Transforming the high-performance 3d-FFT in ABINIT to enable the use of non-blocking collective operations

Torsten Hoefler and Gilles Zérah

Département de Physique Théorique et Appliquée
Commissariat á l'Énergie Atomique
CEA/DAM Ile-de-France BP 12,
91680 Bruyères-le-Châtel Cedex, France

September 13, 2007

Abstract

This article describes the implementation of the three-dimensional Fast Fourier Transformation in the application ABINIT and investigates a possible way to enhance parallel performance by leveraging communication/computation overlap. We propose code transformations that enable the use of non-blocking collective operations which are implemented by an MPI extension library (LibNBC). The transformations and the code are described in detail and fulfil also the role of a source documentation. The results are presented as microbenchmarks that show the performance gain on an InfiniBand network.

1 Introduction

Recent cluster computer interconnection networks, like InfiniBand™, Myrinet, or Quadrics are able to perform most parts of the message transmission process without CPU interaction. Thus, the main CPU is mostly idle during the message transmission process. This enables new optimization techniques that use the free CPU resources during send and receive operations. MPI-2 as the de-facto standard for the programming of parallel systems supports the usage of this "hardware parallelism" by offering non-blocking send and receive operations that can effectively be used to "overlap" computation and communication. Efficient overlap can only be achieved if **both**, the network technology **and** the MPI library support overlap. This was not necessarily true in the past. However, recent MPI libraries like Open MPI [8] support overlap and so called "asynchronous progress" (that enables the message transmission to progress transparently to the user) sufficiently [13].

Traditionally, collective operations have been a "language tool" to support programmers in the design of often-used communication patterns and to ensure highest performance on a given system

and at the same time performance portability between different systems (cf. [10]). Using collective operations, the programmer could be sure that his program will achieve high performance on nearly all parallel architectures, regardless of the underlying hardware/interconnection system. The clear separation of communication and computation enabled computer scientists to optimize the communication patterns independently and the application programmer could concentrate on the algorithm itself. It is clearly suboptimal to use send/receive calls on scenarios where a collective operation could be applied[1].

The two optimization principles, the usage of collective operations and the overlap of computation and communication are unfortunately mutually exclusive in the MPI-2 standard. Our assumption is that non-blocking collective operations could combine both advantages and lead to more efficient parallel programs. Previous work dealt with the design and implementation of non-blocking collective operations as an addition to the MPI-2 standard. The definition of non-blocking collectives and some possible use cases are introduced in [12].

1.1 Implementation of non-blocking collective operations: LibNBC

An implementation of non-blocking collective operations is available with LibNBC[2] [15]. This library implements every collective operation as a collective schedule that is communicator and process specific and executed asynchronously. Every outstanding non-blocking collective operation has its own state that is represented by the current position in its schedule. This enables highest possible asynchrony and the maximum overlap potential. LibNBC bases on MPI-1 (MPI-2 if Fortran is used), and ANSI C and is therewith highly portable. The implemented collective algorithms can be easily exchanged with machine specific ones. The current version is optimized for the InfiniBand network. A detailed description of LibNBC and it's implementation is available as a technical report [14].

2 State of the Art of 3d-FFT

Fourier transforming three-dimensional boxes is used very often in scientific computing. A popular example is the transformation of wave functions from real to reciprocal space and vice versa in *ab initio* calculations to simplify the calculation. Those applications are very computationally demanding and parallelization is necessary to tackle growing problems.

Thus, implementing efficient parallel three-dimensional FFT algorithms has been subject to research since several years. The serial parts of those algorithms are already highly optimized for many processor architectures or follow automatic tuning approaches (e.g., [7]). The big challenge in parallel FFTs is the relatively high communication complexity. The computation grows with $O(N \cdot log(N))$ while the communication growth is only slightly smaller with $O(N)$. Thus, many

[1] Several application developers achieved higher performance with this technique with a single MPI implementation, but this is neither portable nor clean programming. A better way would have been to modify the MPI implementation of the collective operation

[2] `http://www.unixer.de/NBC`

research groups dealt with a communication-efficient parallel implementation of the 3d-FFT. Adelman et al. [1] and Cramer et al. [3] implemented a 3d-FFT on a cluster of workstations. They both chose to perform two 1d transformations along all x and y lines in parallel, do a parallel transpose (MPI_ALLTOALL), and perform the last transformation in the z direction in parallel. This scheme turned out to be more efficient on cluster computers that other distribution patterns (e.g. redistributing the data twice). Another algorithm, developed in the BSP model is presented by Inda et al. in [16].

