Improving the performance of GWW
A dCSE Project

I. Bethune

EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

August 9, 2010

Abstract

In this report, we present the results of investigation into improving the performance of GWW, part of the Quantum Espresso suite of software for *ab initio* simulation. In particular, the 3D Fourier Transform was found to be a significant bottleneck to application scaling. Several alternative methods for the FFT transpose were implemented, and the performance of these was studied on HECToR (Phase 2a and 2b). Speedups of up to 400% (on 128 cores of HECToR Phase 2a) were demonstrated for the 3D FFT in isolation, which delivered benefits of in the range of 4-36% in full application benchmarks. A checkpoint and restart mechanism was also added to help jobs complete in under the 12 hour queue limit on HECToR.

Contents

1 Introduction

This report covers work package 2 of the the dCSE Project "Improving the performance of GWW", carried out at EPCC, The University of Edinburgh. This work package targetted improving the communication performance of the FFT Tranpose operation, a key kernel in GWW calculations. Concurrently with this work, another work package was carried out at the University of Sheffield (Prof. Merlyne De Souza *et al*) to replace the existing 1D domain decomposition of the FFT grids with a 2D decomposition, futher extending the scalability of the algorithm. Comprehensive discussions of this type of optimisation are documented by e.g. Jagode[1] and Sigrist[2].

1.1 HECToR

The work reported here was carried out on both the HECToR Phase 2a (Quad-core XT4) and Phase 2b (24-core XT6) systems. Full details of the HECToR architecture can be found on the HECToR web site[3].

1.2 Quantum Espresso

Quantum Espresso[4] (opEn Source Package for Research in Electronic Structure, Simulation, and Optimization) is a freely available (GPL) suite of software for perfoming materials science using Density Function Theory (DFT). Applications include ground state calculations, structural optimisation, *ab initio* molecular dynamics, linear response (phonons) and spectroscopy. It uses a plane wave basis, in common with similar packages like CASTEP and CPMD, and implements a range of pseudopotentials and exchange correlation functionals, including Hartree-Fock and hybrid functionals (PBE0, B3LYP, HSE). It is developed by an international collaboration of the DEMOCRITOS National Simulation Center (Trieste) with several other materials science centers in Europe and the USA.

Unlike other packages, Quantum Espresso is not a monolithic single executable, but rather a set of independent executables, each of which perform a particular function. For this project the executables of interest were PW.x (Electronic and Ionic Structure, including MD), and PH.x (Phonons using Density Functional Perturbation Theory). Quantum Espresso is written in Fortan 90, consisting of over 300,000 lines of code in nearly 1000 source files. In addition to the source for each executable, there is a common `Modules` directory containing code shared by all the executables. This includes common functionality such as I/O, parallelisation, data types, as well as the FFT. A number of external libraries are required including an FFT library (e.g. FFTW3) as well as BLAS, LAPACK and ScaLAPACK for linear algebra operations.

1.3 GWW

GWW (GW Calculations using Wannier functions), is a recent addition to the Quantum Espresso package. Developed by Dr. Paulo Umari, the code calculates polarisation using a basis of localised Wannier orbitals within the GW approximation, an approach which is around two orders of magnitude faster than convential plane wave basis methods.

The type of calculations carried out by the Sheffield group involve a workflow of 7 individual jobs. Initial investigations showed that these calculations were dominated by the calculation of the dielectric matrix (in particular the head_nscf step of the calculation).

Cores	32	64	128	256
exc_scf (PW.x)	73	68	83	161
exc_nscf (PW.x)	1097	596	378	867
head_scf (PW.x)	158	141	178	
head_nscf (PH.x)	27155	22506	25445	
matrix_scf (PW.x)	179	144	163	
matrix_nscf (PW.x)	1436	909	917	
gww (GWW.x)	194	136	102	

Table 1: Times in seconds against number of cores for each stage of the GWW calcuation, using the Silicon benchmark (see section 1.4)

Output from the inbuilt timing routines in the code indicated that for the longest running parts of the calculation (head_nscf), and also for the other parts of the calculation, a large fraction of the calculation is spent in performing a 3D FFT (typically over 50%), and that much of this was `MPI_Alltoallv` communication derived from the FFT transpose.

1.4 Benchmark Systems

During this work, 3 benchmark systems provided by the Sheffield group were used for testing and benchmarking.

1. CNT40 - Small nanotube of 16 C atoms in unit cell of length 8.5Å
2. Silicon - 64 atoms of crystalline bulk silicon in unit cell of length 20.5Å
3. CNT80 - Larger nanotube of 96 C atoms in unit cell of length 24.3Å

The CNT40 test case is too small to be of much practical interest, but has the advantage of a very short runtime and so is a useful test during development. The other two, particularly CNT80, are closer to the cutting edge of what can be achieved with the code to date.

