

An Analysis of FFT Performance in PRACE Application Codes

Andrew Sunderland^a, Stephen Pickles^a, Miloš Nikolić^b, Aleksandar Jović^b, Josip Jakić^b, Vladimir Slavnić^b, Ivan Girotto^c, Peter Nash^c, Michael Lysaght^c

^aSTFC Daresbury Laboratory, Warrington, UK; ^bInstitute of Physics, Belgrade, Serbia;

^cIrish Centre for High-End Computing, Dublin, Ireland

Abstract

The Fast Fourier Transform (FFT) is one of the most widely used algorithms in engineering and scientific applications and therefore its analysis and performance is of much importance to a range of research fields. On PRACE Tier-0 systems a parallel environment with a great deal of processing power (large number of CPU cores or accelerators such as GPUs) is at disposal for researchers. The FFTs investigated are both in-code and through various numerical libraries, where the algorithm is implemented in both serial and parallel form. The implementations of FFT investigated range from pure MPI, OpenMP versions for multicore, hybrid (OpenMP/MPI) to GPU-based.

The objective of this project is to assess the suitability, performance and scalability of various implementations of FFT for the PRACE large-scale scientific applications Quantum ESPRESSO and DL_POLY.

1. Introduction

The Discrete Fourier Transform (DFT) plays an important role in many scientific and technical applications, including time series and waveform analysis, solutions to linear partial differential equations, convolution, digital signal processing, and image filtering. The DFT is a linear transformation that maps n regularly sampled points from a cycle of a periodic signal, like a sine wave, onto an equal number of points representing the frequency spectrum of the signal. In 1965, Cooley and Tukey devised an algorithm to compute the DFT of an n -point series in $n \cdot \log(n)$ operations. Their new algorithm was a significant improvement over previously known methods for computing the DFT, which required n^2 operations. The revolutionary algorithm by Cooley and Tukey and its variations are referred to as the Fast Fourier Transform (FFT). Due to its wide application in scientific and engineering fields, there has been a lot of interest in implementing FFT on parallel computers.

The scalability of 3-dimensional Fast Fourier Transforms (3D FFTs) is limited by the all-to-all nature of the communications involved. This presents a challenge scaling up those codes that rely heavily on FFT methods to exploit PRACE Petascale systems.

In this study we examine the FFT performance in the molecular dynamics package DL_POLY [1] and Quantum ESPRESSO (QE) (package for electronic structure calculations within Density-Functional Theory and Density-Functional Perturbation Theory) [11] comparing performance of different 3D FFT library routines and exploring their scalability in the strong sense on PRACE hardware.

Our purpose was not to invasively optimize DL_POLY or QE, but rather to compare the FFTs used in DL_POLY and QE with third-party FFT methods. To do this in DL_POLY we use as a basis a synthetic benchmark code developed for a similar project outside PRACE [10]. The developments under PRACE made to this benchmark code include the introduction of FFTW version 3.3.1 routines and a more robust verification stage. The benchmark suite has allowed us to investigate scalability over a range of problem sizes in a controlled way for several candidate routines. The dataset element coefficients generally have no bearing on the computational performance (in contrast to, for example, parallel eigensolvers) and therefore synthetic datasets are representative of those used in DL_POLY and QE providing the grid dimensions are representative of real datasets.

Similar goals as for DL_POLY were set also for QE, but after initial effort to develop a QE benchmark code, only FFT specific input datasets were used with regular (non modified) QE because this way of testing offered enough flexibility for FFT library comparison, although implementation of non-supported FFT libraries to QE (like FFTE), in the time of writing, remained as a work in progress. In addition, specific FFT benchmark codes were developed and used to compare performance of various FFT libraries on the PRACE machines CURIE and JUGENE and these results are also presented in this document.

In section 2, we introduce the various FFT methods used in this study. In section 3, we briefly describe the molecular dynamics code DL_POLY, and its use of FFTs. We also describe the synthetic benchmark code. Section 4 presents DL_POLY benchmark results and the synthetic benchmark code on the JUGENE and CURIE machines. In section 5, we also report on

investigations carried out at ICHEC into the potential benefits of implementing GPU-enabled FFT libraries within DL_POLY. In section 6 we briefly describe Quantum ESPRESSO (QE) suite as well as the implementation of the Fast Fourier Transformation in QE code. This section also gives a description of benchmarking procedures on FFT libraries using both in-house developed FFT test codes and QE. Section 7 presents QE and in-house developed benchmarks results. Finally, in section 8, we summarise our conclusions, discuss related work, and make some recommendations.

2. FFT Libraries and Methods

The main performance bottleneck of parallel 3D FFTs is the communication. Once 3D data is distributed over MPI processes, all-to-all communications are unavoidable. Applications that rely on FFTs adopt different data decomposition strategies; 1D decompositions give each process a complete 2D slab, 2D decompositions give each process a complete 1D pencil, while 3D decompositions give each process a block that does not span the global domain in any dimension. Slab decompositions tend to perform well on small process counts; pencil decompositions scale better, but also eventually run out of steam. Efforts to optimize the performance of 3D parallel FFT libraries have tended to focus on slab and pencil decompositions.

2.1. FFTW

The “Fastest Fourier Transform in the West” has been developed at MIT by Matteo Frigo and Steven G. Johnson [3]. It is open source, and free. It is written in C, but also has Fortran bindings. It supports transforms of arbitrary size. The performance of FFTW is competitive with, and sometimes exceeds, vendor-supplied libraries, and has the advantage that the library and its performance are both highly portable. FFTW achieves portable performance by measuring the speed of many alternative codelets on the target architecture, and making an informed choice at run-time.

Results in this study were obtained using release 3.3.1 of FFTW, the first version to support parallel MPI 3D FFTs. Only slab decompositions are currently supported, so that the 3D grids are decomposed in only one dimension (here we use the z dimension).

A previous versions of FFTW (v2.1.5) is also used in QE as the default FFT library at the time of writing and it is included in the QE distribution.

2.2. FFTE

FFTE [4] has been developed by Daisuke Takahashi of Tsukuba, Japan. The name FFTE, which is an acronym for “Fastest Fourier Transform in the East”, is more of a tribute to FFTW than a signal of any serious attempt to offer a production-ready library to rival FFTW (even though FFTE has been observed to slightly outperform FFTW on very large FFTs). FFTE supports radix 2, 3, and 5 Discrete Fourier Transforms (DFTs), including optimised routines for radix-8, and has parallel flavours, both pure MPI and hybrid MPI+OpenMP. FFTE comes with little documentation, and it is necessary to examine the source code in order to use it. The MPI-parallel version only works correctly when the number of MPI processes is a power of 2, but it does not complain when this constraint is not satisfied. In FFTE, 3D parallel FFTs must be decomposed over MPI processes so that the leading dimension (x) of the 3D arrays (x, y, z) is kept local to each MPI process.

In this study, we use version 5.0 of FFTE. We employ both PZFFT3D, a parallel 3D DFT method which requires that the data is decomposed over MPI processes in the z -dimension (i.e. it supports only a slab decomposition), and PZFFT3DV, which allows data decomposed in both the y - and z -dimensions (i.e. it supports a pencil decomposition). Both PZFFT3D and PZFFT3DV will utilize any additional OpenMP threads, if available at run-time.

FFTE uses `MPI_ALLTOALL` to implement the MPI communication phases in both PZFFT3D and PZFFT3DV.

2.3. DAFT

DAFT (Daresbury Advanced Fourier Transform) is the parallel DFT used by DL_POLY [5]. In contrast to most 3D parallel FFT software, DAFT uses a decomposition in which the global arrays to be Fourier-transformed are distributed across MPI processes in all three dimensions, i.e. the same decomposition into parallelepipeds that DL_POLY uses for the rest of its calculations. The motivations for this design decision are discussed further in [10] and in section 3.1 below.

An important consequence of the DAFT decomposition is that all three dimensions of the DFT require MPI communications. But this is not so daft as it sounds. The alternative, of using a third-party 3D parallel FFT library to perform the DFT, would require two additional communication phases, one to re-distribute the data into the layout expected by the library before the DFT operation, and one to re-distribute the results back into the layout expected by DL_POLY afterwards. Moreover, as argued in [5], the 3D decomposition of DAFT should scale to very large core counts at least as well as a 2D pencil decomposition. We note

that the benchmark system (see section 4.1 below) used in this study is too small to verify this claim.

The 3D decomposition of DAFT has another important consequence. One cannot, in general, employ a third party FFT library to do the 1D FFT operations that arise in the 3D DFT. This is because each piece of the now-distributed 1D FFT must use different “twiddle factors”, and it is not common practice to expose these factors as part of an FFT library’s API. Accordingly, DAFT uses its own implementation of 1D FFTs, based on Clive Temperton’s GPFA algorithm with a modified set of twiddle factors [6].

