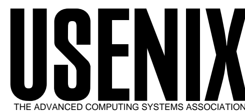
USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000



© 2000 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

The Development and Integration of a Distributed 3D FFT for a Cluster of Workstations

Dr. Christopher E. Cramer
Duke University
cec@ee.duke.edu, http://www.ee.duke.edu/~cec
Dr. John A. Board
Duke University
jab@ee.duke.edu, http://www.ee.duke.edu/~jab

Abstract

In this paper, the authors discuss the steps taken in the formulation of a parallel 3D FFT with good scalability on a cluster of fast workstations connected via commodity 100 Mb/s ethernet. The motivation for this work is to improve the performance and scalability of the Distributed Particle Mesh Ewald (DPME) N-body solver. Scalability issues in the FFT and DPME as an application are presented separately. Also discussed are scalability issues related to the networking hardware used in the cluster. Results indicate that the existence of a parallel FFT significantly improves performance in DPME from a maximum of 5 processors to at least 24 processors on a cluster of workstations. This has an associated increase in speedup from 4 to 12 times faster than the serial version.

1 Introduction

The Fast Fourier Transform (FFT) has many properties useful in both engineering and scientific computing applications (examples include convolution in real space becoming multiplication in Fourier space, audio compression and every DSP technique known to man). Because of these properties it has become a standard tool in many fields and efficient implementations are a subject of much research interest.

Also of increasing research interest is the subject of Cluster Computing. These clusters of workstations are capable of achieving the performance of traditional supercomputers (on certain problems) at a significantly reduced cost. Such clusters are often loosely coupled groups of machines which communicate using commodity networking hardware. Networking is often 100 Mb/s ethernet, although 1 Gb/s ethernet and proprietary solutions such as Myrinet are becoming more common.

Unfortunately, while there exist many vendor implementations of high speed parallel FFTs for tightly coupled traditional supercomputers, there are few parallel implementations available for clusters of workstations. In our work, we had a need for a *portable and efficient* parallel FFT. Not finding one that fit both of our criteria, we decided to implement our own. It was found that the use of this parallel FFT in our code allowed for a great deal of improvement in its performance.

2 Parallel FFTs

The problem with a parallel FFT is that the computational work involved is $O(Nlog_2N)$ while the amount of communication is $O(N)$. This means that for small values of N (we are targeting 64x64x64 3D FFTs), the communication costs rapidly overwhelmed the parallel computation savings. While many tightly coupled parallel processing machines (Cray, SGI Octane, etc) have customized parallel FFT routines, there are few implementations written for a cluster of workstations (i.e. written in PVM or MPI).

Other researchers, when facing this problem, have resorted to using a naive DFT. So long as one is willing to have multiple processors each keep a copy of

the space upon which the DFT is to be performed, the DFT is highly scalable in that the results for any given point in the space do not depend on earlier computation that may have been performed on another processor. While this approach can achieve not only speed up, but also performance improvements over a serial FFT, it seems aesthetically unpleasing and if an efficient, portable, parallel FFT is available, it might not be the best use of processor resources.

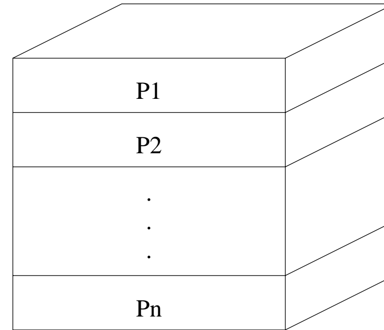


Figure 1: Processor representation of the 3D data space.

2.1 FFTW's MPI-based FFT

One of the faster, yet still portable and freely available, implementations of the FFT is FFTW ([FJ99]). Due to it's portability, we have made use of the library in our own applications.

The latest versions of the FFTW library do have a parallel FFT using MPI for interprocess communications. While our application (DPME) is written using PVM, this would not have been a serious impediment as we were already considering an MPI port of DPME. Unfortunately, the FFT did not scale for problem sizes we are interested in (except for very large FFTs, 256x256x256, the speedup was less than 1). This was even true when using the "transposed" FFT which has significantly less communication (see below for a discussion of transposed FFTs).

