An Efficient and Flexible Parallel FFT Implementation Based on FFTW

Michael Pippig

Abstract   In this paper we describe a new open source software library called PFFT [12], which was developed for calculating parallel complex to complex FFTs on massively parallel architectures. It combines the flexible user interface and hardware adaptiveness of FFTW [7] with a highly scalable two-dimensional data decomposition. We use a transpose FFT algorithm, that consist of one-dimensional FFTs and global data transpositions. For the implementation we utilize the FFTW software library. Therefore we are able to generalize our algorithm straight forward to d-dimensional FFTs, $d \geq 3$, real to complex FFTs and even completely in place transformations. Further retained FFTW features like the selection of planning effort via flags and a separate communicator handle distinguish PFFT from other public available parallel FFT implementations. Automatic ghost cell creation and support of oversampled FFTs complete the outstanding flexibility of PFFT. Our runtime tests up to 262144 cores of the BlueGene/P supercomputer prove PFFT to be as fast as the well known P3DFFT [11] software package, while the flexibility of FFTW is still preserved.
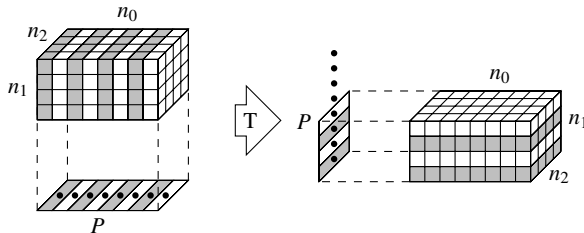
1 Introduction

The fast Fourier transform (FFT) provides the basis of many algorithms in scientific computing. Hence a highly scalable implementation for massively parallel systems such as BlueGene/P is desirable. There are two approaches to parallelize multi-dimensional FFTs, first binary exchange algorithms and second transpose algorithms. An introduction and theoretical comparison can be found in [9]. We concentrate on transpose algorithms, i.e., we perform a sequence of local FFTs and

Michael Pippig

Chemnitz University of Technology, Faculty of Mathematics, 09107 Chemnitz, Germany, e-mail: michael.pippig@mathematik.tu-chemnitz.de

global data transpositions. For convenience we consider a three-dimensional input dataset of size $n_0 \times n_1 \times n_2$ with $n_0 \geq n_1 \geq n_2$.
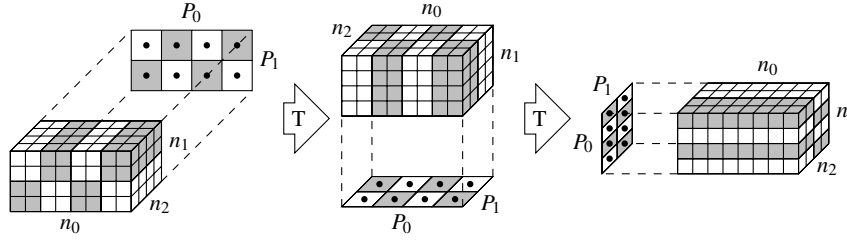
First parallel transpose FFT algorithms were based on a technique called slab decomposition, i.e., the three-dimensional dataset is partitioned along n_0 to distribute it on a given number $P \leq n_0$ of MPI processes. After calculation of the n_0 locally available two-dimensional FFTs of size $n_1 \times n_2$ a data transposition is performed, that corresponds to a call of MPI Alltoall and repartitions the three-dimensional dataset along n_1. Finally $n_1 n_2$ one-dimensional FFTs of size n_0 are computed. For example, implementations of this algorithm are included in the IBM PESSL library [5], the Intel Math Kernel Library [10] and the FFTW [7] software package, that is appreciated for its portable high-performance FFT implementations and flexible user interface. Unfortunately all of these FFT libraries lack high scalability on massively parallel systems because slab decomposition limits the number of efficiently usable MPI processes by n_1. Figure 1 shows an illustration of the one-dimensional distributed parallel FFT agorithm and an example of its scalability limitation.

