Gigaelement FFTs on Apple G5 clusters

R. Crandall, E. Jones, J. Klivington, and D. Kramer

Advanced Computation Group, Apple Computer

17 Aug 04

Abstract: We specify G5 cluster configurations suitable for performing massive (billion-element) fast Fourier transfoms (FFTs), notably 1-dimensional (length 2^{30}), or 2-dimensional ($2^{15} \times 2^{15}$), or 3-dimensional ($2^{10} \times 2^{10} \times 2^{10}$), all for complex-valued float elements. We present timing results for both single- and double-precision floats. We show how clusters can exploit concurrent Altivec-based function calls from Apple's Accelerate framework. We indicate how such distributed "gigaelement" FFTs can be performed using Apple's Xgrid and LnxMPI, or LAM/MPI. In our "canonical configuration"—namely four G5s with simple Ethernet switch—we are able to sustain 2 gigaflops (double-) and 4 gigaflops (single-precision). More extensive hardware options will allow speeds at least as high as 4 gigaflops (double-) and 8 gigaflops (single-precision). These are excellent performance results given the omnipresent issues of communication latency and cache behavior for such enormous signal lengths.

1 Motivation

The need for massive, distributed FFTs arises in much the same way that other expansive phenomena have emerged on the computational scene. Just as Moore's Law predicts exponential growth of transistor density on integrated circuits, modern image processing—which often involves 2-dimensional FFTs—has become a field in which 5000×5000-pixel frames, and even larger sizes than this, are not uncommon. Similarly, across fields from cryptography to computational number theory to data analysis, sampled streams—destined for 1-dimensional FFT—are sometimes in the > 100 Msample range, and again, sometimes even longer than this, reaching into the billions of input elements.

A primary impetus for the present research was a question posed in early 2004 by researchers A. Barty and H. Chapman of Lawrence Livermore National Laboratory (LLNL), namely: How should Apple G5's be configured to effect a $1024 \times 1024 \times 1024$, 3-dimensional FFT? Their application is novel and important: A revolutionary method for protein-structure determination using a new *single-molecule* imaging technique. This method requires X-ray diffraction data from such advanced machinery as the super-powerful Linac Coherent Light Source (LCLS) [6] with the computational problem being that of taking say 1000 separate 1024^3 FFTs to achieve convergence. This means that when we report a time of say 23 seconds for a single such FFT, a required batch of 1000 separate FFTs at LLNL will take about 8 hours of real-time—a "day at the office." This is considered excellent performance, especially when one realizes that these 1000 FFTs amount to more than 10^{14} floating-point operations.

Each of these three scenarios:

- 1-dimensional FFT with length 2^{30},
- 2-dimensional FFT with dimensions $2^{15} \times 2^{15}$,
- 3-dimensional FFT with dimensions $2^{10} \times 2^{10} \times 2^{10}$,

we call "gigaelement" FFTs, that is, about 1 billion complex elements. As we shall see, each of these three modes takes *roughly* the same real-time in a given G5-cluster configuration. One reason for this performance equity is that we can use a method for effecting a 1-dimensional FFT as a "twiddled" 2-dimensional one, so the 1- and 2-dimensional modes differ only by $O(N)$ complexity for n elements. So, in fact, the 3-dimensional mode is usually a tad slower than, but still comparable to the other modes.

It is important to note that the speed advantage that may accrue from parallelization is not the only motivating factor for machine multiplicity. Another consideration is the challenge of keeping an entire massive signal in physical memory. Parallel FFT algorithms that use multiple processes on a single computer are necessary when one has enough physical memory to hold the signal, but the signal doesn't fit entirely in a single address space. Likewise, distributed FFT algorithms that use multiple processes on multiple computers are necessary when one has more data than can fit in the physical memory of one computer.

The specific memory requirements of a cluster designed to run our FFT software depend on the length of the signal, the size of the cluster, and the precision of the data. The values in Table 1 are computed from the formula for required memory μ per machine:

$$\mu = \frac{3}{2} \cdot \frac{N}{M} \cdot b + c, \tag{1.1}$$

where N is the number of elements in the signal, M is the number of machines, b is the number of bytes per complex element, and c is the approximate system overhead. The 3/2 factor comes from the use of a transposition workspace buffer (described later in this paper) half the size again of the actual data. For single-precision complex elements, $b = 8$ bytes, while for double-precision complex elements, $b = 16$ bytes. As mentioned before, we observed that $c \approx 0.5$ GB.

Table 1: Required physical memory μ, per machine, to run the FFT software on varying lengths of signals and varying numbers of computers.

N (signal length)	M (# of machines)	μ (single-precision)	μ (double-precision)
2^{28}	1	3.5 GB	6.5 GB
2^{28}	2	2.0 GB	3.5 GB
2^{30}	2	6.5 GB	N/A
2^{30}	4	3.5 GB	6.5 GB
2^{32}	8	6.5 GB	N/A
2^{32}	16	N/A	6.5 GB
2^{33}	16	6.5 GB	N/A

Because are interested in running both the single- and double-precision gigaelement transforms we choose the ($M = 4$)-machine cluster to be what we call our "canonical cluster."

Note that the physical memory requirements for the computers are yet more stringent if one expects to run any applications while the FFT is being computed. Even the simple act of logging into the computer can use memory needed for holding data. The formula (1.1) for per-machine memory μ is therefore only valid if the computers are used exclusively for FFT computation.

