An Analysis of FFTW and FFTE Performance

Miloš Nikolić, Aleksandar Jović, Josip Jakić, Vladimir Slavnić, and Antun Balaž

Scientific Computing Laboratory, Institute of Physics Belgrade,
University of Belgrade, Pregrevica 118, 11080 Belgrade, Serbia
{milos.nikolic,aleksandar.jovic,josip.jakic,
vladimir.slavnic,antun.balaz}@ipb.ac.rs

Abstract. One of the most frequently used algorithms in engineering and scientific applications is Fast Fourier Transform (FFT). Its open source implementation (Fastest Fourier Transform of the West, FFTW) is widely used, mainly due to its excellent performance, comparable to the vendor-supplied libraries. On the other hand, even if not yet in a fully production state, FFTE (Fastest Fourier Transform of the East) keeps up with FFTW, and outperforms it for very large transform sizes. Here we present results of the performance and scalability tests of FFTW and FFTE libraries. Comparison is done using different compilers and parallelization approaches on CURIE and JUGENE supercomputers.

Keywords: FFT, MPI, OpenMP, Hybrid parallelism.

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 [1] devised an algorithm to compute the DFT of an n-point series in $n*\log(n)$ 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. It presents a challenge scaling up those codes that rely heavily on FFT methods to exploit existing and future Petascale supercomputing systems.

The goal of this paper is to assess the performance and scalability of various implementations of FFT. Specific FFT benchmark codes were developed and used to compare performance of different 3D FFT library routines and explore their scalability in the strong sense on CURIE [2] and JUGENE [3] supercomputers, provided by PRACE [3] association.

M. Dulea et al. (eds.), *High-Perf. Comp. Infrastr. for South East Europe's Research Communities*, Modeling and Optimization in Science and Technologies 2,
DOI: 10.1007/978-3-319-01520-0_20,

In section 2, we introduce various FFT methods used in this study. In section 3 we give a description of benchmarking procedures for FFT libraries using in-house developed FFT test codes. Section 4 presents benchmarks results and, finally, in section 5, we summarize 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 Massachusetts Institute of Technology by Matteo Frigo and Steven G. Johnson [5]. It is open source and free library written in C, but also has Fortran bindings. It supports transforms of arbitrary sizes. 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 coordinate).

2.2 FFTE

FFTE [6] 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. In other cases but

the results will be invalid but the program would run nevertheless, so you should be carefull. In FFTE, 3D parallel FFTs must be decomposed over MPI processes so that the leading coordinate (x) of the 3D arrays (x, y, z) is kept local to each MPI process.

In this study, we used version 5.0 of FFTE. We employed both PZFFT3D, a parallel 3D DFT method which requires that the data is decomposed over MPI processes in the z-coordinate (i.e. it supports only a slab decomposition), and PZFFT3DV, which allows data decomposed in both the y and z coordinate (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.

3 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, at the time. 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 and dimensions of FFT calls (MPI and hybrid with MPI/OpenMP) and we have chosen to use 3D hybrid benchmark codes among them as the most relevant. Obtained measurement results on these codes are presented.

CURIE is located in the computing center of CEA (TGCC) at Bruyères-le-Chatel in France. We used BULLX Fat nodes which have four eight-core Intel Nehalem-EX X7560 processors with 128 GB of memory. JUGENE is located in The Jülich Supercomputing Centre in Germany. It is based on IBM BlueGene/P architecture with four PowerPC 450 32-bit cores and 2 GB of memory in each compute node.

On CURIE, hybrid tests were performed with the number of threads per MPI process varying from 4 to 32 and for the total number of cores ranging from 32 to 1024. On JUGENE, hybrid tests were performed using 1-4 threads per MPI process using 16 to 512 total cores. FFT testing was performed on complex array of varying sizes (up to 230). 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.

4 FFT Benchmark Codes Results and Interpretation

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).

4.1 CURIE Results

As presented in Fig. 1 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.

