3D FFT with 2D decomposition

Roland Schulz

04/27/08

Abstract

Many scientific applications including molecular dynamics (MD) require a fast fourier transform (FFT). As the number of processors for high performance computer increases this transform has to be parallelized to larger number of processors to remove it as a bottleneck for the parallelization. This requires the decomposition to be changed from 1D to 2D. Such a 2D decomposed 3D FFT was implemented as this project. With the 2D decomposition the limiting factor becomes the required global transpose. This transpose can be speeded up by using FFTW transpose instead of the standard MPI `MPI_Alltoall`. Having this 2D decomposed 3D FFTW allows to improve the scaling of the fastest available MD software.

1 Introduction

Fast fourier transform (FFT) is used for many scientific applications. To be able to show its importance and my interest in it I explain its usage in the application I am most interested in which is molecular dynamics (MD) simulations. Because this requires some introduction into MD it is little bit longer and can be skipped if only the FFT itself is of interest to the reader.

MD simulations give unique insides into dynamical processes on the atomic level for biological macromolecules and non-biological materials, because the combination of times and length scales are not accessible to experiments. Recently the importance of high performance MD was discussed in nature-news [1] and portrayed as a competition between Schulten's NAMD and D.E. Shaw's Desmond. But contrary to this article, depending on the used communication library, the fastest or second fastest MD software is GROMACS [2, 3] about even with Desmond as shown in Fig. 1. Because Desmond is not publically available, GROMACS is by large margin the fastest available MD software for systems of the size of this standard benchmark. MD is computational challenging because we would like to simulate systems containing several million atoms for up to one millisecond of simulation time. MD requires to compute the force on each atom for every time-step. A time-step length is limited to 2 to 5fs. Thus to reach the desired simulation times the compute time for each time-step is limited to a thenth millisecond. The

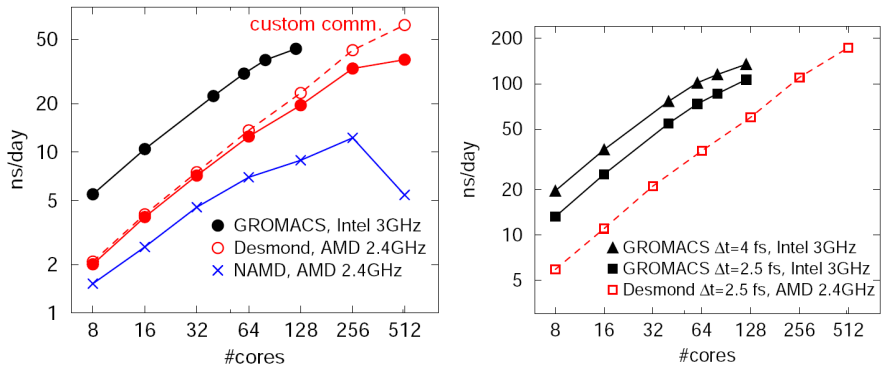

Figure 1: Desmond, NAMD and GROMACS in comparison for standard benchmark protein DHFR. The special communication is a library in replacement of MPI giving lower latency. The left shows the default parameters as specified by the benchmark. The right shows the fastest possible parameters giving accurate results.

forces are computed according to

$$\begin{aligned} V &= \sum_{\text{bonds}} k_b \left(b - b_0\right)^2 + \sum_{\text{angles}} k_\theta \left(\theta - \theta_0\right)^2 + \sum_{\text{dihedrals}} k_\phi \left[1 + \cos\left(n\phi - \delta\right)\right] \\ &+ \sum_{\text{impropers}} k_\omega \left(\omega - \omega_0\right)^2 + \sum_{\text{Urey-Bradley}} k_u \left(u - u_0\right)^2 \\ &+ \sum_{\text{nonbonded}} \epsilon \left[\left(\frac{R_{\min_{ij}}}{r_{ij}}\right)^{12} - \left(\frac{R_{\min_{ij}}}{r_{ij}}\right)^{6}\right] + \frac{q_i q_j}{\epsilon r_{ij}}. \end{aligned} \tag{1}$$

Out of these forces the non-bonded once are critical for the performance. This is because a direct summation scales as $O(N)$, with N the number of atoms, for the bonded once but as (N^2) for the non-bonded once. An algorithm which scales linearly is simple for the van der Waals forces, since this force is short-ranged and thus a cut-off method is justified. Inside a cut-off radius the number of atoms is constant (making the valid assumption of a near constant atom density) and thus also the number of interacting atoms per atom is constant and thus the total number of interaction is $O(N)$.

Because the electrostatic interaction is long-ranged a cut-off method is not sufficient for most simulations making this by far the most difficult force to compute. Different algorithms have been proposed to compute the electrostatic interaction in less than $O(N^2)$ [4]: Ewald ($O(N^{3/2})$), Particle-Particle-Particle-Mesh (P3M, $O(N \log N)$), Particle Mesh Ewald (PME, $O(N \log N)$), Multipole ($O(N)$) and Multigrid ($O(N)$). Besides Multipole they are all based on splitting the interaction into a short-ranged fast varying part and a long-ranged slowly varying part. This is done by introducing (normally gaussion shaped) counter charges for all point charges as shown in Fig 2. The sum of the point charge and the counter charge is neutral and thus their interaction is short-ranged and thus can be computed with a cut-off method. The counter charges are computed separately and are slowly varying. This fact is used by the most popular algorithm PME to compute them in k-space in $O(N)$.

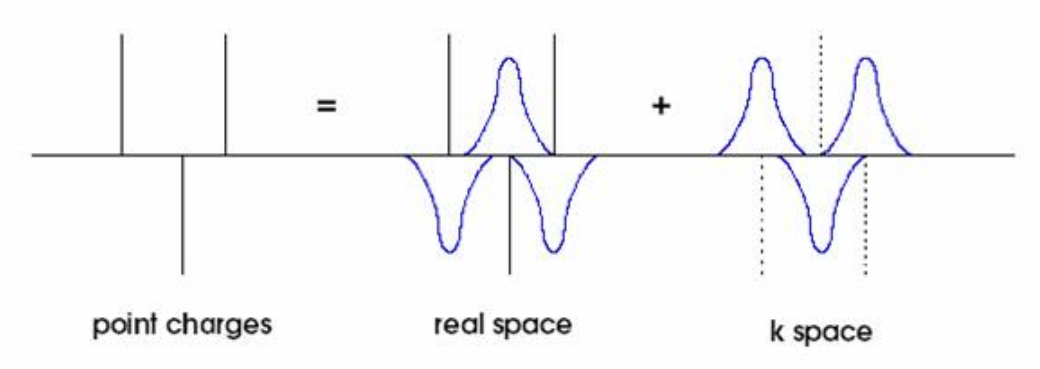

Figure 2: Counter charges used for long range electrostatic methods

Data can be transformed into from the k-space using the three dimensional fourier transform. It has been discovered several times independent from each other (most popular Cooley and Turkey 1965, first Gauss around 1805) that this can be done by a divide and conquer algorithm in $O(N \log N)$. Thus also the full PME scales as $O(N \log N)$.

