An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes

Stefan Goedecker[1], Mireille Boulet[2], Thierry Deutsch[1]

1) Département de recherche fondamentale sur la matière condensée,
SP2M/NM, CEA-Grenoble, 38054 Grenoble cedex 9, France
2) CEA/ DAM Ile de France, DSSI/SNEC
91680 BRUYERES-LE-CHATEL

Abstract
Three dimensional Fast Fourier Transforms (FFTs) are the main computational task in plane wave electronic structure calculations. Obtaining a high performance on a large numbers of processors is non-trivial on the latest generation of parallel computers that consist of nodes made up of a shared memory multiprocessors. A non-dogmatic method for obtaining high performance for such 3-dim FFT's in a combined MPI/OpenMP programming paradigm will be presented. Exploiting the peculiarities of plane wave electronic structure calculations, speedups of up to 160 and speeds of up to 130 Gflops were obtained on 256 processors.

1 Introduction

While the peak speed of computers is increasing rapidly, it becomes also increasingly difficult to obtain a significant fraction of this peak speed in real applications [2]. This is due to an imbalance of the various components of modern computers. A well known imbalance is the discrepancy between the high processor speed and the slow memory access times. Another imbalance found on the latest generation of parallel computers is that the bisectional bandwidth is too low compared to the processing power of a single node consisting of several processors. Such an imbalance does not become visible in certain benchmarks consisting for example of dense linear algebra applications where the number of numerical operations is much larger than the number of data accessed from memory or transfered among the different nodes. Many high quality algorithm such as multigrid methods or fast transformations are however characterized by a number of operations that is proportional to the number of data items or increasing only slightly faster than this number ($N\log(N)$ for instance for N data items). Then, necessarily, both types of imbalance will hamper the performance of such algorithms.

A standard low complexity algorithm is the Fast Fourier transformation [3]. A single one dimensional FFT requires asymptotically $5N\log_2(N)$ operations if a standard radix two approach is adopted. A three-dimensional FFT of a data set consisting of N^3 data requires $15N^3\log_2(N)$ operations. Performing a single 1-dim FFT results in a highly non-local memory access pattern. By doing multiple 1-dim FFT's one can obtain stride one data access and consequently a much higher performance [5]. Multiple 1-dim FFT's can then be used as the building block for 3-dim FFT's. Strategies adapted to various computer architectures, such as vector machines, RISC type workstations and single processor node massively parallel machines were discussed in reference [2], strategies for clusters of vector computers in reference [1]. Obtaining high performance on massively parallel machines is difficult since the total amount of data to be send around is N^3 whereas the total number of operations is only slightly larger namely of the order of $N^3\log_2(N)$. Relatively simple solutions such as the one presented in reference [2] that were efficient on a well balanced machines such as a CRAY T3E are unfortunately not efficient on most massively parallel machines composed of multiprocessor nodes.

The programming paradigm proposed by the computer vendors for such machines is a combination of MPI [6] and OpenMP [7], where OpenMP is to be used to coordinate the work of the various processors on a single node and MPI to coordinate the work among different nodes connected through a high-speed network. The original idea was to use OpenMP for a fine grained parallelism on the node and MPI for a coarse grained parallelism among the nodes. Unfortunately the current OpenMP implementations have not live up to the expectations of a truly fine grained parallelism. For really high performance, OpenMP at present also requires coarse grained parallelism. Consequently OpenMP is frequently not a supplement to MPI but rather a competitor. As will be shown the standard MPI/OpenMP approach also fails for 3-dim FFT's

2 The basic algorithm

The notation of reference [2] will be taken over, i.e. small cap indices refer to untransformed real space indices and capital indices refer to fourier transformed Fourier space indices. In this way $i1, i2, i3$ denotes the initial real space data set and $I1, I2, I3$ the transformed Fourier space data set. The physical dimensions $i1, i2, i3$ and $I1, I2, I3$ can be split into partial dimension $j1, jp1$, $j2, jp2$, $j3, jp3$, and $J1, Jp1$, $J2, Jp2$, $J3, Jp3$, where the first index in each pair runs over the part of the data set that is physically located on a node and the second over the different nodes. In this way $i1, i2, j3, jp3$ denotes the initial real space data set distributed along the last dimension among the nodes. If for example $i3 = 1, 64$, and we are using 4 nodes, then $j3 = 1, 16$, $jp3 = 1, 4$, which means that each node holds 16 xy planes in its memory. Analogously $I1, J2, Jp2, I3$ denotes the Fourier space data set distributed along the second (y) dimension.

In the context of plane wave electronic structure calculations the Fourier space data set is smaller by a factor of 8 compared to the real space data set [4] due to aliasing. The upper half of the frequency spectrum is discarded after the transformation along the x, y and z axis, reducing the data set by a factor of 2 in each step.

