Automatic Generation of the HPC Challenge's Global FFT Benchmark for BlueGene/P

Franz Franchetti[1], Yevgen Voronenko[2], and Gheorghe Almasi[3]
franzf@ece.cmu.edu, yvoronen@gmail.com, and gheorghe@us.ibm.com

[1] Carnegie Mellon University, ECE Department, Pittsburgh, PA 15213, USA
[2] Accuray, Inc., Sunnyvale, CA 94089, USA
[3] IBM Research, T. J. Watson Research Center, Yorktown Heights, NY 10598, USA

Abstract. We present the automatic synthesis of the HPC Challenge's Global FFT, a large 1D FFT across a whole supercomputer system. We extend the Spiral system to synthesize specialized single-node FFT libraries that combine a data layout transformation with the actual on-node FFT computation to improve the network performance through enabling all-to-all collectives. We run our optimized Global FFT benchmark on up to 128k cores (32 racks) of ANL's BlueGene/P "Intrepid" and achieved 6.4 Tflop/s, outperforming ANL's 2008 HPC Challenge Class I Global FFT run (5 Tflop/s). Our code was part of IBM's winning 2010 HPC Challenge Class II submission. Further, we show first single-thread results on BlueGene/Q.

1 Introduction

The HPC Challenge (HPCC) [1] has been developed to provide more in-depth benchmarking of supercomputers beyond the HPL benchmark used for the TOP500 ranking [2]. HPCC contains seven benchmarks: `HPL`, `STREAM`, `RandomAccess`, `PTRANS`, `FFT`, `DGEMM`, and `b_eff`. In this paper we focus on the Global FFT benchmark, which computes a large 1D FFT across a large distributed-memory machine, using FFTE [3]. Given network bandwidth and node (CPU) performance developments, Global FFT is on large machines dominated by the machine's cross-sectional bandwidth and thus the three global transposes required to compute a 1D FFT where input and output are in natural order.

An optimized Global FFT implementation must combine an optimized communication library (e.g., the vendor MPI library) with an optimized FFT library (e.g., the vendor FFT library). However, performance does not compose. To obtain the best performance within each of these two libraries, incompatible data formats are required: (1) The MPI library typically is best optimized to send large messages through collective communication functions (e.g., MPI all-to-all), which requires all data for the same destination processor to be packed into one contiguous memory area. (2) Conversely, FFT libraries usually require the data for FFTs to be contiguous in the node memory to obtain best performance. However, the Global FFT algorithm requires sending neighboring data elements to different target processors. Thus, one either needs to convert the data between the storage formats in between library calls to the two libraries or

one needs to resort to less efficient library functions (e.g., MPI all-to-allv instead of all-to-all). In either case the overhead can be significant.

Contribution. In this paper we present a novel distributed memory 1D FFT algorithm for large processor counts. Our algorithm blocks the FFT's three global transposes so that for each transpose every processor sends only one large message (that is contiguous in memory) to each other processor, using the most optimized collective communication call. The necessary data scrambling before sending and after receiving and the twiddle factor scaling becomes part of modified node FFT libraries. We extend the program generation and autotuning framework Spiral to automatically generate and optimize these modified FFT libraries, and use UPC as communication layer. Our optimized Global FFT reaches 6.4 Tflop/s on 128k cores (32 racks) of ANL's BlueGene/P while FFTE (the original HPCC Global FFT) reached 5 Tflop/s on the same machine in the 2008 Class I HPC Challenge award. Finally, we make a first step towards targeting BlueGene/P's successor—the BlueGene/Q system—and demonstrate Spiral's ability to automatically generate highly optimized single threaded code taking full advantage of the new QPX SIMD vector instruction set.

Related work. One and multi-dimensional FFT algorithms for distributed memory are an extensively studied topic. Most 1D FFT algorithms are building on the *Six Step FFT Algorithm* [4], which breaks a large 1D FFT into two stages of local FFTs on contiguous data plus a twiddle stage and three global transposes. FFTW [5] provides a well-optimized open source single node and MPI FFT library. FFTE is specifically designed for large 1D distributed memory FFTs [3], and the reference implementation of the HPC Challenge's FFT benchmark (Global FFT) [1]. In this work we extend the program generation system Spiral [6, 7], and builds on Spiral's code generation for BlueGene/L's *Double FPU* [8], and multicore CPUs [9]. Our system uses IBM's UPC runtime [10] as communication layer. We are building on Spiral's experimental MPI FFT code generation for fixed problem-size and small and fixed processor count (up to 16) [11]. We are extending these concepts to automatically generate the whole FFT computation required for the Global FFT HPCC benchmark, which needs to be a single library working for any problem size and processor count without recompilation.

2 Background

In this section we discuss the necessary background for this paper. We review the Kronecker product formalism to describe fast Fourier transform (FFT) algorithms, the autotuning and program generation system Spiral, and the BlueGene/P supercomputer.