The discrete fourier transformation can be written as [5]

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{2}$$

where X_k are the transformed values, x_n are the input values and N are the number of input values. By splitting the sum into two parts with the odd and even ones, one gets the recursion formula:

$$\begin{aligned} X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} && (3) \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-\frac{2\pi i}{M} mk} + e^{-\frac{2\pi i}{N} k} \sum_{m=0}^{M-1} x_{2m+1} e^{-\frac{2\pi i}{M} mk} && (4) \\ &= \begin{cases} E_k + e^{-\frac{2\pi i}{N} k} O_k & \text{if } k < M \\ E_{k-M} - e^{-\frac{2\pi i}{N}(k-M)} O_{k-M} & \text{if } k \geq M \end{cases} && (5) \end{aligned}$$

with $M = N/2$. Because O_k (and E_k) can be computed as X_k with the odd (respectively even) part of the input values this formula can be applied recursively until only two numbers are left. Formulas with other radix then 2 also exists. This formula can be easily extended from this one dimensional case to the n-dimensional case by applying the 1D formula in turn onto each dimension of the input data. A special case is real data which is very often the input including for PME. The result is complex but because half of the result is identical to the other half only half as much data has to be computed and stored as for complex input data. Complex data with this symmetry can in turn be back transformed to real data again in half as much CPU time.

"Fastest Fourier Transform in the West" (FFTW) [6] is a very fast implementation of one and multidimensional FFT, of arbitrary size, with real

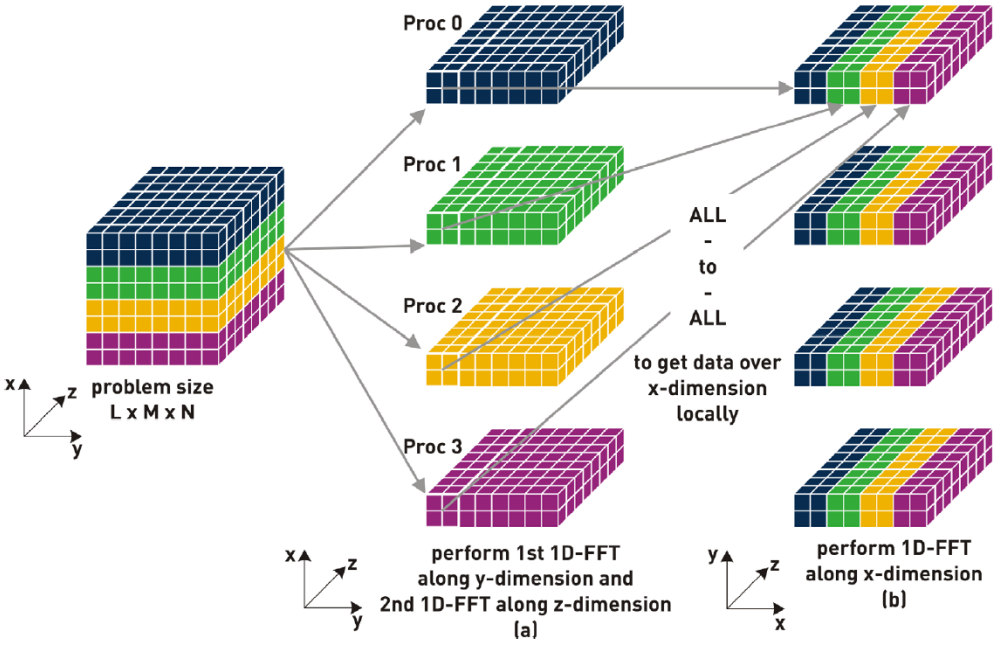

Figure 3: 1D decomposition of 3D FFT [8]

and complex input, both in single and double precision. Since version 3 is is significant faster by utilizing SIMD assembler commands (SSE/Altivec). It also supports Cilk [7] and threads. The 3.2alpha 3 version also supports MPI for parallelization.

2 FFT Decomposition

For parallelization on distributed memory machines most libraries and programs use MPI with a 1D decomposition of the data which is also called slab decomposition. This is also used both by GROMACS and FFTW. The data layout and communication pattern for 3D input data is shown in Fig 3. One can see that the parallel computation of the 3D FFT has three steps using this decomposition:

1. 2D FFT (or two 1D FFT) along the two local dimension
2. global transpose
3. 1D FFT along third dimension

The global transpose is required for the nodes to have the third dimension locally available. This decomposition is the the fasted on limited number of processors because it only needs one global transpose. The disadvantage is that the maximum parallelization is limited to the length of the largest axis of the 3D data. It can be increased by the number of cores per node using a hybrid method combining this decomposition with a thread based parallelization of the individual 1D FFTs. Though this slight increase in parallelization does not change that for (near) cubic data with N data points

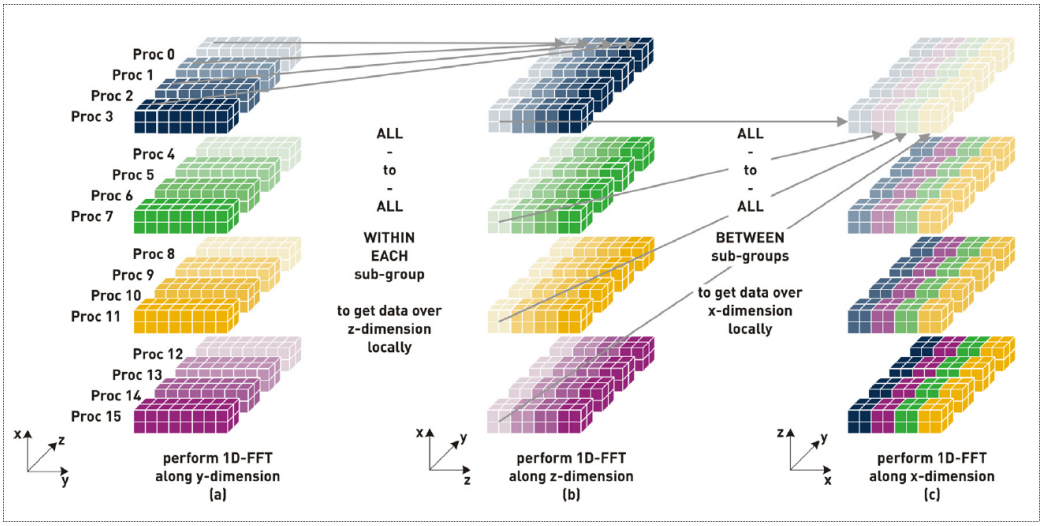

Figure 4: 2D decomposition of 3D FFT [8]. The coordinate system is rotated in each step for better clarity of the communication and datalayout.