2 FFT modes

Some FFT nomenclature will be useful in software descriptions and performance assessment. We shall employ notation that renders clearly the notion that a 2-dimensional FFT involves signal dimensions W, H (width, height); a 3-dimensional FFT involves W, H, L (width, height, length), and so on. Likewise, we shall use running integer indices such as x, y, z to underscore further the notion of spatial dimensions. The 1-dimensional fast Fourier transform (FFT) algorithm calculates the discrete Fourier transform (DFT) of a length-W signal s as a vector S with the p-th of its W components being

$$S_p = \sum_{x=0}^{W-1} s_x \; e^{-2\pi i (x_p/W)}. \tag{2.1}$$

So this 1-dimensional FFT has length W, which for the present purposes we may regard as "width" W. The 2-dimensional FFT of a $W \times H$ signal s is taken to be the $W \times H$ collection

$$S_{pq} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} s_{xy} \; e^{-2\pi i (x_p/W + y_q/H)}.$$

Happily, this 2-dimensional transform S can be effected by FFT of every row of the $W \times H$ signal matrix, in-place, and then FFT of every column, also in-place. In turn, the 3-dimensional FFT of a $W \times H \times L$ signal s is

$$S_{pqr} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} \sum_{z=0}^{L-1} s_{xyz} \; e^{-2\pi i (x_p/W + y_q/H + z_r/L)}.$$

This 3-dimensional transform may also be effected via the expedient of in-place FFT of all rows (for each fixed pair yz, a 1-dimensional FFT along x), then all columns (along y), then all "pencils" (along z).

As for theoretical complexity, each of the 1,2,3-dimensional FFTs requires about

$$\# = C \cdot N \cdot \log_2 N$$

operations, where $N = W, W \cdot H, W \cdot H \cdot L$ respectively, i.e. N is the total element count. For the 1-dimensional Cooley–Tukey standard FFT structure, we have $C = 5$ and this counts all real-valued multiply and add machine operations. Taking for example the $1024 \times 1024 \times 1024$ 3-dimensional FFT, the number of real operations is about

$$\# = 5 \cdot 2^{30} \cdot 30 = 1.6 \times 10^{11},$$

i.e. about 160 gigaops. Thus a gigaelement FFT is running at a gigaflop if the whole run takes 160 seconds.

3 Distributing the FFT

It is instructive to refer to pictorials of the signals throughout this section. We view a 1-dimensional signal as a single row, read left-to-right. We view a 2-dimensional signal as a stack of 1-dimensional signals, each 1-dimensional signal read one after another, starting from the top. We view a 3-dimensional signal as a stack of 2-dimensional signals, each 2-dimensional signal read one after another, starting from the back.

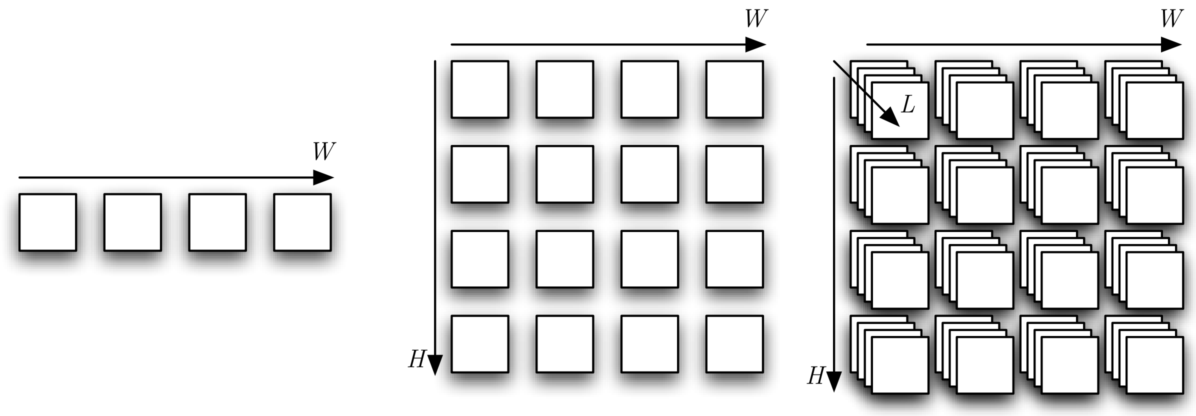

Figure 1: Pictorals of the 1-dimensional, 2-dimensional, and 3-dimension signals, shown in column-major order, and read in memory left-to-right, top-to-bottom, and back-to-front. The x dimension has width W, the y dimension has height H, and the z dimension has length L. In these diagrams, $W = H = L = 4$.

Without loss of generality we may assume higher-dimensional signals are stored in column-major order in memory. For example, if we view a 3-dimensional signal stored in memory as a contiguous 1-dimensional signal of width $W' = W \cdot H \cdot L$, then

$$s'_i = s_{xyz}, \quad \text{where } x = i \bmod W, \quad y = (i/W) \bmod H, \quad \text{and } z = i/(W \cdot H);$$

$$s_{xyz} = s'_i, \quad \text{where } i = x + W \cdot y + W \cdot H \cdot z.$$

For computational purposes we shall therefore view a 2-dimensional signal as L consecutive width-W signals, and a 3-dimensional signal as L consecutive $W \times H$ planes, or $H \cdot L$ consecutive width-W signals.

In our notation it is always the *first* dimension that is read left to right. It should be noted that this is merely notation. Multi-dimensional signals operated on by the software may be input either in row- or column-major order. The transformed signal will be in the same order as the input signal, unless you choose to skip the final transpose.

When multiple processes participate in the calculation of the FFT, each process holds a unique portion of the total signal elements. If P is the number of processes and P evenly divides W, H, and L then each process holds $1/P$ of the signal. For a 1-dimensional signal this means each process has W/P consecutive elements. For a 2-dimensional signal this means each process has H/P consecutive width-W signals. For a 3-dimensional signal this means each process has L/P consecutive $W \times H$ planes. On the canonical four-computer cluster describe in this paper we often ran four processes per computer, so in that case each process held 1/16 of the data.

