \right) \cdot \omega_n^{j_1 k_2} \right] \cdot \omega_{n_1}^{j_1 k_1} \end{aligned} \tag{2}$$

Therefore, the DFT of size n can now be evaluated by first computing n_1 DFTs of size n_2, multiplying by twiddle factors (complex roots of unity) and then computing n_2 DFTs of size n_1. Many algorithms have since been proposed, that achieve similar efficiency; the Prime-Factor (Good-Thomas) algorithm [10, 19] based on the Chinese Remainder Theorem for co-prime factors, the Rader-Brenner algorithm [3], Bruun's algorithm [4], QFT, Winograd's algorithm [20] and Bluestein's algorithms [2]. This class of algorithms are referred to as fast Fourier transform (FFT) algorithms.

The transpose based parallel FFT algorithms are designed for systems with interconnects that do not scale linearly with the number of nodes. These algorithms consider the input as a logical multi-dimensional matrix and break down the DFT computations using Cooley-Tukey algorithm into smaller DFT computations operating along each of the dimensions of the matrix. These algorithms then iterate over the different dimensions of the matrix computing DFTs along those dimensions. In order to ensure that the data required for the DFT computations in the current iteration lie on the same processor, matrix-transpose operations are required along two-dimensional planes of the matrix. Hence, the name transpose FFT algorithms. The simplest form of the transpose FFT algorithm is the two-dimensional FFT algorithm. This is described in Section 2.1. The FFTE algorithm is a variant of the transpose algorithm and is described in Section 2.2.

Algorithm 1: Two-dimensional Transpose FFT Algorithm

1. Distribute columns on each processor - Global Transpose (Alltoall)
2. Compute $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ along the matrix columns
3. Distribute rows on each processor - Global Transpose (Alltoall)
4. Multiply by twiddle factors and Rearrange
5. Compute $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ along the matrix rows
6. Rearrange the output data elements - Global Transpose (Alltoall)

2.1 Two-dimensional Transpose algorithm

The 2D-transpose FFT algorithm considers the input vector as a logical two-dimensional matrix and breaks down the DFT into sub-steps based on the Cooley-Tukey algorithm. The main idea behind the algorithm is to logically express the vector, x, of size N, whose DFT is required to be computed, as a two-dimensional matrix of dimension $n_1 \times n_2$, where $n = n_1 \cdot n_2$.

The Cooley-Tukey algorithm (see equation 2) can now be used to recursively compute the DFT of x in three phases:

- Computation of n_1 DFTs of size n_2 (along each column of the matrix)
- Multiplication of the matrix elements by twiddle factors
- Computation of n_2 DFTs of size n_1 (along each row of the matrix)

A high level description of the two-dimensional Transpose FFT algorithm is presented in Algorithm 1. For simplicity of exposition, we assume that n is an even power of 2 and $n_1 = n_2 = \sqrt{n}$. The data can therefore be considered to form a $\sqrt{n} \times \sqrt{n}$ matrix. The transpose algorithm stripes the matrix into $\sqrt{n}/p$ columns on each processor. Each processor computes $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ (along each column of the matrix). All processors then performs the twiddle factor multiplications on their local data. The next phase is to perform the DFTs along each row of the matrix. However, since the rows of the matrix are distributed amongst the processors, the algorithm first performs a matrix-transpose in order to transform the rows into columns that reside on the same processor. Each processor then performs the $\sqrt{n}/p$ DFTs of size $\sqrt{n}$ (along the new columns).

Typically the input vector is initially distributed over the processors in a way such that n/p contiguous elements of the vector lie on each processor – this can also logically be viewed as a two-dimensional $\sqrt{n} \times \sqrt{n}$ row-major matrix with $\sqrt{n}/p$ rows distributed on each processor. Therefore an additional matrix transpose is required in order to have the columns distributed amongst the processors as required by the transpose algorithm. Similarly, another transpose is required at the end in order to rearrange the output vector so that the contiguous elements reside on the same processor. The matrix transpose can be performed by using the Alltoall collective in conjunction with local transpose of blocks received from each processor (or sent to each processor).

2.2 FFTE Parallel algorithm

The FFTE algorithm is a variant of the Transpose algorithm which is more suited for vector processors. In this algorithm, the input vector is logically considered as a two-dimensional matrix. For simplicity of exposition, we assume the matrix dimensions to be $n^{2/3} \times n^{1/3}$.