(which is proportional to the number of atoms) the maximum number of CPUs scale as $O(N^{1/3})$ and the work scales as $O(N \log N)$ thus resulting in a very bad weak scaling.

This scaling limitation can be overcome by using a 2D decomposition [8, 9] as shown in Fig. 2. The computation is done in five steps

1. 1D FFT the local dimension
2. global transpose
3. 1D FFT.along second dimension
4. global transpose
5. 1D FFT.along third dimension.

These steps assume that the data is local along one axis when calling the 3D FFT. Otherwise another global communication is required as seen in Fig. 5. The same is true for the output data in case the output is needed in a different data layout. As seen in the figure the global transpose requires communication only between subgroups of all nodes. This can be described as within subgroups and between subgroups as in the figure. Alternatively the same is to assume a 2D processor grid. Then the first communication is along one axes of that processor grid and the next is along the other axes.

The decomposition has still a inherent scaling limitation because the maximum number of CPUs only scales as $O(N^{2/3})$ but this limitation is of no practical relevance because for any practical size the network communication time limits the number of nodes significantly more. This is also related to the disadvantage of the 2D decomposition. Because it requires two instead of one global transport it is in general slower than 1D decomposition for a number of processors possible with 1D decomposition [8, 11].

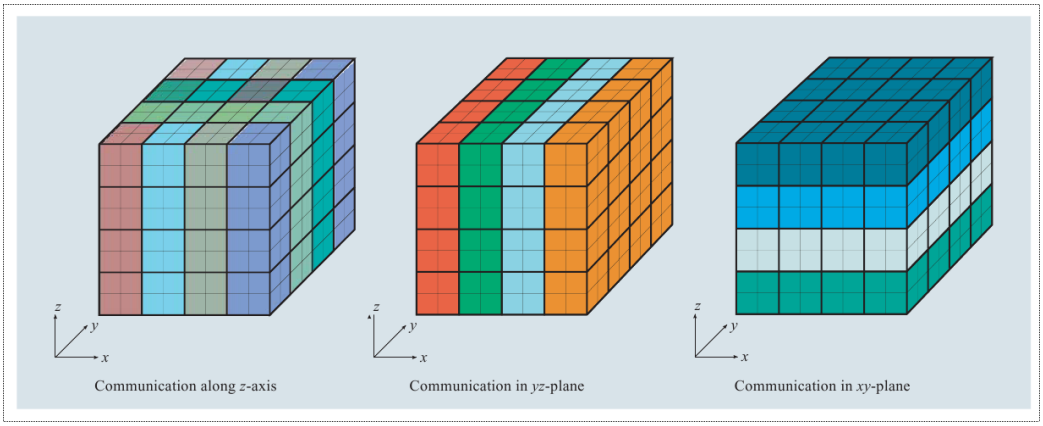

Figure 5: 2D decomposition of 3D FFT without rotated coordinate system and assuming a block distribution of the input data [10]

To my knowledge three parallel FFT libraries using 2D decomposition exist, which are non platform specific (additional there is one specific for Bluegene and implementations part of application code). These are "Parallel FFT Package" by Steve Plimpton [12], FFTE by Daisuke Takahashi [13] and P3DFFT by Dmitry Pekurovsky [14]. Steve's FFT uses for the global transpose `MPI_Send` and `MPI_Irecv`, FFTE uses `MPI_AlltoAll` and P3DFFT uses `MPI_Alltoallv`. All support transform in single and double precision. Only P3DFFT supports transforms from or to real data and input data which cannot be evenly distributed to the number of processors.

3 Implementation

To learn the usage of FFTW I first implemented a 3D FFT local without MPI. As described earlier a 3D FFT is computing by applying the 1D FFT in all three direction in turn. Because FFTW provides a stride parameter for the input and output data vector a local transpose is not necessary. Instead for each of the three 1D FFT calls the stride parameter is specified to get the data values along the correct direction. The usage of FFTW is more complicated than some other libraries because of the usage of so called plans. Thus using FFTW involves three steps: creating the plan prior to the computation, executing the plan and destroying the plan. The advantage of this is that FFTW can measure which algorithm performance best and then reuse this best algorithm for each computation assuming that the program has to do the FFT transformation inside a loop. For this case one should call the planning step with `FFTW_MEASURE` giving a fast computation, for the other case when the computation is done only once `FFTW_ESTIMATE` exists which gives a fast planning fast but less ideal computation time. The correctness of the result was compared both to the 3D FFT provided by FFTW and to Matlab with the criteria that the maximum difference between the results is less than $N^3\epsilon$ with N the box length of the cubic input data and ϵ the smallest number for which the numeric result is $1 + \epsilon > 1$ for the chosen floating point precision.

Since Matlab also internally uses FFTW this does not test the correctness of FFTW but only the correctness of the usage of the library. Since FFTW is very commonly used and has its built in tests, its correctness was assumed.

As a next step the 2D decomposition was implemented with MPI. The global transpose is done with `MPI_Alltoall` and the 1D FFT by calling FFTW. I chose `MPI_Alltoall` for the global transpose because performance measurements on P3DFFT prior to the project showed that `MPI_Alltoallv` is significant slower than `MPI_Alltoall` and implementing an own fast global transpose based on `MPI_Send` and `MPI_Recv` is beyond the scope of the project. I chose to use `MPI_Alltoall` without derived datatypes because I knew that they are not very efficient on the Cray MPI based on MPICH.

`MPI_Alltoall` without derived datatypes requires the data for each processor to be consecutive in memory. Additional the matrix transpose has do be done locally after the submatrix are globally transposed by `MPI_Alltoall`. This local transpose could be avoided by using stride as was done in the local implementation but because the memory has to be rearranged anyhow it is faster to also do the local transpose so that FFTW can operate cache friendly. This also has the advantage to be able to measure this rearrangement time independent of the FFT time.