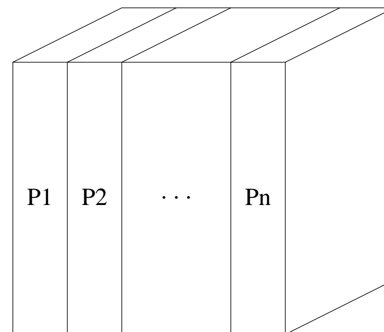


Figure 2: Processor representation of the 3D data space after 2D FFTs and communication step.

2.2 Our Parallel FFT

It was then decided that we needed to create our own parallel FFT which we based on FFTW's sequential FFT libraries. Our FFT was originally written in MPI, but was also ported to PVM for easy incorporation into (the then current version of) DPME. The FFT starts with the assumption that each processor contained one slab of the 3D data space (see Figure 1).

Each slab of the data space contains (for now, exactly) $m = N/P$ 2D slices. Each processor computes a single 2D FFT on each of its m slices, sending the results to all other processors using nonblocking communication. After each processor has received all of the data sent to it, the data is scattered on the processors as seen in Figure 2.

Finally, each processor performs $m * N$ 1D FFTs in the final dimension, yielding the 3D FFT. Note that unless an additional communication step is performed, the resulting data is on different processors than was the original data. This is known as a transposed FFT. In this case, performing either a forwards or backwards FFT results in the Most Significant and the Second Most Significant axes being swapped. So, if the original data layout was stored in ZYX major order, with the data along the Z axis being partitioned to different processors, either a forwards or backwards FFT would result in a changed data layout to YZX major order with the Y axis partitioned to the different processors. The algorithm was designed this way in order to minimize communication, while still being symmetric. In other words, one may call the forward or backward FFT first and have the opposite call put the data back in place.

3 Testbed Configuration

All of the experimental results given in this paper were computed on a cluster of 16 dual processor machines. The processors are Intel Pentium II running at a clock speed of 450 MHz. Each node has 512 MB of RAM as well as an additional 512 MB of virtual memory. The machines are connected using a private 100 Mb/s Ethernet Cisco 2924 switch. All nodes are running the GNU/Linux operating system with kernel release 2.2.12 compiled for SMP.

No kernel patches have been applied to increase network performance as it was found that machines were already capable of 95% of wire speed bandwidth. Kernel patches designed to reduce latency might increase performance slightly at the cost of reducing cluster maintainability.

The parallel programming of this cluster has been performed using both PVM and MPI. PVM ([GBD+94]) results are given using version 3.4.3. MPI ([SOHL+96]) results were found using LAM MPI version 6.3.1.

4 FFT Results

The FFT algorithm described above performs very well provided that one can make use of transposed data in Fourier space. Fortunately, DPME's computation of the electrostatic potential in Fourier space can easily accommodate transposed data. Speed-up results (as compared to a single call to the sequential FFTW 3D FFT routine) for this algorithm are given in Figure 3 and Figure 5 (Figures 4 and 6 give the respective timings). As can be seen from the figures, the method achieves reasonable efficiency for up to 12 processors. The sudden degradation after 12 processors in performance will be discussed in a subsequent section.

The algorithm has also been tested on the IBM SP2 at the North Carolina Super-Computing Center (NCSC). As shown in Figures 7 and 8, the algorithm exhibits moderate speed-up for small FFTs (64x64x64) and good speed-up for medium sized FFTs (128x128x128) - 70% efficiency for up to 32 processors.