There are three Alltoall phases as before. In the first computation phase, each processor computes $n^{2/3}/p$ DFTs, each of size $n^{1/3}$. In the second computation phase, there is a small variation. Each processor, instead of computing $n^{1/3}/p$ DFTs of size $n^{2/3}$ each, breaks down each of these computations in a manner similar to the two-dimensional transpose algorithm itself. Therefore It considers each vector of size $n^{2/3}$ as a matrix of size $n^{1/3} \times n^{1/3}$ and applies the Transpose algorithm again. Note however, that each of these $n^{1/3} \times n^{1/3}$ matrices lie on a single processor; hence the matrix transpose is a local transpose operation and does not require any communication between the nodes. The advantage of performing the transpose on a local node is to reorganize the data so that the elements of the vectors for which the DFTs have to be computed are stored contiguously in memory. The increase in the number of inner-most loops makes the Transpose FFT algorithm more suitable for vector processing.

3 Bottleneck Analysis

As described in the previous Section, parallel FFT algorithms perform FFT computations in five phases, with three phases of global transpose (Alltoall communication) interleaved with two phases of parallel single-node FFTs. If there is no computation-communication overlap the time taken for performing the FFT can be broken down into two parts – the computation time (t_{comp}) and the communication time (t_{comm}): $t_{fft} = t_{comp} + t_{comm}$.

3.1 Computation time

In each computation phase, a processor performs one dimensional FFT computations on a set of vectors present in the local memory. The number of floating point operations needed to compute FFT of a vector of size n is $5n\log(n)$. The time taken for this computation is $5c \cdot n \log n$, where $1/c$ is the floating point performance (FLOPS) of the single-node FFT implementation on the architecture.

Consider the two-dimensional transpose FFT algorithm. For a vector of size n distributed over p processors, in each of the two computation phases, each processor computes $\sqrt{n}/p$ DFTs each of size $\sqrt{n}$. Therefore the parallel computation time is determined by

$$t_{comp} \;=\; 5 \cdot 2\tfrac{\sqrt{n}}{p} \cdot c\sqrt{n}\log\sqrt{n} \;=\; 5\; c \cdot \tfrac{n}{p} \cdot \log n$$

The computational time is inversely proportional to the number of nodes in the system.

3.2 Communication time

In this subsection we discuss the communication performance of the parallel 2-dimensional transpose-FFT algorithm, implemented on a system with a d-dimensional torus network. We first discuss strong and weak scaling limits when the communication and header overheads are ignored and then we modify the analysis take the overheads into account.

3.2.1 Performance without header overheads

Consider the FFT computation of a vector of size n distributed over p processors. Let b be the size (in bytes) of each element of the vector and B be the bisection bandwidth of

the underlying interconnection network in bytes-per-second. All the data (ignoring self-transfers) is exchanged in each phase of the Alltoall communication. Note that one-fourth of the data would cross any bisection in one direction. Therefore, the communication time for performing the Alltoall collective is at least $t_{a2a} = nb/(4B)$.

For a symmetric d-dimensional torus network with link bandwidth l and $p^{1/d}$ processors in each dimension, the bisection bandwidth is given by $B = 2P^{(d-1)/d}l$. Therefore, the communication time for performing three Alltoall collective on such a system is given by $t_{comm} = 3nb/(8 \cdot l \cdot p^{(d-1)/d})$. For the more generic case, when the dimensions of the torus are not equal it is $t_{comm} = 3nb/(8 \cdot l \cdot (p/p_m))$ where p_m is the size of the longest dimension. Thus, the performance of parallel FFT based on a two-dimensional transpose algorithm is given by

$$FLOPS = \frac{5p}{5c + \frac{3}{8}\frac{b}{l}\frac{p^{1/d}}{\log(n)}} \tag{3}$$

Strong Scaling. For determining strong scaling, the problem size (n) is kept fixed and the number of processors (p) is varied. For small values of p (when $\frac{l/b}{1/c} >> \frac{3}{40}\frac{p^{1/d}}{\log(n)}$) the scaling is linear as $FLOPS \approx p/c$. Note that the ratio $(l/b)/(1/c)$, called "flops-to-bandwidth ratio", represents the ratio of time taken to transfer one floating-point element to the time taken to perform one floating-point operation. As p is increased, the second term in denominator of Eq. 3 begins to dominate the first term (when $p >> \left[40/3\left(\frac{l/b}{1/c}\right)\log(n)\right]^d$). In this case the FFT performance is given by