Fast Fourier Transform. Given n real or complex inputs $x_0, \ldots, x_{n-1}$, the discrete Fourier transform (DFT) is defined as

$$y_k = \sum_{0 \le \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \le k < n, \tag{1}$$

with $\omega_n = \exp(-2\pi i/n)$, $i = \sqrt{-1}$. Stacking the x_ℓ and y_k into vectors $x = (x_0, \ldots, x_{n-1})^T$ and $y = (y_0, \ldots, y_{n-1})^T$ yields the equivalent form of a matrix-vector product:

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}. \tag{2}$$

Computing the DFT by its definition (2) requires $\Theta(n^2)$ many operations. FFT algorithms reduce the runtime to $O(n\log(n))$ and can be written in the form of structured sparse matrix factorization using the Kronecker product formalism [6, 12]. The two-point FFT is given by the butterfly matrix,

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{3}$$

In the following, we use I_n to denote an $n \times n$ identity matrix, and

$$A \otimes B = [a_{k\ell}B],\ A = [a_{k\ell}]$$

for the tensor (Kronecker) product of matrices. It replaces every entry $a_{k,\ell}$ of A by the matrix $a_{k,\ell}B$. Most important for FFTs are the cases where A or B is the identity matrix. As examples consider

$$\mathrm{I}_4 \otimes \mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix} \text{ and } \mathrm{DFT}_2 \otimes \mathrm{I}_4 = \left[\begin{array}{cccc|cccc} 1 & & & & 1 & & & \\ & 1 & & & & 1 & & \\ & & 1 & & & & 1 & \\ & & & 1 & & & & 1 \\ \hline 1 & & & & -1 & & & \\ & 1 & & & & -1 & & \\ & & 1 & & & & -1 & \\ & & & 1 & & & & -1 \end{array}\right].$$

Further we introduce the stride permutation matrix defined by

$$\mathrm{L}^{mn}_m :\ jn + i \mapsto im + j, \quad 0 \le i < n,\ 0 \le j < m.$$

L^{mn}_m can be seen as transposing a $n \times m$ matrix which is stored in row-major order and is derived from reshaping a mn-dimensional vector into a $n \times m$ matrix. As example consider

$$\mathrm{L}^8_2 = \begin{bmatrix} 1 & & & & & & & \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & & & & & & 1 & \\ & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{bmatrix}.$$

Equation (4) shows the general mixed-radix Cooley-Tukey FFT algorithm:

$$\mathrm{DFT}_{mn} = \left(\mathrm{DFT}_m \otimes \mathrm{I}_n\right) \mathrm{T}^{mn}_n \left(\mathrm{I}_m \otimes \mathrm{DFT}_n\right) \mathrm{L}^{mn}_m. \tag{4}$$

In (4), T^{mn}_n is a complex diagonal matrix [12]. Using (3) and (4), an 8-point FFT can be derived by two recursive applications:

$$\mathrm{DFT}_8 = (\mathrm{DFT}_2 \otimes \mathrm{I}_4)\, \mathrm{T}^8_4 \left((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2 (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2\right) \mathrm{L}^8_2.$$

Formulas can be manipulated using formula identities like

$$\mathrm{I}_n \otimes (BC) = (\mathrm{I}_n \otimes B)(\mathrm{I}_n \otimes C) \tag{5}$$
$$(BC) \otimes \mathrm{I}_n = (B \otimes \mathrm{I}_n)(C \otimes \mathrm{I}_n) \tag{6}$$
$$\left(\mathrm{L}^{mn}_m\right)^\top = \mathrm{L}^{mn}_n \tag{7}$$
$$(BC)^\top = C^\top B^\top \tag{8}$$
$$A \otimes B = \mathrm{L}^{mn}_m (B \otimes A)\, \mathrm{L}^{mn}_n \tag{9}$$
$$(A \otimes B)^\top = A^\top \otimes B^\top \tag{10}$$
$$\mathrm{L}^{kmn}_n = (\mathrm{L}^{kn}_n \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}^{mn}_n) \tag{11}$$
$$\mathrm{L}^{kmn}_{km} = (\mathrm{I}_k \otimes \mathrm{L}^{mn}_m)(\mathrm{L}^{kn}_k \otimes \mathrm{I}_m) \tag{12}$$
$$\mathrm{L}^{kmn}_k = \mathrm{L}^{kmn}_{km}\, \mathrm{L}^{kmn}_{kn}\,. \tag{13}$$

In (5)–(10), A is a $m \times m$ matrix and B and C are $n \times n$ matrices. Further we denote the conjugation of a matrix A by a permutation P by $A^P = P^\top A P$ and note that for permutation matrices $P^{-1} = P^\top$.

```
void FFT8(_Complex double  *Y, _Complex double  *X) {
    __alignx(16,Y);
    __alignx(16,X);
    _Complex double s34, s35, s36, s37, s38, t100, t101, t102
             , t103, t104, t94, t95, t96, t97, t98, t99;
    t94 = (*(X) + *((X + 4)));
    t95 = (*(X) - *((X + 4)));
    t96 = (*((X + 2)) + *((X + 6)));
    s34 = (__I*(*((X + 2)) - *((X + 6))));
    t97 = (t94 + t96);
    t98 = (t94 - t96);
    t99 = (t95 + s34);
    t100 = (t95 - s34);
    t101 = (*((X + 1)) + *((X + 5)));
    t102 = (*((X + 1)) - *((X + 5)));
    t103 = (*((X + 3)) + *((X + 7)));
    s35 = (__I*(*((X + 3)) - *((X + 7))));
    t104 = (t101 + t103);
    s36 = (__I*(t101 - t103));
    s37 = ((0.70710678118654757 + __I * 0.70710678118654757)*(t102 + s35));
    s38 = ((-0.70710678118654757 + __I * 0.70710678118654757)*(t102 - s35));
    *(Y) = (t97 + t104);
    *((Y + 4)) = (t97 - t104);
    *((Y + 1)) = (t99 + s37);
    *((Y + 5)) = (t99 - s37);
    *((Y + 2)) = (t98 + s36);
    *((Y + 6)) = (t98 - s36);
    *((Y + 3)) = (t100 + s38);
    *((Y + 7)) = (t100 - s38);
}
```