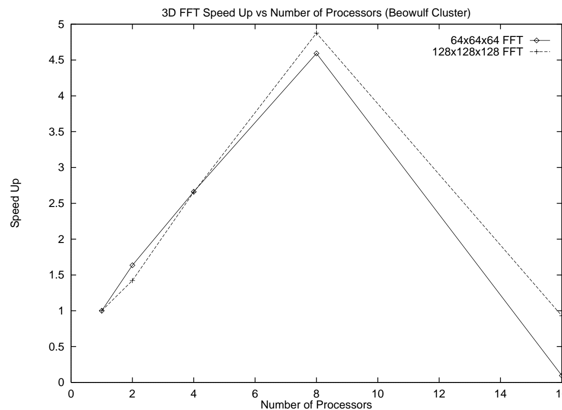


Figure 3: 3D FFT performance on the Duke ECE Beowulf Cluster - speedup

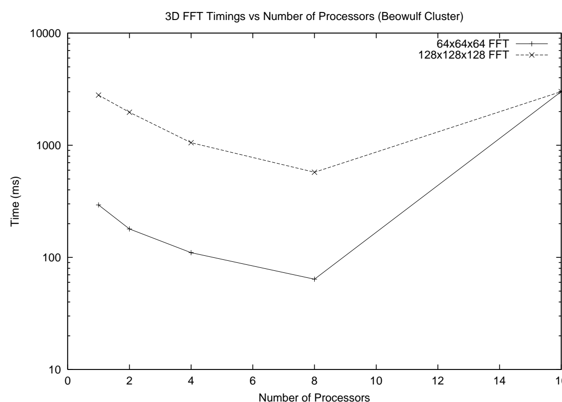


Figure 4: 3D FFT performance on the Duke ECE Beowulf Cluster - timings

4.1 Algorithmic Comparison

The speed up of our algorithm is significantly greater than FFTW's parallel FFT. This is primarily due to the way communication is handled in each algorithm. FFTW has each processor perform all of its 2D FFTs in a single FFTW library call. Then there is a blocking communication step and finally the remaining 1D FFTs are performed. While this decision makes sense in that FFTW achieves its greatest efficiency when performing multiple FFTs, it does not address the real problem in a cluster of workstations: communication. All of the communication in our algorithm is non-blocking. Furthermore, by interleaving communication with computation, we are able to hide a greater degree of the communication overhead involved.

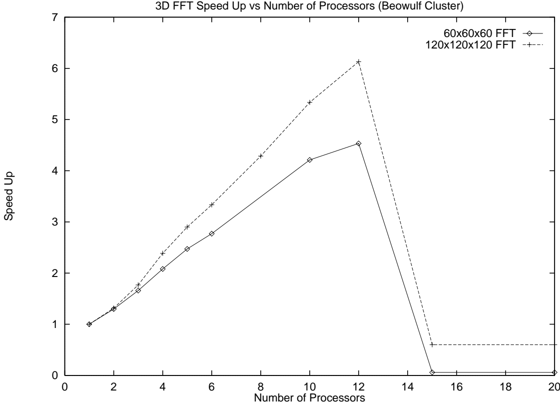


Figure 5: 3D FFT performance on the Duke ECE Beowulf Cluster - speedup

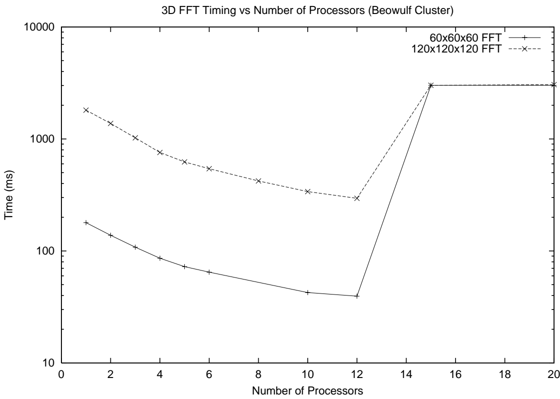


Figure 6: 3D FFT performance on the Duke ECE Beowulf Cluster - timings

4.2 Hardware Considerations

Early results of our parallel algorithm demonstrated scalability only up to four processors. After that point, speedup dropped significantly. Surprised at the suddenness of the scaling drop-off, we began investigating whether the reduced scalability was due to our algorithm or the specific hardware being used.

At the time, the cluster used an Intel 510T 100 Mb/s switch. It was decided to try replacing this switch with a Cisco 2924. After the replacement, we saw improved scalability of the FFT algorithm up to 12 processors. These results were well correlated with those presented by Mier Communications, Inc ([Mie98]) for the number of simultaneous 100 Mb/s full-duplex streams that could be supported by the various switches without dropping packets. It should be noted that the Mier results