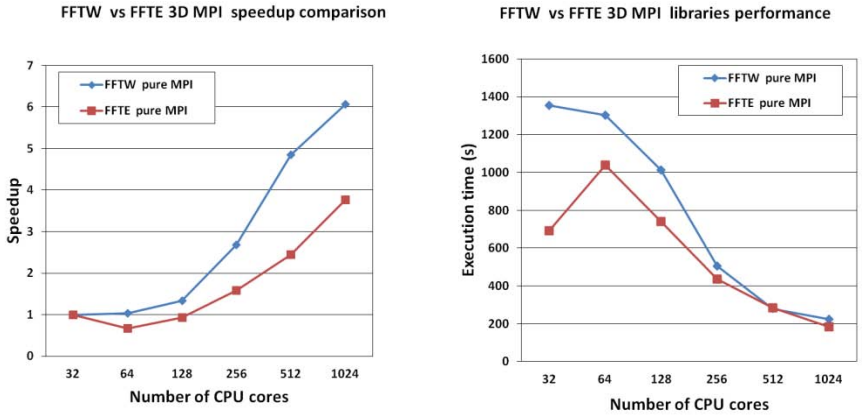

Fig. 1. Comparison of FFTE and FFTW pure MPI performance for 1024^3 dataset on CURIE: (left) speedup plot (32 cores execution times used as a baseline); (right) execution times plot

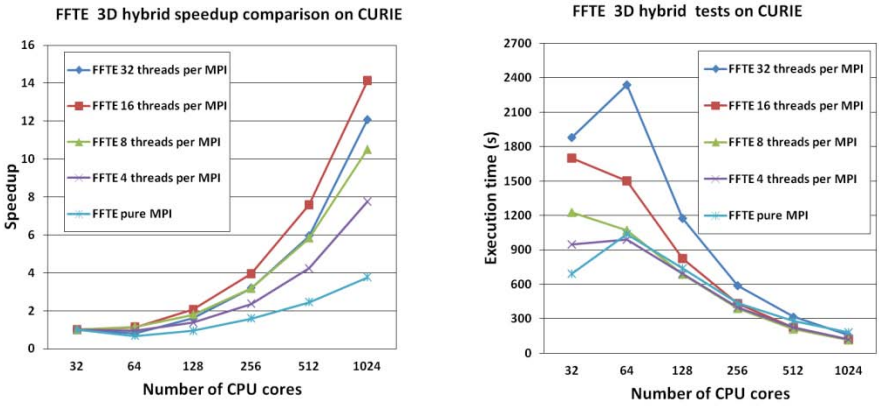

Fig. 2. Comparison of FFTE hybrid performance for different MPI/threads combinations using 1024^3 dataset on CURIE: (left) speedup plot (32 cores execution times used as a baseline); (right) execution times plot

Figure 2 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. 2. (right) 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 can be due to overheads related to the thread initialization and management, but also due to different ways memory allocation is performed in NUMA environment with a process-oriented configuration (MPI) and a thread-oriented configuration (OpenMP).

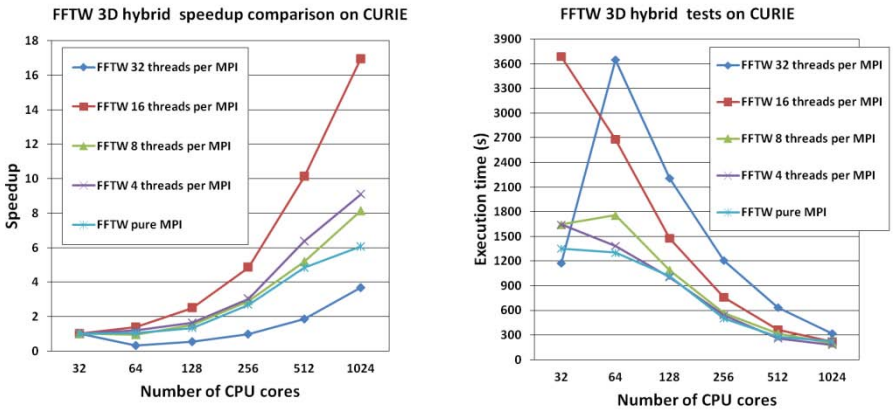

Fig. 3. Comparison of FFTW hybrid performance for different MPI/threads combinations using 1024^3 dataset on CURIE: (left) speedup plot (32 cores execution times used as a baseline); (right) execution times plot

Figure 3 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 of 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.