$$FLOPS \approx \frac{40}{3}\frac{l}{b}\log(n)p^{(d-1)/d}$$

In the transpose algorithm, the Alltoall size (data exchanged between a pair of nodes) is nb/p^2. Therefore, the maximum value of p is equal to $\sqrt{n}$ which gives a maximum performance of

$$FLOPS \approx \frac{40}{3}\frac{l}{b}n^{(d-1)/2d}\log(n)$$

Weak Scaling. In this case, the problem size per node is kept constant as the number of processors is varied. Thus $n = pM/b$ where M is per-node memory allocated to the problem. Considering $p^2 \leq n$, the maximum value p can take is M/b. Therefore, the maximum achievable performance under weak scaling is given by

$$FLOPS = \frac{5M}{5bc + \frac{3}{16}\frac{b^{(2d-1)/d}}{l}\frac{M^{1/d}}{\log(M/b)}}$$

3.2.2 Including header overheads

Considering header overheads, the amount of data exchanged between every pair of nodes is

$$\frac{nb}{p^2} + \left\lceil \frac{nb}{p^2 D_{max}} \right\rceil h$$

where, h is the overhead per packet in bytes (including header, trailer, acknowledgments, etc.), and D_{max} is the largest payload size (in bytes) that can be sent in a packet. In case $nb/p^2 >> D_{max}$, the amount of data exchanged between any pair of nodes can be approximated as $\frac{nb}{p^2}(1 + \frac{h}{D_{max}})$. In this case the performance of the FFT algorithm will be same as before with b replaced by $b(1 + \frac{h}{D_{max}})$.

In the case when $\frac{nb}{p^2} \leq D_{max}$, the amount of data exchanged between all the node pairs is $\frac{nb}{p^2} + h$. Therefore, the total communication time is given by

$$t_{comm} = \frac{3}{4B} \cdot p^2 \left[\frac{nb}{p^2} + h\right] = \frac{3}{8} \frac{(nb + p^2 h)}{p^{(d-1)/d} l}.$$

The FFT performance in this case is given by

$$FLOPS = \frac{5p}{5c + \frac{3}{8}(\frac{b}{l} + \frac{hp^2}{nl}) \frac{p^{1/d}}{\log(n)}} \tag{4}$$

Strong Scaling. Recall that for the transpose algorithm, $p^2 \leq n$. So, if the header overhead, h, is significantly larger than the size of a vector element, b, and $\frac{l/h}{1/c} =<< \frac{3}{40} \frac{p^{1/d}}{\log(n)}$, then the numerator scales as $O(p)$ whereas the denominator scales as $O(p^{2+1/d})$. Therefore, in this range, increasing the number of processors will degrade performance.
Weak Scaling In this case, $n = Mp/b$ and also $p^2 \leq n$. Therefore, under the conditions stated above, the numerator scales as $O(p)$ while the denominator scales as $O(p^{1+1/d})$. Even in this case, there is an absolute limit to which performance may scale.

The absolute performance limits can be estimated from Eq. 4 for strong as well as weak scaling.

4 Optimizations to parallel FFT

There are three avenues for optimization of the FFT algorithm – (i) decrease the computation time, (ii) decrease the communication time, i.e. improve Alltoall performance and (iii) overlap the computation with communication. We describe how we optimized the parallel FFT algorithm on the Blue Gene/L Supercomputer, using all the three optimizations. Since Blue Gene/L is based on a 3-dimensional torus network, the bottleneck for FFT switches from computation to communication as the number of nodes is increased. Therefore, all the optimizations are required for good performance over the full range of Blue Gene/L system sizes. We start with a brief overview of the Blue Gene/L Supercomputer followed by a discussion on our optimizations in the following subsections. The best FFT performance know till date is using the HPC Challenge benchmark [6, 22]. We use this benchmark to study the the effect of our optimizations.

4.1 Blue Gene/L Overview

The Blue Gene/L is a massively parallel supercomputer that scales up to 104K dual-processor nodes [9]. The nodes themselves are physically small, allowing for very high packaging