DAFT supports radix 2, 3, and 5 FFTs, although the radix 3 and 5 code is less highly optimized. The transforms are un-normalized and in-place. The results from a DAFT transform are in a scrambled (“bit-twiddled”) order, and must be accessed via an index table. However, a forward transform followed immediately by an inverse transform will recover the input, provided that the results are normalised by hand (that is, by dividing each element by the size of the 3D array). Communications in DAFT are implemented using MPI point-to-point communications.

2.4. P3DFFT

The Parallel Three-Dimensional Fast Fourier Transforms library (P3DFFT) is an open source library. The main author is Dmitry Pekurovsky and the library started as a project at University of California, San Diego [7]. P3DFFT library supports both 1D and 2D decomposition and it can be used in large-scale calculations because of the large number of tasks that can execute in parallel [14]. The library requires to be built on the top of an optimized 1D FFT library (IBM’s ESSL or FFTW) and supports 2D decomposition strategy in order to compute the 3D FFT. The library’s interface is written in Fortran90 by using the Message Passing Interface and can be used by the programs, by calling the library’s Fortran90 module. A C interface is also available, as are detailed documentation and examples in both Fortran and C. A configuration script is supplied for ease of installation.

3. DL_POLY Application Background

The objective of the first part of the project is to study the performance of both existing and potential 3D FFT routines to be used within DL_POLY and measure their performance on PRACE Tier-0 Architectures.

3.1. DL_POLY Background

DL_POLY is a general purpose classical molecular dynamics (MD) simulation software developed at STFC Daresbury Laboratory by I.T. Todorov and W. Smith. DL_POLY is used worldwide and has a current user-base of over 1600 researchers. All results for DL_POLY in this study were obtained using the first general release of DL_POLY4 [1], downloaded in January 2011, and built from source. The latest minor release of DL_POLY is version 4.03.2, available since March 2012. DL_POLY is the focus of a larger ongoing code development and optimization project in PRACE 1-IP, with input from several European institutions.

In contrast to some computational chemistry codes such as CASTEP [8], FFTs are incidental to the DL_POLY methodology, arising only in problems when the Smooth Particle Mesh Ewald (SPME) method is selected. Accordingly, DL_POLY decomposes its domain into parallelepipeds in all three spatial dimensions, as this is optimal for the majority of problems. As mentioned in section 2, this severely restricts the applicability of third-party 3D FFT libraries in DL_POLY, as these generally expect at least one dimension to be stored locally. In order to avoid the need for additional communications phases before and after the use of a third-party FFT, DL_POLY uses its own 3D FFT software, known as DAFT. It turns out that plugging a third-party 1D FFT library into DAFT to handle the local parts of the computation is far from straightforward, because each local part needs a modified set of “twiddle factors”.

3.2 Synthetic Benchmark

In order to enable comparison between the FFTs used in DL_POLY with third-party FFT methods, we have developed a stand-alone, synthetic benchmark code based on the DL_POLY domain decomposition [2]. This also enables the problem size and decomposition to be controlled more flexibly than when using the full DL_POLY application, which determines the size and decomposition from run-time parameters not directly related to the size of the FFT. This synthetic benchmark code can run DL_POLY’s native DAFT method. It can also run FFTE, provided that the decomposition is restricted to a case in which at least one dimension is stored locally, i.e. to either a pencil or a slab decomposition. In this project, we extended the benchmark code to support the parallel FFTs introduced in FFTW 3.3. The benchmark code can be extended to support other 3D parallel FFT

libraries as they become available in the future.

We have used the benchmark code to study the performance of DAFT, FFTE and FFTW on a range of problem sizes up to the limit of problems attempted using DL_POLY today, and on a range of core and thread counts up to the limit of the available benchmark machines.

4. DL_POLY FFT Benchmark Results on PRACE Tier-0 Machines and their Interpretation

4.1. Context of Benchmarks in DL_POLY

A previous study of FFTs in DL_POLY has reported that typically around 10-20% of overall run time is spent in FFTs for SPME calculations – see Fig. 1 below. The ‘ptile’ setting here relates to the level of occupancy of cores with tasks on the Intel Westmere-based multicore nodes. The PRACE benchmarks use fully occupied nodes where possible and therefore the ‘ptile=24’ results, where around 20% of overall runtime is spent in the calculating the FFT, are most relevant to this report.

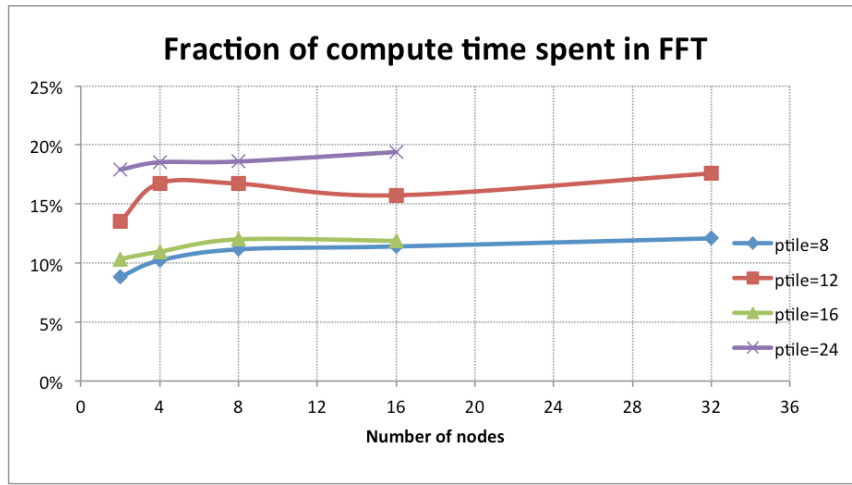


Fig. 1. Fraction of DL_POLY compute time spent in 3D Fourier transforms. Dataset is NaCl, 1,728,000 ions on Fujitsu Intel Westmere system (from OPL research report [2])

4.2. Synthetic Benchmarks for DL_POLY Analysis

In this section, we present results obtained with the synthetic benchmark code, described in section 3.2 above, on PRACE Tier-0 architectures. We examine four routines included in the libraries described in section 2, FFTE’s PZFFT3D and PZFFT3DV, FFTW and DL_POLY’s DAFT. We benchmarked using a range of problem sizes (i.e. the global size of the distributed 3D array being Fourier transformed), including 128^3 , 256^3 , and 512^3 . Although array sizes that are not powers of 2 could well introduce other considerations arising out of the relative quality of radix-3 and radix-5 implementations, we ruled these out of scope in the present study.

The benchmarks were run on a range of core counts, from 32 (equivalent to 1 node of CURIE) up to 1024 cores on JUGENE (512 is currently the largest power of 2 node count that can be scheduled on CURIE). We report the time taken for a forward transform immediately followed by the inverse transform, averaged over 10 repeats, as measured on process 0 of the MPI application.

We explore different decompositions over MPI processes, where supported by the FFT method. We use the convention that np denotes the total number of processes; np_x , np_y , and np_z denote the number of processes over which the data is geometrically decomposed in the x , y , and z dimensions respectively; $np = np_x * np_y * np_z$; and the x index is the fastest varying in all array declarations. For FFTW and PZFFT3D, which only supports a slab decomposition, $np = np_z$, $np_x = np_y = 1$. For PZFFT3DV, which supports also a pencil decomposition, $np_x = 1$. For DAFT, all of np_x , np_y and np_z can be greater than 1, though due to time constraints we only investigated the slab and block decompositions.

We benchmark the performance and scalability (in the strong sense) of FFTs in DL_POLY, measuring the time taken by the FFT operation as a whole. The scaling plots for FFTW and FFTE are limited by $np = np_z$. The performance figures from the DAFT block distributions are used as a baseline figure for performance as this is the method and decomposition currently used in DL_POLY. We are using test cases at the larger end of the problem sizes to which the codes are applied today, while still being

small enough so that we can begin to explore their scalability characteristics on the benchmark system.

4.3. JUGENE Results

All results presented were run using submission jobs specifying ‘VN’ mode, where all four cores on a node are occupied with MPI threads. All jobs in this study requested exclusive use of their nodes.

