Improving the Performance of CP2K on the Cray XT

Iain Bethune, EPCC, The University of Edinburgh

ABSTRACT: CP2K is a freely available and increasingly popular Density Functional Theory code for the simulation of a wide range of systems. It is heavily used on many Cray XT systems, including 'HECToR' in the UK and 'Monte Rosa' in Switzerland. We describe performance optimisations made to the code in several key areas, including 3D Fourier Transforms, and present the implementation of a load balancing scheme for multi-grids. These result in performance gains of around 30% on 256 cores (for a generally representative benchmark) and up to 300% on 1024 cores (for non-homogeneous systems). Early results from the implementation of hybrid MPI/OpenMP parallelism in the code are also presented.

KEYWORDS: CP2K, Cray XT, MPI, OpenMP, FFT, Load Balancing

1 Introduction

1.1 CP2K

CP2K[1] is a freely available density functional theory (DFT) package, licensed under the GPL. The code is developed by a number of individuals and groups, and currently numbers over 600,000 lines of Fortran 95 code. Key developers include members of Prof. Jurg Hutter's group in the Physical Chemistry Institute of the University of Zurich, in particular Dr. VandeVondele who collaborated closely throughout the project.

The key difference between CP2K and other DFT codes is its implementation of the Quickstep algorithm, which uses a dual basis - atom-centred Gaussian functions to represent the wave-functions, and plane waves/regular grids for the electronic density. See the Quickstep paper [2] for full details of the algorithm. In practise this approach requires an efficient and scalable method for transforming between the two representations, which is the subject of most of the work undertaken here. Like most other DFT codes, CP2K also makes use of 3D Fast Fourier Transforms (FFTs) to convert from the real space to reciprocal space (plane wave) representations.

1.2 HECToR

HECToR[3] is a Cray XT5h system sited at the University of Edinburgh, consisting of 5664 XT4 compute nodes and 28 X2 vector compute nodes. In its 'Phase 1' configuration, used for generating the performance data reported below, each node contained a single AMD 2.8GHz dual-core Opteron processor with 6GB RAM. The system has since been upgraded ('Phase 2a') by replacing the processors with quad-core 2.3GHz quad-core Opterons with 8GB RAM per node. HECToR is expected to be upgraded to a Cray XT6 in 2010, with two 12-core processors per node, allowing up to 24 cores in a single shared memory region.

The HECToR service includes a Distributed Computational Science and Engineering (dCSE) support programme, where HPC experts work closely with users and code developers to enable them to make best use of the system. The work reported on here is the result of two 6 month dCSE projects, beginning in Aug 2008, in collaboration with Dr. Slater (HECToR user, University College London) and Dr. VandeVondele.

1.3 Rosa

Monte Rosa[4] is a Cray XT5 system at CSCS in Switzerland, consisting of 1844 XT5 compute nodes, each with two 2.4GHz hexa-core Opteron processors and 8GB RAM per processor. We are grateful to Prof. Hutter for providing access to this system.

2 Halo Swaps

A key step in the Quickstep algorithm is the mapping of Gaussian basis functions, stored as coeffi-

cients in a distributed sparse matrix, onto real space multi-grids, prior to Fourier Transformation. This process is known as 'collocation' and the inverse (calculating coefficients from the grids and storing them in the matrix), 'integration'. The real space grids may be distributed or replicated across MPI tasks, depending on the size of the grids. In the distributed case, each MPI task has a cuboid subsection of the entire grid, and maps the corresponding Gaussians onto its section of the grid. Because the Gaussians have a finite extent, it is necessary for each process to have a 'halo' region surrounding it's 'core' section of the grid, in order that it can map Gaussians which extend outside it's core region. The data in the halos is then exchanged with neighbouring MPI tasks, such that every process ends up with the complete data for all Gaussians that intersect it's core region of the grid. Because the size of the halos is fixed by the radius of the largest Gaussian, but the core region shrinks as the grid is divided up over more and more processes, this halo swap can become a bottleneck to parallel scalability. For example, a typical 125^3 grid on 512 MPI tasks has a local domain size of 16^3, and a halo width of 18. As a result, the halos make up 97% of the total grid data, and communication becomes costly.