density in order to realize optimum cost-performance ratio. Each node has two embedded 700 MHz PPC440 processor cores, allowing the system to run in two different modes. In the *coprocessor mode* one of the processors is dedicated to messaging and one is available for application computation. In the *virtual node mode* each node is logically separated into two nodes, each of which has a processor and half of the physical memory. Each processor is responsible for its own messaging. In this mode, the node runs two application processes, one on each processor. Each node has 32-KB L1 instruction and data caches and a 4-MB embedded DRAM L3 cache. The Blue Gene/L uses five interconnect networks for I/O, debug, and various types of interprocessor communication. The most significant of these interconnection networks is the $64 \times 32 \times 32$ three-dimensional torus that has the highest aggregate bandwidth and handles the bulk of all communication. Each node supports six independent 1.4 Gbps bidirectional nearest neighbor links, with an aggregate bandwidth of 2.1 GB/s. The torus network uses both dynamic (adaptive) and deterministic routing with virtual buffering and cut-through capability. The messaging is based on variable size packets, each $n \times 32$ bytes, where $n = 1$ to 8 "chunks". The first eight bytes of each packet contain link-level information, routing information and a byte-wide cyclic redundancy check (CRC) that detects header data corruption during transmission. In addition, a 32-bit trailer is appended to each packet that includes a 24-bit CRC.

4.2 Single-node performance

The effort to optimize the FFT computational kernel was reasonably limited in scope, but resulted in reasonable performance improvements. In order to utilize the SIMD load-store and FMA units, we replaced the C code with Blue Gene/L intrinsics. Intrinsics have a one-to-one correspondence with assembly instructions and are conventionally viewed as moving from compiler hints (such as providing alignment and disjoint pragma) to compiler commands.

These instructions are not identical to assembly instructions because the individual instructions are still scheduled by, and the registers allocated/managed by, the compiler. This is advantageous in terms of productivity and, typically, the compiler does a good job with register allocation and scheduling. However, there are disadvantages to this arrangement that one can address in a number of different ways.

Because the compiler handles the register allocation, there can be register spills. While the compiler spills and restores in a very efficient manner, these spills do take up execution slots that could be used for other purposes and they can degrade performance. This can be addressed in two ways. First, it is our experience that reducing the number of pseudo registers asked of the compiler, the spill rate will go down. Second, we could replace the intrinsics with assembly instructions and handle the register allocation ourselves.

Further, the compiler appears to schedule instructions as if the data of interest was resident in the L1 data cache. During FFT operations this is not always the case. Unfortunately, the appropriateness of this approximation made to reality (by the compiler) becomes apparent when the remedies are considered.

While it is possible to schedule the instructions in assembler (or by turning off the scheduler and using intrinsics) so as to allow a longer load to use latency (12 cycles for covering L2 latency instead of 5 for covering L1 latency), there are resource limitations that force this strategy to yield limited returns. The cores of the Blue Gene/L system are limited to han-

dling 4 outstanding (non-L1) loads in 3 different cache lines. By arranging the strides of the load so consecutive loads address different cache lines (and loading the following quad-words in the shadow of the L1 load), it is possible to improve the covered latency by approximately 50%, but further improvements appear difficult. It should be pointed out that the L2 prefetch unit makes this limitation considerably less problematic than it would be on a system not so equipped.

4.3 Alltoall collective algorithm

As shown in Section 3, the performance of the Alltoall communication collective is bandwidth bound on systems such as Blue Gene/L, where the bisection bandwidth does not scale linearly with the number of nodes. Therefore congestion build-up can have adverse effects on the performance of the Alltoall performance. In this Section, we discuss an algorithm for the Alltoall collective, called *barrier-synchronization* algorithm that we proposed [12] for the Alltoall collective, specially designed for the torus-interconnect. This Alltoall algorithm gives significant improvements over the Blue Gene/L product MPI Alltoall – up to 35% on 64K systems. We briefly summarize the approach here for completeness.

The main idea behind the barrier-synchronization algorithm is to periodically clear up any congestion that may have built up in the communication network by draining the network completely. This eliminates long term effects of congestion build-ups. In order to do this, we divide the Alltoall communication into multiple phases and require these phases to exhibit certain properties:

Load-balancing of links: We ensure that in each phase, the load on all the links of the 3D-torus interconnect is the same. This is to avoid local hot-spots as far as possible.

Load-balancing of phases: We ensure that in each phase, the total traffic load (in terms of byte-hops) is the same.

Synchronization: We group these phases and then separate them with the fast Blue Gene/L hardware barrier. This has two advantages. First, it ensures that the network is completely drained after each phase – so that congestion effects are not carried over across the grouped phases. Second, it prevents network buffers from becoming full by limiting the total number of packets entering the network. The number of phases to be grouped together is determined by the amount of data to be exchanged with each node and the network buffers available on the Blue Gene/L.

In order to ensure good load-balancing of links, it is very important to decide which other nodes a given node will communicate with in each iteration.