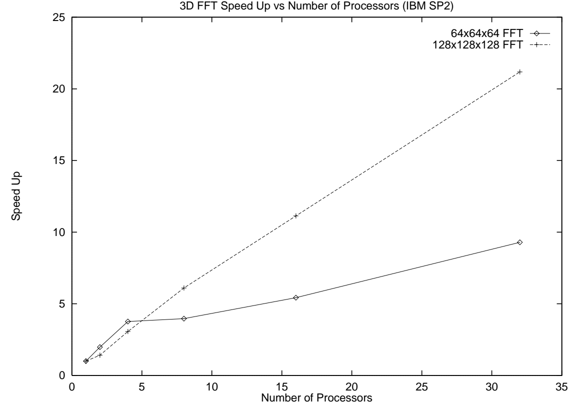


Figure 7: 3D FFT performance on the IBM SP2 - speedup

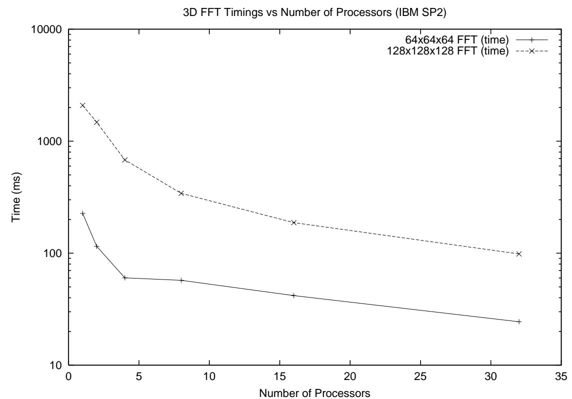


Figure 8: 3D FFT performance on the IBM SP2 - timings

were for the Cisco 2916 and the Intel 510T. However, the Cisco 2916 and 2924 have similar backplane architectures.

The Mier Communications tests also included a comparison with the Bay 350. We then tried using a switch in the same family, the Bay 450. Again, the scaling results on our algorithm were well correlated with the number of simultaneous 100 Mb/s full-duplex streams that the switch backplane could handle without dropping packets.

It is uncertain why the switches perform so differently. The backplanes of each switch are all rated at over 2 Gb/s (Intel - 2.1 Gb/s, Bay - 2.5 Gb/s, Cisco - 3.2 Gb/s). Latency also does not appear to be a factor as all of the switches have minimum latency times of $10\mu s$ or less. The most likely reason behind the differences is that the backplane architecture of the various switches results in the Intel switch drop-

ping packets under full load. Since MPI and PVM communicate using TCP/IP protocols, this results in the packet being resent.

Therefore, the strong correlation between the number of simultaneous full-duplex streams (found by Mier) and our own scaling results is likely due to the nature of our algorithm which aims to fully utilize the switch architecture by having each processor send data to every other processor after each 2D slice of the data has been converted to Fourier space. So, each processor is not only sending data to all other processors, but is also receiving data from each of the other processors. The amount of data in each message depends on the size of the FFT and the number of processors being utilized. In general, it is: $B * N^2/P$ where B is the data size (for the complex doubles we are using, this is 16 bytes), N is the size along one dimension of the FFT and P is the number of processors. For a $64x64x64$ FFT on 4 processors, the messages are 16 kB in size. The number of messages is $N * P$. The number and size of these messages insures that the switch is fully utilized on the P ports. Therefore, the number of simultaneous full-duplex streams capable of being supported is very important.

Determining whether the most important factor in the algorithm's performance is the switch bandwidth or the overall latency is a fairly difficult task. The various switches we used on our local cluster all had similar latencies and had the same (theoretical bandwidths), however, they performed in vastly different manners. The IBM SP2 had a completely different switch architecture, with over 1 Gb/s bandwidth and very low latencies. However, some analysis of the amount of data being sent should give us some idea of which factor (bandwidth or latency) is the most important.

Consider a $64x64x64$ parallel 3D FFT running on 4 processors. Each 2D slice of the data is 64 kB in size. Each message (to the 3 other processors) for this slice is 16 kB in size. On a 100 Mb/s switch, the theoretical minimum time that this message could cross the network is approximately 1.3ms. Switch latencies are on the order of $10\mu s$. When the TCP stack latencies are considered, the latencies involved are still roughly an order of magnitude less than the time to transfer the message. Of course if more processors are used, if the FFTs are smaller, or if the switch bandwidth increases, the latencies inherent in the switch and the TCP stack become significantly more important.