2 Checkpointing

Although not originally part of the project plan, it became clear early on that runtimes for the largest test case (CNT80) was approaching the 12 hour queue limit set on HECToR (See figure 1). Even with the proposed improvements to the FFT, it was not clear that with increasingly large systems being studied at Sheffield that it would be possible to complete a single step of the calculation in under the twelve hour limit. For example, on 64 cores of HECToR Phase2a, the exc_scf step of CNT80 system takes a total of 8 hours. Closer examination found that this is in fact made up of two sub-steps; firstly an iterative diagonalisation procedure (using ScaLAPACK) taking around 2h20m, followed by the 'dft_exchange' step (mainly FFT) taking around 5h40m. While the time taken for the second part would be reduced by the modifications planned in the dCSE project, it was decided to implement a checkpoint/restart mechanism, that allows the two parts of this step of the calculation to be performed independently.

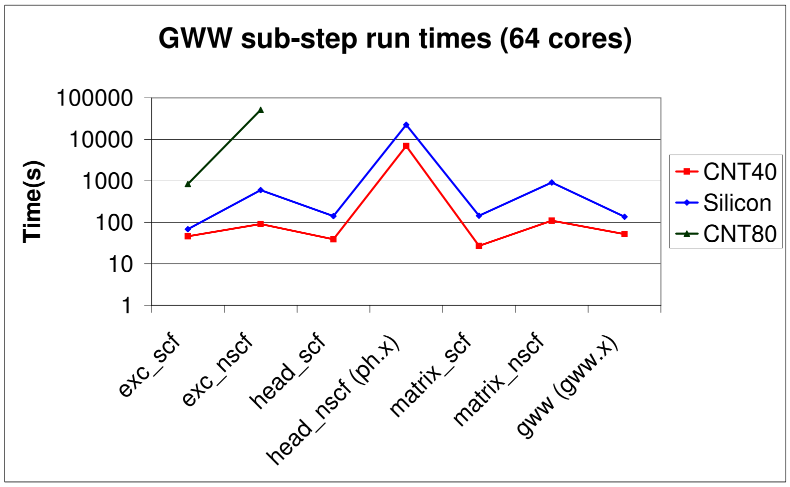

Figure 1: Run times of each step of the GWW calculation on 64 cores

This was implemented by a new flag in the input `&control` section called `split_calculation`. If set to zero or omitted, both sections of the calculation would be performed as normal. If set to 1, only the first step (diagonalisation) would run, and if set to 2, the output from the first step would be read in, and the dft_exchange step would run.

It should also be noted that while this part of the calculation is already very expensive, we expect the head_nscf (PH.x) step to be even more expensive based on experience with the smaller two test cases. However, this calculation already has checkpointing built in, and will automatically restart from the most recent saved point if the job is stopped (e.g. by hitting the 12 hour wallclock limit), so is already suitable for running in multiple 12 hour blocks.

3 FFT Implementation in Quantum Espresso

As mentioned earlier, the 3D FFT is a key component of Quantum Espresso, and is implemented as a set of modules which are shared between several of the executables. In particular PW.x and PH.x both make use of the same FFT code, so optimisations made here will benefit both.

At a high level, the FFT grids are distributed as slices (planes) in real space, and as columns (rays) in Fourier space. The reason for the column distribution in Fourier space (often used in a full 2D decomposition), is that not every column will necessarily have the same number (or any) of non-zero Fourier coefficients. By dividing in columns, the number of columns will typically be much larger than the number of MPI processes, and hence the columns can distibuted among the processes to load balance the number of non-zero Fourier coefficients per process. A full description of this technique is presented by Giannozzi *et al*, 2004[5]. Quantum Espresso maintains two grids of varying resolution, with the finer grid typically containing around twice as many grid points as the coarse grid. However, there are many more coarse grid FFTs performed each timestep.

As a result of this decomposition, each 3D FFT consists of a 2D FFT performed on local data in planes, a global transpose using `MPI_Alltoallv` across all processes, followed by a 1D FFT on the newly localised data in columns (or vice versa for the inverse transform).

3.1 FFT Test Harness

For easy development and testing of the FFT routines the first task of the project was to isolate the FFT code in a simple benchmark code. In order to minimise the amount of code needing to be written for this test harness, a new executable was created based on the PW.x code. Here, the main FFT routine `tg_cft3s` is wrapped by a routine `cft3s`. This was replaced with code which rather than execute a single FFT as requested, repeatedly performed a fixed number of forward and reverse FFT loops, before finally checking the output, and exiting. This has the effect of allowing the code to start up as normal, reading the normal set of input files, before the first FFT is intercepted and the FFT benchmark is performed.

The wallclock time taken by the FFTs is recorded using `MPI_Wtime` and reported on completion of the benchmark. A test for correctness is made by saving the initial contents of the grid, and comparing this to the final grids. Both the total error (absolute sum of errors in each grid element across all processes) and the maximum single element error are reported. Using the original, unmodified FFT routines and the CNT80 input files a maximum single element error of 10^{-14} was found after 1000 forward and inverse FFTs. This gives an error of 10^{-17} per iteration, which is consistent with the level of accuracy expected from double precision arithmetic. High values of either the single element or total error would flag a warning in the output file, and so the code could easily be tested for correctness during development.