Fig. 1. 8-point FFT, using complex C99 data types and the IBM XL C dialect.

Spiral. Recursive application of rules like (3) and (4) yields many different algorithms for a FFT size. Spiral [6] uses this fact to search for the fastest on a given platform. A user-specified transform (like DFT_{256}) is expanded by Spiral using rules into

SPL construct	code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1]);`
$y = (\mathrm{I}_m \otimes A_n)x$	`for (i=0;i<m;i++)` `  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`
$y = (A_m \otimes \mathrm{I}_n)x$	`for (i=0;i<m;i++)` `  y[i:n:i+m-1] = A(x[i:n:i+m-1]);`

$y = \mathrm{L}^{km}_k\, x$	`for (i=0;i<k;i++)`
	`  for (j=0;j<m;j++)`
	`  y[i+k*j]=x[m*i+j];`

$y = \mathrm{T}^{km}_k\, x$	`for (i=0;i<k*m;i++)` `  y[i]=T_km_k[i]*x[i];`

Table 1. Compiling SPL into code is done by recursively using the above correspondences. x denotes the input and y the output vector. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e`, and extracted at stride `s`. `T_km_k` is a array of precomputed constants.

a formula, which is then translated into a C program by a special formula compiler. The formula compiler is based on a translation table similar to Table 1 and uses traditional compiler techniques like unrolling, array scalarization, constant folding, and strength reduction to produce high quality fixed-size FFT functions from a given formula. The runtime of the program is measured and fed into a search module, which triggers, in a feedback loop, the generation of a modified formula based on a search strategy. Upon termination, Spiral out the fastest program found. Figure 1 shows an 8-point FFT generated by Spiral for BlueGene/P, using the complex data type extension of C99.

For sizes too large to be implemented as a single basic block, Spiral is automatically generating a recursive mixed-radix FFT library [7] similar to FFTW [5]. Spiral employs a rewriting system to symbolically expand breakdown rules like (4) to find a closure of recursive functions that is needed to implement the recursive FFT library. It then automatically implements these recursive functions as well as recursion leafs (codelets) for a sufficiently large set of sizes. At runtime, a planner autotunes the recursive decomposition of the FFT in an one-time setup effort. After tuning, a fast FFT library call for the respective problem size is available.

The key insight is that a straightforward implementation of (4) suggests four steps corresponding to the four factors, where two steps call smaller DFTs. However, to improve locality, the initial permutation L^{mn}_m is usually not performed but interpreted as data access for the subsequent computation, and the twiddle diagonal T^{mn}_n is fused with the subsequent DFTs. This strategy is chosen, for example, in the library FFTW 2.x and the code can be sketched as shown in Figure 2. A simplified description of performing this process by hand can be found in [13].

BlueGene/P. BlueGene/P is the second generation BlueGene architecture from IBM, succeeding BlueGene/L [14]. In its compute nodes BlueGene/P uses four PowerPC 450 cores operating at 850 MHz with a double precision, dual pipe floating point

```
void dft(int n, complex *y, complex *x) {
   int k = choose_factor(n);
   // t1 = (I_k tensor DFT_m)L(n,k)*x
   for(int i=0; i < k; ++i)
      dft_iostride(m, k, 1, t1 + m*i, x + m*i);
   // y = (DFT_k tensor I_m) diag(d(j))
   for(int i=0; i < m; ++i)
      dft_scaled(k, m, precomp_d[i], y + i, t1 + i);
}

// DFT variants needed
void dft_iostride(int n, int istride, int ostride, complex *y, complex *x);
void dft_scaled(int n, int stride, complex *d, complex *y, complex *x);
```

Fig. 2. Recursive FFT implementation in the style of FFTW 2.X.

unit per core. Each node has 13.6 Gflop/s peak performance (3.4Gflop/s per core) and 2 GB RAM with 13.6 GB/s memory bandwidth. Each core has a private 32 kB L1 cache and the four cores of a node share an 8 MB L3 cache. The compute nodes are connected with multiple interconnection networks including a 3-D torus (used for standard messaging), a global collective network (used for reductions), and a global barrier network. Each node has six bi-directional network links supporting 425 MB/s in each direction into the torus network leading to 5.1 GB/s bidirectional bandwidth per node. The BlueGene/P system "Intrepid" installed at Argonne National Laboratory (ANL) consists of 40 BlueGene/P racks. Each rack contains 1,024 compute nodes (32 node cards, each holding 32 compute nodes), and each compute node four cores (one quad-core CPU).