This fact has been recognized by IBM in developing the SP2 in that there are two interfaces for accessing the switch. The standard method routes a user's network requests through the kernel to the adapter and then to the switch. This results in a throughput of 440 Mb/s primarily because of the latency of $165\mu s$. The second method bypasses the kernel and goes directly from user space to the adapter, resulting in the throughput increasing by a factor of 2.4 to approximately 1 Gb/s due in part to the latency being decreased by a factor of 6.9 to $24\mu s$. If we repeat the previous examination of bandwidth versus latency for the two SP2 network interfaces, on the first interface we find that at a speed of 440 Mb/s, each message only requires $297\mu s$ to traverse the switch. In this case, the $165\mu s$ latency is within the same order of magnitude as the transport time, causing latency to be a much greater factor than for our 100 Mb/s switches. For the second interface, the transport time is $123\mu s$ while the latency is $24\mu s$. So, on the SP2 we see that latency is more important primarily because of the maximum speed of the switch itself.

5 Application: DPME

The application for which the parallel 3D FFT was developed is Distributed Particle Mesh Ewald (DPME) which is a parallel N-body solver (with Periodic Boundary Conditions) based on the Particle Mesh Ewald (PME) method developed by Tom Darden [DYP93, EPB+95]. Ewald Summation ([Ewa21]) is a technique for finding the electrostatic potential of particles in an infinite lattice. In the method, a single cell of a crystal lattice is assumed to be infinitely replicated in all dimensions. Furthermore, periodic boundary conditions (PBC) are assumed, meaning that as particles leave the cell from one side, they enter from the opposite replication and so emerge on the opposite side of the cell.

5.1 Ewald Summation

The common trait of all Ewald methods is that they split the original problem space of electrostatic point charges into two separate problem spaces. The first, designated the real (or direct) space part contains the original point charges plus proportional charge

distributions centered at the same locations in space as each point charge, with the opposite charge. The second problem space is known as the reciprocal or Fourier space. This space contains only the negative of the charge distributions from the direct space. Therefore, adding the two problem spaces yields the original point charge distribution (see Figure 9).

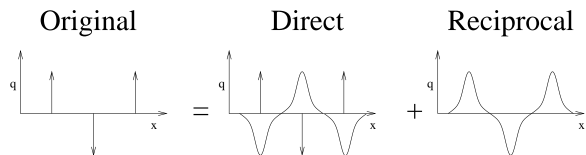


Figure 9: 1D representation of the original, direct and reciprocal problem spaces.

By choosing the appropriate amplitude for the charge distribution, the point charge plus the charge distribution will have an electrostatic potential of 0 at an infinite distance or at any distance where the charge distribution is equal to 0. For infinite distributions, the electrostatic potential can be approximated as 0 at a distance r less than infinity, with a known error bound. Therefore, to solve the direct space portion of the problem, one simply evaluates the all-pairs interaction of each point charge and charge distribution combination with all other combinations centered at a distance less than the cut-off radius.

Due to the periodic boundary conditions, the reciprocal sum is a set of N periodic functions. By solving this problem in the Fourier domain, the infinite periodic functions converge rapidly. The result can then be converted back into real space and summed with the direct space results to obtain the electrostatic potential for the original problem space.

5.2 Particle Mesh Ewald

Particle Mesh Ewald is a derivative of the Ewald method that again splits the problem space into a direct and a reciprocal space. The direct sum is solved by directly computing the interactions between all particles within each particle's cut-off radius. Assuming constant density of the particles being simulated, this is an O(N) problem.

To solve the reciprocal sum, one first discretizes the problem space as a 3D mesh. The charge functions are then interpolated onto the mesh. A 3D FFT is then performed on the mesh, transforming it into Fourier space. The electrostatic potential of the the mesh is computed (still in Fourier space). The mesh is transformed back into real space (by means of an inverse FFT). Finally, the electrostatic potentials of the original point charge locations are interpolated, based on the values in the 3D mesh.

The direct and reciprocal space electrostatic potentials can then be summed to find the electrostatic potential of the total problem space at the locations of each point charge. As was previously mentioned, solving the direct space portion of the problem is O(N). The reciprocal sum's order of complexity is: $O(Mlog(M) + p^3 * M)$ where p is the order of interpolation when converting to or from the mesh and M is the number of mesh points. The 3D FFT is $O(Mlog(M))$ and the interpolation is $O(p^3 * M)$. If M is approximately N, then the complexity of computing the reciprocal sum is $O(Nlog(N))$.