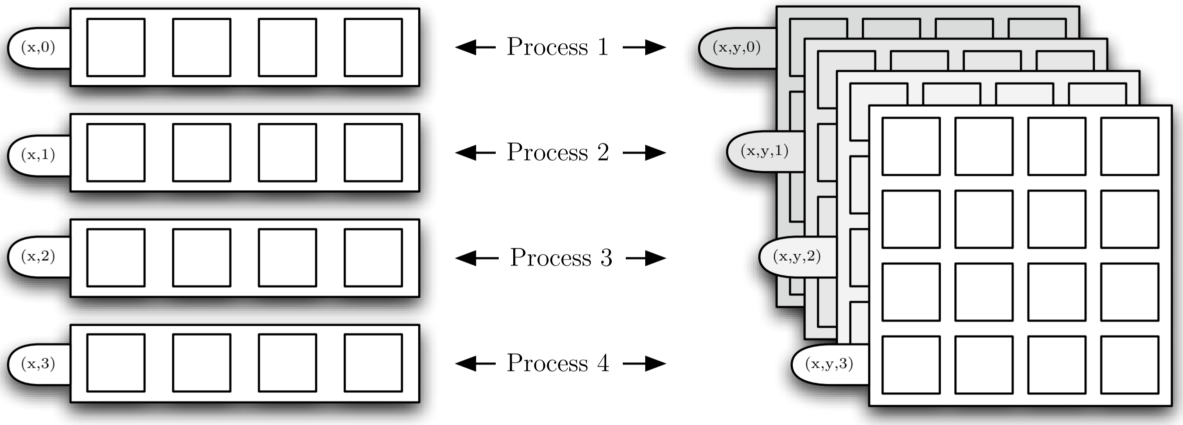

Figure 2: Diagrams of the data distribution of 2-dimensional and 3-dimensional signals across $P = 4$ processes. In this example, each process holds one quarter of the entire data. In the case of the 2-dimensional signal each process holds one quarter of the rows. In the case of the 3-dimensional signal each process holds one quarter of the planes.

As mentioned before, all of the higher dimensional FFTs can be computed by performing in-place 1-dimensional transforms along each of the dimensions. If we permit ourselves to perform in-place 2-dimensional transposes of dimensions held locally as well as 2-dimensional transposes of dimensions distributed across many processes, it is possible to compute all of the higher dimensional FFTs using only row transforms.

While a 1-dimensional in-place FFT of a row is quickly computed using existing library functions, assuming the width W is not too large[1], computing in-place FFTs of 1-dimensional columns and pencils may be less efficient than in-place FFTs of rows, because the elements of columns and pencils are not stored contiguously in memory. Furthermore, when a 2-dimensional signal is distributed across multiple processes, each process holds entire rows, but no process holds an entire column. Likewise, when a 3-dimensional signal is distributed across multiple processes, each process holds entire rows and columns, but no process holds

[1]The performance constraint on W is that the entire row must fit into memory, hopefully all into cache. A method for computing a 1-dimensional FFT with massive W—via a "twiddled" 2-dimensional $\sqrt{W} \times \sqrt{W}$ FFT—is described later in this section.

an entire pencil. Because it is inefficient and sometimes impossible to perform column or pencil transforms, we decide to make use of row FFTs exclusively in the following algorithms.

To compute a 2-dimensional FFT using only row FFTs, we first transform along x all of the rows. We then transpose the rows and columns, thus swapping the x and y dimensions. We next transform along y all of the new rows (formerly columns), and finally transpose the rows (originally columns) and columns (originally rows) to get the signal back in the original order.

To compute a 3-dimensional FFT using only row FFTs, we first transform along x all of the rows. We then transpose the rows and columns, thus swapping the x and y dimensions. We next transpose the columns (originally rows) and pencils, thus swapping the x and z dimensions. Next we transform along y the new rows (originally columns). We then transpose the rows (originally columns) and columns (originally pencils), thus swapping the y and z dimensions. Next we transform along z all of the new rows (originally pencils). Finally transpose the rows (originally pencils) and pencils (originally rows) to get the signal back in the original order.

Thus we can compute 2-dimensional and 3-dimensional FFTs, using only row FFTs and 2-dimensional transposes. These "row-only" algorithms can be modified to distribute the work across P processes running on M machines, as in Figure 3. To compute the 2-dimensional transform each process does concurrently a height H/P strip of width-W row FFTs, then the full $W \times H$ matrix is transposed over the network, so that next each process can concurrently do a height W/P strip of width H row FFTs, equivalent to column FFTs on the original index order. Optionally, if one wishes the final FFT to be reordered back to original index order, one can expend one final transpose over the network.

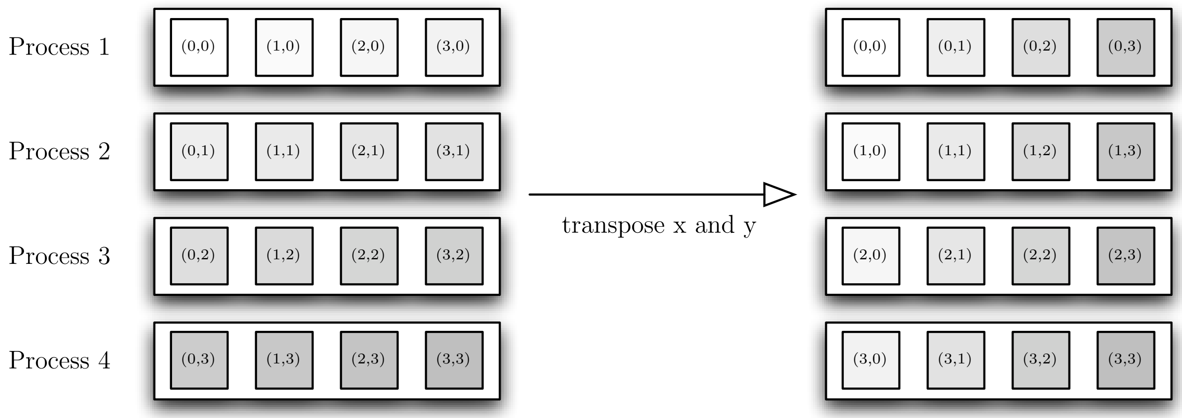

Figure 3: Diagram of the movement of data elements during a distributed transpose of a 2-dimensional signal. First, each process computes the FFT of the row it is holding, then the processes engage in the distributed transpose, and finally each process computes the FFT of the new row it is holding.