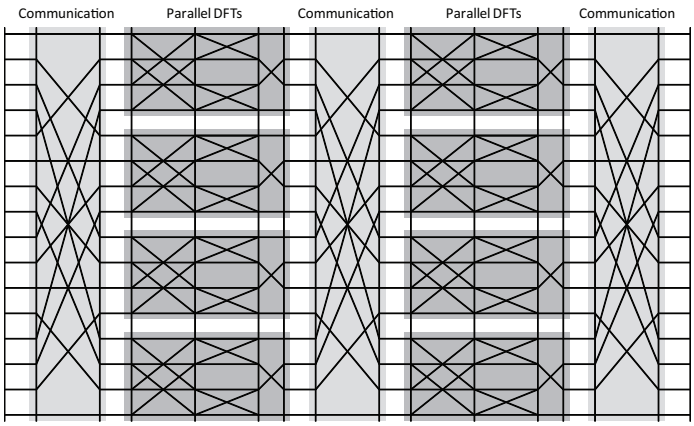
BlueGene/P messaging layer. We use the IBM UPC runtime system as messaging layer. It provides an equivalent to the MPI all-to-all collective operation that fully utilizes BlueGene/P's 3D torus interconnection network. To achieve best performance, exactly one large message of equal size that is contiguous in the node memory should be sent from every processor to every other processor.

BlueGene/Q. BlueGene/Q is the third generation BlueGene architecture from IBM, succeeding BlueGene/P [15]. The Blue Gene/Q Compute chip [16] is a system-on-a-Chip (SOC) ASIC with 16 user-accessible 4-way SMT (Symmetric Multi Threading) A2 cores clocked at 1.6 GHz. A quad floating unit implementing the QPX instruction set is associated with each core. The BlueGene/Q A2 chip achieves 204.8 Gflop/s peak performance. At 1024 chips per rack, the 48 rack ANL "Mira" systems achieves a peak performance of 10 Pflop/s and the 96 rack LLNL "Sequoia" system 20 Pflop/s, respectively.

3 Global FFT Algorithm

We now derive our novel 1D Global FFT algorithm, which is a variant of the Six Step FFT algorithm. Like the Six Step FFT algorithm, it has three global data exchanges. However, we block the global transpositions so that exactly one pair of large messages that are contiguous in memory are exchanged between every pair of processors in every communication step. This can be mapped efficiently to collective communication functions (all-to-all). We formally merge the ensuing data scrambling necessary to produce

consume the contiguous messages with the on-node FFT computations and derive modified FFT libraries (working on custom scrambled data format) that perform the reordering at no extra cost compared to standard FFT libraries. We use the Kronecker product formalism to derive the algorithm and use Spiral to automatically build the modified node FFT libraries from the Kronecker product specification. Finally, we show pseudo-code for the top-level parallel (single program multiple data, SPMD) function that calls the modified node FFT libraries.



$$\mathrm{L}_4^{16}\left(\mathrm{I}_4\otimes\big((\mathrm{DFT}_2\otimes\mathrm{I}_2)T_2^4(\mathrm{I}_2\otimes\mathrm{DFT}_2)L_2^4\big)\right)\mathrm{L}_4^{16}\,T_4^{16}\left(\mathrm{I}_4\otimes\big((\mathrm{DFT}_2\otimes\mathrm{I}_2)T_2^4(\mathrm{I}_2\otimes\mathrm{DFT}_2)L_2^4\big)\right)\mathrm{L}_4^{16}$$

Fig. 3. Six-step FFT for $n = 2^4$ and $k = m = \sqrt{n} = 4$.

Algorithm derivation. Using (5)–(13), the Six Step FFT algorithm can be derived from (4):

$$\mathrm{DFT}_{mn} = \mathrm{L}_m^{mn}\left(\mathrm{I}_n\otimes\mathrm{DFT}_m\right)\mathrm{L}_n^{mn}\,\mathrm{T}_n^{mn}\left(\mathrm{I}_m\otimes\mathrm{DFT}_n\right)\mathrm{L}_m^{mn}. \tag{14}$$

By flipping both tensor products into their parallel form ($\mathrm{I}_n\otimes\mathrm{DFT}_m$ and $\mathrm{I}_m\otimes\mathrm{DFT}_n$), the algorithm is guaranteed to perform all DFT computations within the local memory of each node. This is achieved by reshaping the data vector of length mn into a $n\times m$ matrix and explicitly transposing it back and forth (a total of three transpositions is required). Typically, choosing $m\approx\sqrt{mn}$ and $n = mn/m$ gives the so-called "square-root decomposition" which maximizes the number of processors that the DFT can be run on in parallel and provides good load balancing. A visual (data flow) representation of the 16-point six-step FFT is shown in Fig. 3. Details can be found in [17].