Fig. 1 Distribution of a three-dimensional dataset of size $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ on a one-dimensional process grid of size $P = 8$. After the transposition (T) half of the processes remain idle

A volumetric domain decomposition was used in [4, 2, 1] to overcome the scalability bottleneck and a software library [3] for power of two FFTs customized to BlueGene/L systems was implemented. The dataset is partitioned along two dimensions and therefore the number of MPI processes can be increased to at most $n_1 n_2$. Figure 2 shows an illustration of the two-dimensional distributed parallel FFT algorithm and its improved scalability in comparison to the example in Fig. 1. Portable implementations based on two-dimensional data decomposition are the FFT package from Sandia National Laboratories [13] and the three-dimensional FFT library called P3DFFT [11]. While the FFT algorithm from Sandia National Laboratories is less restrictive on data distributions, runtime tests proved P3DFFT to perform better in most cases. It naturally turns up to ask for a public available parallel FFT library, that unifies the flexibility of FFTW and the highly scalable two-dimensional data decomposition.

This paper is divided into the following parts. First we give the necessary definitions and assumptions, that will endure through the whole paper. Next we show the

Fig. 2 Distribution of a three-dimensional dataset of size $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ on a two-dimensional process grid of size $P_0 \times P_1 = 4 \times 2$. None of the processes remains idle in any calculation step

key ideas to implement a parallel FFT algorithm totally based on the FFTW software library. Since P3DFFT is a well known and public available software package for parallel FFTs, we compare features of P3DFFT and our new PFFT package in Sect. 4 and their runtimes in Sect. 5. Finally, we summarize the most important results in Sect. 6.

2 Definitions and Assumptions

Consider a three-dimensional dataset of $n_0 \times n_1 \times n_2$ complex numbers $g_{k_0 k_1 k_2} \in \mathbb{C}$, $k_s = 0, \ldots, n_s - 1$ for all $s = 0, 1, 2$. We write the three-dimensional forward discrete Fourier transform (DFT) as

$$\hat{g}_{l_0 l_1 l_2} := \sum_{k_0=0}^{n_0-1} \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} g_{k_0 k_1 k_2} \exp\left(-2\pi i \left(\frac{l_2 k_2}{n_2} + \frac{l_1 k_1}{n_1} + \frac{l_0 k_0}{n_0}\right)\right) \in \mathbb{C},$$

where $l_s = 0, \ldots, n_s - 1$ for all $s = 0, 1, 2$. It is well known, that a multi-dimensional DFT can be calculated efficiently by successive one-dimensional fast Fourier transforms (FFTs).

We assume further the dataset to be mapped onto a two-dimensional processor grid of size $P_0 \times P_1$ such that every processor owns a block of $n_0/P_0 \times n_1/P_1 \times n_2$ complex numbers. For convenience we claim n_s to be divisible by P_r for all $s = 0, 1, 2$ and $r = 0, 1$. In order to make the following algorithms more flexible we can easily overcome these requirements. Depending on the context we interpret the notation n_s/P_r either as a simple division or as a partitioning of the dataset along dimension n_s on P_r processors in equal blocks of size n_s/P_r, for all $s = 0, 1, 2$ and $r = 0, 1$. This notation allows us to compactly represent the main characteristics of data distribution, namely the transposition and the partitioning of the dataset, with diagrams. For example we interpret the notation $n_2/P_1 \times n_0/P_0 \times n_1$, as transposed dataset of size $n_0 \times n_1 \times n_2$ that is distributed on P_0 processors along the first dimension and on P_1

processors along the last dimension. We assume such multi-dimensional datasets to be stored in C typical row major order, i.e., the last dimension lies consecutively in memory. Therefore partitioning a multi-dimensional dataset can be done most efficiently along the first dimension.

3 The Parallel Three-Dimensional FFT Algorithm

Our parallel three-dimensional FFT implementation is based on one-dimensional FFTs and two-dimensional data transpositions from FFTW. Therefore, we give a brief overview of the algorithms supported by the FFTW software library and show how to use them to build a parallel three-dimensional FFT.