4 FFT Transpose

As each column of the fourier grid intersects every plane of the real-space grid, in principle each process has to transfer some data to every other process between the two local FFT steps. In practise this is not always the case e.g. if the number of processes is greater than the number of slices, some processes will receive (or send for the inverse FFT) no data. In general, each process may send and receive differing amounts of data to every other process, since the number of processes may not divide exactly into the number of planes or columns. To illustrate this, consider the CNT40 test case on 16 cores. Here the coarse grid is 30x30x30 elements, and so there are 30 planes of data, and 900 columns, of which 657 happen to contain non-zero Fourier coefficients.

Dividing these between the 16 cores we find that 14 processes have 2 planes, with the remainder having 1. 15 processes have 41 columns each, and the final one has 42. Taking into account all combinations of domain sizes, there will be four different message sizes used for the Alltoallv - 84 elements, 82 elements, 42 elements, and 41 elements. In this case each grid element is a double-precision complex number i.e. 16 bytes per element.

The transpose operation is implemented in the function `fft_scatter`, which includes packing the data from the grid itself into contiguous blocks for each destination process in a send buffer, the MPI_Alltoallv call itself, followed by a corresponding unpacking operation. In the existing code there were already two alternative implementations of this function - the MPI_Alltoallv version described above, and a similar version where the collective communication is replaced by a number of non-blocking point-to-point communication calls. The choice of which version to use is controlled by preprocessor macros which can be defined in the `make.sys` file which controls the compilation.

Following the same scheme (using macros to conditionally compile each specific function) several alternative methods for performing the transpose were implemented and these are detailed below. Performance results are reported together in section 5 for clarity.

4.1 Shared Memory Alltoallv

The first approach to be attempted was to implement a buffer shared between all processes on the same SMP node (4 processes per node on the XT4, and up to 24 processes per node on the XT6), using the Unix SHM API to allow each process to acceess the buffer. This follows the approach of the 2Decomp library [6] used in Incompact3D and makes use of code from David Tanqueray of Cray for identifying which processes belong to which SMP node, and setting up the shared buffers. It should be noted that this approach is not portable, as it relies on details of the `/proc` filesystem on the Cray platform.

The aim is that by combining all the data from each process into a single send buffer on a single process (the 'root') on each SMP node, firstly, the number of processes involved in the MPI_Alltoallv operation can be reduced by a factor of the SMP width. As the number of messages exchanged is asymptotically $O(p^2)$, this dramatically reduces the number of messages e.g. using 64 cores of HECToR Phase 2a, this would reduce the number of messages from 4032 to 240. Secondly, by aggregating messages together, we reduce the impact of network latency, which is proportional to the number of messages. This relies on the fact that the copy of memory into and out of the shared buffer is relatively cheap as this is an extra step not present in the original algorithm.

The implementation of this method makes use of the `fft_dlay_descriptor` type, which contains parameters relating to the FFT grids (dimensions, indexing arrays etc.) to store the shared send and receive buffers, as this type is preserved from one FFT iteration to the next, saving the need for repeatedly allocating and deallocating the buffer.

There is a substantial amount of additonal 'book-keeping' required as each process needs to know enough about how much data is being sent from each other process in the SMP node that it can copy its data to/from the correct regions of the shared buffer. There is also the need for the creation of two extra communicators, one containing only the root process of each SMP node (used for the MPI_Alltoallv), and one containing just the processes in the SMP node (used for exchanging send and receive counts).

Compiling the code using the SHM Alltoall requires the `__FFT_SHM` macro to be defined and requires a compiler support Cray Pointers (e.g. gfortran using `-fcray-pointer`).

4.2 Padded Alltoall

The second approach is motivated by the fact that although differing amounts of data are sent and recieved by each process, requiring the use of MPI_Alltoallv (see section 4), if the data was padded such that every process sent and recieved the same amount of data, MPI_Alltoall could be used instead. For 'short' messages, MPI_Alltoall uses an optimised algorithm, known as 'store and forward' which trades off increased bandwidth usage for a reduction in message latency. Specifically, whereas the default algorithm's cost scales as O$(p + n)$, where p is the number of processes and n is the message size, the store and forward algorithm scales as O$(log(p) + nlog(p))$. When the number of processes is large, and the message size is small, the is expected to give a significant performance improvement, which should overcome the cost of sending the extra padding data. This can be seen clearly in figure 2, which shows that when the message size is sufficiently small, MPI_Alltoall is significantly faster than MPI_Alltoallv - in fact up to 5 times faster, for 256B messages on 256 cores.

In fact, using the same example as before (CNT40 on 16 cores), the total data size being sent by all processes is 1232 elements (20kB). If each process has additional padding added to equal the largest individual send size, the total data to be sent is now 1344 elements, only a 9% increase in the total volume of data. For messages of this size (1kB), the bandwidth component of an individual message is of order 10^{-7} compared with the latency of order 10^{-6}, so the extra cost associated with the padding is negligable.