However, even if their approach uses collective communication, the computation and the communication is clearly separated and the communication units are idle during computation and vice versa. First experiences with overlapping in parallel 3d FFTs have been gained by Calvin et al. [2]. Dubey at al. analyzed in [4] different communication patterns for parallel FFTs. They found that non-blocking send/receive communication with overlap performs slower than collective operations. Goedecker et al. use a combination of OpenMP and MPI to utilize the available resources more efficiently in [9]. However, their scheme is limited by the number of available processing units per node.

Other FFT implementations, optimized for special parallel hardware, have been proposed in [5, 6, 17].

2.1 Implementation in ABINIT

ABINIT uses two different kinds of FFTs. A normal, also called full FFT is used to transform the electron density, and a zero-padded FFT is used to transform the wave function. Zero-padding the FFT is a way to save computational effort by utilizing special properties of ab initio calculations. The real-space grid has to be bigger than the reciprocal space grid[3], this means that the forward transformation can just ignore zero elements and the backward transformation can fill those elements with zeroes. This saves a considerable amount of computation and communication for the calculation. Many different implementations of the FFT algorithm are available to the user, and we here concentrate on the implementation of the algorithm described in [9], in the version using only MPI.

The following subsections are meant as a documentation for the source code. It is highly recommended to read the cited source-code sections in addition.

2.1.1 Full transformation

Forward The full forward transformation is implemented in `forw.F90`. It starts with the planes distributed in i3 direction. Every i3-plane is first transformed in i1 direction (`fftstp`), repacked with `unswitch` and transformed in i2 direction (`fftstp`) and immediately packed into the communication buffer with `unmpiswitch`. The communication buffer `zmpi1` is so packed i3-plane by i3-plane. The data is packed in the `zmpi1` array as I1,i2,i3,ip2 where ip2 runs from 1 to *nproc*. This scheme allows a redistribution with MPI_ALLTOALL that i2 is distributed among the different processes after the operation.

[3] usually a factor of two is chosen

A blocking MPI_ALLTOALL is performed in the whole buffer after all planes have been packed. The last step transforms the i3-lines line by line with `fftstp`. The data is unpacked from the buffer with the `unscramble` routine.

Backward The full backward transformation, implemented in `back.F90` starts with the i2-planes distributed among the processes. It transforms all i1-i3-lines with `fftstp` and packs the transformed data into the communication buffer `zmpi1` (subroutine `scramble`). The buffer is packed as I1,i2,i3,ip3 and suited to redistribute from i2-planes to i3-planes.

A blocking MPI_ALLTOALL is performed to do this redistribution after all lines have been packed to the buffer. The last step is to unpack every i2-plane with `mpiswitch` and to transform in i1 and i2 direction.

2.1.2 Zero padding

Forward The zero padded forward transformation is implemented in `forw_wf.F90`. It starts with the i3-planes distributed among different processes like the full transform. The difference is that the transformation is not done completely. The i3-planes are again transformed plane by plane. The transformation in i1 direction is done fully (all lines are transformed). The second transformation in the i2 direction is only done for the needed box size. The ratio between real and reciprocal box sizes is determined by the `boxcut` which is usually about 2 for *ab initio* calculations. Thus, we assume a boxcut of 2 for all further explanations. After this two transformations, the box shrunk to 1/4th of its original size (two times halved). This means that only 1/4th of the data has to be communicated in the MPI_ALLTOALL exchange. However, this requires special packing. The routine `unmpiswitch_cent` performs this packing of only the central elements (all others can just be ignored because they are not used in further transformations). The MPI_ALLTOALL is performed, the data is unpacked in `unscramble` and the last transformation in i3 direction is performed on the center region (only 1/2 of the full i3 dimension).