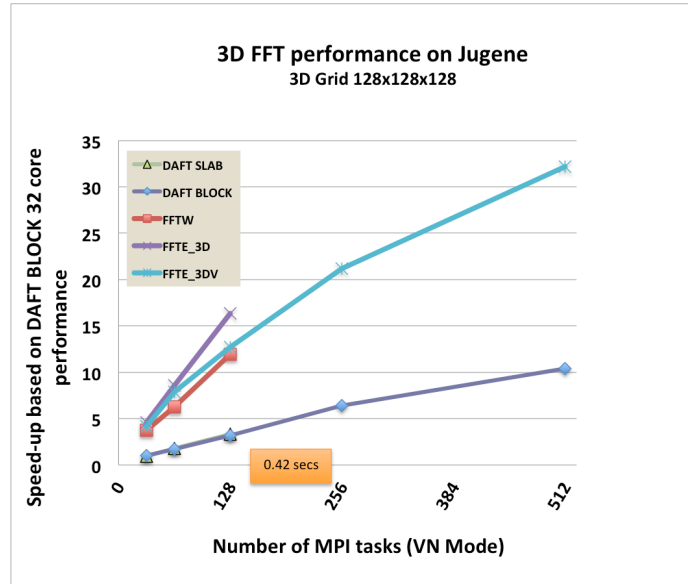


Fig. 2. Relative performance of FFT library routines for 128^3 dataset on JUGENE. Timing label refers to runtime of DAFT Block run on 32 cores.

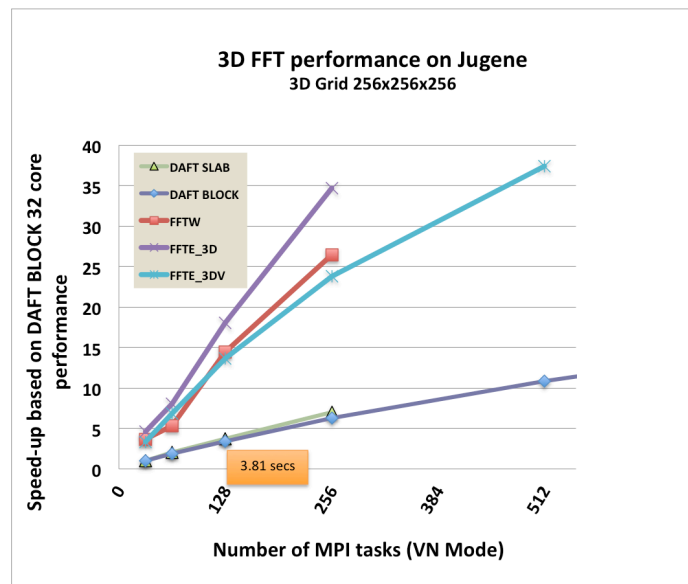


Fig. 3 Relative performance of FFT library routines for 256^3 dataset on JUGENE. Timing label refers to runtime of DAFT Block run on 32 cores.

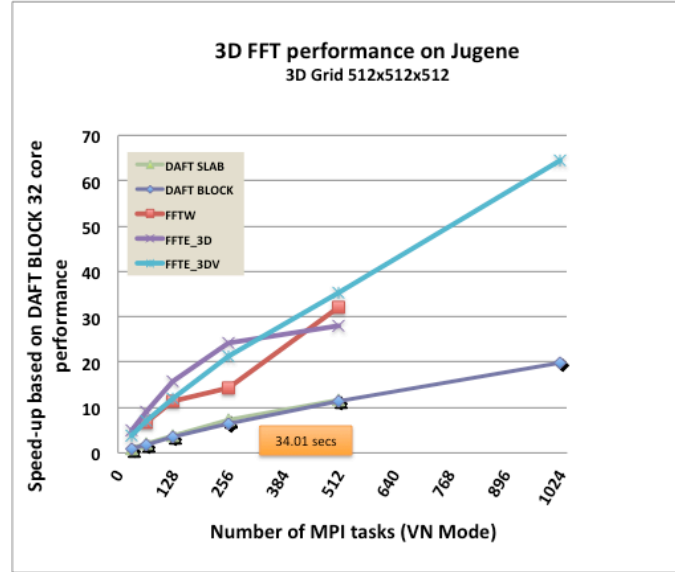


Fig. 4. Relative performance of FFT library routines for 512^3 dataset on JUGENE. Timing label refers to runtime of DAFT Block run on 32 cores.

The FFT performance results from JUGENE in

Fig. 2-4 show generally good scaling behaviour for all the tested routines and datasets. However, the speed of both FFTE and FFTW is noticeably faster, generally around 2-3 times faster, than the existing DL_POLY DAFT routines. FFTE_3DV is slightly slower than FFTE_3D for the smaller datasets but is the best performing and best scaling routine tested here for the 512^3 dataset.

4.4. CURIE Results

All runs on CURIE used 32 MPI tasks per node, thereby fully populating the 32-way nodes with MPI tasks. All jobs in this study requested exclusive use of their nodes.

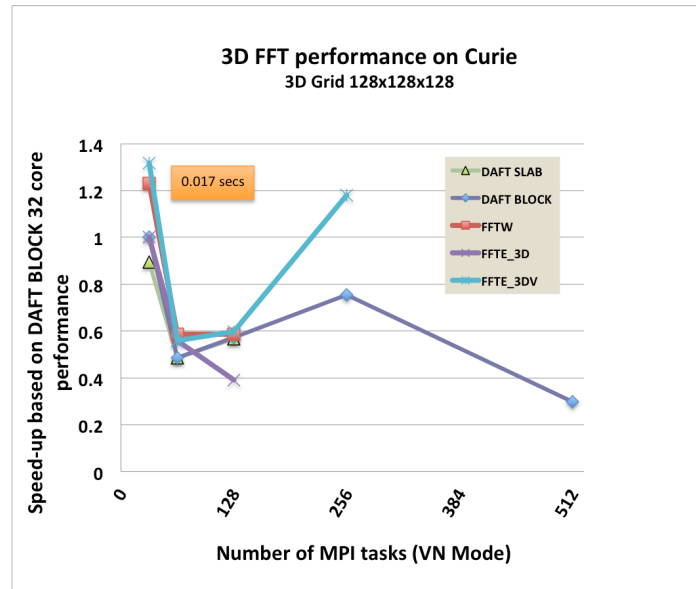


Fig. 5. Relative performance of FFT library routines for 128^3 dataset on CURIE. Timing label refers to runtime of DAFT Block run on 32 cores.

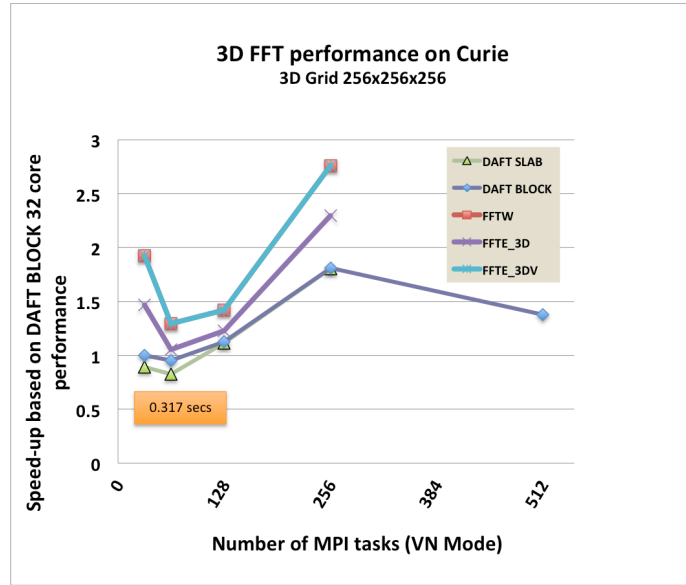


Fig. 6. Relative performance of FFT library routines for 256^3 dataset on CURIE. Timing label refers to runtime of DAFT Block run on 32 cores.

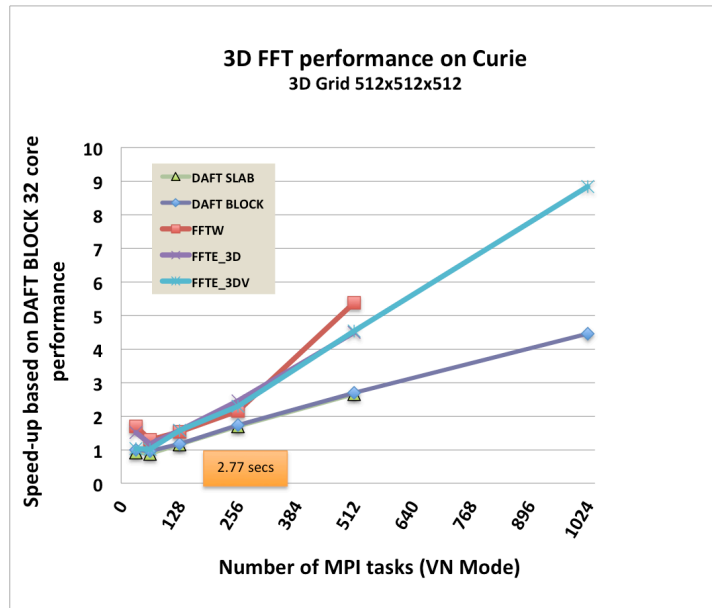


Fig. 7. Relative performance of FFT library routines for 512^3 dataset on CURIE. Timing label refers to runtime of DAFT Block run on 32 cores.