Below assume p processors and $p \mid m, n$. Using (5)–(13) and associativity and distributivity the stride permutation L_m^{mn} can be expressed as three permutation stages. First we use (12) to obtain

$$\mathrm{L}_m^{mn} = \left(\mathrm{I}_p\otimes\mathrm{L}_{m/p}^{mn/p}\right)\left(\mathrm{L}_p^{np}\otimes\mathrm{I}_{m/p}\right). \tag{15}$$

Next we use (5) to obtain

$$\mathrm{L}^{np}_{p} = \big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{n/p}\big)\big(\mathrm{I}_{p} \otimes \mathrm{L}^{n}_{p}\big). \tag{16}$$

Inserting (16) into (15) yields

$$\mathrm{L}^{mn}_{m} = \big(\mathrm{I}_{p} \otimes \mathrm{L}^{mn/p}_{m/p}\big)\big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\big)\big(\mathrm{I}_{p} \otimes \mathrm{L}^{n}_{p} \otimes \mathrm{I}_{m/p}\big) \tag{17}$$

after further simplification.

Equation (17) describes treating the $n \times m$ matrix as block matrix of $p \times p$ blocks with block size $nm/p \times nm/p$. Each of the p processor holds p blocks in its local memory. (17) states that a distributed matrix can be transposed by transposing all blocks (p^2 local transpositions, each transposing a local block of size $nm/p \times nm/p$) followed by transposing the blocks (one $p \times p$ transposition moving whole blocks, implemented as all-to-all collective communication). The mechanics of the Kronecker product formalism requires three factors to describe the two steps. In our algorithm derivation we also require a transposed version of (17) where we first swap m and n in (17) and the apply (7) to obtain the transposed expression for L^{mn}_{m}, leading to

$$\mathrm{L}^{mn}_{m} = \big(\mathrm{I}_{p} \otimes \mathrm{L}^{m}_{m/p} \otimes \mathrm{I}_{n/p}\big)\big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\big)\big(\mathrm{I}_{p} \otimes \mathrm{L}^{mn/p}_{m}\big). \tag{18}$$

Inserting (17) and (18) into (14) and regrouping the ensuing expression using (5) leads to the final algorithm (for a more detailed derivation see [11, 18]),

$$\begin{aligned}\mathrm{DFT}_{mn} = {} & \underbrace{\big(\mathrm{I}_{p} \otimes (\mathrm{L}^{m}_{m/p} \otimes \mathrm{I}_{n/p})\big)}_{\text{local transpose}} \underbrace{\big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\big)}_{\text{all-to-all}} \underbrace{\big(\mathrm{I}_{p} \otimes (\mathrm{DFT}_{m} \otimes \mathrm{I}_{n/p})\big)}_{\text{inplace FFT library call}} \\ & \underbrace{\big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\big)}_{\text{all-to-all}} \underbrace{(\mathrm{T}^{mn}_{n})^{(\mathrm{I}_{p} \otimes \mathrm{L}^{m}_{m/p} \otimes \mathrm{I}_{n/p})}\Big(\mathrm{I}_{p} \otimes (\mathrm{L}^{m}_{p} \otimes \mathrm{I}_{n/p})(\mathrm{I}_{m/p} \otimes \mathrm{DFT}_{n})\ \mathrm{L}^{mn/p}_{m/p}\Big)}_{\text{out-of-place scaled FFT library call}} \\ & \underbrace{\big(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\big)}_{\text{all-to-all}} \underbrace{\big(\mathrm{I}_{p} \otimes (\mathrm{L}^{n}_{p} \otimes \mathrm{I}_{m/p})\big)}_{\text{local transpose}}. \end{aligned} \tag{19}$$

Eq. (19) makes the minimal necessary changes to the Six Step algorithm to make it compatible to highly optimized all-to-all communication calls, and to allow for specialized high-performance local recursive FFT libraries. Reading (19) from right to left, first each processor performs local data scrambling (*local transpose*) in their own memory space to produce the first set of contiguous messages. This cannot be folded into any FFT library call but could be merged with computation that produces the input data. Next all processors invoke *all-to-all* collective communication; all p processors send one message of size mn/p^2 to every of the p processors (including themselves). Then a modified node FFT—the *out-of-place scaled FFT library*—is called to perform the local FFT computation on scrambled data and performs twiddle scaling. Next the same *all-to-all* call is invoked a second time, followed by the second modified node FFT library, an *inplace FFT library* operating on scrambled data. Note that too make this stage inplace, one needs to chose (17) and (18) carefully. Lastly, the same *all-to-all*

collective communication is called a third time to redistribute the data to the target processor and a final *local transpose* unscrambling phase puts the data back into natural order. This final scrambling cannot be merged with any of the modified libraries but could be merged with the code consuming the transformed data.

Specialized FFT node libraries. Our derivation extracted the formal definition of two modified node FFT libraries that are invoked independently but in parallel on all p processors. The first node library is specified as

$$(\mathrm{T}^{mn}_{n,i})^{(\mathrm{L}^m_{m/p} \otimes \mathrm{I}_{n/p})} \Big((\mathrm{L}^m_p \otimes \mathrm{I}_{n/p})(\mathrm{I}_{m/p} \otimes \mathrm{DFT}_n)\ \mathrm{L}^{mn/p}_{m/p} \Big) \tag{20}$$

with $\mathrm{T}^{mn}_{n,i}$ being the global FFT twiddle factors for processor i. The library specified by (20) performs an out-of-place batch FFT (m/p FFTs of size n) plus twiddle scaling on a block-matrix data format. The second library is specified as

$$\mathrm{DFT}_m \otimes \mathrm{I}_{n/p} \tag{21}$$

and performs an inplace strided batch FFT (n/p FFTs of size m) that can be viewed as column FFT. The modified node FFT libraries are automatically generated from the specification using Spiral's general size library generation framework [7].