Careful analysis of the communication pattern revealed that some of the data that was being sent ended up in the halo regions of neighbouring processes, and was therefore not needed for the subsequent FFT. In this case, it was possible to reduce the amount of data sent by discarding data that is known to end up in a halo region. When done in 3 dimensions this gives substantial savings, shown in table 1 below, in terms of both the amount of data to be sent, and the amount of time taken packing buffers. The 'rs2pw libtest' is an artificial benchmark which executes only the real space to plane wave transfer part of the code, which includes the halo swap described above.

2.1 Benchmarks

To test the results of these changes the 'bench_64' test case was used. Bench_64 is a molecular dynamics simulation of 64 water molecules in a cubic, periodic cell of side 12.42Å, described by a TZV2P basis set in an NVE ensemble at a temperature of 300K. The system is evolved for 50 timesteps of 0.5fs each. This simulation takes a few minutes to complete on 256 cores and is considered typical of the smaller systems that might be simulated using CP2K. The table below gives the runtimes on a range of processor counts on HECToR and the resulting speedup:

	Before	After
Avg. Message Size (bytes)	194688	91008
Time in SendRecv (s)	0.468	0.22
Time packing X bufs (s)	0.107	0.002
Time unpacking X bufs (s)	0.189	0.003
Time packing Y bufs (s)	0.060	0.005
Time unpacking Y bufs (s)	0.096	0.017
Time packing Z bufs (s)	0.054	0.054
Time unpacking Z bufs (s)	0.091	0.091

Table 1: 60 iterations of the rs2pw libtest on 512 cores, before and after optimisation

Cores	16	32	64	128	256	512
Before(s)	952	541	318	268	217	264
After(s)	938	519	296	247	190	235
Speedup(%)	2	4	7	9	14	12

Table 2: Comparison of bench_64 runtime before and after rs2pw optimisation

3 Fast Fourier Transforms

CP2K makes use of FFTs to transform from the Gaussian (real space) to plane wave basis. As efficient computation of a 1D FFT is closely tied to the architecture of the machine in question, CP2K provides an implementation-agnostic interface to perform 1D FFTs. A number of libraries can be optionally linked in to the code to implement this functionality, including FFTW2/3[5], ACML, MKL, ESSL and CUDA (for GPU support). A default FFT implementation FFTSG[6] is also provided for the case that no FFT library is available. In practice, FFTW 3 has been found to provide competitive performance with proprietary FFT libraries[7], and is widely available.

As the plane wave grids containing data for the FFT are distributed in CP2K, a parallel 3D FFT is required, and follows the conventional approach of transposing the data using MPI_Alltoallv between successive 1D FFTs. CP2K can make use of a 1D (plane, slice, slab) decomposition of the grids when the number of MPI tasks is less than the shortest grid dimension, or a 2D (ray, pencil) decomposition for larger numbers of tasks, with 1 or 2 transpose steps required respectively.

3.1 Caching FFT metadata

Detailing profiling of the FFT routines using the Cray Performance Analysis Toolkit (CrayPAT), revealed a number of MPI routines used for e.g. generating the MPI sub-communicators used for the transpose steps, were being called once per FFT, despite the fact that this mapping is constant throughout the execution of the program. The code was modified so these communicators and several other pieces of metadata (coordinates, grid bounds etc.) were calculated once, stored in a data structure, and reused at each iteration. This alone had the effect of of a 12% speedup on 512 cores for the bench_64 test (see table3).

Cores	64	128	256	512
Before(s)	366	264	191	238
After(s)	363	250	177	213
Speedup(%)	1	6	8	12

Table 3: Comparison of bench_64 runtime before and after FFT caching optimisation

3.2 FFTW Planning

FFTW achieves good performance on a range of architectures by combining a set of 'codelets', generated at compile-time, to perform a complete FFT of a given length. Precisely how the codelets are combined is determined by a planning step, before the execution of the FFT. The cheapest method of planning, `FFTW_ESTIMATE`, chooses a combination of codelets based on heuristic knowledge about the system. However, better plans (i.e. better runtime FFT performance) can be achieved by using more expensive forms of planning - `FFTW_MEASURE`, and for FFTW 3, `FFTW_PATIENT` and `FFTW_EXHAUSTIVE` - which run a number of FFTs using different combinations of codelets and pick the combination with the best performance. Clearly, doing so takes more time than generating an estimated plan, and so is appropriate to use if many FFTs of the same size are performed, allowing the plan to be generated once and re-used many times.