Unlike the performance results from JUGENE, the parallel scaling curves obtained from CURIE and shown in Fig. 5-7 show that there is a noticeable degradation in performance when runs involve more than one 32-way compute node. As expected, these findings suggest therefore that undertaking off-node communications on CURIE is less efficient than within-node communications and can have a significant impact upon parallel performance. This slow-down appears most significantly in the results from the smaller datasets, where a multitude of frequent all-to-all small messages are required and the parallel performance of all the library routines investigated here suffers to a similar extent. The scaling problem is less apparent in Fig. 7, where off-node messages involve larger sections of data and off-node communication latency is less critical. The routine FFTE_3DV provides the best overall parallel performance.

4.5. Comparison of JUGENE and CURIE Results

The performance results for the FFTW, FFTE and DL_POLY's native FFT routine DAFT are reported in the figures above for the PRACE systems JUGENE and CURIE. The analysis shows that the parallel scaling for the FFT routines on JUGENE is superior to CURIE but outright performance is generally faster on CURIE (due to CURIE's faster processors), especially at lower core counts. On 32 cores an FFT using FFTE_3DV for the 256^3 DL_POLY dataset takes 1.14 seconds on JUGENE, compared to 0.33 seconds on CURIE. At higher core counts the performance of the two machines is closer, as the superior scaling properties of JUGENE begin to impact. For example, an FFT using FFTE_3DV on 1024 cores for the 256^3 DL_POLY dataset takes 0.065 seconds on JUGENE, compared to 0.036 seconds on CURIE.

The comparisons are somewhat unfair to DAFT, because FFTW and FFTE only support slab and pencil decompositions, and are therefore not a plug-in replacement for DAFT in DL_POLY. The results are somewhat ambiguous. Neither FFTE nor FFTW emerges as a clear winner on all problem sizes and architectures. Hence both FFTW and FFTE have potential for improvement. DAFT, although outperformed by both FFTE and FFTW on slab and pencil decompositions at low process counts, does show promising scaling characteristics.

5. A GPU-Enabled FFT Library

With the possible performance advantages of using third-party FFT libraries within DL_POLY in mind, the performance of the Parallel Three Dimensional Fast Fourier Transforms (P3DFFT) library [7] has recently been investigated at ICHEC. The library is currently being developed at the San Diego Supercomputer Centre and has recently been ported to GPUs by a group at the Georgia Institute of Technology (where the ported library is dubbed DiGPUFFT [9]). It claims to be optimized for large datasets and uses a 2D, or pencil, decomposition as opposed to a 1D or slab decomposition. It is written using Fortran 90 and MPI. On each node, P3DFFT computes local 1D FFTs using third party FFT libraries, which by default is FFTW, though IBM's ESSL and Intel's MKL may serve as drop-in replacements. For DiGPUFFT, a custom cuFFT wrapper was developed for use within P3DFFT, making NVIDIA's cuFFT library an additional local FFT option.

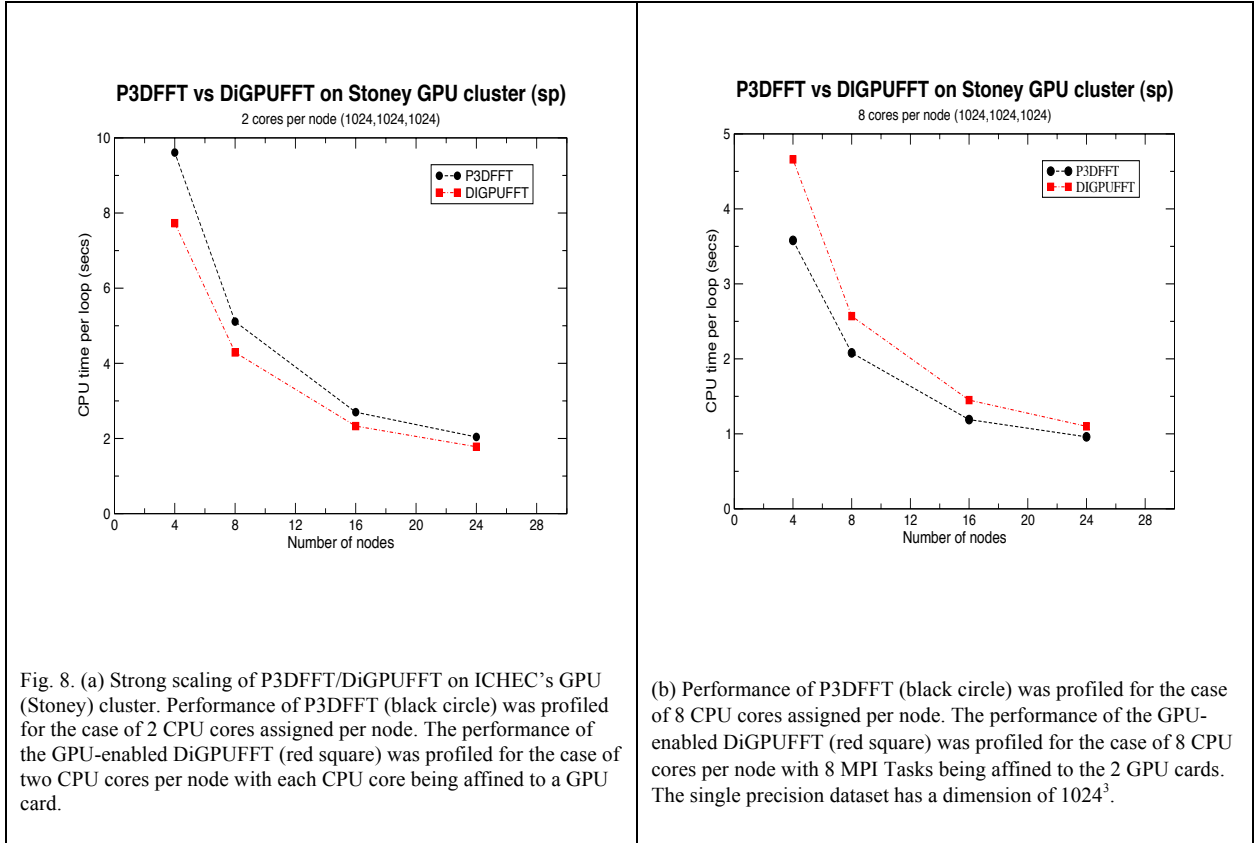


Fig. 8 shows the performance profile of P3DFFT/DiGUFFT run over all 24 nodes of ICHEC’s Stoney GPU cluster. Each of Stoney’s compute nodes has two 2.8 GHz Intel (Nehalem EP) Xeon X5560 quad-core processors and 48 GB of RAM. Twenty-four of the nodes have two NVIDIA Tesla M2090 cards installed, with each card providing 512 GPU cores and 6 GB of GDDR5 memory. P3DFFT was compiled using the FFTW library. The *driver_rand.F90* program supplied as part of the P3DFFT/DiGUFFT installation package was used in all benchmarks where timers intrinsic to this program were used to obtain results.

5.1. Under-population of compute nodes:

Fig. 8(a) shows results for the case of 2 CPU cores assigned per compute node (with the 2 MPI tasks affined to two GPU cards for the DiGUFFT results). The dataset used was randomly generated and has a dimension of 1024^3 . All calculations were carried out using single precision (currently double precision computation is unavailable in DiGUFFT). Fig. 8(a) shows that for a low node count (4 nodes), the GPU-enabled DiGUFFT achieves a performance gain of ~20% over its pure MPI P3DFFT equivalent and that the performance improvement becomes less significant as the node count increases.

5.2. Full-population of compute nodes:

Fig. 8(b) shows a similar set of results with the number of CPU cores per node increased to eight. With the nodes fully populated it can be seen that P3DFFT outperforms DiGUFFT by ~20% for lower node counts where the difference in performance, once again, becomes less significant at higher node counts. It should be noted that within DiGUFFT it is only possible to affine the extra CPU cores to MPI tasks and not OpenMP threads. Therefore the CPU/GPU hybrid setup in 8(b) is not regarded as optimal, as only a single MPI task per GPU card is recommended. However this is currently the only way to fully utilize all the cores within a node on the Stoney GPU cluster using DiGUFFT. A comparison between figures 8(a) and 8(b) shows that, for both P3DFFT and DiGUFFT, populating the nodes fully results in a factor ~1.5-2.0 improvement in performance.

We have also carried out a performance analysis of DiGUFFT for datasets with dimensions 256^3 and 512^3 . However, as expected, for the smaller FFT sizes, the CPU outperforms the GPU, due partly to the CPU still doing the local transposes, and the high cost of host-to-device memory transfers. As the FFT dimensions grow, the GPU’s faster compute time quickly overcomes this additional overhead. We have found that a dataset of dimension 512^3 is close to the performance cross-over point between the CPU and GPU implementations, which reflects this GPU memory transfer overhead.