The Padded alltoall requires much less additional calculation that then SHM approach, as each process already has all the data needed to calculate the global maximum message size. Once the buffers are packed, data is sent using a call to MPI_Alltoall, then unpacked. By ensuring that the real data is always at the start of the block of data sent between processes (and the padding is at the end), it is easy to just read only the amount of data that would be expected if the communication had been done using MPI_Alltoallv, and ignore the padding.

Compiling the code using Padded Alltoall requires the `__FFT_ALLTOALL` macro to be defined at compile time.

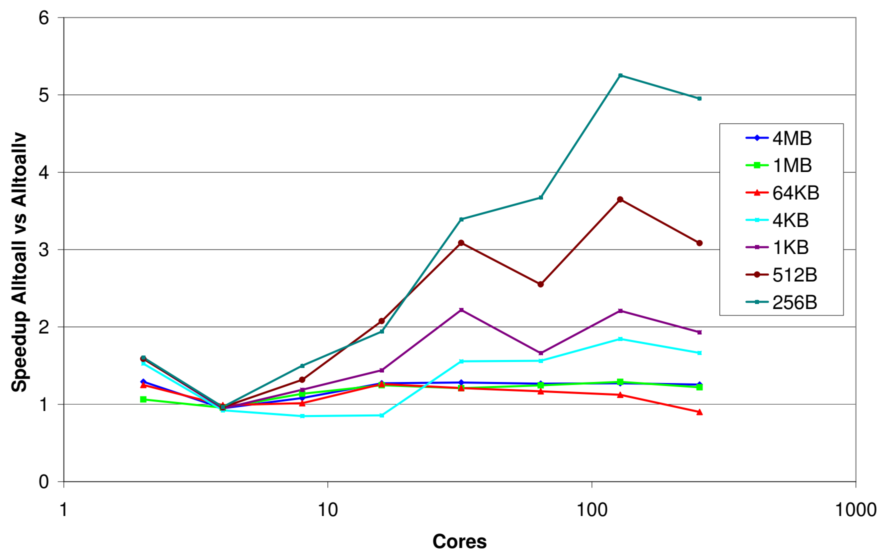

Figure 2: Relative speedup of MPI_Alltoall over MPI_Alltoallv on the Intel MPI Benchmarks

4.2.1 Combining the first two approaches

As the SHM and Padded Alltoall modifications are orthogonal, they can be combined in a straightforward way. Each simply copies its data into the shared buffer as before, but instead of packing contiguously as before, instead the data is packed so that every block is spaced equally, according to the size of the largest individual message size.

However, as seen in section 5 this method does not give as good performance as the SHM Alltoallv approach on its own, so it has not been included in the final code.

4.3 Scatter/Gather Alltoallv

Thirdly, a similar approach to the SHM Send Buffers was implemented. However, to avoid the limitations of having non-portable code, and having to use a shared memory API which breaks the MPI Distributed Memory model, an alternative using MPI_Gatherv and MPI_Scatterv to collect data on the root node of each SMP was used. This follows a similar implementation in CASTEP[7]. While similar in structure to the SHM implementation, an extra step is needed at both the packing and unpacking stage. Since all data sent from a given process to the root on its SMP node by MPI_Gatherv must be contiguous, it needs to be unpacked from this recieve buffer into the send buffer that will be used for the MPI_Alltoallv call that performs the global transpose. A corresponding operation is also needed after than MPI_Alltoallv, before the data is send back to each process in the SMP node using MPI_Scatterv.

Compiling the code using the Scatter/Gather Alltoallv requires the `__FFT_LOCAL_COMM` macro to be defined at compile time.

5 Benchmark Results

5.1 FFT Benchmark

Each of the four implementations, as well as the existing MPI_Alltoallv and point-to-point implementation were run on the HECToR Phase 2a and Phase 2b systems for each of the three benchmark systems, CNT40, Silicon and CNT80. The results are shown below. In all graphs, the X axis shows the number of cores, and the Y axis the speedup relative to the original MPI_Alltoallv code on 1 core.