For exposition, we first consider a 1D-torus of length p. Assume that p is a suitable multiple of 4. Let x be any node in the ring. In phase i, the node x exchanges data with nodes $x+i$, $x-i$, $x+(p/4-i)$ and $x-(p/4-i)$ for $i=1$ to $p/4-1$, where addition & subtraction are *modulo* p. In another phase (phase 0), data is sent to nodes $x+p/4$, $x-p/4$ and $x+p/2$ with the data to farthest node being split equally in both the directions. It is easy to verify that with this scheme, the total load on each of the links is the same for each phase (except phase 0).

Now consider a 3D-torus of dimension $p_x \times p_y \times p_z$. n each phase, a node (x, y, z) sends

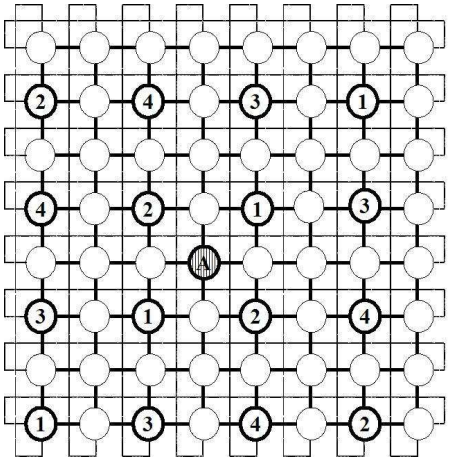


Figure 1: Communication patterns (for sender A) in different phases on a 2D 8x8 torus.

data to exactly four other nodes given by[1] $(x+i, y+j, z+k)$, $(x-i, y-j, z-k)$, $(x+p_x/2-i, y+p_y/2-j, z+p_z/2-k)$ and $(x-p_x/2+i, y-p_y/2+j, z-p_z/2+k)$. The values of i, j, k are chosen in phases such that the full space of nodes is spanned. It can be verified that except in the cases $(i, j, k) \in \{(0,0,0), (\pm P_x/2, \pm P_y/2, \pm P_z/2)\}$: (a) the four nodes are always distinct (b) load across bottleneck links is perfectly balanced and (c) all phases have the same load (byte-hops). The phases corresponding to $(i, j, k) \in \{(0,0,0), (\pm P_x/2, \pm P_y/2, \pm P_z/2)\}$ require special handling but are few and do not take significant time. Figure 1 shows the traffic pattern for node A in a 2 dimensional 8x8 torus network. One may verify that with this scheme, the links in each dimension as well as the phases are perfectly balanced.

We implemented our Alltoall algorithm directly using the underlying lower layer communication primitives.

4.4 Computation Communication Overlap

The partitioning of parallel FFT algorithms into computation and communication phases makes it an attractive option to overlap computation and communication. This requires careful partitioning of data into smaller data sets which are independent of each other.

For this optimization, we replace the FFTE algorithm with the simpler two-dimensional Transpose FFT algorithm. It is easier to conceptualize and implement the computation-communication overlap for this simpler case. Although this is not the most efficient implementation for FFT, it allows us to achieve the overlap more easily.

Note that the $\sqrt{n}/p$ DFT computations involved in step 2 of the transpose FFT algorithm (Algorithm 1) along each column are independent of each other. Similarly the DFT computations in step 5 along each row are also independent of each other. Therefore, it is possible to overlap the computations of phases 2 and 5 with the communication of phases 3 and 6 respectively. The optimized algorithm incorporating computation-communication overlap is illustrated in Algorithm 2. In order to achieve the overlap, the $\sqrt{n}/p$ vectors on each processor in each phase are divided into m blocks of $b = \sqrt{n}/pm$ vectors each. The All-

[1] Addition and subtraction along dimensions x, y, z are modulo p_x, p_y, p_z respectively.