To turn the algorithmic advantage into a performance advantage, the automatically generated libraries need to be of equivalent performance as FFTW or the vendor library ESSL. Since we are targeting Global FFT for 128k processors, the largest FFT sizes are up to $mn = 2^{38}$, and thus m and n can be up to 2^{19}. Thus, the node FFT libraries built from the specifications (20) and (21) need to provide good performance for batches of large FFTs. The generated libraries must perform all state-of-the-art optimizations including SIMD vectorization for the Double FPU [8, 9] and must be parallelized across the four cores of a BlueGene/P node [9] when running in SMP mode. Further, aggressive memory hierarchy optimizations like buffering and vector recursion need to be applied [5, 9]. All these optimizations need to be performed fully automatically [7].

Full Global FFT code. In Figure 4 we show the full HPCC Global FFT algorithm using a partitioned global address space (PGAS) abstraction similar to Unified Parallel C (UPC). The data vectors `x` and `y` are block distributed (mn/p elements reside in the local memory of each of the p nodes) and all parallel for loops are run across p nodes of the parallel machine. For simplicity, on-node threading and SIMD vectorization is omitted.

4 Experimental Results

We experimentally evaluated our optimized Global FFT benchmark on BlueGene/P configurations from one node card (32 quadcore nodes or 128 cores) up to 32 racks (32k quadcore nodes or 128k cores), with one process per node. We used the IBM UPC runtime for process and thread management and as messaging layer. The benchmark is executed as UPC program that calls external (C/C++) libraries for the on-node FFT computation. UPC uses IBM's XL C compiler as backend, and our generated synthesized on-node libraries were compiled with IBM's XL C compiler and options "-O3 -qarch=440d".

```
// HPCC Gloabal FFT
// data is block distributed on p processors
// the p iterations of parallel for loops are executed across p nodes

// all to all data exchange
//
// implements a block transpose of a p x p matrix on vectors of n elements
// exchange between all pairs of p processors packets of size n
//
// input/output:  x[p*p*n], block distributed across p processors
// x := L^{p^2}_p (x) I_n * x
//
void all_to_all(int p, int n, _Complex double *x) {
    int i, j;

    par_forall (i=0; i<p; i++)
        for (j=i+1; j<p; j++)
            SENDRECV(i, j, x+n*(i*p+j), x+n*(i+j*p), n);
}


// local transpose
//
// transposes a n x m matrix of vectors of v complex elements,
// stored in row major order in local node memory
// x := L^{mn}_m (x) I_v * x
//
void transpose(int mn, int m, int v, _Complex double x) {
    int i, j, k, n = mn/m;

    for (i=0; i<n; i++)
        for (j=i; j<m; j++)
           for (k=0; k<v; k++)
               SWAP(x[v*(i*m+j)+k], x[v*(i+j*n)+k]);
}


// global FFT of size m*n on p processors
// y = DFT_mn * x
//
// input:  x[m*n], block distributed across p processors
// output: y[m*n], block distributed across p processors
//
void global_fft(int m, int n, int p, _Complex double y, _Complex double x) {
    int i, j, k;

    par_forall (i=0; i<p; i++)
        transpose(n, p, m/p, x+i*m*n/p);

    all_to_all(p, m*n/(p*p), x);

    par_forall (i=0; i<p; i++)
        fft_scaled(n, m, p, n/p, m*n, m/p, m*n/p, x+i*m*n/p, y+i*m*n/p);

    all_to_all(p, m*n/(p*p), x);

    par_forall (i=0; i<p; i++)
        fft_inplace(m, m*n/p, m, n/p, y+i*m*n/p);

    all_to_all(p, m*n/(p*p), x);

    par_forall (i=0; i<p; i++)
        transpose(m, m/p, n/p, y+i*m*n/p);
}
```

Fig. 4. HPCC Global FFT implementation using a UPC-like PGAS syntax, implementing (19).