5.3 Distributed Particle Mesh Ewald

Distributed Particle Mesh Ewald (DPME) is a distributed implementation of the PME method, written by Abdulnour Yakoub Toukmaji for his Ph.D. work in Duke University's Scientific Computing research group ([Tou97]).

DPME uses a Master/Slave model for performing parallel computations. The direct sum is performed on the set of Slave processors spawned by PVM. The Master processor performs the reciprocal sum serially. This decision was primarily made due to the lack of a parallel FFT with a speed-up greater than 1 for a cluster of workstations.

DPME has several options which affect performance. The most significant of these is the Ewald Parameter (α) which is inversely proportional to the width of the Gaussian charge distribution. By adjusting the width of the Gaussian, one can shift work between the direct sum and the reciprocal sum. For example, a very narrow Gaussian charge distribution would allow for a small cut-off radius in the direct sum without incurring a large error penalty. However, a narrow Gaussian distribution would require a large number of mesh points in the reciprocal sum. Conversely, a wide charge distribution would result in a nearly uniform reciprocal sum and would therefore require few mesh points, however, the cut-off radius would have to be proportionally larger.

Since the reciprocal sum in DPME is sequential, the obvious strategy to increase its scalability is to set the Ewald parameter relatively small (a wider Gaussian charge distribution). This places most of the work in the direct sum which has a great deal of scalability. Unfortunately, there is a limit to how wide the charge distribution can be. If it is too wide, then the work in the direct sum approaches $O(N^2)$ complexity minimizing the advantages of using a larger number of processors. In general, it was found that optimal performance was achieved using 8 direct sum processors along with the 1 reciprocal sum processor. However, these results were obtained on an older generation of processors. On the cluster we are currently developing under, the performance was significantly worse (see Figure 10).

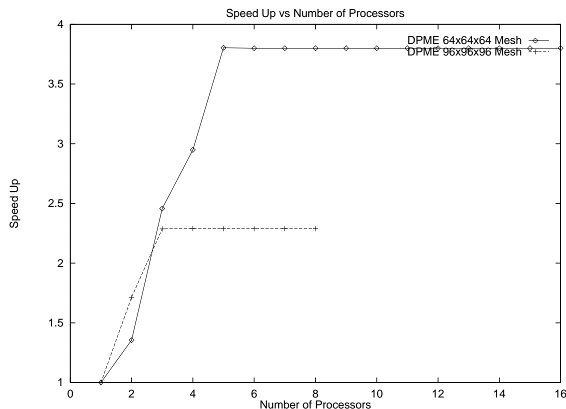


Figure 10: Original DPME speed up curves for 64x64x64 and 96x96x96 meshes

6 Application Speedup

The existence and inclusion of a parallel 3D FFT has allowed us to parallelize the remainder of the reciprocal sum in DPME. This does make it somewhat more difficult to examine speedup in DPME. Before, if you had N processors, then N-1 were dedicated to the direct sum, and one was given to the reciprocal sum. With a parallel reciprocal sum, there can be N-1 ways to divide the work on N processors, since at least one processor must be used in both the direct and reciprocal sum parts. However, since the reciprocal sum is still less efficiently parallelized than the direct sum, it is possible to begin with a single reciprocal sum processor and increase the number until the optimal operating point is found. The following results were computed by taking the desired number of processors and dividing them into direct and reciprocal processors as discussed above until the optimal partition was found. This optimal point is then given as the timing for the number of processors. Performance results for DPME with the parallelized reciprocal sum as compared to the serial reciprocal sum are given in Figure 11 and Figure 12.