Algorithm 2: Block-vector Transpose FFT Algorithm Pseudo-code

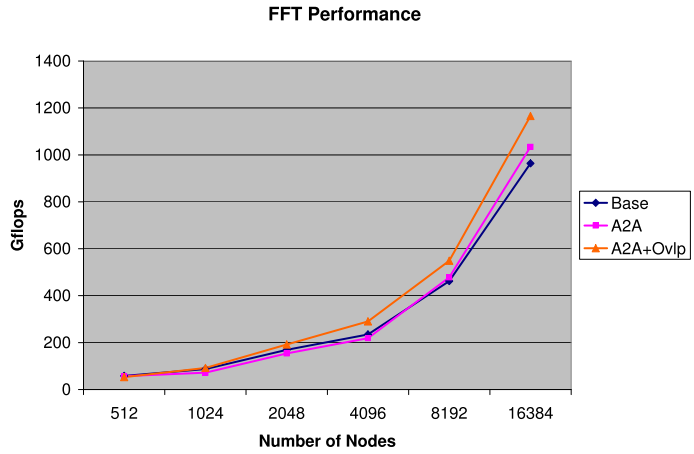
1. Distribute columns on each processor - Global Transpose (Alltoall)
2. /* Compute DFTs of matrix columns and overlap with Alltoall */
   a. Compute DFTs of 1st block of column vectors, i.e., first $\sqrt{n}/pm$ columns
   b. For $j = 1$ to $m - 1$ do
      i. Initiate DFTs of $(j + 1)^{th}$ block of column vectors on second core/thread
      ii. Distribute j^{th} block of column vectors using Alltoall
   c. Distribute m^{th} block of column vectors using Alltoall
3. Multiply by twiddle factors and Rearrange
4. /* Compute DFTs of matrix columns and overlap with Alltoall */ Similar to Step 2
5. Rearrange the output data elements - Global Transpose (Alltoall)

toall communication of phase 3 and 6 are also split into m Alltoall communication calls each. The computation of the $(j + 1)^{th}$ block of vectors can now be overlapped with the Alltoall communication of the j^{th} block of vectors (already computed in the previous iteration).

When the computation and communication are divided into m sub-phases, the smaller of the two costs (computation and communication) gets hidden behind the other, except for one iteration of pre-processing/post-processing that is not overlapped. Therefore, the total time taken by the FFT algorithm reduces to $max\{(1/m)t_{comp} + t_{comm}, t_{comp} + (1/m)t_{comm}\}$, where the first term corresponds to the case where communication is the bottleneck and the second term corresponds to the case where computation is the bottleneck.

Some of these overlap optimizations have been previously discussed in [11] – however there, the authors proposed the use of asynchronous send and receive operations for performing the communication. This rules out the use of optimized Alltoall collectives for which many optimization techniques can be applied (as described in the previous section) therefore obtaining much better performance for the communication phase in comparison to asynchronous send/receives. We propose techniques for overlapping computation and communication that still use the Alltoall collective which can be implemented on systems supporting multiple threads – with one thread being dedicated to computation and the other to Alltoall communication. This technique lends itself naturally to the Blue Gene/L system that has dual-core nodes whereby one of the cores can be dedicated to computation and the other to Alltoall communication.

On Blue Gene/L, the dual-core nodes are used in the overlapping of computation and communication. The main thread executes on core 0 (master-core). It allocates the work of DFT computations to core 1 in Step 2.b.i. (and Step 4.b.i.) of the algorithm above. After initiating the DFT computations, it proceeds to perform the Alltoall communication for the already computed DFTs (Steps 2.b.ii. and 4.b.ii.). Once it completes the communication, it waits on core 1 to complete the computations, so that it can proceed to the next iteration. As mentioned in Section 4.1, the caches of the dual-core nodes on Blue Gene/L are not coherent. Therefore special care needs to be taken to flush/invalidate the caches. We flush the cache on core 0 before initiating work on core 1 and we flush the cache on core 1 after it completes the computations. This ensures that there is no stale data in either of the caches.



Num Nodes	Base (Gflops)	A2A (Gflops)	A2A+Ovlp (Gflops)
512	58	57	53
1024	87	72	92
2048	169	154	192
4096	235	219	290
8192	462	478	549
16384	964	1034	1166
65536	2311	2875	2779

Figure 2: Comparison of base FFT algorithm with algorithm incorporating optimized Alltoall (A2A), and both Alltoall and computation-communication overlap (A2A+Ovlp)

5 Performance Results

In this Section, we discuss the performance of parallel FFT obtained with our optimizations. We present weak scaling and strong scaling results.