Due to CP2K's library-agnostic interface, there was no way to store and re-use plans, so for every FFT, an estimated plan was created, executed, and destroyed. The ability to cache library-dependant data (similarly to the metadata above) was added to the FFT interface, which allows the storage and reuse of FFTW plans. As well as dramatically reducing the number of calls to the planner, this also allows the more advanced forms of planning to be used, where the number of FFTs to be performed is large enough that the cost of planning will be outweighed by the increase in runtime performance. This choice is exposed to the user via an option in the CP2K input file, defaulting to `FFTW_ESTIMATE`, as before. Table 4 shows the speedup obtained by each planning method over 2000 FFTs. Typical Molecular Dynamics simulations might perform many more FFTs, and see a corresponding increase in performance.

	Time(s)	Speedup(%)
Original Code	997	
FFTW_ESTIMATE	995	0.2
FFTW_MEASURE	989	0.8
FFTW_PATIENT	975	2.3
FFTW_EXHAUSTIVE	1081	

Table 4: Time and speedup for 2000 3D FFTs using different plan types

3.3 FFT Transpose

The transpose step between 1D FFTs involves using MPI_Alltoallv to globally redistribute the data on the plane wave grids. This operation scales poorly with the number of processors, and can be a bottleneck for some systems on greater than 1000 cores, for example. MPI_Alltoallv is required as typically the grid dimensions do not divide evenly into the number of MPI tasks, so each process has a slightly different amount of data to send and recieve. However, the difference is usually small (some processes have one more or less row of the grid than the others), so with the addition of a small amount of padding to even out the distribution of data, it would be possible to use MPI_Alltoall instead. Micro-benchmarking using the Intel MPI Benchmarks [8] showed that MPI_Alltoall outperforms MPI_Alltoallv (up to 500% for small messages on 128 cores) for the same amount of data transferred (see figure 1).

The FFT routines were modified to insert and remove the padding during the existing buffer packing steps, and initial benchmarking of the 3D FFT in isolation was promising, showing a 43% speedup for a 125^3 grid on 256 cores. However, when a full benchmark (e.g. bench_64) was run, only a 2% improvement was seen. Further investigation showed

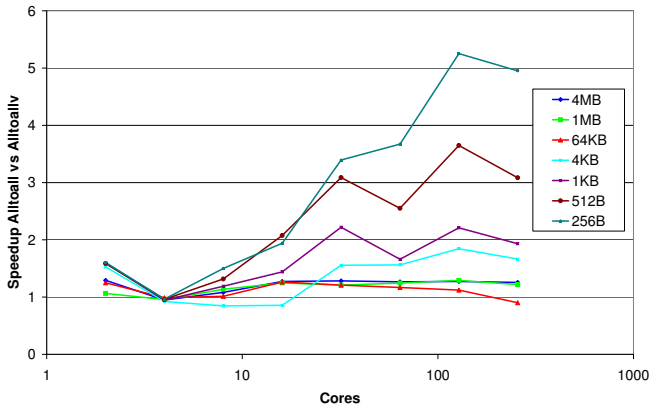

Figure 1: Graph of improvement of MPI_Alltoall over MPI_Alltoallv

that this was due to poor synchronisation - the performance improvement seen in the micro-benchmark is only realised if all processes are well synchronised. Nevertheless, this does raise the hopes that significant improvements could be made other codes, particularly when the message size is very small, and large numbers of cores are used.

4 Load Balancing

As mentioned in section 2, a significant part of the computation in CP2K is mapping Gaussian 'tasks' from sparse matrices onto the real space multigrids. This introduces the potential for load imbalance, as a single MPI task will map Gaussians that reside on its local section of the grids. As the Gaussians are spatially distributed depending on the location of atoms in the system being simulated, an inhomogeneous sytem such as a solid/gas interface or cluster (e.g. molecule in vacuo) will cause a disproportionate number of Gaussians to be allocated to a particular region of the grid.