It is non trivial to implement this local memory rearrangement correctly because of the number of involved indices and the rotating coordinate system. Thus I implemented it in steps to reduce the initial complexity. The final implementation requires only that the input data is cubic and that the data can be evenly distributed on the processor grid. It allows any size of cube as complex input both in single and double precision. The notation to describe the implemention steps is: N box length of data, P total number of processor, Px and Py are the number of processors along the sides of 2D processor grid, $Nx = N/Px$ and $Ny = N/Py$. The steps are then:

1. each processor has only one input number, squared processor grid
   $Nx = Ny = 1$

2. any size of data, squared processor grid
   $Px = Py$, $Nx = Ny$, $Nx \bmod Px = 0$

3. any size of data, any processor grid able to evenly divide data
   $Nx \bmod Px = 0$, $Ny \bmod Py = 0$

The first implementation did not require any local transpose and the second implementation had much simpler local transpose. The limitation of the final implementation to allow only evenly dividable data is caused by the choice of `MPI_Alltoall` which does not allow to communicate different amounts of data per destination respectively source. The same reason applies to the limitation to not allow transforms from or to real data but only complex to complex. The amount of data for real is $(N + 1)/2$. This means for most number of processor and input cube sizes the data will not be dividable evenly both for real and complex at the same time and thus the amount of data which would have to be communicated would vary. I started to implement a version which also allows non cubic input data, but it complicate

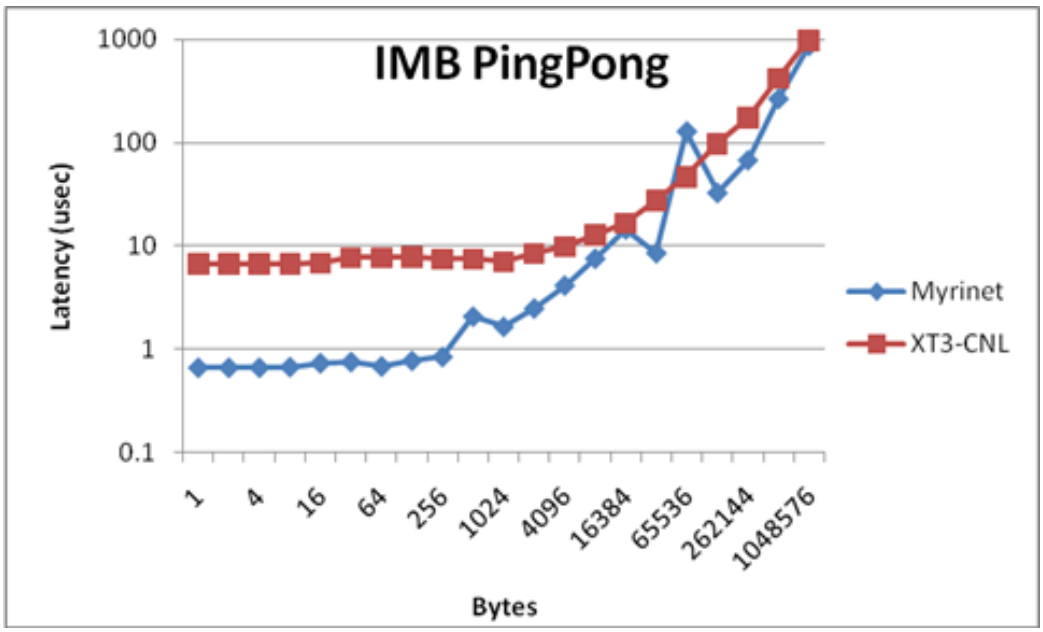

Figure 6: Point to point latency for Seastar and Myrinet. Measurement done partly by me and partly by Sadaf Alam independent of this project.

the local transform without a benefit to the goal of a test implementation for performance tests. The correctness of each implementation is tested against the 3D FFT built into FFTW using the same crateria as for the local 3D FFT.

For performance the local transpose loops are reordered for cache efficiency and the FFTW measurement option is used. Because `MPI_Alltoall` cannot be used in-place the local matrix has to be stored twice which could be a problem for larger systems but not for current MD applications. Because the matrix is stored twice anyhow FFTW is also not used in-place even though it is possible because the out-of-place implementation is usually faster according to their benchmark [15]. For the usage of FFTW it is important that the length of the 1D FFT is not a multiple of a large prime. But because a single 1D FFT is not parallelized this cannot be affected by the parallelization and thus has to be considered by the user.

4 FFT Benchmark

The benchmark were carried on Helics2 [16] a Dual-Core Opteron 2.8GHz with 10G Myrinet interconnect and Jaguar [17] a Dual-Core Opteron 2.6GHz with Cray Seastar interconnect. While Seastar has benefits regarding very large cluster including router error correction the point to point latency as measured with Intel MPI Benchmark (IMB) [18] is higher than for 10G Myrinet as seen in Fig. 6.

The measurement is done by averaging the time of 100 FFT and also averaging over all used processors. The time is taken for the total local transpose, the total time and the two MPI calls individual. The time is

measure by `MPI_Wtime`. From measurement done prior to this project I know that 2D decomposition can be faster than 1D decomposition starting from around $N = 128$. Benchmarking my FFT implemented showed directly that most of the time is spent in the two `MPI_Alltoall` calls:

CPU Grid	Local transpose	FFT	1st MPI	2nd MPI
2x64	0.30+/-0.01	0.66+/-0.01	122.93+/-87.64	3.24+/-26.72
4x32	0.25+/-0.01	0.65+/-0.01	59.01+/-42.27	4.04+/-25.74
8x16	0.25+/-0.01	0.65+/-0.01	92.38+/-26.07	8.94+/-33.20
16x8	0.26+/-0.01	0.65+/-0.01	84.88+/-33.86	19.64+/-58.73

This data is for Helics2 but the conclusion is true also for Jaguar. The large standard deviation originates from load implance. This can be seen by adding a `MPI_Barrier` before the first `MPI_Wtime` and after the second `MPI_Wtime`. This way timing of a task cannot be affected by the load imbalance from the task before. Of course one cannot add up the timings of the individual task in this case because the effect of the load imbalance is not measured accurately anymore. The effect can be seen in the next measurement:

CPU Grid	Local transpose	FFT	1st MPI	2nd MPI
16x8	0.25+/-0.00	0.65+/-0.01	4.18+/-13.03	75.93+/-60.12

The percentage of the `MPI_Alltoall` increases with larger number of processors because `MPI_Alltoall` decreases much slower than linearly. Because most time is spent in the global transpose I focused by benchmarking effort on this.

5 Global Transpose Benchmark

For the global transpose benchmark I used IMB both on Helics2 and Jaguar. I chose not to use my own code, so I wouldn't have the overhead of the FFT and can clearly see the FFT by itself. Also IMB provides a nice matrix for the timing in dependence both of message size and the number of processors, which is convenient not to have to write by oneself. For the MPI implementation I used both the default implementation which is in both cases a vendor modified version of MPICH and the OpenMPI implementation with backend for the highperformance interconnect. For Jaguar the version use were xt-mpt/2.0.33 and openmpi/1.3a1r17992. On Helics2 I used MPICH-MX 1.2.7..5 and OpenMPI 1.2.5.