3.1 One-Dimensional Serial FFT Supported by FFTW

Assume a three-dimensional dataset of $N_0 \times N_1 \times N_2$ complex numbers. The FFTW software library includes algorithms to Fourier transform single dimensions of a multi-dimensional dataset. We sketch those algorithms by the following diagrams

$$
\begin{aligned}
&N_0 \times N_1 \times N_2 \stackrel{\mathsf{FFT0}}{\to} \hat{N}_0 \times N_1 \times N_2 \;, &&N_0 \times N_1 \times N_2 \stackrel{\mathsf{FFT2}}{\to} N_0 \times N_1 \times \hat{N}_2 \;, \\
&N_0 \times N_1 \times N_2 \stackrel{\mathsf{FFT1}}{\to} N_0 \times \hat{N}_1 \times N_2 \;,
\end{aligned}
\tag{1}
$$

where FFTX indicates a one-dimensional FFT of dimension X and the hats denote Fourier transformed dimensions. In addition we are able to combine one-dimensional FFTs with cache oblivious array transpositions [8] by using the most powerful user interface of FFTW. Due to the fact that FFTW can not combine arbitrary in place transpositions with the calculation of one-dimensional FFTs [7], we restrict ourself to the interchange of two successive dimensions. Taking into account, that we can substitute two successive dimensions into a single one, we get five different transposition algorithms that we indicate by the resulting order of dimensions

$$
\begin{aligned}
&N_0 \times N_1 \times N_2 \underset{012}{\to} N_0 \times N_1 \times N_2 \;, &&(N_0 \times N_1) \times N_2 \underset{201}{\to} N_2 \times (N_0 \times N_1) \;, \\
&N_0 \times N_1 \times N_2 \underset{102}{\to} N_1 \times N_0 \times N_2 \;, &&N_0 \times (N_1 \times N_2) \underset{120}{\to} (N_1 \times N_2) \times N_0 \;, \\
&N_0 \times N_1 \times N_2 \underset{021}{\to} N_0 \times N_2 \times N_1 \;.
\end{aligned}
\tag{2}
$$

Note, that all combinations of algorithms from (1) and (2) can be performed in place as well as out of place.

3.2 Two-Dimensional Parallel Transpositions Supported by FFTW

Suppose a two-dimensional dataset of $N_0 \times N_1$ complex numbers is mapped on P processors, such that every processor holds a block of size $N_0/P \times N_1$. The FFTW3.3alpha1 software library includes a parallel matrix transposition (T) to remap the array into blocks of size $N_1/P \times N_0$. This algorithm is also used for the one-dimensional distributed parallel FFT implementations within FFTW. If we call this library function with the flags FFTW_MPI_TRANSPOSE_OUT and FFTW_MPI_TRANSPOSE_IN, we will be able to handle the following matrix transpositions

$$
\begin{aligned}
N_0/P \times N_1 & \xrightarrow{\mathsf{T}} N_1/P \times N_0 \,, & N_0/P \times N_1 & \xrightarrow[\text{OUT}]{\mathsf{T}} N_0 \times N_1/P \,, \\
N_1 \times N_0/P & \xrightarrow[\text{IN}]{\mathsf{T}} N_1/P \times N_0 \,, & N_1 \times N_0/P & \xrightarrow[\text{IN,OUT}]{\mathsf{T}} N_0 \times N_1/P \,,
\end{aligned}
\tag{3}
$$

where OUT and IN indicate that the corresponding flag was set. There are great advantages of using the parallel transposition algorithms of FFTW instead of direct calls to corresponding MPI functions. FFTW compares different matrix transposition algorithms to get the fastest one. This provides us with portable hardware adaptive communication functions. Furthermore all transpositions can be performed in place, which is impossible by calls to MPIs standard Alltoall functions and hard to program in an efficient way with point to point communications. In addition we remain many features of the flexible but easy to use interface of FFTW for our PFFT algorithms. Those features will be explained in more detail in Sec. 4.