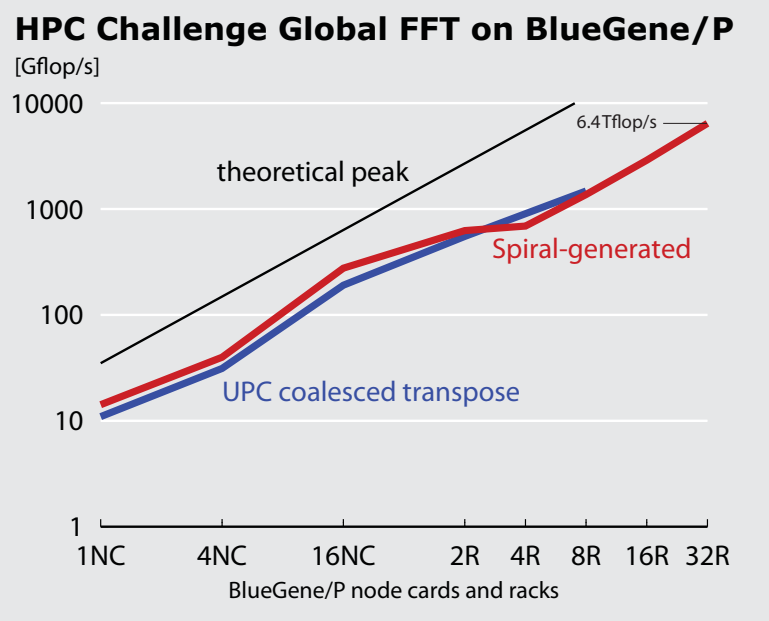


Fig. 5. Performance of the HPC Challenge Global FFT Benchmark on BlueGene/P from 128 cores (1 node card) up to 128k cores (32 racks).

We implemented a baseline Global FFT version that uses IBM's BlueGene/P ESSL for local FFTs and UPC coalesced transpose (the equivalent of MPI all-to-all) for messaging. This implementation requires explicit data reordering between the UPC messaging and the invocation of ESSL but provides best performance for the FFT computation and the messaging in separation. This implementation is part of IBM's winning 2010 HPC Challenge Class II UPC submission.

Figure 5 summarizes the performance results. We run the UPC+ESSL baseline benchmark on the IBM T.J. Watson BlueGene/P system for up to eight racks. We run our Spiral-generated library from one node card to 2 racks on the T.J. Watson machine and on ANL's "Intrepid" from 4 racks to 32 racks. The Spiral-generated Global FFT generally outperforms the UPC+ESSL baseline which shows that (a) Spiral's automatically generated node libraries offer performance competitive with ESSL, and (b) the memory traffic savings obtained by merging data scrambling with the node-libraries improves performance. Finally, the Spiral-generated Global FFT reaches 6.4 Tflop/s on 32 racks of "Intrepid". The winning 2008 ANL HPC Challenge Class I submission reported 5 Tflop/s Global FFT performance on the same machine. Thus, the combination of algorithmic optimization and library generation improved the Global FFT on "Intrepid" by 1.4 Tflop/s or 28%.

Single Node performance. Figure 6 shows single node performance on the BlueGene/P quadcore PowerPC 450D. We compare the GNU Scientific Library (GSL) [19] to Spiral-generated sequential and multi-threaded scalar and Double FPU-vectorized code. Spiral-generated scalar single-core code significantly outperforms the GSL for in-cache sizes and performs equally to the GSL for memory-bound sizes, demonstrat-

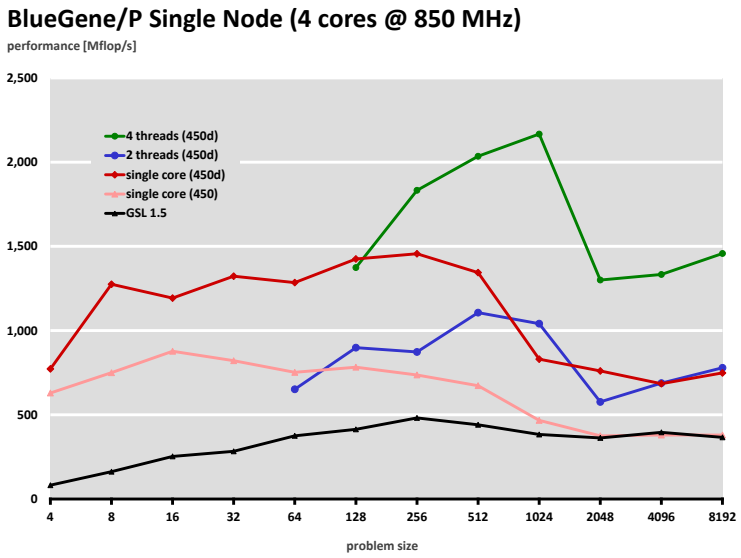


Fig. 6. Single node performance on BlueGene/P quadcore CPU.

ing the quality of Spiral's base line code generation on BlueGene/P. Spiral's Double FPU two-way SIMD vector code provides between 50% and 2x speed-up on top of the scalar base-line. Using all four cores of the BlueGene/P multicore CPU yields speed-up of 2x–2.5x except for the smallest sizes where parallelization overhead makes sequential code the fastest choice. Using only two threads is never a winning strategy.

Towards Global FFT on BlueGene/Q. We are in the process of porting the Spiral Global FFT code generation to the next generation BlueGene machines, BlueGene/Q. One major difference is that BlueGene/Q features a new 4-way SIMD vector unit called QPX that is twice as wide as the Double FPU of BlueGene/P. In Figure 7 we show first performance results of Spiral-generated QPX code run on a single thread of a BlueGene/Q node. We observe that for small FFT sizes Spiral-generated code substantially outperforms both FFTW and ESSL. We are currently porting and adapting the remaining two levels of parallelism of the Global FFT (intra-node threading and inter-node message passing) to BlueGene/Q.