A cost model exists within the code that can accurately estimate the computational cost of a given task, and this is used to calculate the cost of tasks on the distributed grid levels. The tasks from the replicated grid levels (which can be mapped by any MPI task) are then used to even out the imbalance. This scheme is successful for moderately imbalanced systems (e.g. bench_64 on 64 cores):

```
 At the end of the load_balance_distributed
 Maximum load:                  75667
 Average load:                  68312
 Minimum load:                  13060

 At the end of the load_balance_replicated
 Maximum load:                 123552
 Average load:                 123457
 Minimum load:                 123374
```

However, very inhomogenous systems (such as W216 - a cluster of 216 water molecules in a 34Åbox) do not have enough replicated tasks to balance out the imbalance in distributed tasks, and so exhibit significant load imbalance - in this case 6:1 between maximum and minimum loads:

```
 At the end of the load_balance_distributed
 Maximum load:                1738978
 Average load:                 176232
 Minimum load:                      0

 At the end of the load_balance_replicated
 Maximum load:                1738978
 Average load:                 475032
 Minimum load:                 286053
```

To address this, an indirection was added between 'real ranks' - the MPI rank of the process - and 'virtual ranks' - the ID of the corresponding section of the grid. This allows processes to 'own' different sections of the grid at each level, so heavily loaded segments of the grid on one level can be matched up with underloaded sections of the grid on another level. Using this scheme, the load of the most heavily loaded process in the W216 problem above was reduced by 30%. For this particular problem it is not possible to find a perfect load balance, as there are several sections of a single grid level which have more load that the total average load. It is possible to improve the load balance further by having more grid levels, and hence less load on each grid level. However, if it is possible to balance the load perfectly, then this algorithm will succeed, as shown in figures 2 and 3

5 Benchmarks

Following the above sections of work, comprising the first of the two dCSE projects funded, the bench_64 and W216 benchmarks were run to determine the overall performance improvements to the code. A maximum speedup of 31% on 256 cores for bench_64

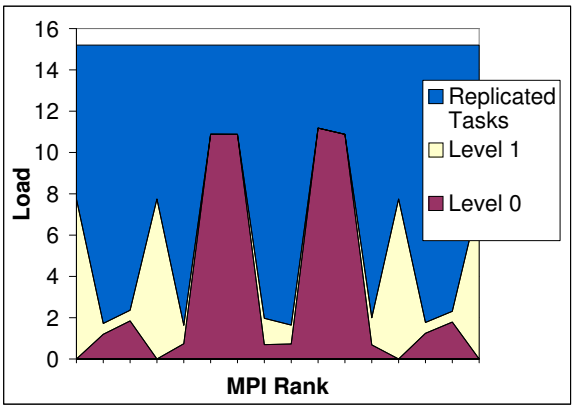

Figure 2: W216 load balance on 16 cores - perfect load balance achieved

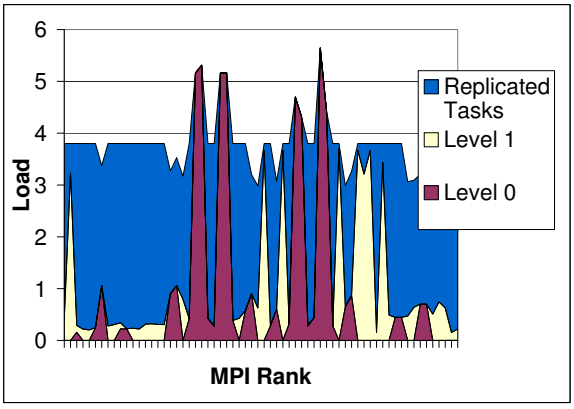

Figure 3: W216 load balance on 64 cores - several heavily loaded grid sections stop perfect local balance

and 341% on 1024 cores for W216 were found, as shown in tables 5 and 6.

It should be noted that in addition to the work performed within the dCSE project, other work was undertaken by the CP2K development group, which would affect the benchmarks of W216. In particular, the load balancing was modified to allow heavily loaded processes to shift some work to neighbouring processes, provided that their realspace grid halos still contain the entirety of a Gaussian to be mapped. This has the effect of reducing the highest peaks of load (see figure 3). However, the FFT and halo swap optimisation also has a significant effect, especially on larger numbers of cores. Due to the concurrent nature of development, and the fact that W216 is relatively expensive to run, these changes were not benchmarked at each step in development.