5.3. 3D-cuFFT on a single GPU:

Finally, we have also implemented a call to NVIDIA’s 3D cuFFT routine inside DL_POLY. Currently, NVIDIA has no distributed version of its cuFFT library available so any call to the 3D cuFFT routine must occur on a single CPU affined to a single GPU. With this configuration we have found a factor of ~8 speedup over DAFT running on a single CPU for a dataset of typical size used in DL_POLY. While such an implementation will obviously not scale, the significant speedup may point to the potential benefit of using a ‘gather-scatter’ approach to the FFT problem when running DL_POLY on small GPU clusters.

6. Quantum ESPRESSO Application Background

Quantum ESPRESSO (QE) is an integrated suite of computer codes for electronic-structure calculations and materials modelling at the nanoscale [11]. It is based on density-functional theory, plane waves, and pseudo-potentials (both norm-conserving and ultrasoft).

Part of this work was to study the performance of 3D FFTs, as used within QE. Performance and scalability of QE is benchmarked by measuring the total time taken in the FFT. A 3D FFT performance comparison is undertaken between the current version 3.3.1 of FFTW and the ‘internal’ version of FFTW that is currently bundled with QE using the datasets provided both from QE distribution and from the QE FFT developers.

6.1. FFT Implementation in Quantum ESPRESSO

3D FFT is a key component of Quantum ESPRESSO, and it is implemented as a set of modules that are shared between

several executables. 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 the FFT grids by columns, the number of columns will typically be much larger than the number of MPI processes, and hence the columns can be distributed among the processes to achieve better load balancing of non-zero Fourier coefficients per process.

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 global communication, followed by a 1D FFT on the newly localised data in columns (or vice versa for the inverse transform).

6.2. Benchmarking of FFT libraries using developed in-house codes

For the purpose of performance and scalability testing of various FFT libraries, in-house benchmark codes were developed on a local PARADOX cluster at the Institute of Physics Belgrade (IPB) using C, Fortran77 and Fortran90 programming languages and the latest versions of FFTW (3.3.1) and FFTE (5.0) libraries. Since the FFTE package is distributed with Fortran source files only, a suitable FFT library was created. A comparison of FFTW and FFTE libraries was performed on CURIE and JUGENE for different types of FFT calls: 1D, 2D and 3D FFT (MPI and hybrid with MPI/OpenMP), but only results for 3D are presented in this white paper.

In order to test the hybrid FFT implementations, we have chosen to use 3D hybrid benchmark codes among all types of FFT calls (1D, 2D and 3D) as the most relevant, and they were tested similarly as in other cases. On CURIE, hybrid tests were performed with the number of threads per MPI process varying from 4 to 32 (single CURIE fat node has 32 cores) and for the total number of cores ranging from 32 to 1024. On JUGENE, hybrid tests were performed using 1-4 threads (each JUGENE node has 4 cores) per MPI process using 16 to 512 total cores. FFT testing was performed on complex array of varying sizes (up to 2^{30}). Input datasets were chosen to be comparable with the ones used in FFTW and FFTE developers test examples, both in size and operational complexity. In order to allow detailed performance analysis of the execution time of our implementation, the forward FFT was looped (in-place) 120 times on CURIE and 1000 times on JUGENE.

6.3. Benchmarking of FFT libraries used in QE

Benchmarking of QE has been performed using the internal QE FFTW copy and FFTW 3.3.1 library on CURIE. A specific dataset, developed together with QE developers, was used in order to maximize the fraction of FFT execution time in the total code execution time, and thus make it suitable for assessment and comparison. The execution was performed with the `-ntg` flag, which enables many FFTs to be performed at the same time and extends the potential scalability of the FFT.

7. Quantum ESPRESSO and in-house developed FFT benchmark codes results and interpretation

7.1. Previous Performance Results From Quantum ESPRESSO

A previous study of FFTs in Quantum ESPRESSO [16] has shown that the 3D parallel distributed FFT is the main performance bottleneck for the application. Speedup results when using pure MPI and mixed MPI-OpenMP implementations are shown in Fig. 9. As can be seen from the figure, the use of a hybrid MPI/OpenMP approach improves the performance of the code in some cases, but still the overall scalability is quite poor.

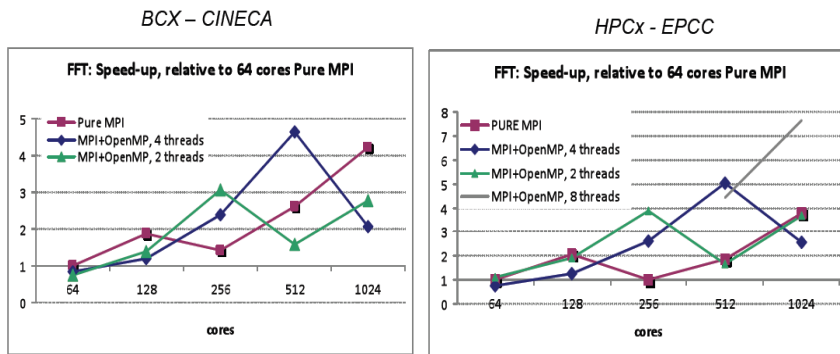


Fig. 9 Quantum ESPRESSO FFT library scalability plot for pure MPI and hybrid runs on BCX and HPCx systems, from [14].

7.2. Quantum ESPRESSO FFT results using a FFT specific input dataset

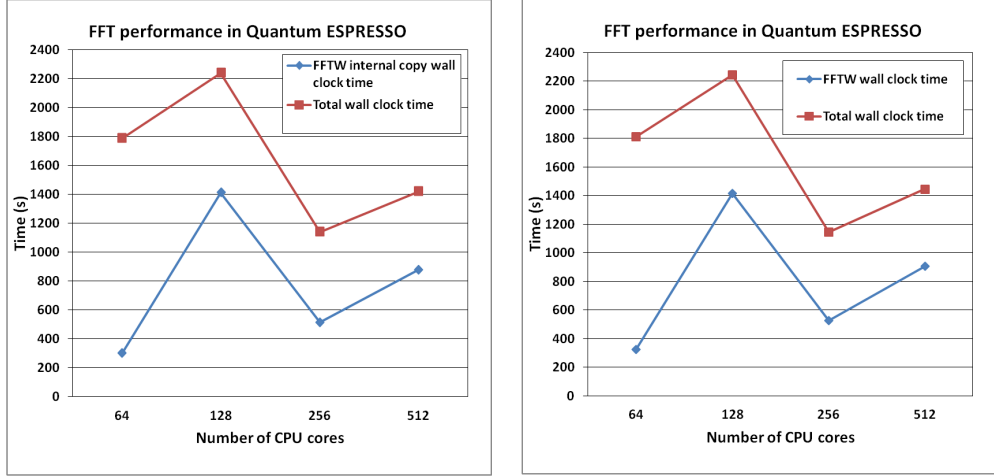


Fig.10 FFT performance in Quantum ESPRESSO on CURIE using a specific dataset emphasizing FFT operations.

Fig.10 shows the total wall clock time and the time spent on FFT for QE using the internal FFT implementation (graph on the left) and for QE linked against the FFTW 3.3.1 library (graph on the right). These runs were performed on the CURIE machine. As we can see, the results are practically identical, and both show adverse effects on the performance when a large number of MPI processes are used. We also stress large differences between CPU times and wall clock times, which may point to extensive I/O overheads.

7.3. In-house developed FFT benchmark codes results

Using the in-house developed FFT benchmark code, we have compared the execution times of the considered libraries for 3D Fourier transform computation of the 3D mesh with dimensions 1024^3 on CURIE and 256^3 on JUGENE (due to the memory limitations of the JUGENE nodes, a smaller grid was used in this case).