3.3 Parallel Three-Dimensional FFT Based on FFTW

Now we suggest a new method for a parallel three-dimensional FFT, that can be computed by a careful combination of serial one-dimensional FFTs (1), local data transpositions (2) and global data transpositions (3). Assume a three-dimensional dataset of $n_0 \times n_1 \times n_2$ complex numbers that is distributed on a two-dimensional processor grid of size $P_0 \times P_1$. Then every processor holds a local data block of size $n_0/P_0 \times n_1/P_1 \times n_2$. An algorithm to perform a parallel three-dimensional FFT on this dataset is given by the following sequence of one-dimensional FFTs and data transpositions

$$
\begin{aligned}
n_0/P_0 \times n_1/P_1 \times n_2 & \xrightarrow[201]{\mathsf{FFT2}} (\hat{n}_2 \times n_0/P_0) \times n_1/P_1 \xrightarrow[\text{IN}]{\mathsf{T}} (\hat{n}_2/P_1 \times n_0/P_0) \times n_1 \\
& \xrightarrow[201]{\mathsf{FFT2}} (\hat{n}_1 \times \hat{n}_2/P_1) \times n_0/P_0 \xrightarrow[\text{IN}]{\mathsf{T}} (\hat{n}_1/P_0 \times \hat{n}_2/P_1) \times n_0 \\
& \xrightarrow[102]{\mathsf{FFT2}} \hat{n}_2/P_1 \times \hat{n}_1/P_0 \times \hat{n}_0 \,.
\end{aligned}
$$

The first step can be obtained by a combination of (1) with the substitutions $N_0 = n_0/P_0, N_1 = n_1/P_1, N_2 = n_2$ and (2) with the substitutions $N_0 = n_0/P_0, N_1 = n_1/P_1, N_2 = \hat{n}_2$, while the second step arises from (3) with the substitutions $N_0 = \hat{n}_2 \times n_0/P_0, N_1 = n_1, P = P_1$. Thereby mapping a two-dimensional block of size $\hat{n}_2 \times n_0/P_0$ on P_1 processors results in blocks of size $\hat{n}_2/P_1 \times n_0/P_0$, because of row major memory order. All remaining steps can be derived by analogous substitutions. Note that we would have to perform at least two further transpositions to return to the initial data layout. Instead we expect users to work with transposed data in Fourier space and use the same sequence as above on the transposed dataset $\hat{n}_0 \times \hat{n}_2 \times \hat{n}_1$ and transposed processor grid $P_1 \times P_0$ to calculate a three-dimensional backward FFT

$$
\begin{aligned}
\hat{n}_2/P_1 \times \hat{n}_1/P_0 \times \hat{n}_0 \; &\overset{\text{FFT2}}{\underset{201}{\to}} \; (n_0 \times \hat{n}_2/P_1) \times \hat{n}_1/P_0 \; \overset{\text{T}}{\underset{\text{IN}}{\to}} \; (n_0/P_0 \times \hat{n}_2/P_1) \times \hat{n}_1 \\
&\overset{\text{FFT2}}{\underset{201}{\to}} \; n_1 \times (n_0/P_0 \times \hat{n}_2/P_1) \; \overset{\text{T}}{\underset{\text{IN}}{\to}} \; n_1/P_1 \times (n_0/P_0 \times \hat{n}_2) \\
&\overset{\text{FFT2}}{\underset{102}{\to}} \; n_0/P_0 \times n_1/P_1 \times n_2 \, .
\end{aligned}
$$

This gives a considerable gain in performance and finally returns to the initial data layout. Of course there are many other ways to combine the algorithms of FFTW to get a three-dimensional FFT. In our implementation the user is free to choose between different combinations and their resulting data layouts.