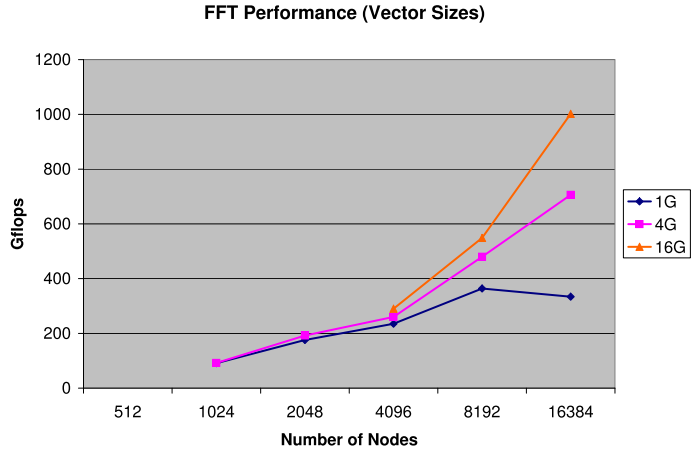
Weak Scaling. Figure 2 compares the performance of the FFT algorithm with various optimizations. The *base* algorithm refers to the HPC Challenge FFT algorithm in which the single node FFT code is optimized to use the double hummer features of Blue Gene/L. The *A2A* algorithm refers to the base algorithm with the Alltoall collective modified to use the optimizations mentioned in Section 4.3. Finally, the *A2A+Ovlp* algorithm refers to the base algorithm modified to include both – the Alltoall optimizations and the computation-communication overlap optimizations. The size of the vectors in these runs are determined such that the vector size per-node ranges between 2M to 4M elements per node and the total vector size is a perfect square.

It can be seen that the performance of A2A algorithm is a little worse than the base algorithm but improves with increasing number of nodes. This is expected because the proposed Alltoall algorithm introduces barrier overheads to clear up the network. These overheads degrade performance for small sizes. The Alltoall optimizations are intended to regularly clear up congestion hot-spots, which impact performance much more for large number of nodes. The performance of the optimized algorithm with both optimizations improves over the base algorithm by as much as 20% for large systems.

On the 64K node Blue Gene/L system, the FFT time is dominated by the Alltoall communication time. For a vector of size 274877906944 of double precision complex numbers ($b = 16$), the base algorithm performs the FFT in 22.6 seconds at a rate of 2311 Gflops. The time spent in the computation and communication phases is 5.5 seconds and 17.1 seconds respectively. More than 75% of the time is spent in performing global Alltoall collectives. Our calculations indicate that this Alltoall performance is 63% of the theoretically achievable peak performance (after factoring in header and other overheads). Our Alltoall algorithm takes 4.1 seconds for the same communication – about 85% of the theoretically achievable

peak performance. This gives a performance of 2875 Gflops for the FFT algorithm using the Alltoall optimizations alone. This is a 20% improvement over the base algorithm and is better than the currently best reported HPC Challenge FFT performance record [22]. The performance does not improve using computation-communication overlap optimizations along with the Alltoall optimizations (2779 Gflops). The reason is that the overlap techniques result in dividing the Alltoall into multiple phases as well - resulting in an Alltoall size of 256 bytes for the above sizes. Therefore, overheads related to header, trailer and acknowledgments become costly, resulting in $nb/p^2 < D_{max}$ (See Section 3.2.2). Hence, the overlapping techniques are useful only if the Alltoall size is sufficiently large.

Strong Scaling. In Figure 3, we compare the FFT performance for different vector sizes ranging from 1G to 16G elements over different number of nodes. Performance scales well with increasing number of nodes. However, after scaling to a certain number of nodes, the performance starts to drop. This can be observed for the case of 1 GigaElements vector size, where the performance drops on 64K nodes. This is because the Alltoall size drops to $nb/p^2 = 2^{30} \cdot 16/16348^2 = 64$ bytes for this case. This is much smaller than the maximum data transferable per packet which is $D_{max} = 240$ bytes. Therefore, as discussed in Section 3.2.2, the header/trailer and acknowledgment overheads become large compared to the data being transferred, resulting in inefficient use of the communication network. Hence, performance scales well as long as the Alltoall size $nb/p^2 > D_{max}$.



Num Nodes	1 Giga elements	4 Giga elements	16 Giga elements
1024	91	92	-
2048	176	192	-
4096	235	260	290
8192	364	480	549
16384	334	706	1002

Figure 3: Performance of optimized FFT algorithm for different vector sizes

6 Conclusions

In this paper, we analyzed the performance of two-dimensional transpose-based FFT algorithm on a massively parallel system with a d-dimensional torus network. On the Blue Gene/L Supercomputer, which is based on a 3-dimensional torus network, we identified the performance bottlenecks and optimized the performance by (i) improving the performance of the single-node FFT algorithm; (ii) improving the Alltoall collective algorithm and (iii) overlapping computation and communication. Using these optimization, we obtained the best known performance numbers (2875 Gflops) for FFT on any system.