5.1.1 HECToR Phase 2a

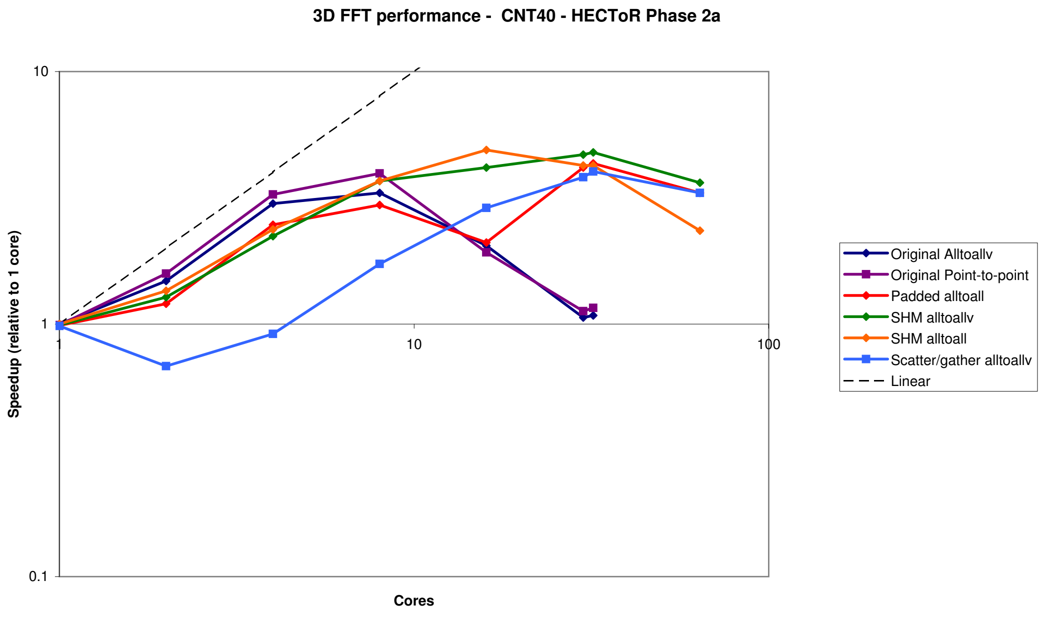

Figure 3: Performance of 3D FFT Benchmark using CNT40 benchmark on HECToR Phase 2a

The CNT40 benchmark (figure 3) is small, and with the original code, only scales up to 8 cores before the performance drops off. It should be noted that in this case, the point-to-point method gives slightly better performance than the MPI_Alltoallv method. This is due to the fact that asynchronous message passing is used, so some of the buffer packing and unpacking cost is hidden as it takes place while other communications are outstanding. With the MPI_Alltoallv, the data for all processes must be packed first, the communication takes place synchronously, followed by unpacking again.

The padded Alltoall gives similar performance to the original Alltoallv code, with the slight slowdown due to the extra overhead of sending the padded data. However, beyond 16 cores, the message sizes drop below 1kB, and the impact of the store-and-forward algorithm can clearly be seen. However, as the grid size is only 30^3, the FFT no longer scales beyond 32 cores.

The SHM Alltoallv and SHM Alltoall both give similar performance, and perform best of the options available at the larger core counts (8-32).

The Scatter/Gather Alltoallv shows good scalability up to 32 cores, which we would expect as this is mainly due to the reduced number of processes participating in the MPI_Alltoallv operation. However, the extra overhead of scattering and gathering the data, along with the required extra data packing, mean that it does not perform nearly as well as the SHM version. It should be possible to remove the dip in performance for 2-4 cores, since in this case we could replace the packing, MPI_Alltoallv and unpacking with a simple data copy. However, as the focus of this work was extending the scalability of the FFT, this was not done.

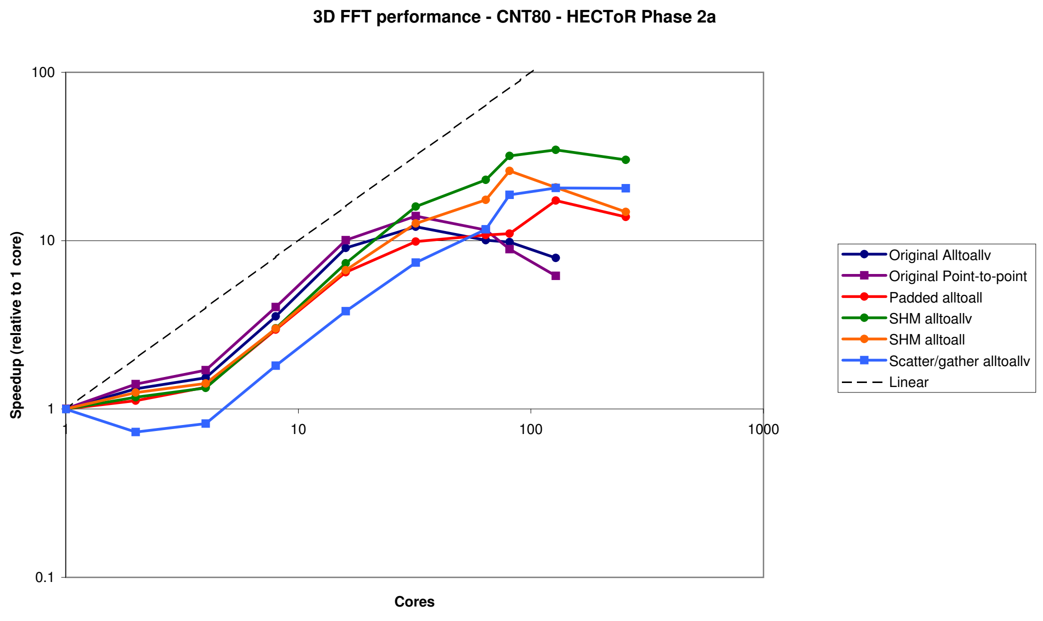

Figure 4: Performance of 3D FFT Benchmark using CNT80 benchmark on HECToR Phase 2a

The CNT80 benchmark (figure 4) is larger - a 81^3 grid - so all the different algorithms scale well. The same trends as for the CNT40 case can be seen here, and again the SHM Alltoallv is clearly the best choice, giving good performance up to 128 cores.