By searching on the Internet for advice on global transpose I found the "FFTW MPI Transposes" [19] in FFTW 3.2alpha3. It is more general than the standard `MPI_Alltoall` but can be used as a compatible replacement for `MPI_Alltoall`. It is not as general as `MPI_Alltoallv` but does support different message sizes per processor. It has three builtin implementation for the communication. One is using direct point-to-point communication with precomputed ordering to reduce the number of collisions. The second is using a recursive "radix-r" approach to reduce the number of messages to $O(p \log p)$ for p processors. The third implementation calls directly `MPI_Alltoall` or `MPI_Alltoallv` depending on whether the data is evenly dividable.

I had not seen this transpose in the FFTW documentation before because I mainly used the released version and did not look at the details of their

MPI implementation because it only uses slab decomposition. I added an implementation to IMB calling the FFTW MPI Transpose. As for the FFT, FFTW uses a planning phase to measure the fastest possible communication pattern out of the three mentioned implementations. The timing is without the planning phase assuming that it neglectable for long runs because it only has to be executed once.

I used the normal scheduler and did not request a special reservation for the benchmark. Thus the processors were in most cases not contiguous assigned. Heike Jagode has shown that on BlueGene the precise placing on the 3D torus is very important [8]. Because Jaguar also uses a 3D torus is could potentially also be very important, even though according to John Levesque in a personal communication this effect should be small on Jaguar. Since I did not want to request a special reservation for this project I could not test it. Helics2 has a full-bisection diameter-3 Clos network and thus I do not expect large effect on the placing. Again I did not request a special reservation to test assumption.

All timing results are in μs. Along the y-direction the message size in bytes is given and in the x direction the number of processors is shown. For AlltoAll by the standard MPI library on Jaguar I measured:

AlltoAll	2	4	8	16	32	64	128
0	0.07	0.07	0.07	0.07	0.07	0.07	0.07
1	14.32	47.41	59.72	81.9	107.84	170.44	221.84
2	14.2	47.41	59.69	82.17	131.79	170.71	222.02
4	14.11	47.53	59.85	100.56	131.68	175.37	226.63
8	14.12	47.89	73.12	100.95	133.17	177.03	231.23
16	14.29	47.68	73.3	102.36	134.7	177	247.94
32	16.36	59.77	74.29	102.96	137.89	184.96	274.81
64	16.36	59.52	75.26	106.93	144.09	215.79	355.28
128	16.15	59.11	77.8	109.8	167.51	267.72	535.16
256	16.31	59.44	80.63	134.23	197.14	385.22	905.91
512	16.83	61.26	97.46	153.29	282.1	660.94	1667.26
1024	17.15	62.89	155.02	358.16	769.05	1619.8	3313.07
2048	21.07	72.11	183.02	416.35	898.14	1892.51	3948.11
4096	23.42	83.73	210.89	480.43	1039.05	2279.3	5210.17
8192	27.79	100.56	274.76	650.2	1500.21	3543.02	9191.78
16384	36.8	137.25	379.46	1108.1	2639.44	6775.03	17428.15
32768	57.31	218.34	738.8	2106.87	5243.45	12965.68	32434.54
65536	97.52	454.29	1524.2	4356.3	10311.28	24638.28	61710.51

The most of the other results I will present as relative results to make the comparison simpler. The number are always the fraction of the larger time divided by the smaller time. Which one is slower is represented by the color. Red means that the method which is compared is slower than the reference

and black means the reference is slower than the method which is compared. I highlight very large ratio by a large font size.

AllToAllV	2	4	8	16	32	64	128
0	**5.4**	**6.6**	**9.6**	**15.7**	**25.9**	**46.3**	**22832.0**
1	1.1	1.1	**1.6**	**2.6**	**4.2**	**5.7**	**8.9**
2	1.1	1.1	**1.6**	**2.6**	**3.4**	**5.7**	**9.0**
4	1.1	1.1	**1.6**	**2.1**	**3.5**	**5.5**	**8.9**
8	1.1	1.1	1.3	**2.1**	**3.4**	**5.5**	**8.7**
16	1.1	1.1	1.3	**2.1**	**3.4**	**5.5**	**8.3**
32	1.1	1.2	**1.7**	**2.6**	**4.2**	**6.5**	**9.4**
64	1.1	1.1	**1.7**	**2.5**	**4.0**	**5.6**	**7.2**
128	1.1	1.1	**1.6**	**2.5**	**3.4**	**4.7**	**5.5**
256	1.1	1.1	**1.6**	**2.1**	**2.9**	**3.6**	**7.3**
512	1.1	1.1	1.3	**1.8**	**2.1**	**3.2**	**10.7**
1024	1.1	1.1	1.1	1.2	1.2	**2.2**	**2.0**
2048	1.1	1.1	1.1	1.2	1.1	**2.4**	**2.7**
4096	1.1	1.1	1.1	1.1	**1.9**	**2.4**	**7.0**
8192	1.1	1.1	1.1	1.2	**2.3**	**1.9**	**3.1**
16384	1.0	1.3	1.2	**1.8**	**1.8**	1.4	1.3
32768	1.0	1.3	1.2	1.4	1.5	1.3	1.2
65536	1.0	1.0	1.1	1.2	1.3	1.2	1.2

Measuring AllToAllV with the standard MPI on Jaguar, one can see that AllToAllV is significant slower than AllToAll for many system sizes and number of processors. The factor is often more than 3 and goes to close to 10. It is only above 10 for the unimportant case of zero message length.