All the techniques discussed in this paper are also applicable to parallel transpose-based algorithms for computing 2D and 3D-FFT. These algorithms also use single-node FFT computations and Alltoall communication. In these cases, the Alltoall is generally restricted to smaller communicators (a subset of the nodes). On torus interconnects, these communicators are typically mapped to planes of the torus. Therefore the Alltoall optimizations can still be applied in two-dimensions. Finally, the computation-communication overlap techniques are also applicable as these algorithms also perform computations and matrix transpositions in separate phases.

7 Acknowledgments

We would like to thank several people without whose valuable help, this work could not have been accomplished. We thank James Sexton for closely working with us on the HPC Challenge benchmarks. We thank Sameer Kumar and Philip Heidelberger for useful discussions related to the low-level communication library for Blue Gene and Alltoall optimizations.

References

[1] Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994. *A high performance parallel algorithm for 1-D FFT.* In Proceedings of the IEEE Conference on Supercomputing. 1994. pp. 34-40.

[2] Bluestein, L. I., *A linear filtering approach to the computation of the discrete Fourier transform,* Northeast Electronics Research and Engineering Meeting Record 10, 218-219 (1968).

[3] Brenner, N., and Rader, C., *A New Principle for Fast Fourier Transformation,* IEEE Acoustics, Speech & Signal Processing 24: 264-266. 1976.

[4] Bruun, G., z-Transform DFT filters and FFTs, IEEE Trans. on Acoustics, Speech and Signal Processing, 1978, pp. 56-63.

[5] Cooley, James W., and John W. Tukey, *An algorithm for the machine calculation of complex Fourier series,* Math. Comput. 19, 297301 (1965).

[6] Dongarra, J., and Luszczek, P., Introduction to the hpc challenge benchmark suite. TR ICL-UT-05-01, ICL, 2005.

[7] Eleftheriou, M., Moreira, J. E., Fitch, B. G., Germain, R. S., *A Volumetric FFT for BlueGene/L.* HiPC 2003: 194-203.

[8] Frigo, M. and Johnson, S. G., *The Design and Implementation of FFTW3,* Proceedings of the IEEE 93 (2), 216231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[9] Gara, A., Blumrich, M. A., Chen, D., Chiu, G. L.-T., Coteus, P., Giampapa, M. E., Haring, R. A., Heidelberger, P., Hoenicke, D., Kopcsay, G. V., Liebsch, T. A., Ohmacht, M., Steinmacher-Burow, B. D., Takken, T., and Vranas, P., Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49:195–212, 2005.

[10] Good, I. J. *The interaction algorithm and practical Fourier analysis,* J. R. Statist. Soc. B 20 (2), 361-372 (1958). Addendum, ibid. 22 (2), 373-375 (1960).

[11] Gupta, S. K. S., Huang, C.-H., Sadayappan, P. and Johnson, R. W., *A technique for overlapping computation and communication for block recursive algorithms,* Concurrency: Practice and Experience", Vol 10(2), pp 73–90, 1998.

[12] Kumar, S., Garg, R., and Heidelberger, P., *Performance Analysis and Optimization of All-to-all communication on the Blue Gene/L Supercomputer.* To appear in the proc. of the IEEE International Conference on Parallel Processing, 2008.

[13] Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin-Cummings Publishing Co., Inc., 1994.

[14] Perry, M., *Using 2-D FFTs for object recognition,* ICSPAT, pp. 1043-1048, DSP World Expo., 1994.

[15] Rader, C. M., *Discrete Fourier transforms when the number of data samples is prime,* Proc. IEEE 56, 11071108 (1968).

[16] Sridharan, S., Dawson, E., and Goldgurg, B., *Speech encryption using discrete orthogonal transforms,* ICASSP-90, pp. 1647-1650, Albuquerque. NM, April 1990.

[17] Stearns, S.D. and David, R.A., *Signal Processing Algorithms,* Englewood Cliffs, NJ, Prentice Hall 1993.

[18] Takahashi, D., *High-performance parallel FFT algorithms for the HITACHI SR8000,* Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, 2000. pp. 192-199 vol.1

[19] Thomas, L. H., *U*sing a computer to solve problems in physics, in Applications of Digital Computers. 1963.

[20] Winograd, S., *On computing the discrete Fourier transform,* Math. Computation 32: 175-199 (1978).

[21] MPI FFTW. *http://www.fftw.org/fftw2_doc/fftw_4.html.*

[22] HPC Challenge Results. *http://icl.cs.utk.edu/hpcc/hpcc_results.cgi.*