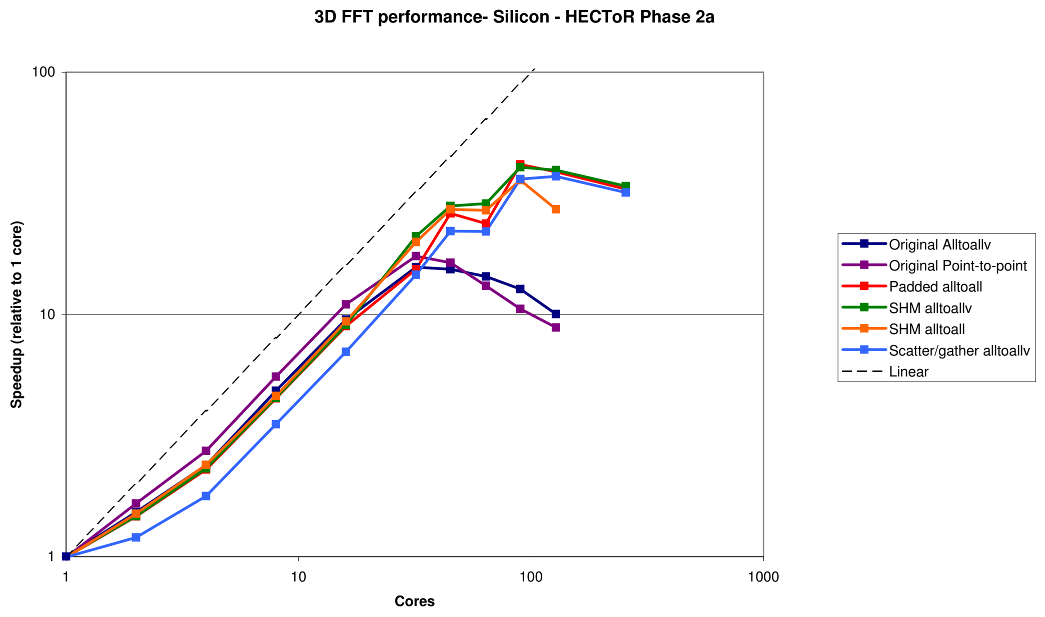

Figure 5: Performance of 3D FFT Benchmark using Silicon benchmark on HECToR Phase 2a

The Silicon benchmark (figure 5) is also slightly larger - 90^3 - and is very similar to the CNT80 case. Best performance here is achieved with 90 cores, and in this case, the Padded Alltoall and SHM alltoallv perform very similarly, as the padding overhead is very small since the number of processes divides the grid dimension exactly. There is also a clear gain in performance at 45 cores, indicating that picking the number of processors to match the grid dimensions can be a useful technique.

It is also worth making the point that in the region where the new FFT implementations stop scaling (e.g. 128 cores for the Silicon benchmark), they are around 4 times faster than the original implementations.

5.1.2 HECToR Phase 2b

The same benchmarks were run on the HECToR Phase 2b system, with some significant differences in results. Starting with the CNT40 case (figure 6), whereas the original code only scaled up to 8 cores on the XT4, there is good scalability up to 24 cores on the XT6. This is due to the fact that there are now 24 cores on a single node, so all communication can go through the shared memory device rather than onto the network. However, beyond 24 cores, there is a very sharp drop-off in performance, as all 24 cores on the node attempt to send messages to another node. Cray's next generation 'Gemini' interconnect, which will be added to HECToR Phase 2b in late 2010, promises to provide higher message throughput, so should go some way towards addressing this particular issue.

The Padded alltoall method does somewhat better beyond 24 cores, due to the reduced latency costs compared to the alltoallv, performance is still poor as the number of messages to be exachanged remains the same.

The SHM and Scatter/gather methods have even poorer performance when the number of cores is less than 24, but this level of performance is maintained to higher processor counts. However, as this is a relatively small system, with a grid size of 30 in each dimension, these results show it is best to use just a single node of the XT6 and avoid the inter-node communication entirely.