4 Parallel FFT Software Library

Our open source parallel FFT software library (PFFT) [12] is available at `www.tu-chemnitz.de/~mpip`. We now compare the features of our PFFT software library to the P3DFFT software library. Both packages are portable open source libraries which include implementations of parallel three-dimensional FFT based on the highly scalable two-dimensional data decomposition. While P3DFFT is restricted to real to complex FFTs and written in Fortran, our software library calculates complex to complex FFTs and is written in C. Similar to the FFTW software package the user interface of PFFT is split into three layers. The basic interface depends only on the essential parameters of PFFT and is intended to provide an easy start. The possibility to calculate PFFT on a user specified communicator and the ability to change the planning flags of FFTW without recompilation of the PFFT library denote remarkable advantages over the interface of P3DFFT. Both libraries offer in place and out of place FFTs. However, P3DFFT uses buffers roughly three times the size of the input array, while PFFT supports in place algorithms that roughly half the memory consumption in comparison to the out of place algorithms.

More sophisticated adjustments to the algorithm are possible with the advanced user interface. Like P3DFFT we also support a ghost cell creation algorithm to duplicate data near the border at neighboring processors after the FFT calculation finished. In addition we offer the adjoint ghostcell algorithm, that sums up all ghost-

cells of neighboring processors. As another outstanding feature PFFT gives natural support to oversampled FFTs, i.e., the input array is filled up with zeros to reach the size of the larger output array. If we add all zeros before calling a parallel FFT, we will loose a considerable amount of performance, because some processes get parts of the input array that are filled with zeros. Our algorithm omits FFTs on vectors filled with zeros and distributes the work equally. Furthermore the advanced interface can be easily used to calculate d-dimensional FFTs, where d must be larger than 2, with the two-dimensional data decomposition. That is possible because our algorithms can work on tuples of complex numbers. To calculate a d-dimensional FFT, we call our PFFT library on complex tuples of size $n_3 \times \cdots \times n_{d-1}$ and add one call of a $(d-3)$-dimensional FFT for the last dimensions.
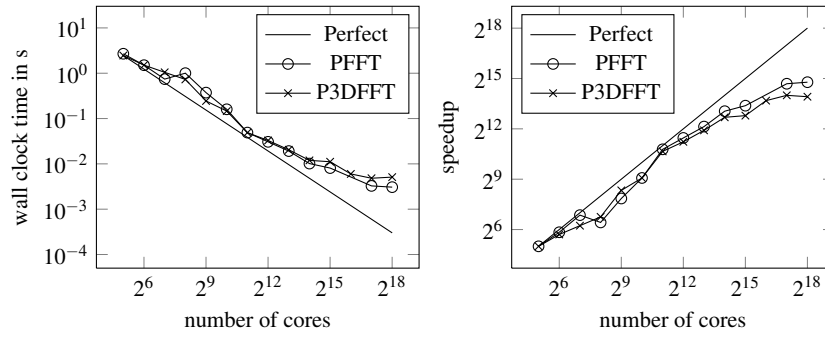
Finally the guru interface is the right choice for users who want to change the combination of serial FFTs and global data transpositions from Sect. 3.3.
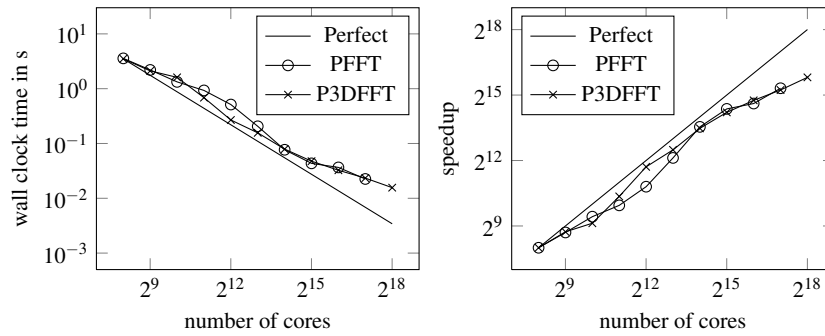
5 Runtime Measurements