7.3.1. CURIE results

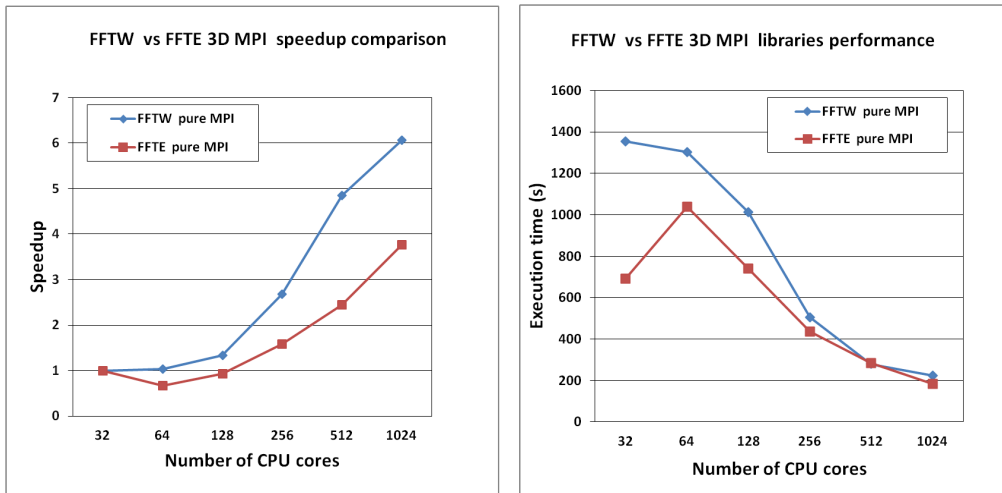


Fig. 11 Comparison of FFTE and FFTW pure MPI performance for 1024^3 dataset on CURIE: (a) speedup plot (32 cores execution times used as a baseline); (b) execution times plot.

As presented in Fig. 11, the FFTW 3.3.1 library demonstrates better scalability than FFTE, but FFTE performs faster (achieves lower execution times) than FFTW when pure MPI implementations are compared on CURIE.

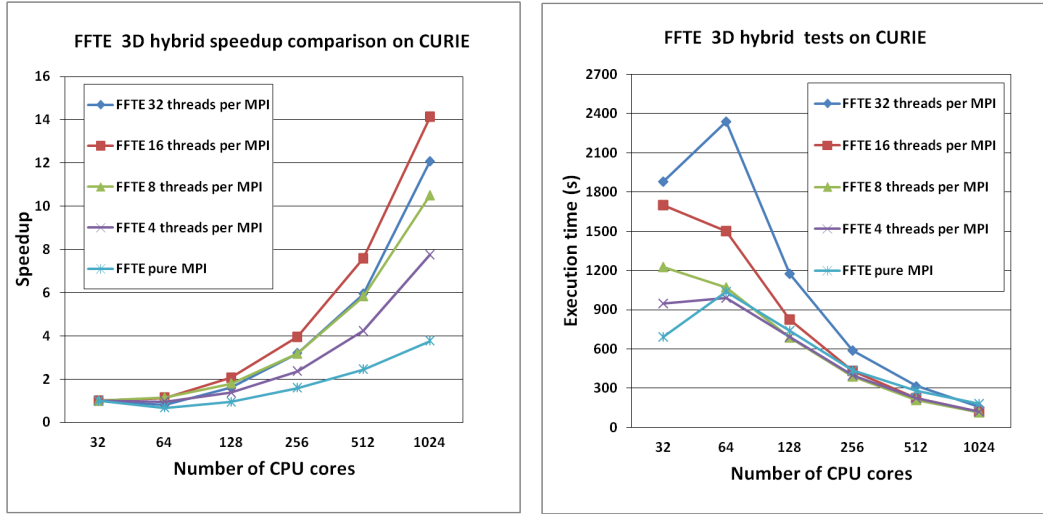


Fig. 12. Comparison of FFTE hybrid performance for different MPI/threads combinations using 1024^3 dataset on CURIE: (a) speedup plot (32 cores execution times used as a baseline); (b) execution times plot.

Fig. 12 shows that the best scaling is achieved when running with 16 threads per MPI process and that the fastest hybrid combination is the one with 4 threads per MPI process. From this figure we can also see that the FFTE library implemented with pure MPI scales worse than the hybrid implementation for all tested combinations of processes and threads. However, Fig. 12(b) shows absolute execution times, and we see that tests performed with pure MPI are faster than hybrid tests with both 32 and 16 threads per MPI process, and are comparable to hybrid runs with 4 and 8 threads per MPI process. As it can be observed, execution times for threaded runs increase as the number of threads per MPI process increases. This points to severe overheads related to the thread initialization and management.

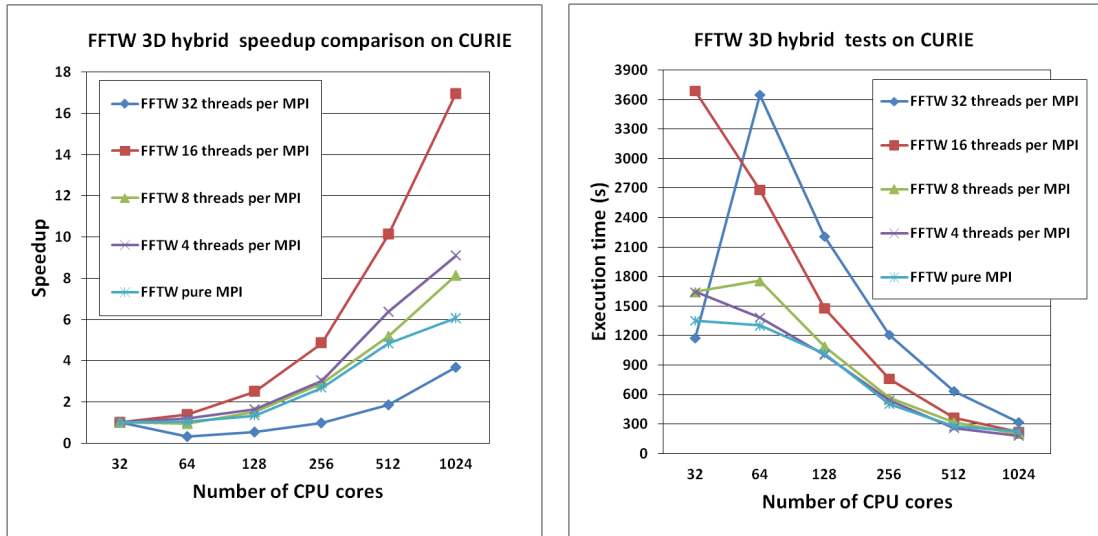


Fig. 13. Comparison of FFTW hybrid performance for different MPI/threads combinations using 1024^3 dataset on CURIE: (a) speedup plot (32 cores execution times used as a baseline); (b) execution times plot.

Fig. 13 shows hybrid tests for the FFTW library with 4, 8, 16 and 32 threads per MPI processes. The tests performed on CURIE show that the best scaling is achieved when running with 16 threads, as in the case of the FFTE library. Also, the fastest hybrid combination is the one with 4 threads, the same as in the case of FFTE library. Pure MPI results are shown for comparison and it

can be seen that pure MPI results are comparable with the fastest hybrid implementation. We have observed unusual performance for the case with a single MPI process and 32 threads, where performance is significantly better. This is probably due to the internal implementation of the hybrid version of the library, and this case needs further investigation using appropriate tools.

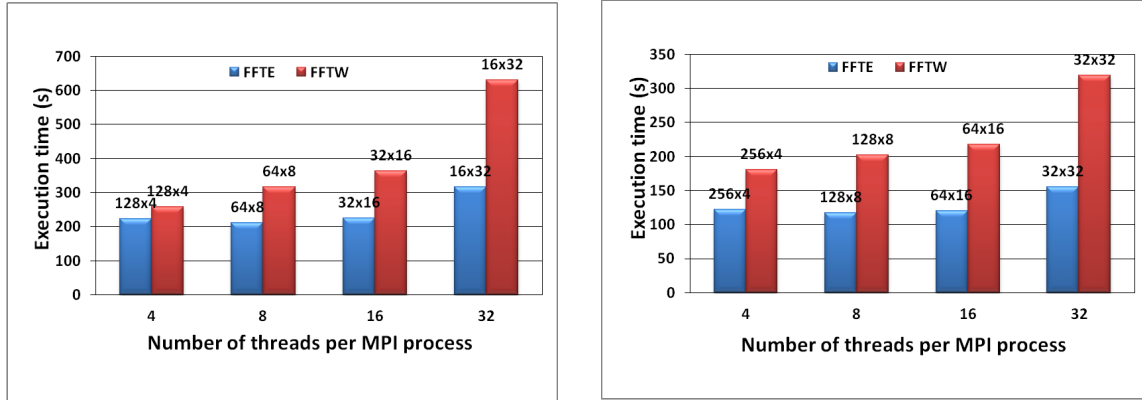


Fig. 14. Comparison of FFTE and FFTW hybrid performance on CURIE: (a) 512 total cores; (b) 1024 total cores

Fig. 14 shows that FFTE library performs faster than FFTW for all hybrid combinations, which were tested on 512 and 1024 total cores on the CURIE machine.