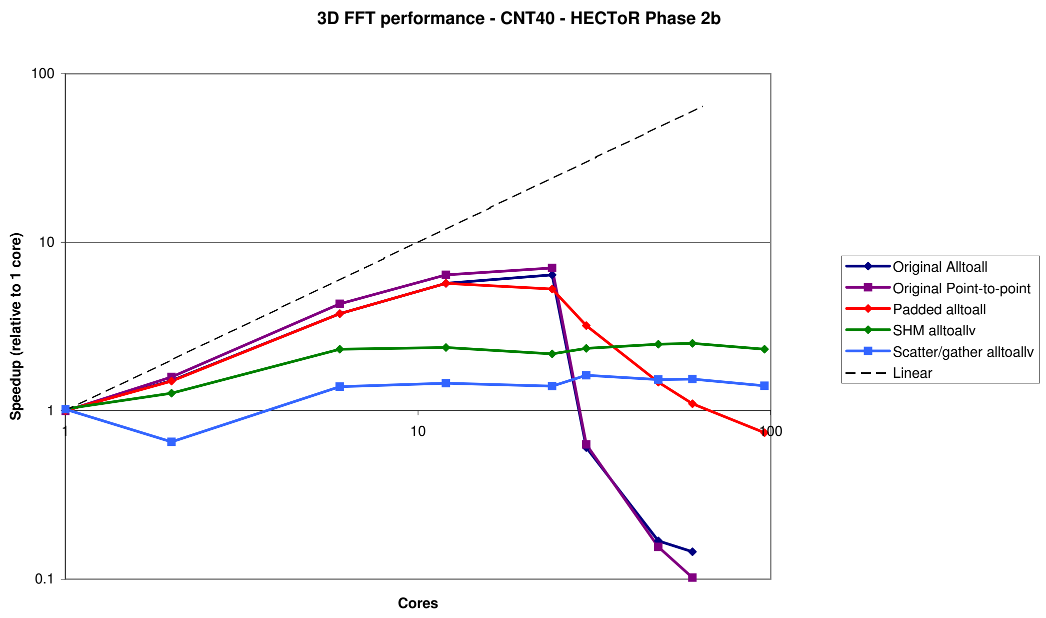

Figure 6: Performance of 3D FFT Benchmark using CNT40 benchmark on HECToR Phase 2b

For CNT80 (figure 7), we see a similar drop-off beyond 24 cores for all the methods in which each core participates in the Alltoall communication. However, the padded alltoall does get some benefit at 96 cores and above from the switch to the store-and-forward algorithm.

The SHM and Scatter/gather code performs somewhat better for this larger example than CNT40, giving reasonable scalability all the way to 288 cores (12 nodes). It this case, there is a clear advantage in only using a single process per node for the communication, as the total number of messages drops from 82656 (of which 79488 would cross the network) to 132!

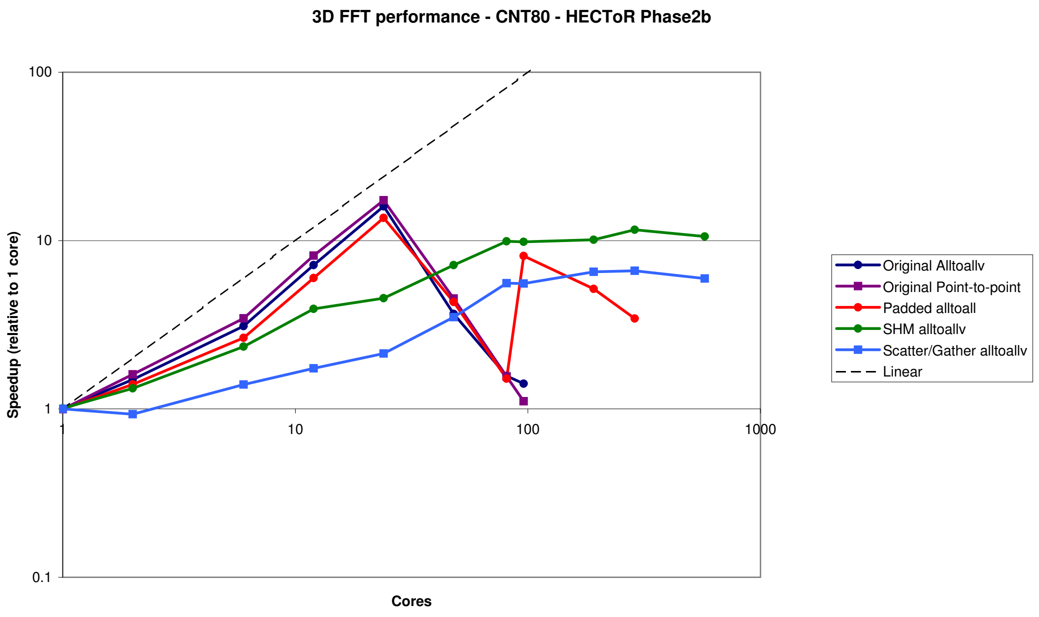

Figure 7: Performance of 3D FFT Benchmark using CNT80 benchmark on HECToR Phase 2b

Finally, the Silicon benchmark (figure 8) again does not scale beyond 24 cores with the original code. Although the Padded alltoall does mitigate the drop-off in performance somewhat, to achieve better scalability (up to 288 cores), the SHM Alltoallv is required, similarly to the CNT80 case.