Figure 4 shows that FFTE library performs faster than FFTW for all hybrid combinations, which were tested on 512 and 1024 cores on the CURIE machine.

Apart from the case with the total of 32 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.

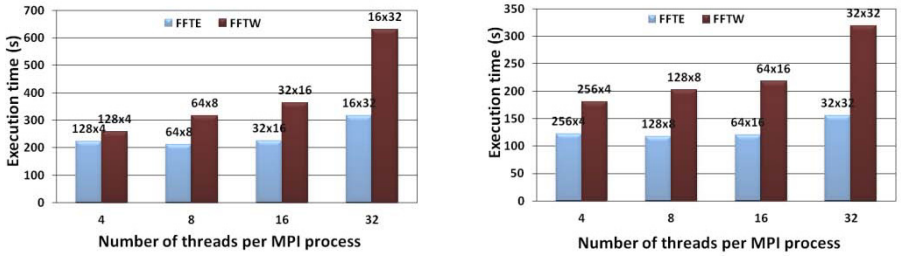

Fig. 4. Comparison of FFTE and FFTW hybrid performance on CURIE: (left) 512 total cores; (right) 1024 total cores

4.2 JUGENE Results

Figure 5 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.

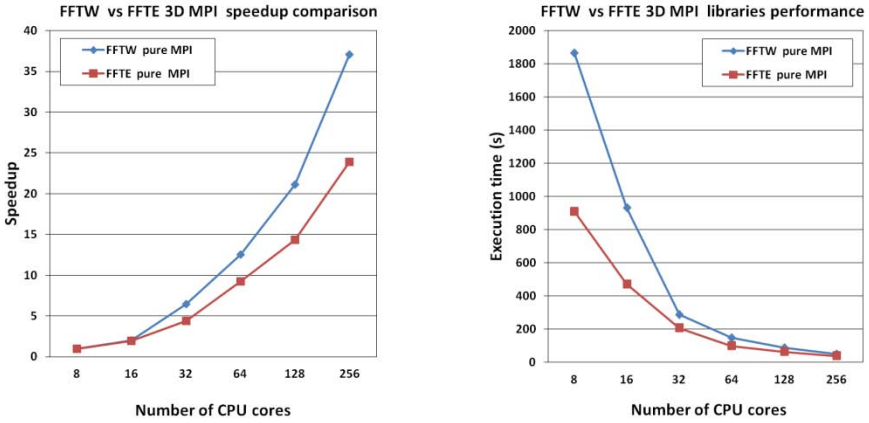

Fig. 5. Comparison of FFTE and FFTW pure MPI performance for the 256^3 dataset on JUGENE: (left) speedup plot (8 cores execution times used as a baseline); (right) execution times plot

Figure 6 presents hybrid tests for the FFTE library with pure MPI, as well as for two and four threads per MPI process. The tests performed on JUGENE show that better scaling is achieved when four threads are used. However, again in Fig. 6 (right) we see that the pure MPI implementation is the fastest.

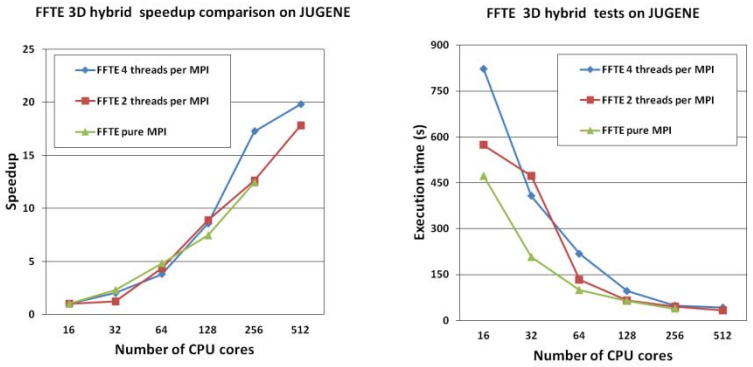

Fig. 6. Comparison of FFTE hybrid performance for different MPI/threads combinations using 256^3 dataset on JUGENE: (left) speedup plot (16 cores execution times used as a baseline); (right) execution times plot