In the following the organization of the algorithm with respect to the different nodes will be described. This part is based on MPI and is similar to the approach adopted in the CPMD electronic structure code [4]. For the moment we will assume that one node has only one processor. The OpenMP part that concerns the fact that one node consists of several processors will be introduced later.

We start with the initial real space data set distributed along the z dimension on the nproc nodes.

Input: i1,i2,j3,jp3

where $i1 = 1, n1$, $i2 = 1, n2$, jp1 $= 1, nproc$, j3 $= 1, n3/nproc$.
We first transform along the second dimension to obtain

multiple 1-dim FFT: i1,I2,j3,jp3

Since the upper half of the high frequency components is removed in electronic structure calculations we remove this part in the subsequent single processor rotation. Depending on the convention the high frequency part can either be located in the middle or in the corners of the data set. In any case I2 has then only n2/2 elements

Rotation and removal: I2,i1,j3,jp3

The same steps are repeated to transform along the x axis:

multiple 1-dim FFT: I2,I1,j3,jp3

Rotation and removal: I1,I2,j3,jp3

The previous two transformations along the x and y axis involved always x-y planes that were available on a single processor. For the transformation along the z axis data distributed among several processors are needed and hence MPI calls. The previous data set can equivalently be written as

Previous data set reformatted: I1,J2,Jp2,j3,jp3

We now switch indices by local (single processor) copying

Copy: I1,J2,j3,Jp2,jp3

The next step switches the last two indices by invoking the MPI ALLTOALL routine

MPI_ALLTOALL: I1,J2,j3,jp3,Jp2

The result can be reformatted:

Previous data set reformatted: I1,J2,i3,Jp2

Now all data for a single FFT along the z axis are on one processor and we transform to obtain

FFT: I1,J2,I3,Jp2

We could now bring this output data set into the input format form $I1, I2, J3, Jp3$, but this would require another MPI_ALLTOALL and another copy. It is preferable instead just to reorder locally to get

Copy: I1,I3,J2,Jp2

and to work then with Fourier space data that have this format and are consequently distributed among the nodes along the y axis.

To transform from Fourier space back to real space all the steps just described are run backwards

This approach has the following advantages. It requires only 1 MPI_ALLTOALL compared with 2 for a optimized general FFT [2]. As a consequence the number of single node copy operations is also reduced. The overhead of the parallel version compared to the serial version is therefore less than 20 percent. The amount of data moved around by MPI_ALLTOALL is reduced by a factor of four compared to a general FFT, since the MPI_ALLTOALL routine is only invoked at the last step (along z direction) where the data set is already shrunk by a factor of 4 (dimension n1/2 * n2/2 * n3) due to the removal of the high frequencies.

3 Timing results

The timing results presented in this section were obtained on the Hewlett Packard Compaq AlphaServer SC ES45 massively parallel computer, clocked at 1 GHertz. The dimension of the FFT was 128^3 in Fourier space and 256^3 in real space. The timing measures the following sequence of operations. A Fourier transformation of the wavefunction from Fourier space into real space, a multiplication of the wavefunction in real space with the potential and a back-transformation of the wavefunction into Fourier space. This sequence of operations was applied to 64 wavefunctions unless specified differently, but the reported timing is the average time for one single sequence. In spite of the averaging considerable fluctuations are however present in the measurements.

Table 1 shows the timing results for the MPI only implementation. The speedup on a single node is rather poor on this machine (2.91/.869=3.35) and is nearly equal to the speedup if the 4 processors are distributed among 4 different nodes. The single node speedup that was also measured on an older Compaq SC232 and surprisingly we obtained a much better speedup of 3.9. The striking feature of the result is however the following. If one uses more than one node one cannot substantially accelerate the computation by using 4 processes per node instead of using 2 or even only one. In other words, for more than 4 processes one obtains a faster execution time if one assigns only one MPI process per node instead of clustering the same number of processes on the smallest possible number of nodes (i.e. having 4 processes per node). This surprising feature is due to the the behavior of MPI_ALLTOALL routine. For its efficiency the bisectional bandwidth counts [2] and for a constant number of processes this bisectional band width is best if the processes are distributed among the largest possible number of nodes. Roughly speaking, if 4 processes are running per node they can not send their data fast enough through the single port to the network that is available on each node. The highest speed obtained in this series of measures on 128 processors was 52 Gflops.

Table 1: Timing in sec of the MPI version. Horizontally the number of used processors per node varies (the maximum being 4), vertically the number of nodes varies. The number of processors is thus the product of the entries in the top line and leftmost column.