Comparing the OpenMPI implementation of AllToAll and AllToAllV to the standard MPI based on MPICH one can see for both AllToAll

AlltoAll	2	4	8	16	32	64	128
0	1.2	1.4	1.2	1.2	1.2	1.2	1.4
1	**8.3**	1.1	**2.5**	1.5	**1.6**	1.3	1.3
2	**8.7**	1.1	**2.5**	1.5	1.3	1.3	1.4
4	**8.2**	1.1	**2.4**	1.2	1.3	1.3	1.3
8	**8.6**	1.1	**2.0**	1.2	1.2	1.3	1.3
16	**8.3**	1.1	**2.0**	1.2	1.3	1.3	1.3
32	**9.3**	1.1	**2.0**	1.2	1.2	1.3	1.2
64	**9.1**	1.1	**1.9**	1.2	1.3	1.2	1.1
128	**9.0**	1.1	**1.9**	1.2	1.2	1.2	1.0
256	**8.6**	1.1	**1.8**	**2.6**	**4.1**	**4.9**	**7.0**
512	**8.3**	1.1	**1.6**	**2.5**	**3.0**	**3.9**	**4.3**
1024	**7.0**	1.1	1.0	1.1	1.3	**2.2**	**2.4**
2048	**6.7**	1.2	1.1	1.1	**1.6**	**2.5**	**2.4**
4096	**3.0**	1.0	1.1	1.3	**1.5**	**1.7**	**1.8**
8192	**2.5**	1.0	1.1	1.3	**1.5**	**1.7**	**1.5**
16384	**1.8**	1.1	1.3	1.3	**1.6**	1.4	1.3
32768	1.5	**1.6**	**1.5**	1.4	1.3	1.2	1.0
65536	1.3	1.3	1.2	1.1	1.2	1.1	1.0

and for AllToAllV

AllToAllV	2	4	8	16	32	64	128
0	**3.8**	**4.2**	**4.8**	**6.1**	**6.7**	**7.5**	**2663.7**
1	**9.4**	1.2	**1.5**	**1.7**	**1.7**	**1.7**	**2.0**
2	**9.5**	1.2	**1.5**	**1.7**	**1.7**	**1.7**	**2.0**
4	**9.5**	1.2	**1.5**	**1.6**	**1.7**	**1.7**	**1.9**
8	**9.3**	1.2	1.5	**1.6**	**1.7**	**1.7**	**2.1**
16	**9.4**	1.2	1.5	**1.6**	**1.7**	**1.7**	**2.1**
32	**10.4**	1.0	1.2	1.3	1.3	1.4	**1.6**
64	**10.1**	1.0	1.2	1.3	1.3	1.4	**1.8**
128	**9.9**	1.0	1.2	1.3	1.3	1.4	**1.8**
256	**9.7**	1.0	1.2	1.3	1.4	**1.5**	1.0
512	**8.7**	1.0	1.2	1.3	1.4	1.5	**2.5**
1024	**7.5**	1.0	1.2	1.3	**2.1**	1.2	1.2
2048	**7.0**	1.1	1.0	1.3	**2.0**	1.1	1.2
4096	**3.3**	1.0	1.1	**1.6**	1.5	1.1	**3.4**
8192	**2.7**	1.0	1.0	**1.6**	1.2	1.1	**2.0**
16384	**2.0**	1.2	1.1	1.2	1.3	1.1	1.0
32768	**1.5**	**1.8**	1.5	**1.7**	1.4	**1.7**	
65536	1.3	1.1	1.3	1.4	1.4	**1.7**	

that OpenMPI is for almost all practical cases slower than the MPICH

based MPI. The only expectation for practical cases and significant difference is for some message sizes for 128 processors.

Very interesting is that the FFTW is for some message sizes and number of processors faster than the standard MPI AllToAll implementation:

AlltoAll – FFTW	2	4	8	16	32	64
4	1.0	**1.5**	1.0	1.1	1.1	1.0
8	1.0	1.0	1.1	1.2	1.1	1.0
16	1.0	1.2	1.2	1.2	1.1	1.0
32	1.0	**1.7**	1.0	1.1	1.1	1.1
64	1.0	**1.7**	1.0	1.1	1.0	1.2
128	1.0	**1.6**	1.0	1.1	1.0	1.2
256	1.0	**1.6**	1.0	1.1	1.0	1.2
512	1.0	**1.5**	1.1	1.1	1.0	1.1
1024	1.0	1.3	**1.4**	**1.8**	**1.6**	**1.4**
2048	1.0	1.2	**1.3**	**1.3**	1.3	1.0
4096	1.0	1.0	1.1	1.0	1.1	**1.4**
8192	1.0	1.0	1.0	1.1	1.1	1.3
16384	1.0	1.1	1.1	1.0	1.0	1.2
32768	1.0	1.0	1.0	1.0	1.0	1.1
65536	1.0	1.1	1.0	1.0	1.0	1.1

The main advantage can be seen for 4 processors or 1kb or 2kb message size. As mentioned earlier FFTW uses `MPI_Alltoall` or `MPI_Alltoallv` automatically in case they are faster than its own implementation. To compare FFTW to `MPI_Alltoallv` for the case of non evenly dividable data I deactivated the usage of `MPI_Alltoall`. The two other implementations in FFTW do not have any special treatmeant of the evenly dividable case. This measurement gave the same results as for the non modified FFTW. Also FFTW allows to do the global transpose both in the linear data layout as used by `MPI_Alltoall` and in a 2D matrix layout with each processor having a submatrix of the global 2D matrix. This data layout enables to do the global transpose without prior rearrangement for the 2D case and also in special cases for the 3D case. Also this data layout is required for the unevenly case. Comparing the linear and the matrix data layout showed no difference in the required time. Thus it makes sense to compare `MPI_Alltoallv` which is required using MPI for unevenly distributed data to FFTW transpose because it supports this case to though with a different data layout:

FFTW – AlltoAllV	2	4	8	16	32	64	128
4	1.1	***1.3***	***1.7***	***2.5***	***3.8***	***6.6***	***10.9***
8	1.1	***1.3***	***1.5***	***2.5***	***3.7***	***6.5***	***10.2***
16	1.1	1.1	***1.4***	***2.3***	***3.5***	***5.8***	***9.8***
32	1.1	1.3	***1.7***	***3.0***	***4.6***	***7.0***	***9.7***
64	1.1	***1.3***	***1.7***	***2.8***	***3.9***	***5.7***	***7.5***
128	1.1	1.3	***1.6***	***2.6***	***3.5***	***4.9***	***5.8***
256	1.1	1.2	***1.6***	***2.2***	***2.9***	***3.8***	***7.5***
512	1.1	1.2	***1.4***	***1.9***	***2.1***	***3.5***	***14.1***
1024	1.1	1.1	1.2	***1.5***	***1.5***	***3.6***	***2.9***
2048	1.1	1.1	1.2	1.1	***1.3***	***2.7***	***2.6***
4096	1.1	1.0	1.0	1.0	***1.7***	***2.2***	***6.2***
8192	1.1	1.0	1.1	1.2	1.0	***1.6***	***2.9***
16384	1.0	1.1	1.1	1.1	***1.4***	1.2	1.2
32768	1.0	***1.5***	1.1	1.0	1.1	1.0	1.0
65536	1.0	1.0	1.1	1.0	1.0	1.0	1.0

This speedup is very impressive as it is often over 4x faster than the standard MPI implementation.