The 3-dimensional FFT can follow the basic prescription of the 2-dimensional FFT, with, of course, more intricate transposition work, as seen in Figure 4. Starting with the dimensions in the order $W \times H \times L$, each process does concurrently a length L/P stack of height-H strips of width-W row FFTs, then locally transposes each strip of rows so that the matrix dimensions end up $W \times L \times H$. Then the full matrix is transposed over the network so that the dimensions end up $L \times W \times H$. Next each process does concurrently a length H/P stack of height W strips of width-L row FFTs. Then each process transposes each strip of rows so that the matrix dimensions end up $L \times H \times W$, and then does a length W/P stack of height H strips of width-L row FFTs. Optionally, if one wants the final FFT to be reordered back to original index order, one can expend one final transpose over the network.

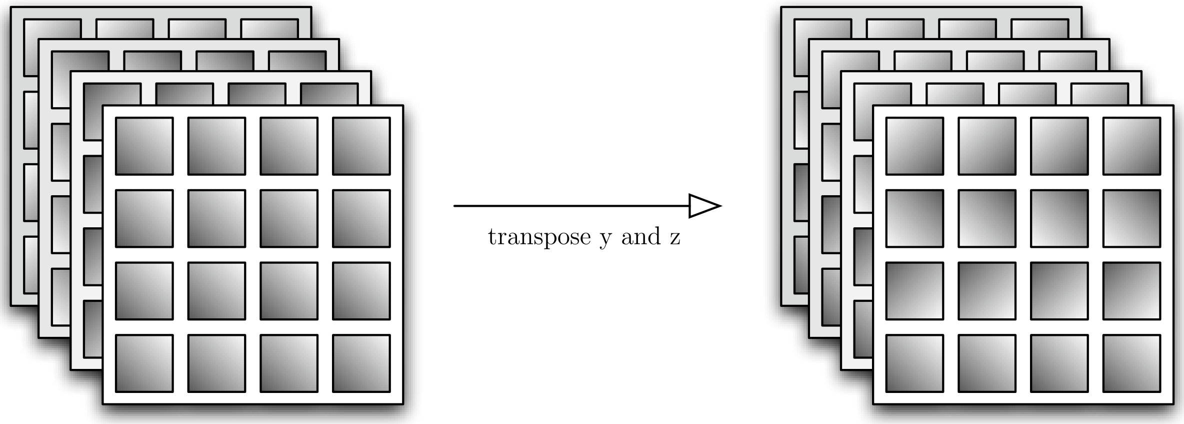

Figure 4: Diagram of the movement of data elements during a distributed transpose of a 3-dimensional signal. First, each process computes the FFT of the plane it is holding, then the processes engage in the distributed transpose and some local transposes, and finally each process computes the FFT of the new rows it is holding

Finally, to compute a 1-dimensional FFT for very large W, we may use the 4-step factored matrix approach described in [1] and [3]. The idea is, one may perform a 1-dimensional FFT of length $W = 4^k$ by "column-wise" ordering of the original signal into a square matrix of dimensions $\sqrt{W} \times \sqrt{W}$, then "pretending" to do a 2-dimensional FFT on a the square matrix, except that between the row and column FFT steps, one "twiddles" the x, y-th matrix element by a factor $\exp(-2\pi ixy/\sqrt{W})$. This is the 1-dimensional method we used to obtain timings. When we compute the FFT of a 1-dimensional signal of width $W = 4^k$, we actually perform all of the steps previously described to compute the FFT of a 2-dimensional signal with dimensions $W \times H = 2^k \times 2^k$, with an additional twiddle step run concurrently on all the processes in the middle, immediately following the first set of row FFTs, before the distributed transpose.

4 Precise hardware/software configuration

In putting together the "canonical cluster" for use in this research, we endeavored to specify standard components on behalf of prospective Apple users. Performance may indeed be improved by using specialized components (see later sections, Re: more machines and faster network); the canonical cluster described below is intended as a solid price/performance compromise.

Canonical cluster:

For the purposes of testing and timing four dual-processor 2.0 GHz PowerMac G5 computers were used, as shown in Figure 5. Each computer was was running Mac OS X 10.3.4 and was loaded with 6.5 GB of RAM. The computers were each connected to a Netgear GS108 8-port gigabit switch via Ethernet cables.

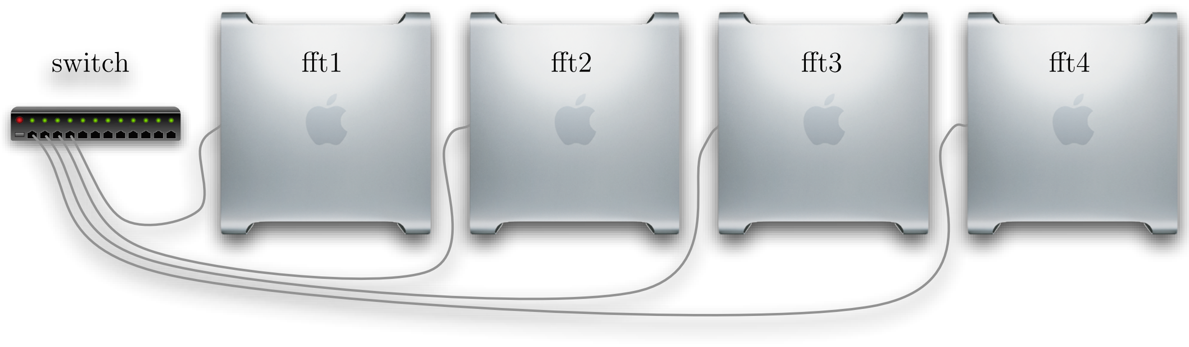

Figure 5: Diagram of the "canonical cluster." This cluster consists of four PowerMac G5 computers, four Ethernet cables, an Ethernet switch, and four power cables. (Power cables not shown.)