In this section we compare the strong scaling behavior of P3DFFT and PFFT up to the full BlueGene/P machine in Jülich Research Center. During the Jülich Blue-Gene/P Scaling Workshop 2010 we were able to run FFTs of size 513^3 and 1024^3 on up to 64 of the available 72 racks, i.e., 262144 cores. Since P3DFFT only supports real to complex FFTs we applied P3DFFT to the real and imaginary part of a complex input array to get comparable times to the complex to complex FFTs of the PFFT package. The test runs consisted of 10 alternately calculations of forward and backward FFTs. Since these two transforms are inverse except for a constant factor, it is easy to check the results after each run. The average wall clock time as well as the average speedup of one forward and backward transformation can be seen in Fig. 3 for FFT of size 512^3 and in Fig. 4 for FFT of size 1024^3. Memory restrictions force P3DFFT to utilize at least 32 cores on BlueGene/P to calculate a FFT of size 512^3 and 256 cores to perform a FFT of size 1024. Therefore we chose the associated wall clock times as references for speedup and efficiency calculations. Note that PFFT can perform these FFTs on half the cores because of less memory consumption. However we only recorded times on core counts which both algorithms were able to utilize to get comparable results.

Unfortunately the PFFT test run of size 1024^3 on 64 racks died with a RAS event. Nevertheless our measurements show that the scaling behavior of PFFT and P3DFFT are quiet similar. Therefore we expect roughly the same runtime for PFFT of size 1024^3 on 64 racks as we observed for P3DFFT.
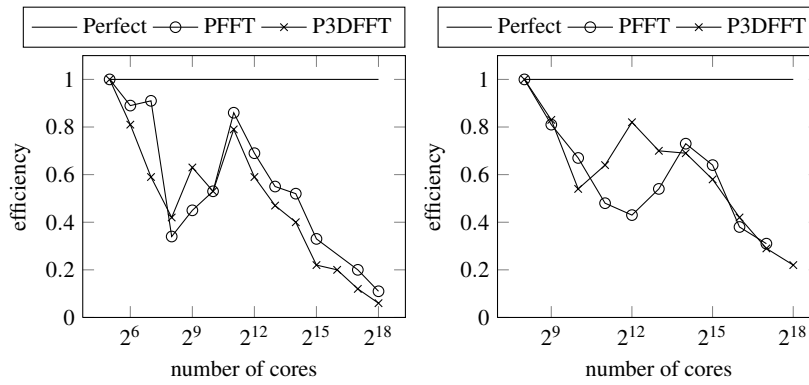
Note that 262144 is the maximum number of cores we can efficiently utilize for a FFT of size 512^3. This also means that every core calculates only one local FFT of size 512 in each of the three calculation steps. Therefore the communication takes the largest part of the runtime. The growing communication ratio for increasing core counts also explains the FFT typical decrease of efficiency seen in Fig. 5.

Fig. 3 Runtime measurements for FFT of size 512^3 up to 262144 cores

Fig. 4 Runtime measurements for FFT of size 1024^3 up to 262144 cores

Fig. 5 Efficiency for FFT of size 512^3 (left) and 1024^3 (right) up to 262144 cores

Our flexible PFFT software library can also be used to calculate d-dimensional FFTs, $d \geq 3$. For example, we analyzed the scalability of a four-dimensional FFT of size 64^4. Since our algorithm uses the two-dimensional data decomposition, we are able to efficiently utilize up to 4096 cores. Note that FFT algorithms based on the one-dimensional decomposition are limited by 64 processes. The runtime measurements given in Fig. 6 and Fig. 7 again show the high performance of our PFFT algorithms.
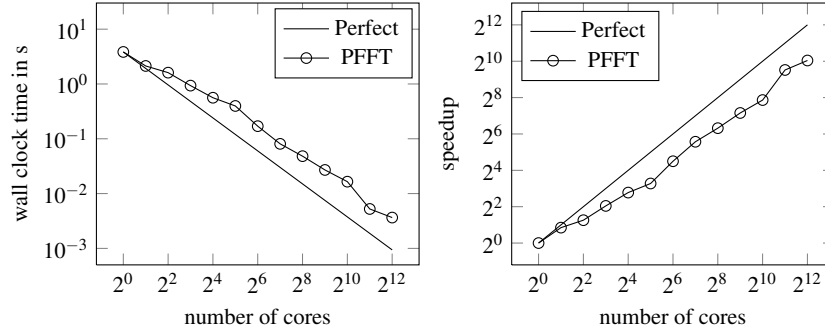
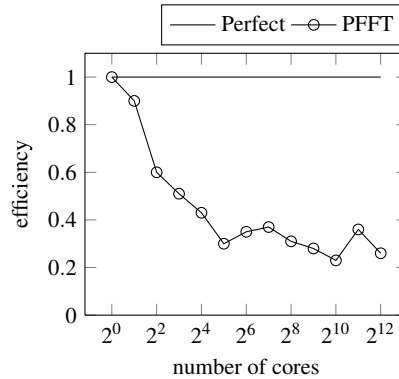**Fig. 6** Runtime measurements for FFT of size 64^4 up to 4096 cores