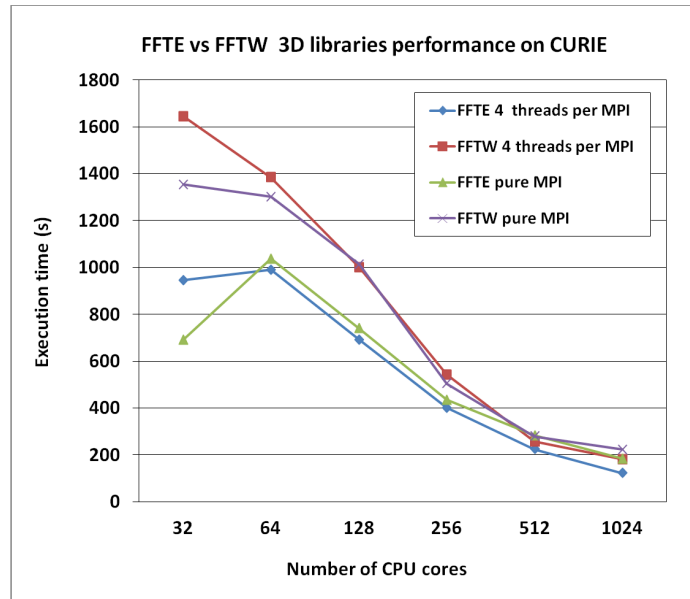


Fig. 15. Summary of the best-achieved results of FFT libraries on CURIE.

Fig. 15 shows execution times of the best hybrid (4 threads per MPI process) and pure MPI implementations results for both libraries. Apart from the case with 32 total cores, both the MPI and hybrid versions show very similar performance, with hybrid versions performing slightly faster as the number of cores grows (clearly visible in the case of the FFTE library). Because of that, we recommend using a hybrid implementation when the total number of cores is sufficiently large.

7.3.2. JUGENE results

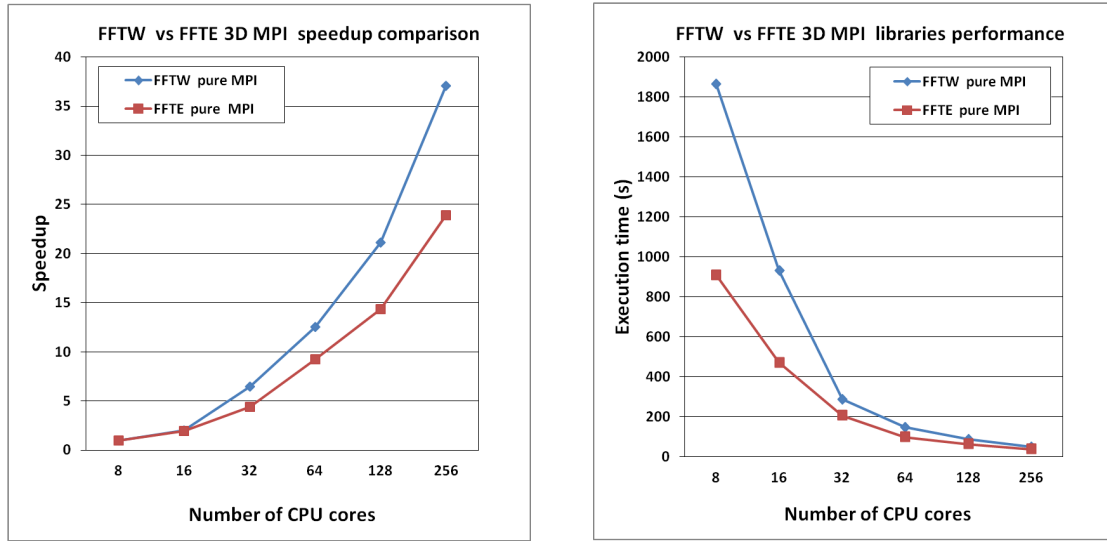


Fig. 16. Comparison of FFTE and FFTW pure MPI performance for the 256^3 dataset on JUGENE: (a) speedup plot (8 cores execution times used as a baseline); (b) execution times plot.

Fig. 16 shows that again FFTW 3.3.1 library scales better than FFTE, but the FFTE library is faster than FFTW in absolute execution times when implemented with pure MPI on JUGENE.

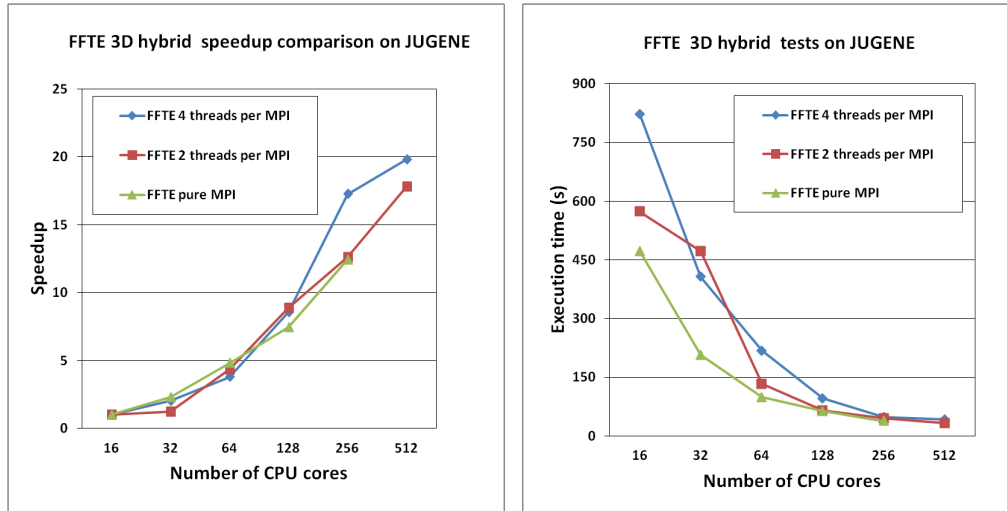


Fig. 17. Comparison of FFTE hybrid performance for different MPI/threads combinations using 256^3 dataset on JUGENE: (a) speedup plot (16 cores execution times used as a baseline); (b) execution times plot.

Fig. 17 presents hybrid tests for the FFTE library with pure MPI, as well as for 2 and 4 threads per MPI process. The tests performed on JUGENE show that better scaling is achieved when 4 threads are used. However, again in Fig. 17(b) we see that the pure MPI implementation is the fastest.

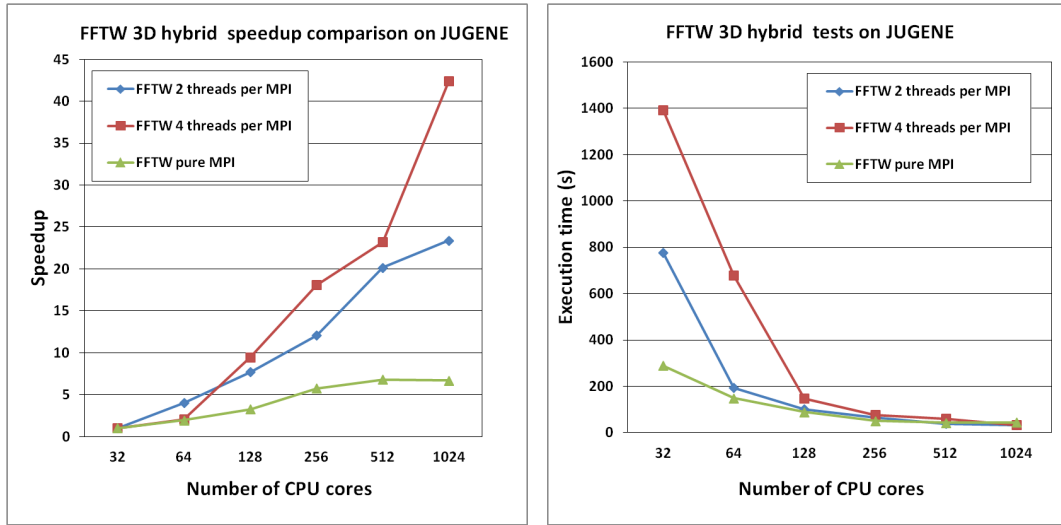


Fig. 18. Comparison of FFTW hybrid performance for different MPI/threads combinations using 256^3 dataset on JUGENE: (a) speedup plot (32 cores execution times used as a baseline); (b) execution times plot.

Fig. 18 shows the hybrid tests for FFTW 3.3.1 library with pure MPI, as well as for 2 and 4 threads per MPI process. Tests performed on JUGENE show that better scaling is achieved when running with 4 threads than with 2 threads per MPI process. It is interesting to notice that in both cases this library shows excellent scaling on the JUGENE system. Fig. 18(b) shows that the FFTW library is faster for tests with 2 threads per MPI process than tests with 4 threads in all cases, but that the pure MPI implementation outperforms all others.