Each computer was given a unique Computer Name. The computers were named fft1, fft2, fft3, and fft4. Each computer was configured to allow Remote Login. Each computer was set to boot without logging anyone in. The Apple Xcode Developer Tools [10], the LAM/MPI 7.0.6 [5] Mac OS X Package, and the Xgrid Technical Preview 2 [11] software with LnxMPI [7] were installed on each computer.

To set up a gigaelement-FFT system, there are hardware and software tasks, as laid out below:

Hardware/software configuration:

- Obtain and configure the hardware:
  - Obtain the computers, the memory, the switch, and the cables. Note the "canonical cluster" defined above, and used for most of the present timings.
  - Install the memory in computers.
  - Connect the computers and the switch via Ethernet cables as shown in the Figure 5.
  - Connect the switch to the public network, if desired.
  - Connect the power cables to the computers and the switch.
- Obtain and configure the software (on each computer):
  - Attach a keyboard, mouse, and monitor to the computer.
  - Turn on the computer.
  - Install Mac OS X 10.3.4 and the Apple Xcode Developer Tools.
  - Open System Preferences and select Sharing.
  - Set the Computer Name to a unique name.
  - Check the Remote Login checkbox.
  - Return to System Preferences and select Accounts.
  - Select the Login Options button.
  - Un-check the "Automatically log in as:" checkbox.
  - Download, install and configure either Xgrid (Option 1) or LAM/MPI (Option 2); see below for details.
  - Obtain and configure the `dist_fft` software.
  - Shutdown computer.
  - Detach keyboard, mouse, and monitor.

You do not need to install both LAM/MPI and Xgrid. You may choose which software package you prefer to use. LAM/MPI is appropriate if you are most comfortable with the command-line interface, otherwise Xgrid is appropriate.

Option 1 (Xgrid option; see below for LAM/MPI option):

- Configure Xgrid:
  - Open System Preferences and select Xgrid.
  - Configure the agent password and the controller to which to bind.

  - Turn on the agent on each computer.
  - Configure the controller passwords.
  - Turn on the controller on a *single* computer.

- Compile the source code:
  - Open the `dist_fft` Xcode project file and click the Build button; or,
  - Use the `cc` compiler:

    ```
    fft1:~/Projects/dist_fft admin$ cc -o dist_fft_test \
        -ILnxMPI -framework Accelerate \
        -O3 -faltivec -fstrict-aliasing \
        -mcpu=G5 -mtune=G5 -mpowerpc64 \
        LnxMPI/LnxMPI_S.c *.c
    ```

- Run the timing application with Xgrid:
  - Open the Xgrid application.
  - Connect to a controller.
  - Create a new MPILaunch job.
  - Select the `dist_fft_test` executable.
  - Enter the number of processors you want to use.
  - Click the Submit button.

Option 2 (LAM/MPI option; see above for Xgrid option):

- Configure LAM/MPI:
  - Create identical private and public `ssh` identity keys on each computer and add the public key to the `authorized_keys` files:

    ```
    fft1:~ admin$ ssh-keygen -t dsa
    fft1:~ admin$ cp .ssh/id_dsa.pub .ssh/authorized_keys
    fft1:~ admin$ scp -r .ssh fft2.local.:
    fft1:~ admin$ scp -r .ssh fft3.local.:
    fft1:~ admin$ scp -r .ssh fft4.local.:
    fft1:~ admin$ scp .ssh/known_hosts fft2.local.:.ssh
    fft1:~ admin$ scp .ssh/known_hosts fft3.local.:.ssh
    fft1:~ admin$ scp .ssh/known_hosts fft4.local.:.ssh
    ```

  - Create a `bhosts` file the specifies the names of the computers and the number of CPUs in each computer:

```
fft1.local. cpu=2
fft2.local. cpu=2
fft3.local. cpu=2
fft4.local. cpu=2
```

- Compile the source code:
  - Use the `mpicc` compiler:

```
fft1:~/Projects/dist_fft admin$ cc -o dist_fft_test \
    -framework Accelerate \
    -O3 -faltivec -fstrict-aliasing \
    -mcpu=G5 -mtune=G5 -mpowerpc64 \
    *.c
```

- Run the timing application with LAM/MPI:
  - Run the `lamboot` command passing the path of the `bhosts` file:

```
fft1:~ admin$ lamboot bhosts
```

  - Use `mpirun` to run the application across the cluster, one process per CPU:

```
fft1:~ admin$ mpirun -s n0 C Projects/dist_fft/dist_fft_test
```

Before being used for timings the computers should first be restarted to ensure that the most free memory possible will be available. Immediately following the restart a simple application should be run on each CPU in order to exercise the memory allocation systems of Mac OS X. The memory-exercising application is described in Section 6.

For more information on configuring and using the `dist_fft` software see the documentation included with the `dist_fft` project. For more information on running LnxMPI-based programs with Xgrid see the release notes included with Xgrid.

5 Software overview

The software created for this paper for the purpose of performing a distributed FFT is called `dist_fft`. The `dist_fft` project was designed with the goal of optimizing for use with Mac OS X and G5. We wanted to put together a lightweight solution that made effective use of existing system services.

The software is based around the distributed transpose algorithm and implementation from the Fastest Fourier Transform in the West (FFTW) project [4]. All of the distributed transpose code comes directly from MPI FFTW code [8]. The basic functionality provided by the MPI FFTW code is to perform a 2-dimensional transpose of a matrix whose elements are distributed across many processes. Three source files, along with headers, were copied from the FFTW project to the `dist_fft` project; `sched.c`, `TOMS_transpose.c`, and `transpose_mpi.c`. The one major change we made to the distributed transpose code was to replaced the original out-of-place rectangular matrix transpose function with a call to the equivalent function in the vDSP library, resulting in significant performance benefits.

For calculating row FFTs the software makes exclusive use of the vDSP library from the Accelerate framework. The Accelerate framework is included with Mac OS X 10.3 and later and includes signal processing functions that are optimized for Altivec and G5.