Fig. 7 Efficiency for FFT of size 64^4 up to 4096 cores

6 Concluding Remarks

We developed an efficient algorithm to compute d-dimensional FFTs ($d \geq 3$) in parallel on a two-dimensional processor grid. As well for one-dimensional FFTs as for MPI based communication we exploited highly optimized algorithms of the

FFTW software library [6]. Therefore the interface of our PFFT software library [12] could be easily derived from the flexible user interface of FFTW. This includes the support of parallel in place FFTs without release of high performance. Our runtime tests up to 262144 cores of the BlueGene/P supercomputer prove PFFT to be as fast as the well known P3DFFT software package [11], while the flexibility of FFTW is still preserved. To our knowledge no public available parallel FFT library has been tested to such great core counts by now. These measurements alleviate the decision-making process, whether a parallel FFT should be used to exploit the full BlueGene/P system.

References

1. Eleftheriou, M., Fitch, B.G., Rayshubskiy, A., Ward, T.J.C., Germain, R.S.: Performance measurements of the 3d FFT on the Blue Gene/L supercomputer. In: J.C. Cunha, P.D. Medeiros (eds.) Euro-Par 2005 Parallel Processing, *Lecture Notes in Computer Science*, vol. 3648, pp. 795 – 803. Springer (2005)
2. Eleftheriou, M., Fitch, B.G., Rayshubskiy, A., Ward, T.J.C., Germain, R.S.: Scalable framework for 3d FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements. IBM Journal of Research and Development **49**, 457 – 464 (2005)
3. Eleftheriou, M., Moreira, J.E., Fitch, B.G., Germain, R.S.: Parallel FFT subroutine library. URL `http://www.alphaworks.ibm.com/tech/bgl3dfft`
4. Eleftheriou, M., Moreira, J.E., Fitch, B.G., Germain, R.S.: A volumetric FFT for BlueGene/L. In: T.M. Pinkston, V.K. Prasanna (eds.) HiPC, *Lecture Notes in Computer Science*, vol. 2913, pp. 194 – 203. Springer (2003)
5. Filippone, S.: The IBM parallel engineering and scientific subroutine library. In: J. Dongarra, K. Madsen, J. Wasniewski (eds.) PARA, *Lecture Notes in Computer Science*, vol. 1041, pp. 199 – 206. Springer (1995)
6. Frigo, M., Johnson, S.G.: FFTW, C subroutine library. `http://www.fftw.org`. URL `http://www.fftw.org`
7. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**, 216 – 231 (2005)
8. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Ann. Symp. on Foundations of Comp. Sci. (FOCS), pp. 285 – 297. IEEE Comput. Soc. (1999)
9. Gupta, A., Kumar, V.: The scalability of FFT on parallel computers. IEEE Transactions on Parallel and Distributed Systems **4**, 922 – 932 (1993)
10. Intel Corporation: Intel math kernel library. URL `http://software.intel.com/en-us/intel-mkl/`
11. Pekurovsky, D.: P3DFFT, Parallel FFT subroutine library. URL `http://www.sdsc.edu/us/resources/p3dfft`
12. Pippig, M.: PFFT, Parallel FFT subroutine library. URL `http://www.tu-chemnitz.de/˜mpip`
13. Plimpton, S.: Parallel FFT subroutine library. URL `http://www.sandia.gov/˜sjplimp/docs/fft/README.html`