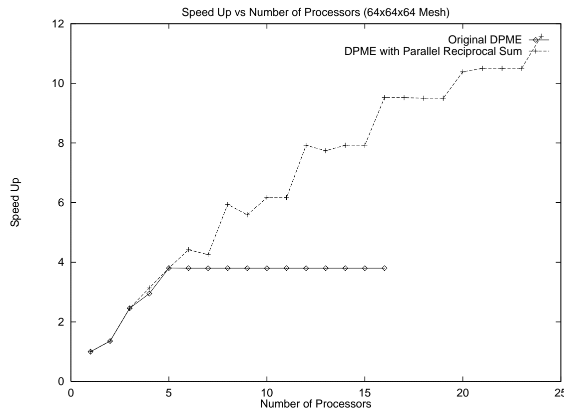


Figure 11: Original DPME speed up versus that of DPME with a parallel reciprocal sum (64x64x64 point mesh)

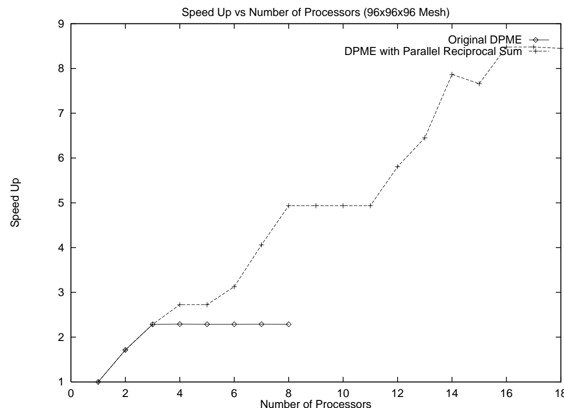


Figure 12: Original DPME speed up versus that of DPME with a parallel reciprocal sum (96x96x96 point mesh)

7 Conclusions

In this paper, we have described the formulation of a parallel 3D FFT capable of achieving speed up on a cluster of workstations connected via 100 Mb/s ethernet. By incorporating this parallel FFT, we have been able to increase the scalability of DPME from a maximum of 5 processors to a maximum of

24 processors, with a corresponding increase in the speedup from 4 to 12.

The FFT itself is general purposed and unlike much other work in the field, it will achieve reasonable speed up on a cluster of workstations. It should give some hope to people contemplating the massively scalable DFT as a replacement for FFT code.

8 Current and Future Work

Currently, we are working on or planning to work on several more improvements in the FFT code for DPME. One area is whether there is an advantage to be had in writing the FFT code to exploit two levels of parallelism on SMP machines: inter-node and intra-node. FFTW does have a thread-based version capable (in our tests) of a speed-up of 1.8 by using two SMP processors on a single node. So, rather than using 8 processes on 4 nodes (8 processors) where each processor had a portion of the data that it operated on, it might prove better to divide the data amongst the 4 nodes. Each node could then put its two processors to work on performing the computation faster. Hopefully, this would shift the operating point of the FFT algorithm to a place where the communication was more efficient (fewer communication calls with more data per call).

Another place where the FFT code might be optimized is in the amount of work done per communication call. As our algorithm stands now, each 2D slice of the data has a 2D FFT performed on it and it is then divided and sent to all other processors. It might prove to be more efficient to perform multiple 2D FFTs before sending the data to the other processors. Of course, if all slices were computed before sending the data, then our algorithm would be identical to the FFTW parallel algorithm. However, there may be an optimal point somewhere between 1 and all slices at a time.

9 Acknowledgments

We would like to thank NIH for funding work on DPME. We would also like to the North Carolina Super-computing Center for grants of time on the Cray T3D and IBM SP2.

References

[DYP93] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh Ewald: An n*log(n) method for Ewald sums in large systems. *Journal of Chemical Physics*, 98(12):10089–10092, 1993.

[EPB+95] Ulrich Essmann, Lalith Perera, Max Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[Ewa21] P. Ewald. Die berechnung optischer und elektrostrischer gitterpotentiale. *Anals of Physics*, 64:253, 1921.

[FJ99] Matteo Frigo and Steven G. Johnson. *FFTW User's Manual*, 2.1.2 edition, 1999.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. Massachusetts Institute of Technology, 1994.

[Mie98] Product lab testing comparison. Technical report, Mier Communications, Inc., 1998.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[Swa82] P.N. Swarztrauber. Vectorizing the FFTs. In G. Rodrigue, editor, *Parallel Computations*, pages 51–83. Academic Press, 1982.

[Tou97] Abdulnour Yakoub Toukmaji. *Efficient Methods for Evaluating Periodic Electrostatic interactions on high performance compters*. PhD thesis, Duke University, 1997.