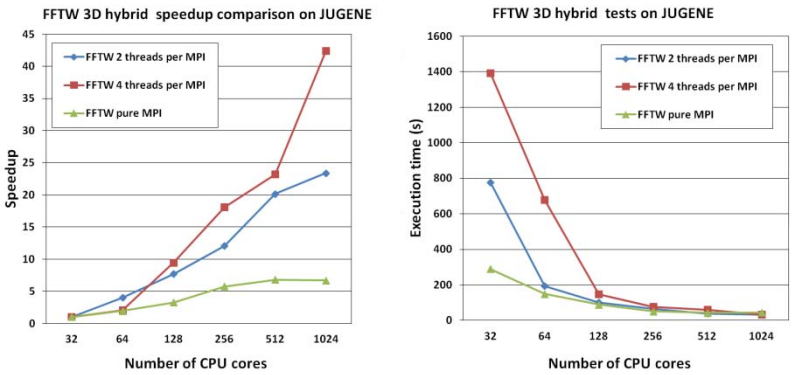

Fig. 7. Comparison of FFTW hybrid performance for different MPI/threads combinations using 256^3 dataset on JUGENE: (left) speedup plot (32 cores execution times used as a baseline); (right) execution times plot

Fig. 7 shows the hybrid tests for FFTW 3.3.1 library with pure MPI, as well as for two and four threads per MPI process. Tests performed on JUGENE show that better scaling is achieved when running with four threads than with two threads per MPI process. It is interesting to notice that in both cases this library shows excellent scaling on the JUGENE system. Fig. 7 (right) shows that the FFTW library is faster for tests with two threads per MPI process than tests with four threads in all cases, but that the pure MPI implementation outperforms all others.

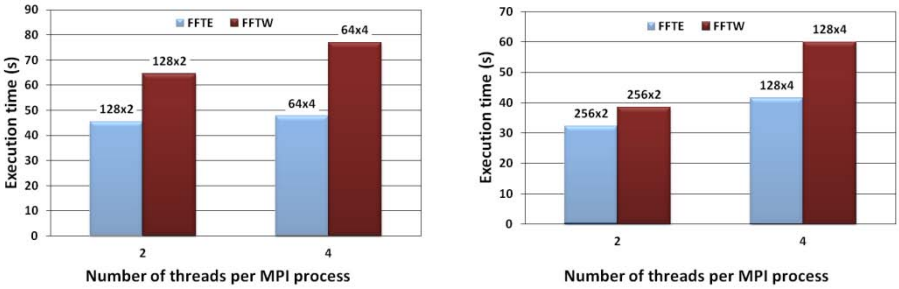

Fig. 8. Comparison of FFTE and FFTW hybrid performance on JUGENE: (left) 256 cores; (right) 512 cores

5 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. . Given the current state of affairs, it is difficult for application developers to rely on third party libraries to achieve portable and scalable FFT performance. 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 benefits of shared memory within a node can help improve the scalability and for this reason using a hybrid implementation, when the total number of cores is sufficiently large, is recommended.

Acknowledgements. The work is achieved using the PRACE Research Infrastructure resources [BULL Bullx (CURIE), France; Blue Gene/P (JUGENE), Germany] and PARADOX Cluster at the Scientific Computing Laboratory of the Institute of Physics Belgrade, supported in part by the Serbian Ministry of Education, Science and Technological Development under projects No. ON171017 and III43007, and by the European Commission under FP7 projects HP-SEE, PRACE-2IP, PRACE-3IP and EGI-InSPIRE.

References

1. Cooley–Tukey FFT algorithm, `http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`
2. CURIE Supercomputer, `http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm`
3. JUGENE Supercomputer, `http://www.fz-juelich.de/jsc/jugene`
4. PRACE Home Page, `http://www.prace-ri.eu/`
5. FFTW Home Page, `http://www.fftw.org/`
6. FFTE: A Fast Fourier Transform Package, `http://www.ffte.jp/`