The `dist_fft` software performs distributed 1-dimensional, 2-dimensional, and 3-dimensional FFT computations. The 1-dimensional FFTs are subject to the constraint $W = 4^k$ for integer k in the range $2..15$. The 2-dimensional FFTs are subject to the constraint $W = H = 2^k$ for integer k in the range $2..15$. The 3-dimensional FFTs are subject to the constraint $W = H = L = 2^k$ for integer k in the range $2..10$.

The software has two compile-time options that affect the computation performed; real numbers may be stored as either single-precision or double-precision floating-point numbers, and complex numbers may be stored in split format or interleaved format.

Complex numbers are represented by a pair of real numbers, called the real part and the imaginary part. In split format, the real part and the imaginary part are stored in separate memory buffers. In interleaved format, the real part and the imaginary part are stored adjacent each other in the same memory buffer.

The software also has a run-time option to use a separate “workspace” memory buffer. This buffer used during the network transpose for very large W. When using the workspace the transform itself remains in-place. The workspace is used as an intermediate scratch space. For split-format the workspace is half the size of the signal. For interleaved-format the workspace is the same size as the signal.

It was found that for the sizes we were interested in the most effective options were to use single-precision split-format complex numbers and a workspace.

6 Software implementation

Many modifications were made to the basic distributed FFT algorithms during the implementation and optimization of the software. The algorithm implementations are described in this section by pseudo-code procedures that take three arguments; the signal, the order of the input signal, and the desired order of the output signal.

The 1-dimensional algorithm can accept and output signals in any combination of "column-wise" and "row-wise" order. Row-wise order is the order you get if you simply read the 1-dimensional signal into memory, in order. Column-wise is the order you get after treating the 1-dimensional signal as if it were a 2-dimensional signal and transposing it.

The 1- and 3-dimensional algorithms can accept and output signals in any combination of "row-major" and "column-major" order. Column-major order was described in Section 3. As mentioned before, the multidimensional algorithms we describe in this paper work equally as well for signals in both row- and column-major order, and it is merely for notational convenience that we assume that the signal is in column-major order.

We begin with the implementation of the 2-dimensional algorithm. Very little differs from the basic algorithm described before. The first modification we made in the implementation is that we "prefetch" each row before performing the FFT. We empirically determined that for large widths the FFTs of rows ran faster if we first walked through the components of the elements of each row and loaded the contents of each location. We conjecture that by walking through the rows sequentially prior to computing the FFTs, we were able to get the rows loaded into the G5's caches more efficiently than the FFT computation alone was able to. See the description of the implementation of the 1-dimensional algorithm for more information on how we first discovered the positive effects of prefetching.

The second modification we made was to make the final distributed transpose optional by allowing the output signal to be in the opposite (row- vs. column-major) order as the input signal.

```
Distributed-Two-Dimensional-FFT(signal, input-order, output-order)
1  for each row in signal
2        do Prefetch(row)
3           FFT(row)
4  Distributed-Transpose(signal)
5  for each row in signal
6        do Prefetch(row)
7           FFT(row)
8  if input-order = output-order
9     then Distributed-Transpose(signal)
```

The implementation of the 1-dimensional algorithm is similar to the 2-dimensional algorithm, as described before, with the addition of the twiddle step. The Twiddle procedure has been vectorized by hand for maximum performance. The algorithm implementation follows the prescription for the twist operation described in [1].

We hypothesized that we would make better use of the caches if we twiddled each row immediately before or immediately after transforming each row, rather than doing all the twiddles before all of the transforms, or doing all of the transforms before all of the twiddles. We were astonished to discover that the total time it took to perform the twiddle of a row followed immediately by computing the transform of that row was *less* than the time it took to simply compute the transform of that row without performing the twiddle at all. Thus we implemented the algorithm so that we twiddled after the first distributed transpose, immediately preceding the second transform, rather than twiddling before the transpose.

By distilling out the functionality of the twiddle that resulted in improved transform performance, we then added the aforementioned prefetching step immediately before the row transform that wasn't preceded by the twiddle.

For our final modification of the 1-dimensional algorithm we make the initial and final distributed transposes optional. We make the initial distributed transpose optional by allowing the input signal to be in column-wise order and we make the final distributed transpose optional by allowing the output signal to be in column-wise order.

```
Distributed-One-Dimensional-FFT(signal, input-order, output-order)
 1  if input-order ≠ Column-Wise
 2     then Distributed-Transpose(signal)
 3  for each row in signal
 4        do Prefetch(row)
 5           FFT(row)
 6  Distributed-Transpose(signal)
 7  for each row in signal
 8        do Twiddle(row)
 9           FFT(row)
10  if output-order ≠ Column-Wise
11     then Distributed-Transpose(signal)
```

The implementation of the 3-dimensional algorithm is somewhat more intricate. The distributed transpose library we used only supported transposing the last two dimensions, and we only have enough of the signals in each process to transpose the first two dimensions locally in each process. Thus to effect a transpose of the the first and last dimension we must perform a local transpose followed by a distributed transpose followed by one more local transpose.

To transpose the 1st and 2nd dimensions, no interprocess communication is required. Each process simply does an in-place transpose of each plane of rows and columns it is holding. The Transpose procedure has been hand vectorized for maximum peformance. An additional optimization we made was to transpose each plane of rows and columns immediately after transforming the rows of the plane, rather than transforming all of the rows of all of the planes, and then transposing all of the planes.

As before, we make the final transposes optional by allowing the output signal to be in the opposite (row- vs. column-major) order as the input signal. Unlike before, we did not add prefetch steps because they were not observed to improve the performance of the algorithm.

```
Distributed-Three-Dimensional-FFT(signal, input-order, output-order)
 1  for each plane in signal
 2       do for each row in plane
 3              do FFT(row)
 4          Transpose(plane)
 5  Distributed-Transpose(signal)
 6  for each plane in signal
 7       do for each row in plane
 8              do FFT(row)
 9          Transpose(plane)
10          for each row in plane
11              do FFT(row)
12  if input-order = output-order
13     then for each plane in signal
14              do Transpose(plane)
15          Distributed-Transpose(signal)
16          for each plane in signal
17              do Transpose(plane)
```

We made one additional implementation modification when using split-format data that applies to all of the algorithms. Since the complex data is split into two separate buffers for real and imaginary components, when we perform the distributed transpose we first perform a distributed transpose of the real components followed by a distributed transpose of the imaginary components. By doing these operations sequentially the workspace is allowed to be only half the size of the signal when the data is in split format.