Cores	64	128	256	512
Before(s)	317	264	230	294
After(s)	316	237	176	224
Speedup(%)	1	11	31	31

Table 5: Comparison of bench_64 runtime before and after 1st dCSE project

Cores	128	256	512	1024
Before(s)	11555	7250	5640	4865
After(s)	4800	2859	2096	1425
Speedup(%)	241	254	268	341

Table 6: Comparison of W216 runtime before and after 1st dCSE project

6 Adding OpenMP to CP2K

Following the success of the work reported so far, a second dCSE project was funded to improve the scalability of CP2K using a mixed-mode OpenMP/MPI approach. This was motivated by a number of considerations, firstly that by using OpenMP threads rather than MPI tasks to utilise compute cores, the effect of algorithms that scale poorly with the number of MPI tasks would be reduced (e.g. FFT transpose, realspace to planewave transfer). Also, by reducing the number of MPI tasks, the more efficient 1D decomposition for the FFT grids can be maintained for higher core counts. Secondly, as the Cray XT architecture is going increasinly multi-core (XT4 - 2/4/6 cores per node, XT5 - 8/12 cores, XT6 - 24 cores), minimising access to the network which is shared by all cores on a node will become increasingly desirable to avoid a performance bottleneck. Thirdly, as of 2009, CP2K contains a highly scalable Hybrid Functional (HFX) [9] implementation which uses OpenMP to access the full memory of a node in a single MPI Task. This currently scales to 32,000 cores, and at this point, the non-OpenMP rest of the code stops the code scaling further. Fourthly, we hope to reduce load imbalance by allowing a high-level load balancing to take place per MPI task, as before, and manage to distribution of work over OpenMP threads by dynamic scheduling.

The work detailed below was performed on Rosa (section 1.3), which allows up to 12 OpenMP threads per MPI task, on a 12 core XT5 node. This is expected to provide a better indicator of the performance expected on XT6 systems than the current HECToR XT4.

6.1 Implementation Details

Three key routines were identified where using OpenMP might be of benefit - realspace to planewave transfer (halo buffer packing), FFTs, and Gaussian collocation/integration. These routines take up the majority of the runtime for most systems. In order to achieve good performance, it is important that as much of the code as possible makes use of OpenMP, otherwise cores will remain idle for large portions of the run time.

A simple approach was taken where OpenMP parallel regions would be created in each of the areas mentioned. MPI communications would be done in single-threaded regions, as the Cray Message Passing Toolkit does not provide a high performance thread-safe MPI implementation.

In the realspace to planewave transfer routines the most expensive operations are buffer packing, and these are now done in parallel using OpenMP threads. It is worth noting that although the OpenMP standard allows Fortran Array operations to be parallelised automatically by the `workshare` directive, this behaviour is not implemented in gfortran prior to version 4.5.0, released in April 2010, and so array operations have been parallelised manually by calculating array bounds for each thread to work on. Compared to the original MPI code, running the realspace to planewave transfer in isolation, the maximum performance is approximately doubled, and the most effective number of cores is increased from 288 to 4608 when using 6 threads per MPI task. Using 12 threads per task (the maximum available on Rosa) did not give any further benefit, and in most cases had a detrimental effect, due to the overhead of both processors sharing access to the same arrays.

For the FFT, as well as buffer packing for the transpose, the 1D FFTs were parallelised by dividing the plane or pencil to be FFTed up, and assigning a section to each thread. Care is taken to ensure each grid section is aligned correctly to allow FFTW to generate plans using SSE vector instructions for maxmium performance. In the case of an odd number of rows, there is also the need for a second FFT plan for threads which have a differing number of elements to transform. As expected, using mixed-mode OpenMP/MPI allows the FFT to scale much further (from 288 up to 2304 cores), and increased the peak performance by a factor of 3.

The collocation and integration routines provided the greatest challenge to effective use of OpenMP. In the case of collocation, each thread is assigned a number of tasks taken from the density matrix using a dynamic schedule. As each of these tasks is to be written to the same grid, and could potentially overlap with the region being updated by another thread, a seperate scratch copy of the grid is assigned to each thread, and these are then reduced (in parallel) into the final grid once all tasks have been mapped. However, this overhead means that this routine only effectively uses two threads before overheads begin to dominate. In contrast, in the integration routine the grids are read-only, and so can be shared directly by all threads. By grouping tasks into sections which only access a single block of the matrix, each thread can operate completely independently. This is reflected in the fact that this routine easily scales up to 12 threads with little or no overhead.