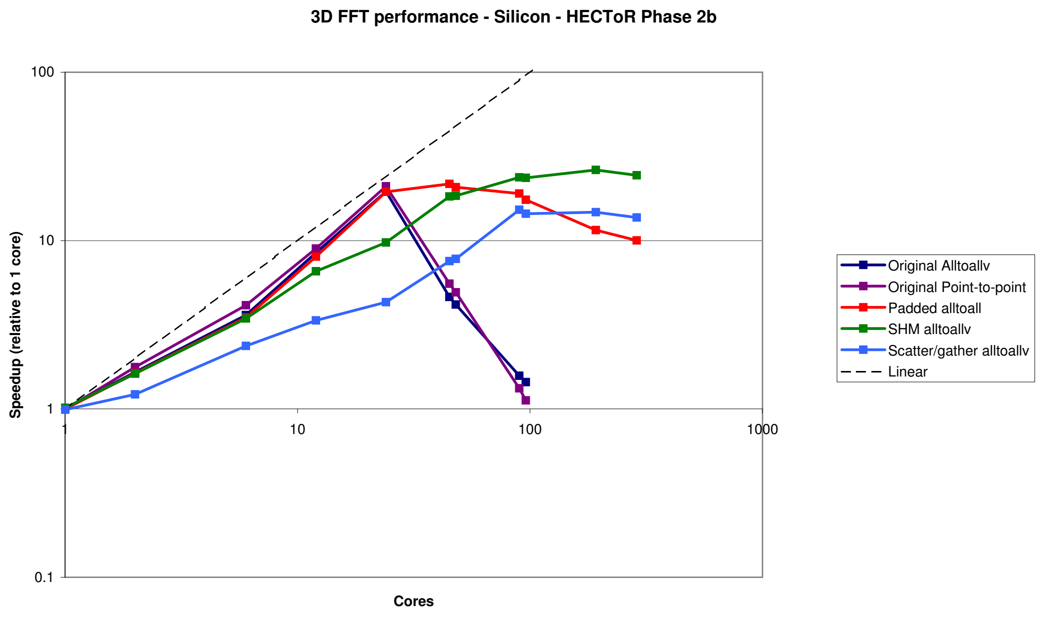

Figure 8: Performance of 3D FFT Benchmark using Silicon benchmark on HECToR Phase 2b

5.2 Application Benchmarks

To demonstrate the benefits of these changes to the user, the full PW.X executable was compiled with the original Alltoallv implementation, and the new padded Alltoall and SHM alltoallv modifications. This was then used to run the first two stages of the GWW calculation (on HECToR Phase 2a). The results are shown in table 2.

	CNT40 (16 cores)		Silicon (64 cores)		CNT80 (64 cores)		
Version	exc_scf	exc_nscf	exc_scf	exc_nscf	exc_scf	exc_nscf1	exc_nscf2
Original alltoallv	18.9	33.0	102	657	595	2h15m	4h55m
Padded alltoall	16.7	29.0	76	559	441	2h10m	5h15m
Speedup	12%	12%	25%	15%	26%	4%	-7%
SHM alltoallv	14.6	21.0	77	466	424	2h10m	3h16m
Speedup	23%	36%	25%	29%	29%	4%	34%

Table 2: Comparison of FFT transpose methods for application benchmarks (times in seconds except where specified)

In all cases but one, both the Padded alltoall and SHM alltoallv methods are faster than the original alltoallv implementation. Note that for CNT80, the nscf step is split in two (see section 2) and the first part is dominated by linear algebra, rather than the FFT. In all cases the SHM alltoallv outperfoms the padded alltoall, so it is recommended that this method always be used on HECToR

6 Conclusion

In summary, three alternative methods for performing the global communication in the 3D FFT transpose were implemented - SHM Alltoallv, Padded Alltoall and Scatter/Gather Alltoallv. The SHM Alltoallv is found to perform well, and scales the best of the three implementations. Speedups of up to 400% (on 128 cores of HECToR Phase 2a) were demonstrated for the 3D FFT in isolation, which delivered benefits of in the range of 4-36% in full application benchmarks. Even with these improvements, some large jobs would not fit into the 12 hour queue limit on HECToR, so a checkpoint and restart mechanism was added for non-SCF calculations using the PW.X code.

The performance of the FFT on HECToR Phase 2b (XT6) was found to be disappointing beyond 24 cores (1 node) due to the high number of messages requiring to cross the shared network interface. However, the forthcoming installation of Cray's new Gemini interconnect in Q4 2010 is expected to address this limitation and bring performance more closely in line with the Phase 2a system (XT4).

When combined with the ongoing work at Sheffield to implement a full 2D domain decomposition, even further scalability will be achieved. Nevertheless these modifications alone will deliver real improvements to the performance and scalability of Quantum Espresso for HECToR users.

References

[1] Fourier Transforms for the BlueGene/L Communication Network, Heike Jagode, 2006, http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0506/hjagode.pdf

[2] Optimizing parallel 3D Fast Fourier Transformations for a cluster of IBM POWER5 SMP nodes, Ulrich Sigrist, 2007, http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0607/2298876-27h-d07rep1.1.pdf

[3] HECToR website, http://www.hector.ac.uk

[4] Quantum Espresso, http://www.quantum-espresso.org/

[5] First-principle molecular dynamics with ultrasoft pseudopotentials: Parallel implementation and application to extended bioinorganic systems, P. Giannozzi, F. De Angelis and R. Car, J. Chem. Phys. 120 (2004)

[6] 2DECOMP&FFT User Guide, Ning Li, 2010, http://www.hector.ac.uk/cse/distributedcse/reports/incompact3d/UserGuide.html

[7] An LPAR-customized MPI_AllToAllV for the Materials Science code CASTEP, Martin Plummer and Keith Refson, 2004, http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0401.pdf