5 Conclusion

The increased complexity and performance levels of high performance and supercomputing systems makes the automatic generation and tuning of performance libraries for the petascale and beyond a promising alternative to hand-tuning. In this paper we present an novel 1D Global FFT algorithm for the HPC Challenge. Extending the Spiral system, we automatically generate specialized node FFT libraries that support the data layout required by the messaging layer while providing FFT performance of the native

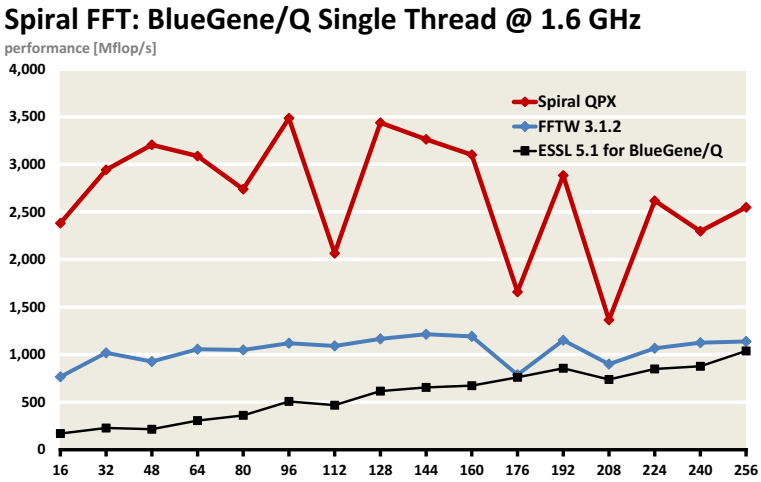


Fig. 7. BlueGene/Q QPX single thread performance.

FFT data layout. The resulting reduction in memory traffic and high node performance enabled us to reach 6.4 Tflop/s on 128k cores of ANL's BlueGene/P system, improving performance by 28% over the previously reported Global FFT Class I benchmark. Finally, we show first single-node results on BlueGene/Q in which we significantly outperform FFTW and ESSL.

Acknowledgement

The authors acknowledge support by NSF through awards 0702386 and 1116802, and by the U.S. Army through contract W911NF-10-1-0004. Yevgen Voronenko was partially supported by a Kauffman Entrepreneur Postdoctoral Fellowship. The authors wish to thank Kalyan Kumaran, Scott Parker and Vitali Morozov of Argonne National Laboratory for access and help with ANL's BlueGene/P "Intrepid" and BlueGene/Q "VEAS"/"VESTA".

References

1. Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D.: The HPC Challenge (HPCC) benchmark suite. In: SC06 Conference Tutorial. (2006)
2. Meuer, H.W.: The top500 project: Looking back over 15 years of supercomputing experience (2008)
3. Takahashi, D.: An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors. In: Proc. Int'l Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA). (2006) 1178–1187
4. Bailey, D.H.: FFTs in external or hierarchical memory. J. Supercomputing **4** (1990) 23–35

5. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231 special issue on "Program Generation, Optimization, and Adaptation".
6. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE **93**(2) (2005) 232–275 special issue on "Program Generation, Optimization, and Adaptation".
7. Voronenko, Y., de Mesmay, F., Püschel, M.: Computer generation of general size linear transform libraries. In: Proc. Code Generation and Optimization (CGO). (2009) 102–113
8. Franchetti, F., Kral, S., Lorenz, J., Püschel, M., Ueberhuber, C.W., Wurzinger, P.: Automatically tuned FFTs for BlueGene/Ls Double FPU. In: High Performance Computing for Computational Science (VECPAR). Volume 3402 of Lecture Notes in Computer Science., Springer (2004) 23–36
9. Franchetti, F., Püschel, M., Voronenko, Y., Chellappa, S., Moura, J.M.F.: Discrete Fourier transform on multicore. IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores" **26**(6) (2009) 90–102
10. UPC Consortium: UPC language specifications, v1.2 (2005) Lawrence Berkeley National Lab Tech Report LBNL-59208.
11. Bonelli, A., Franchetti, F., Lorenz, J., Püschel, M., Ueberhuber, C.W.: Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: Proc. International Symposium on Parellel and Distributed Processing and Applications (ISPA). (2006)
12. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
13. Chellappa, S., Franchetti, F., Püschel, M.: How to write fast numerical code: A small introduction. In: Lecture Notes in Computer Science. Volume 5235., Springer (2008) 196–259
14. Alam, S., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, C., Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., Worley, P., Yu, W.: Early evaluation of IBM BlueGene/P. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. SC '08, Piscataway, NJ, USA, IEEE Press (2008) 23:1–23:12
15. Team, T.B.G.: Blue Gene/Q: by co-design. Computer Science - Research and Development (2012) 1–9
16. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., gara, a., Chiu, G., Boyle, P., Chist, N., Kim, C.: The ibm blue gene/q compute chip. IEEE Micro **32**(2) (2012) 48–60
17. Franchetti, F., Püschel, M.: Fast Fourier Transform. In: Encyclopedia of Parallel Computing. Springer (2011)
18. Chellappa, S.: Computer Generation of Fourier Transform Libraries for Distributed Memory Architectures. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2010)
19. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F.: GNU Scientific Library Reference Manual - Third Edition (v1.12). Network Theory Ltd. (2009)