6.2 Benchmarks

The Bench_64 and W216 test cases were both used for benchmarking the mixed-mode code. Runs were made using pure MPI, and 2,6 and 12 OpenMP threads per MPI task. The same range of core counts were used for each variant (e.g. 144 MPI tasks, 72x2 threads, 24x6 or 12x12). The results of the W216 benchmark are plotted in figure 4. For clarity, the performance (inverse of time) is plotted, so ideal scaling is represented by a straight line through the origin. The graph clearly shows a large increase in scalability, as the peak performance increases from 576 cores (pure MPI) to 9216 cores (with 12 threads per task). The maximum performance also increases by a factor of 2.5, thus decreasing the minimum time to solution. For the 6 and 12 thread runs, there is a significant decrease in performance at lower processor counts, mostly due to routines remaining that do not currently make use of threads. However, using 2 threads per task gives similar or better performance to pure MPI across all processor counts tested, so could be recommended for general use.

7 Further Work

Analysis of the benchmark results has revealed a number of areas that still require attention. In particular, for low processor counts, the functional evaluation routines take a great deal of time and currently do not make use of OpenMP. This will at least partly address the poor performance of the multi-threaded code on low processor counts. The collo-

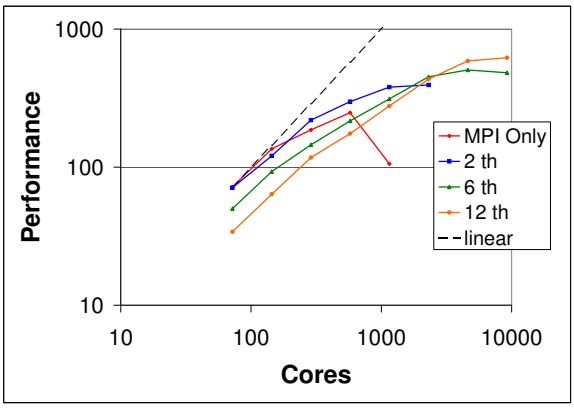

Figure 4: Comparison of W216 MPI and mixed-mode benchmark

cation routine will also be revisited to introduce the task grouping used in the integration. This should allow the scratch grids to be reduced after each grid level is completed, rather than after all grid levels, potentially improving cache utilisation and hence reducing the overhead of this method. There is also a new sparse matrix library (DBCSR) recently introduced in CP2K, and work is ongoing to improve its parallel efficiency and use of OpenMP.

8 Conclusion

As a result of the work reported in this paper, and ongoing development work from the CP2K Development Team over the last 2 years, the performance of CP2K on the Cray XT has increased dramatically. Time to solution for small systems on the scale of 100s of cores has been more than halved, and scalability has been extended into the 1,000s of cores for such systems using OpenMP. Furthermore, larger systems and hybrid functional calculations have been demonstrated to scale up to 10,000s of cores, which prepares CP2K well for the massively parallel, multicore future of the Cray XT architecture.

9 Acknowledgements

This work was funded by two HECToR dCSE (Distributed Computational Science and Engineering) grants administered by NAG Ltd on behalf of EPSRC.

Dr. Ben Slater at University College London is acknowledged for developing the scientific justification, which was key in securing the funding for this work.

The expertise and helpfulness of Prof. Juerg Hutter's group at the Institute of Physical Chemistry, University of Zurich was key to the success of this project and is gratefully acknowledged.

10 About the Author

Iain Bethune is an HPC Applications Consultant at EPCC, University of Edinburgh. He can be reached by email: ibethune@epcc.ed.ac.uk

References

[1] CP2K Developers Home Page, `http://cp2k.berlios.de`

[2] Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed , M.Parrinello, T. Chassaing and J. Hutter, Comp. Phys. Comm. 167, 103 (2005)

[3] HECToR: UK National Supercomputing Service, `http://www.hector.ac.uk`

[4] CSCS Swiss National Computing Centre, `http://www.cscs.ch`

[5] The Design and Implementation of FFTW3, M. Frigo and S. Johnson, Proceedings of the IEEE 93 (2), 216.231 (2005).

[6] An efficient 3-dim FFT for plane wave electronic structure calculations on massively parrallel machines composed of multiprocessor nodes, S. Goedecker, M. Boulet and T. Deutsch, `http://pages.unibas.ch/comphys/comphys/SOFTWARE/FFT/fft.ps`

[7] FFT Benchmark Results, `http://www.fftw.org/speed/`

[8] Intel®MPI Benchmarks 3.2, `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`

[9] Robust Periodic Hartree-Fock Exchange for Large-Scale Simulations Using Gaussian Basis Sets, M. Guidon, J. Hutter, J.VandeVondele, J. Chem. Theory Comput. 5(11) (2009)