The implementation of Distributed-Transpose procedure comes from MPI FFTW. A full description of the algorithm is beyond the scope of this paper but we note that this is the one step that requires interprocess communication. The Distributed-Transpose implementation uses the Message Passing Interface (MPI) to specify which data to send to the other processes.

We mentioned before that we used an application to reliably put the computer into a warmed-up state after a reboot. The memory-exercising application had a ten-iteration loop. During each iteration it would allocate two buffers, one 2^{30} bytes, and the other 2^{29} bytes, then write zero to every byte of each buffer, and then deallocate the buffers. The memory-excercising application was created because it was observed that the first few large allocations and deallocations of memory by a process running immediately following a restart resulted in continuous background kernel activity that reduced the performance of the computations for a period of a few minutes. This background activity caused the timings of computations taken while the computer was in the initial slower state to differ significantly from the timings of the computations taken in the later faster state. In order to achieve consistent results across reboots, a reliable method was devised for putting the computers into the faster state immediately following each reboot.

7 Performance measurements

Measurements of performance of the software were obtained by adding to the software appropriate wall-timing code. After each stage the software printed out the time elapsed for that stage. At the end of the transforms, the total time was printed, as well as the average time. The software was configured to perform a forward transform followed by a inverse transform, and repeat, for some number of iterations. The timings reported here are average over both directions of transform, and average over some number of iterations.

We begin by reporting the best average times measured on the cluster described in Section 4 of this paper. For single- and double-precision, split-complex data of total element count $N = 2^{30}$, with a workspace, the average real-times appear in Table 2.

Table 2: Average time per FFT and throughput for the canonical 4-computer cluster.

signal dimensions	precision	time per FFT (sec)	throughput (Gflops)
2^{30}	single	35.4	4.5
$2^{15} \times 2^{15}$	single	35.1	4.6
$2^{10} \times 2^{10} \times 2^{10}$	single	43.5	3.7
2^{30}	double	85.8	1.9
$2^{15} \times 2^{15}$	double	86.4	1.9
$2^{10} \times 2^{10} \times 2^{10}$	double	120.8	1.3

The 2-dimensional single-precision transform was the fastest of the six, achieving 4.6 Gflops of throughput.

We also ran the software on an eight-computer cluster connected via Myrinet instead of Ethernet and used the vendor-supplied MPI library and IBM's XLC compiler. The computers were dual-processor 2.0 GHz Xserve G5s with 3.5 GB of memory per machine.

For single- and double-precision, split-complex data input data of total element count $N = 2^{30}$, with a workspace, running on four and eight computers, the average real-times appear Table 3. Note that the cluster computers did not have enough memory each to compute the double-precision transform on four computers.

Table 3: Average time per FFT and throughput for the Myrinet-connected cluster.

signal dimensions	precision	# of machines	time per FFT (sec)	throughput (Gflops)
2^{30}	single	4	30.9	5.2
$2^{15} \times 2^{15}$	single	4	30.7	5.2
$2^{10} \times 2^{10} \times 2^{10}$	single	4	40.1	4.0
2^{30}	single	8	18.0	8.9
$2^{15} \times 2^{15}$	single	8	17.9	8.9
$2^{10} \times 2^{10} \times 2^{10}$	single	8	22.4	7.1
2^{30}	double	8	43.1	3.7
$2^{15} \times 2^{15}$	double	8	42.3	3.8
$2^{10} \times 2^{10} \times 2^{10}$	double	8	59.7	2.7

Using four computers the Myrinet-connected cluster was about 10% faster than the canonical cluster for single-precision FFTs. The fastest transform on the Myrinet-connected cluster was the 2-dimensional single-precision FFT, achieving 8.9 Gflops of throughput on eight computers, and running about 70% faster on eight computers than on four computers.

8 Accuracy of massive FFTs

It is natural to wonder whether massive signal lengths lead to unacceptable numerical error in the FFT result. A related, but perhaps more important question is: What is the expected numerical error in an FFT-based convolution (correlation)? This question leads to intricate numerical and theoretical analysis; indeed, though some questions have been satisfactorily solved (see literature references) there remain to this day open problems in regard to FFT error. In what follows, we give the briefest of tours of some of the rules of thumb, in the form of "expected" errors and "worst-case" errors.

We concentrate on 1-dimensional FFTs; the results for 2- and 3-dimensional FFTs are quite similar. A first observation: It is a happy fact that a length-W FFT having complexity $O(W \log_2 W)$ is generally *more* accurate than a literal DFT of complexity $O(W^2)$. The reason for this is simple: The FFT involves fewer operations. If we think of the sums (2.1) as amounting to a random walk in W dimensions with W steps, caused by random elements s_x (say, uniformly random across the unit complex circle) then we expect the magnitude error $\mathcal{E}$ for a single literal DFT to be

$$\mathcal{E} \approx \; c\, \epsilon\, \sqrt{W},$$

where ϵ is the machine precision, e.g. $\epsilon = 2^{-53}$ for double-precision (IEEE 754/854 compliant), and c is a constant, roughly $c \approx 1$.

Second, an FFT thought of as a random walk of length $\log_2 W$ in W dimensions should have:

- Magnitude error $\mathcal{E} \approx c_1\ \epsilon\ \sqrt{\log_2 W}$, for random (unit) input;
- Magnitude error $\mathcal{E} \leq c_2'\ \epsilon\ \ \log_2 W$, for conspiratorial (unit) input.