For Helics2 the absolute time in μs with the standard MPI is

AlltoAll	2	4	8	16	32	64
4	2.41	5.46	25.32	42.27	73.11	115.98
8	2.47	5.6	25.55	46.4	74.62	140.53
16	2.5	5.79	28.05	47.76	94.43	142.85
32	2.53	5.84	29.03	61.47	96.98	150.06
64	2.46	5.64	37.2	62.44	98.78	190.29
128	2.55	6.04	38.19	64.84	133.27	281.88
256	2.67	6.45	40.4	90.84	206.01	317.86
512	3.4	7.95	60.96	186.19	436.3	951.54
1024	3.55	10.04	61.57	186.74	438.36	944.12
2048	4.53	14.29	99.99	303.17	709.82	1773.94
4096	6.61	23.4	178.6	539.98	1305.49	2969.52
8192	11.17	50.77	192.05	613.35	1735.33	3538.46
16384	20.04	79.65	369.57	1301.51	3425.31	7182.7
32768	23.09	252.19	725.82	2602.21	7169.08	15406.9
65536	40.38	138.29	1406.8	4046.22	9907.08	24991.65

Relative to this the time for `MPI_Alltoallv` on Helics2 with standard MPI is slower mainly for more than 16 processor and message size smaller than 256 byte. The difference is not as big as shown before for Jaguar:

AllToAllV	2	4	8	16	32	64
4	1.0	1.0	1.2	**2.2**	**3.1**	**4.1**
8	1.1	1.0	1.2	**2.0**	**3.0**	**3.4**
16	1.1	1.0	1.1	**2.0**	**2.4**	**3.4**
32	1.1	1.0	1.1	**1.6**	**2.3**	**3.2**
64	1.1	1.0	1.0	**1.8**	**2.7**	**3.1**
128	1.1	1.0	1.0	**1.8**	**2.1**	**2.1**
256	1.1	1.0	**1.5**	**2.1**	**2.1**	**3.0**
512	1.1	1.2	1.0	1.0	1.0	1.0
1024	1.0	1.0	1.0	1.0	1.0	1.0
2048	1.0	1.0	1.0	1.0	1.0	1.2
4096	1.0	1.1	1.0	1.0	1.0	1.0
8192	1.1	1.1	1.0	1.0	1.0	1.1
16384	1.2	1.1	1.0	1.0	1.0	1.2
32768	1.2	1.0	1.0	1.1	1.0	1.1
65536	1.3	**4.1**	1.1	**1.5**	**1.7**	**1.6**

As for Jaguar, though not as much, OpenMPI is slower than the standard MPICH based MPI:

OpenMPI AllToAll	2	4	8	16	32	64
4	1.3	1.0	1.2	1.2	1.4	1.3
8	1.4	1.2	1.2	1.3	1.1	**1.5**
16	1.4	1.2	1.1	1.1	1.3	**1.5**
32	1.4	1.2	1.3	1.2	1.3	1.1
64	1.4	1.2	1.0	1.2	1.1	1.0
128	1.4	1.2	**1.6**	1.2	1.1	1.0
256	1.3	1.2	1.5	**2.0**	**2.1**	**2.9**
512	**1.5**	1.2	1.0	1.0	1.0	1.0
1024	1.3	1.3	**1.6**	**1.6**	**1.6**	**1.6**
2048	1.3	1.2	1.4	1.4	1.4	1.2
4096	1.2	1.3	1.3	1.3	1.2	1.2
8192	1.1	1.1	1.3	1.2	1.0	1.1
16384	1.0	1.1	1.3	1.1	1.0	1.1
32768	**1.7**	1.4	1.4	1.1	1.1	1.1
65536	**1.8**	**2.3**	1.5	1.3	1.2	1.0

I tested also on both Helics2 and Jaguar the FFTW transpose compiled with OpenMPI. Since this is also slower one can conclude that also the point-to-point communication with OpenMPI is slower although I did not benchmark this directly.

The comparison of the standard `MPI_Alltoall` and `MPI_Alltoallv` to the FFTW transpose show as for Jaguar a clear improvement by FFTW:

FFTW	2	4	8	16	32	64
4	***1.7***	***1.7***	***2.1***	***2.0***	***2.1***	***1.8***
8	***1.7***	***1.7***	***2.2***	***1.8***	***2.1***	***2.0***
16	***1.7***	***1.7***	***1.7***	***1.7***	***1.8***	***1.9***
32	***1.7***	***1.8***	***2.2***	***1.7***	***1.7***	***1.9***
64	***1.7***	1.5	***1.8***	***1.6***	***1.6***	***1.6***
128	***1.7***	1.5	***1.8***	***1.6***	***1.5***	1.5
256	***1.6***	1.2	***1.8***	1.4	1.5	1.2
512	1.5	1.0	***1.7***	***1.9***	***2.2***	***2.2***
1024	1.4	1.1	1.0	1.4	1.5	1.4
2048	1.3	1.0	1.1	***1.5***	1.4	1.5
4096	1.2	1.0	1.2	1.4	1.4	1.3
8192	1.1	1.2	1.0	1.0	1.0	1.0
16384	1.0	1.0	1.0	1.0	1.0	1.0
32768	1.2	***3.5***	1.0	1.1	1.1	1.1
65536	1.3	1.1	1.1	1.4	1.0	1.0

FFTW – AllToAllV	2	4	8	16	32	64
4	***1.8***	***1.6***	***2.9***	***4.5***	***6.7***	***9.6***
8	***1.7***	***1.5***	***2.9***	***3.9***	***6.2***	***6.9***
16	***1.6***	***1.6***	***2.4***	***3.8***	***4.3***	***6.3***
32	***1.6***	***1.6***	***2.6***	***2.7***	***4.0***	***6.0***
64	***1.6***	1.3	***1.9***	***3.0***	***4.5***	***4.8***
128	***1.6***	1.4	***1.9***	***2.8***	***3.0***	***2.9***
256	***1.5***	1.1	***2.7***	***3.2***	***3.1***	***3.7***
512	1.4	1.2	***1.7***	***1.9***	***2.3***	***2.3***
1024	1.4	1.0	1.0	1.4	1.4	***1.5***
2048	1.3	1.0	1.3	***1.5***	1.4	1.3
4096	1.2	1.1	1.3	1.4	1.4	1.4
8192	1.2	1.1	1.0	1.0	1.1	1.1
16384	1.2	1.1	1.0	1.0	1.1	1.2
32768	1.2	***3.4***	1.0	1.1	1.0	1.2
65536	1.0	***4.1***	1.2	1.1	1.3	1.4

Eventhough one usually cannot choose the hardware to improve the library performance, I think it is non the less interesting to compare Seastar to Myri 10G. Relative to Seastar Myri 10G is faster by:

4	**5.9**	**8.7**	**2.4**	**2.4**	**1.8**	**1.5**
8	**5.7**	**8.6**	**2.9**	**2.2**	**1.8**	1.3
16	**5.7**	**8.2**	**2.6**	**2.1**	1.4	1.2
32	**6.5**	**10.2**	**2.6**	**1.7**	1.4	1.2
64	**6.7**	**10.6**	**2.0**	**1.7**	1.5	1.1
128	**6.3**	**9.8**	**2.0**	**1.7**	1.3	1.1
256	**6.1**	**9.2**	**2.0**	1.5	1.0	1.2
512	**5.0**	**7.7**	**1.6**	1.2	**1.5**	1.4
1024	**4.8**	**6.3**	**2.5**	**1.9**	**1.8**	**1.7**
2048	**4.7**	**5.0**	**1.8**	1.4	1.3	1.1
4096	**3.5**	**3.6**	1.2	1.1	1.3	1.3
8192	**2.5**	**2.0**	1.4	1.1	1.2	1.0
16384	**1.8**	**1.7**	1.0	1.2	1.3	1.1
32768	**2.5**	1.2	1.0	1.2	1.4	1.2
65536	**2.4**	**3.3**	1.1	1.1	1.0	1.0

for `MPI_Alltoall` and by

FFTW helics-jagaur	2	4	8	16	32	64
4	**9.8**	**9.6**	**4.9**	**4.4**	**3.5**	**2.8**
8	**9.7**	**14.4**	**5.6**	**3.4**	**3.2**	**2.6**
16	**9.3**	**11.6**	**3.8**	**3.2**	**2.4**	**2.4**
32	**10.7**	**10.8**	**5.6**	**2.5**	**2.3**	**2.5**
64	**11.0**	**9.4**	**3.7**	**2.4**	**2.4**	**2.1**
128	**10.3**	**9.2**	**3.7**	**2.5**	**1.9**	**1.6**
256	**9.9**	**7.2**	**3.6**	**1.9**	**1.4**	**1.7**
512	**7.2**	**5.3**	**2.6**	**1.4**	**1.4**	**1.7**
1024	**6.3**	**5.3**	**1.9**	**1.5**	**1.6**	**1.7**
2048	**5.6**	**4.3**	**1.6**	**1.6**	**1.4**	**1.6**
4096	**4.1**	**3.6**	1.3	1.2	1.2	**1.4**
8192	**2.7**	**2.5**	**1.4**	1.2	1.1	1.3
16384	**1.9**	**1.9**	1.2	1.2	1.2	1.1
32768	**2.0**	**3.0**	1.0	**1.4**	1.3	1.0
65536	**1.9**	**3.3**	1.1	**1.3**	1.1	1.1

when one uses FFTW on both platforms. One sees that Myri 10G only is faster for small data size (either small message size or small number of processors). In these cases the lower latency is seen. For large data size the time is bandwidth limited and then both interconnects have similar speed.

Based on the results, that FFTW is on both platforms faster, I added support for FFTW transpose to my 3D FFT with linear data layout. The 2D processor grid used for the two global transposes can be chosen either

(almost) square or very rectangular. Thus in the two global transpose the number of processors communicating to eachother is equal to the size of the grid in this direction. The message size is for both communication the same. From the transpose results one can see that the time is increasing faster than linearly with the number of processors thus it is better to use a very rectangular processor grid unless the application using the 3D FFT requires a square processor grid. Thus I did not add a planning phase, as used by FFTW, to my 3D FFT because no parameter has to be optimized.

6 Possible further steps

It is also possible to use the matrix data layout for 3D matrices because FFTW transpose has a parameter "howmany" to specify blocks of data which are kept consecutive during the transpose. Since this can only be applied to consecutive data it applies only to dimemsion 1 (number from slowest dimension) thus it can only be applied to a transpose of dimension 2 and 3. As can be seen in the diagrams showing the commincation pattern for the 2D decomposed FFT these two dimensions are not transposed but instead dimension 1 and 2 and 1 and 3 are transposed. Thus also the matrix data layout requires a local data reordering. I was not able to implement this after I got the results from the transpose benchmarking.

Having implemented this data reordering, the matrix data layout FFTW transpose will allow unevenly distributed data without a performance penalty. This allows for any size of input data and also real to complex transforms. The reduced amount of data for the real data should according to the transpose measurements give an improvement of 30%. The usage of real data and FFTW transpose should make the library faster than all other libraries and more general than all but P3DFFT.

7 Conclusion

I have implemented a correct 3D FFT which is not limited by the data decomposition as is FFTW and the FFT implemented in GROMACS. The limiting factor is the global transpose which was optimized by using the FFTW transpose. The measurements have shown that this FFTW transpose is faster both on Seastar and Myri 10G interconnect. This enables to extend this implementation also to not dividable input sizes and real to complex transforms.

References

[1] Brendan Borrell. Chemistry: Power play. *Nature News*, 2008.

[2] B. Hess, C. Kutzner, D. Vanderspoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.*, February 2008.

[3] Gromacs. http://www.gromacs.org, 2008.

[4] C. Sagui and T. A. Darden. Molecular dynamics simulations of biomolecules: long-range electrostatic effects. *Annu Rev Biophys Biomol Struct*, 28:155–179, 1999.

[5] http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm.

[6] http://www.fftw.org.

[7] http://supertech.csail.mit.edu/cilk.

[8] Heike Jagode. Fourier transforms for the bluegene/l communication network. Master's thesis, The University of Edinburgh, 2005.

[9] Maria Eleftheriou, José E. Moreira, Blake G. Fitch, and Robert S. Germain. A volumetric fft for bluegene/l. pages 194–203. 2003.

[10] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain. Scalable framework for 3d ffts on the blue gene/l supercomputer: Implementation and early performance measurements. *IBM Journal of Research and Development*, 49, 2005.

[11] Dmitry Pekurovsky. personal communication, 2008.

[12] Steve Plimpton. http://www.sandia.gov/~sjplimp/docs/fft/README.html.

[13] Daisuke Takahashi. http://www.ffte.jp/.

[14] Dmitry Pekurovsky. http://www.sdsc.edu/us/resources/p3dfft.php.

[15] http://www.fftw.org/speed/opteron-2.2GHz-64bit/.

[16] http://helics.iwr.uni-heidelberg.de.

[17] http://www.nccs.gov/computing-resources/jaguar.

[18] http://www3.intel.com/cd/software/products/asmo-na/eng/219848.htm.

[19] http://www.fftw.org/fftw-3.2alpha3-doc/FFTW-MPI-Transposes.html.