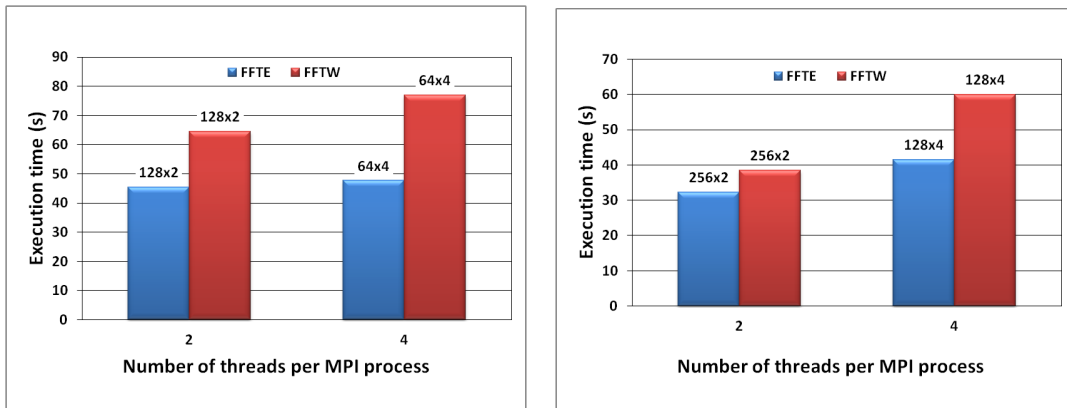


Fig. 19. Comparison of FFTE and FFTW hybrid performance on JUGENE: (a) 256 total cores; (b) 512 total cores

As shown in Fig. 19, FFTE outperforms FFTW for all mentioned hybrid combinations.

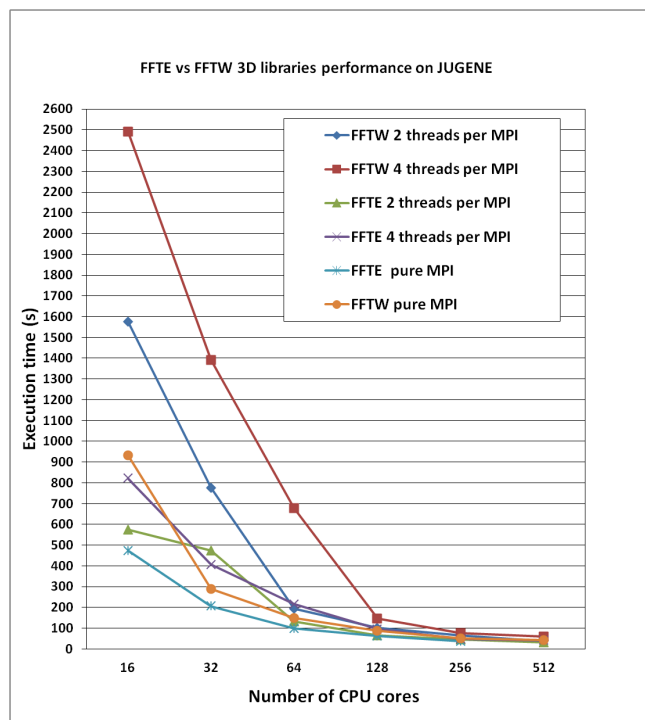


Fig. 20. Summary of all FFT libraries comparison results on JUGENE.

Finally, Fig. 20 summarizes execution times for all the MPI and hybrid runs of both libraries on JUGENE.

8. Conclusions and Recommendations

The scalability of parallel 3D FFTs remains inherently limited, owing to the all-to-all communications involved. Likewise the variety of data decompositions supported by the available libraries is also limited. While these limitations of numerical library routines remain to be the case, we will continue to see FFT-dependent applications using custom parallel FFTs with bespoke communications, and little re-use of library code, often restricted to serial or threaded FFTs within a single MPI process.

It is clear that exploiting shared memory within a node can help improve the scalability of slab and pencil decompositions [2], and from Fig. 20 there is some evidence that hybrid MPI-OpenMP implementations in some of the libraries considered are benefitting from this. Given the current state of affairs, it is difficult for application developers to rely on 3rd party libraries to achieve portable and scalable FFT performance. Due to the limitations of slab decompositions it will be necessary to switch to pencil or even to block decompositions to both i) enable parallel runs on large numbers of nodes and cores associated with petascale architectures and ii) exploit these architectures for best performance. However, the precise cross-over points depend heavily on problem size and system characteristics.

Compared to other parallel molecular dynamics and computational chemistry codes, DL_POLY is distinctive in employing a 3D block decomposition throughout. Consequently, to employ a 3rd party parallel FFT library in DL_POLY would require additional all-to-all communication phases to redistribute data before and after the library call. For the developers of DL_POLY, it is likely that DAFT is not the best choice at all problem sizes and core counts. Certainly, DL_POLY can expect to benefit more from shared memory, and the current work on a hybrid MPI & OpenMP implementation is well motivated. Whether it is worthwhile to modify DL_POLY so that it can exploit 3rd party parallel FFT libraries is a decision that the developers will have to make.

We have also investigated the potential benefits of using GPU-enabled third-party FFT libraries within DL_POLY. While other investigations point to a potential performance benefit of implementing third-party FFT libraries within DL_POLY, the performance benefit of using a GPU-enabled FFT library over a pure MPI-based FFT library is, for the moment, less clear. While figure 7(a) does indicate a performance advantage in using DiGUFFT over P3DFFT, it should be noted that the dataset is 16 times larger than typical datasets used in DL_POLY. It should also be noted that the results shown here are for a single precision dataset, whereas DL_POLY uses double precision datasets. The DiGUFFT library is currently only enabled for single precision calculations and it is expected that any performance advantage currently seen over P3DFFT will diminish further when using double precision. FFTE is found to be faster than other FFT implementations. Presented results show that FFTW 3.3.1 library has better scalability than FFTE, but FFTE performs faster (achieves lower execution time) than FFTW when pure MPI

implementations are compared. The best scaling for hybrid runs is achieved when running with 16 threads on CURIE and 4 threads on JUGENE, for both libraries. In total, the FFTE hybrid implementation with 4 threads per MPI process showed the best performance on CURIE, but FFTE pure MPI implementation was comparable and it can even outperform the hybrid one on some platforms, as JUGENE results have shown.

The main bottleneck in performing the 3D FFT in Quantum Espresso is communication between 1D and 2D instances of the FFT function. Therefore, future work should include implementing 3D FFT function to replace the 1D/2D combination for performing 3D FFT, thus overcoming current problems. Based on results presented in this whitepaper, future work should also include an implementation of the FFTE library into Quantum Espresso. P3DFFT library performs only real-to-complex and complex-to-real FFT transforms and as such is not suitable for use by Quantum Espresso, as it requires library that can perform complex-to-complex transform.

Although, while performing FFT 3D transforms, MPI communication remains a main performance bottleneck, additional parallelization levels (on electron states, on k-points, if available) would allow extended scalability.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources [BULL Bullx (CURIE), France; Blue Gene/P (JUGENE), Germany, PARADOX Cluster at IPB].

The authors would also like to express their gratitude to Ian Bush of NAG Ltd UK for providing information on the use of FFTs in DL_POLY.

References

1. The DL_POLY Molecular Simulation Package: http://www.cse.scitech.ac.uk/ccg/software/DL_POLY/
2. S.M. Pickles, "A Study of FFT Usage in DL_POLY and CASTEP", Open Petascale Libraries Research Report, <http://www.openpetascale.org/index.php/public/page/documents>
3. FFTW Home Page, <http://www.fftw.org/>
4. FFTE: A Fast Fourier Transform Package, <http://www.ffte.jp/>
5. I.J. Bush and I.T. Todorov, "A DAFT DL_POLY distributed memory adaptation of the smoothed particle mesh Ewald method", *Comp. Phys. Commun.* 175 (5) 323-329 (2006), doi:10.1016/j.cpc.2006.05.001.
6. Clive Temperton, "A Generalized Prime Factor FFT Algorithm for any $N = 2^p 3^q 5^r$ ", *SIAM J. Sci. and Stat. Comput.*, 13(3), 676-686.
7. P3DFFT Home Page, <http://code.google.com/p/p3dfft/>
8. CASTEP Home Page: www.castep.org
9. DiGPUFFT Home Page, <http://code.google.com/p/digpufft/>
10. Open Petascale Library Project home page, <http://www.openpetascale.org/>
11. <http://www.quantum-espresso.org>
12. <http://supertech.csail.mit.edu/cilk/index.html>
13. Daisuke Takahashi, "A Hybrid MPI/OpenMP Implementation of a Parallel 3D FFT on SMP Clusters", *Proc. 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005)*, Lecture Notes in Computer Science, No. 3911, pp. 970-977, Springer-Verlag (2006)
14. <http://www.sdsc.edu/us/resources/p3dfft/>
15. http://users.aims.ac.za/~sandro/files/tutorial_para.pdf
16. http://www.ichec.ie/education_training/qe_workshop/material/05/qews_cc_parallel.pdf