Here, c_1, c_2 are constants (very roughly 1 or 2), and by "conspiratorial" input we mean input that is not random, rather conspires to give worst-case error. Thus for example, a single-precision FFT of length 2^{30} could have an error amounting to about 6 bits lost on the FFT output values.

Third, and again perhaps the most important set of observations, convolution involving FFT steps has much more significant error than a single FFT. If all initial elements for a convolution of two length-W signals all lie in a complex circle of radius r, it can be shown that for constants $c_3, c_4 \approx 2$ (again very roughly) the error $\mathcal{F}$ in a single, final convolution element is approximated and bounded respectively as

- Convolution error $\mathcal{F} \approx c_3\ \epsilon\ r^2 \sqrt{W \log_2 W}$, for random digits in complex r-circle,
- Convolution error $\mathcal{F} \leq c_4\ \epsilon\ r^2\ W \log_2 W$, worst-case,

using the methods of [9]. But this means, for double-precision-float convolution of initial *integer* digits all in $[-b/2, b/2)$, one has

$$\mathcal{F} \approx 2^{-53} \cdot b^2 \cdot 2^{30} \cdot 30 \approx 2^{-18} b^2,$$

so that convolution errors accrue in the (necessarily) integer final ouputs if b has more than about 9 bits. This notion of integer signal elements and final integer data is not specious; in fact, many modern computation number theory efforts involve just this type of convolution; see [9] [2], where more precise results, graphs of "expected" and "worst-case" errors appear, and derivations for the constant factors c_i are presented. Incidentally, problems involving signals as large as $W = 2^{33}$ are contemplated for large prime-search and factoring projects on the theme that even double-precision does not allow very large integer digits of size b for exact, integer convolution.

Let us close this section, acknowledging the heuristic nature of the approximations above, by summarizing in terms of useful rules of thumb.

Rules of thumb for massive-FFT accuracy:

- For a solitary, gigaelement FFT (either single- or double-precision):
  - For random input data, roughly 3 bits of error in the float mantissa expected.
  - Worst-case, conspiratorial input data could lose more like 6 bits.
- On the other hand, for gigaelement convolution with double-precision floats (single-precision convolution tends to be risky in the gigaelement region, for reasons given), and for input data sets of size-b elements, the magnitude error in the standard, three-FFT convolution should be roughly, for each final convolution element:
  - $\mathcal{F} \approx 2^{-38}b^2$, for random input elements; so, $(38 - 2\log_2 b)$ good mantissa bits.
  - Roughly $\mathcal{F} \leq 2^{-18}b^2$ worst-case; so, $(18 - 2\log_2 b)$ good mantissa bits.

9 Extensions and future research

It should be fairly clear how to generalize to other dimension values, for example to 1-dimensional FFTs of length not a power-of-four, 2-dimensional FFTs with $W \neq H$, or 3-dimensional FFTs with no two box sides equal. What is not so clear, and might give rise to some good research, is how 2- and 3-dimensional FFTs might be related, to advantage, by twiddle-matrix methods, just as the 1- and 2-dimensional cases already are.

The software already supports larger element counts N than the limit 2^{30} previously mentioned, up to a theoretical limit of $N = 2^{60}$, although we remind ourselves of the machine and memory constraints laid out in Table 1.

However, beyond such geometrical considerations, there are certain directions for optimization. One such is to exploit the very common scenario in which the input data is pure-real. Then a real-signal transform can be used at each step, with intricate combinatorics but a tremendous reward: Run times can be cut almost in half. Even network communication times can be brought down because of less data. The details of real-signal, Hermitian-symmetric transforms for 2 dimensions are discussed in [1]. Generalization to 3 dimensions is, again, intricate but should bring almost a factor of 2 in overall speed.

Another avenue for increasing throughput is through changing the hardware. Faster processors should reduce the time for each FFT, as should faster memory. Faster network interconnects, such as Myrinet and Infiniband, also may reduce the time for each FFT, as demonstrated in the previous section. Even if we hold the speed of the computers and network constant, simply increasing the number of computers should reduce the time for each FFT, as long as a large enough network switch is used.

For the future, it would be good to generate comprehensive data on performance vs. machine-count M. Table 3 of Section 7 is an initial glimpse into variation on M: In spite of our concentration on $M = 4$ as a canonical configuration, we notice that $M = 8$ offers faster FFT speed, and of course, allows more total memory.

Another interesting development direction would be the creation of an Xgrid plug-in dedicated to massive transforms—not just FFTs but wavelet transforms, and so on.

10 Acknowledgments

The authors thank D. Bailey, A. Barty, J. Buhler, H. Chapman, I. Ollman, E. Prabhakar, A. Sazegari for research discussions, and E. Mayer and C. Percival for their theoretical results in regard to FFT numerics. We thank M. Frigo and S. Johnson for their wonderful implementation of the FFTW library.

References

[1] R. Crandall and J. Klivington, "Supercomputer-style FFT Library for PowerPC G4," 2000, `http://www.apple.com/acg/`

[2] R. Crandall, E. Mayer, and J. Papadopoulos, "The twenty-fourth Fermat number is composite," *Math. Comp.* **72**, 2003, 1555-1572.

[3] R. Crandall and C. Pomerance, *Prime numbers: A computational perspective*, Springer-Verlag, New York, 2001.

[4] FFTW, `http://www.fftw.org/`

[5] LAM/MPI, `http://www.lam-mpi.org/`

[6] LCLS, `http://www-ssrl.slac.stanford.edu/lcls`

[7] LnxMPI, `http://exodus.physics.ucla.edu/appleseed/dev/developer.html`

[8] MPI FFTW, `http://www.fftw.org/fftw2_doc/fftw_4.html#SEC55`

[9] C. Percival, "Rapid multiplication modulo the sum and difference of highly composite numbers," *Math. Comp.* **72**, 2002, 387-395.

[10] Xcode, `http://developer.apple.com/tools/macosxtools.html`

[11] Xgrid, `http://www.apple.com/acg/xgrid/`