	1	2	4
1	2.91	1.57	0.869
2	1.63	0.952	0.599
4	0.875	0.578	0.393
8	0.539	0.303	0.230
16	0.249	0.163	0.127
32	0.128	0.0863	0.0456
64	0.0664	0.0456	
128	0.0404		

Table 2 shows the timing results for a pure OpenMP parallelization. The number of processes per node is in this case limited by the number of processors per node which is 4 for the Compaq SC ES45 parallel computer which we are considering. For comparison we have however in Table 3 also included the results on a single Hewlett Packard EV7 node, where one can go up to 16 processors. The OpenMP parallelization is rather straightforward. The entire serial FFT part is cache blocked [2] for optimal efficiency. This outer cache blocking loop can easily be parallelized with OpenMP resulting in a fairly coarse grained parallelism. Nevertheless the OpenMP speedup of 3.2 is slightly worse than the MPI speedup on a single node.

We have implemented for this FFT an traditional MPI/OpenMP approach where each MPI process consists of 4 OpenMP threads. Such an

Table 2: Timing in sec of the OpenMP version on a single node of the Hewlett Packard Compaq AlphaServer SC ES45 as a function of the number of OpenMP threads. The speedup is given in parentheses.

Numb. Threads	1	2	4
time	2.89	1.50 (1.93)	0.903 (3.20)

Table 3: Timing in sec of the OpenMP version on the Compaq EV7 as a function of the number of OpenMP threads. The speedup is given in parentheses.

Num. Threads	1	2	4	8	16
time	2.74	1.48 (1.85)	0.841 (3.26)	0.501 (5.47)	0.263 (10.42)

approach has the advantage that the total number of messages that have to be send around during the MPI_ALLTOALL call is reduced by a factor of 16 since the number of MPI processes is reduced by a factor of 4. However the total amount of data to be sent remains the same. Since for the case of the FFT bandwidth rather than latency is the limiting factor, this approach offers no advantage over the pure MPI approach. This was confirmed by our timings. In addition this approach can not overcome the fundamental communication bottleneck. During the communication step performed by one thread all the other threads idle.

If one would just like to do a single FFT as fast as possible on a parallel machine of this type the conclusion is clear. Grasp the largest possible number of nodes that can be used to do the FFT (i.e. 128 for a 128^3 FFT) and use then the pure MPI implementation with one process per node. In practice two things are different. First one needs to do multiple FFT's, second of all it is economically wasteful not to use the majority of the processors on each node. Both requirements can be reconciled as will be shown in the following.

From the previous discussion it is clear that the most precious resource on a machine where each node consists of a multiprocessor is the bandwidth among the nodes. Therefore we should strive to keep the network busy all the time. This is not the case in the two previous implementations. Phases of computation during which the network is idle are alternating with communication phases. The situation is similar to the traffic problems in a big city. If everybody starts to work at the same time in the morning there

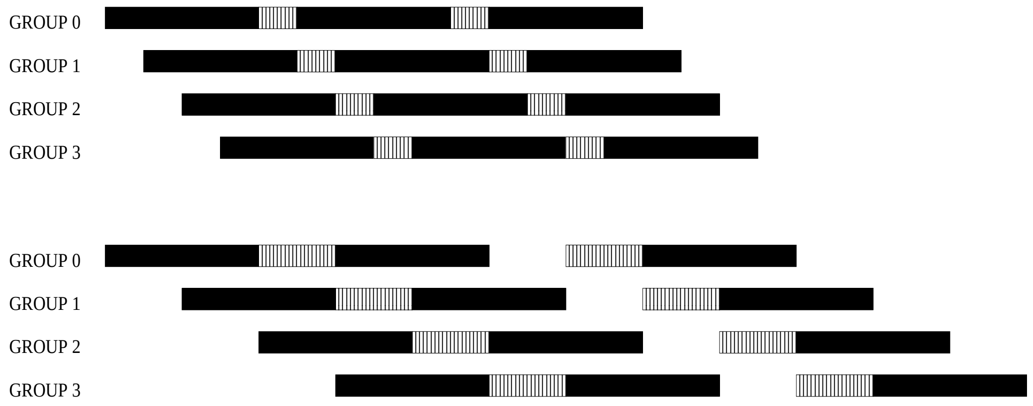

Figure 1: Illustration of the computation phases (black) and the communication phases (hatched) of the 4 task groups. Time is progressing fro the left to right. The requirement for the alignment of the communication stages is that they are not allowed to overlap. In the scenario shown in the lower part of the Figure this requirement results in idle time for the processes.

will be a traffic collapse. By staggering the beginning of the working hours a larger amount of rush hour traffic can be handled. Applying this simple idea to FFT's means the following. We subdivide all the processes on our nodes into 4 groups. The first group contains the threads number 0 on all the nodes, the second group all the threads number 1 and so on. Group 0 is the first to start the calculation. Once it arrives at the communication stage the second group starts the calculation and hopefully the first group has finished the communication stage once the second arrives at this stage. The third and fourth group start with additional delays the calculation and in the ideal case the communication stages of these 4 groups are not in conflict. The idea is illustrated in Fig. 1. In the upper part we assume that the communication time is less than a quarter of the computation time. In this case the throughput will be exactly 4 times larger compared to the case where we use only one process per node. This means that if we have a large enough number of multiple FFT's we can do 4 times as many FFT's with the same number of nodes. The case where the communication part is more than a quarter of the computation part is illustrated in the lower part of Fig. 1. In this case the throughput is increased by less than a factor of 4, but nevertheless the throughput is still increased.

The results of this approach are shown in Table 4. Adding more processes per node gives now a better speedup for a small number of processors. For larger configurations going from 2 to 4 threads per node does again not help very much since in this case the communication time is roughly half of the total time. The largest performance is obtained on 128 nodes independently of whether one uses 2 or 4 processors per node. The speedup at the peak performance is 160 and the speed 130 Gflops which is more than double of the speed that was obtained in the pure MPI approach. The 4 groups were defined as OpenMP threads because in this way it is easiest to classify the processes into threads belonging to one node. The same goal could however be achieved by using MPI task groups. Let us point out that this way to use OpenMP is highly non-standard. The OpenMP parallelization over the different multiple FFT's is logically on top of the single FFT MPI-parallelization. Since the calculation of the different 3-dim FFT's are completely decoupled very few of the OpenMP features are needed.

Table 4: Timing in sec of the non-conventional OpenMP/MPI implementation for multiple FFTs. Horizontally the number of threads (which equals the number of processors) per node varies (the maximum being 4), vertically the number of nodes varies.

	1	2	4
1	2.93	1.72	0.844
2	1.62	0.842	0.447
4	0.878	0.463	0.247
8	0.467	0.245	0.144
16	0.239	0.128	0.0805
32	0.0130	0.0753	0.0502
64	0.0671	0.0371	0.0319
128	0.0361	0.0186	0.0180

Evidently this non-conventional OpenMP/MPI approach does not offer any advantage in case one has to do only a single FFT. Table 5 shows however that already for a modest number of FFT's one can make substantial gains and that the asymptotic performance is reached at around 32 multiple FFT's

Table 5: Timings in sec of the OpenMP/MPI approach as a function of the number of multiple FFT's allowing for 4 threads per node. Horizontally the number of multiple FFT varies, vertically the number of nodes.

	4	8	16	32	64
1	0.862	0.877	0.867	0.875	0.875
2	0.518	0.533	0.485	0.457	0.446
4	0.297	0.297	0.270	0.254	0.245
8	0.175	0.168	0.155	0.147	0.145
16	0.0961	0.0891	0.0887	0.0825	0.0812
32	0.0767	0.0659	0.0549	0.0481	0.0500
64	0.0613	0.0490	0.0431	0.0358	0.0313
128	0.0369	0.0244	0.0206	0.0179	0.0182

4 Conclusions

Using state of the art serial optimization techniques and innovative parallel optimization techniques we were able to obtain very high performances of up to 130 Gflops on 256 processors for the most time consuming part in plane wave electronic structure calculations, namely the application of the local potential to the wave function. This part consists essentially of a three-dimensional FFT. By exploiting other sources of parallelism, such as the parallelism over the different k points, it should be possible to obtain performances of more than a Teraflop in electronic structure calculations on the largest computers available today.

5 Acknowledgment

We thank the Hewlett Packard Company for giving us access to their latest EV7 multiprocessor. We also acknowledge the support of Gilles Zerah for this project. SG thanks Phlippe Blaise and Gilles Civario for interesting discussions on programming problems.

References

[1] D. Takahashi, Applied parallel computing, proceedings lecture notes in computer science **1947** 316 (2001)

[2] S. Goedecker, A. Hoisie, *Performance Optimization of numerically intensive codes*, SIAM publishing company, Philadelphia, USA 2001 (ISBN 0-89871-484-2)

[3] Charles van Loan *"Computational Frameworks for the Fast Fourier Transform"*, SIAM, Philadelphia (USA) 1992 ; E. Oran Brigham *"The Fast Fourier Transform and its Applications"*, Prentice Hall, Englewoods Cliffs, NJ (USA), 1988

[4] D. Marx and J. Hutter, in *"Modern Methods and Algorithms of Quantum Chemistry"*, J. Grotendorst (Ed.), http://www.fz-juelich.de/wsqc/proceedings

[5] S. Goedecker, Comp. Phys. Commun. **76**, 294 (1993)

[6] P. Pacheco, *"Parallel Programming with MPI"*, Morgan Kaufmann, San Fransisco, USA, 1996

[7] R. Chandra et al. , *"Parallel Programming in OpenMP"*, Morgan Kaufmann, San Fransisco, USA, 2001