Backward The zero padded backward transformation is similar to the full backward besides the fact that it starts on a small box and ends on a larger one. This requires to pad each dimension (line) to the larger box before the transformation is performed on all elements. The transformation starts similar to the normal backward with the i2-planes distributed. All i1-i3-lines are filled up with zeroes by the subroutine `fill_cent`. The transformation on each line is performed immediately and the transformed data is packed line by line into the `zmpi2` array (`scramble`). The following exchange (MPI_ALLTOALL) is again performed with 1/4th of the data. The unpacking of every i3-plane is performed by `mpiswitch_cent` that also pads the data with zeroes. The following transformation is done in i1 direction, rearranged (with zero padding) and then in i2 direction.

The zero padding FFT is depicted in Figure 1.

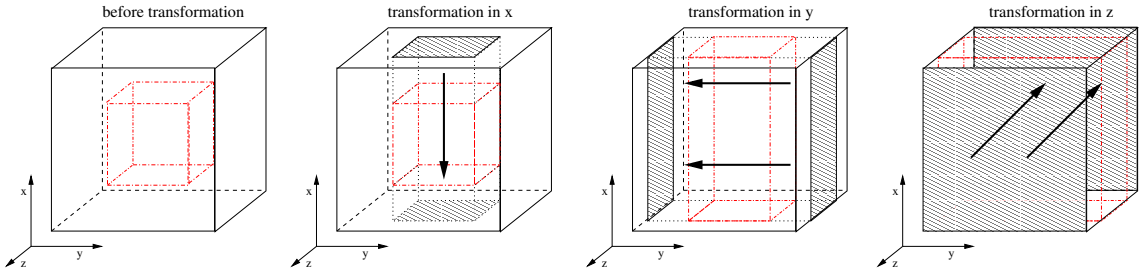

Figure 1: 3d FFT Backward Transformation with zero padding.

3 Code Transformations

We want to apply a pipelining scheme to the transformation. This means that the all-to-all communication of a plane is started as soon as this plane is transformed. The disadvantage is that one has to wait for all communications to end and to unpack the data before the first i3-line can be transformed. The overlap potential decreases for every plane and the latest plane communication is immediately waited for (no overlap is possible). This pipelining technique is only efficient if there is a sufficient number of planes available to pipeline. A second issue is that the overlap potential of LibNBC is highest for large messages sizes. Thus, we need to implement a scheme to collect multiple planes for a single non-blocking communication step. Other tuning features like a window size (limiting number of outstanding requests) or a specific test interval (to progress LibNBC internally) have not been implemented yet.

The whole implementation is guarded with a CPP define (`MPI_NBC`) that enables the configure script to indicate if LibNBC is available or not.

Full Forward Transformation The forward transformation remains mainly unchanged. The current implementation transforms already plane by plane in i3 direction. The new parameter `mpi_enreg%fftplanes_fourdp` indicates the number of i3 planes to accumulate before a non-blocking communication is started. The main difference is that the planes have to be packed differently for the finer-grained communication operation. The communication takes now place on n small all-to-all buffers instead of a single one. The different packing is performed in the new subroutine `unmpiswitch_htor`. This subroutine packs it accordingly to the number of accumulated planes. The different packing schemes for a 3x3x3 box on 3 processes are shown in Figures 2 and 3. The `zmpi1` array is used as a three dimensional array of dimensions $n1 \times n2/p \times n3$. The first dimension is just consecutively filled with data $(1..i1)$. The indexes of the second and third dimension are corrected, according to the required packing scheme.

All outstanding communication operations are finished after the transformation of all i3-planes. The unpacking must of course also obey the new structure of the array `zmpi2`. Thus, the new routine `unscramble_htor` has been introduced. This routine unpacks the selected number of planes into the work array.

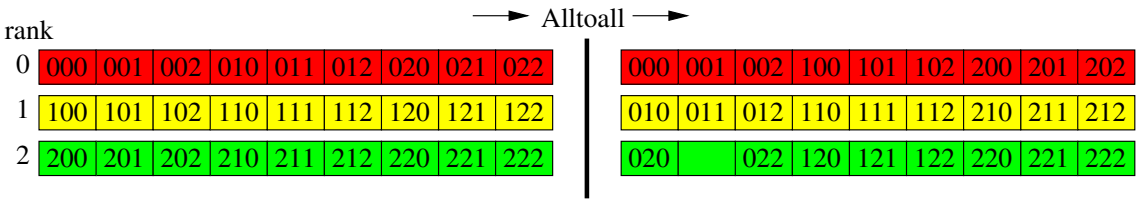

Figure 2: Memory layout for a single Alltoall operation.

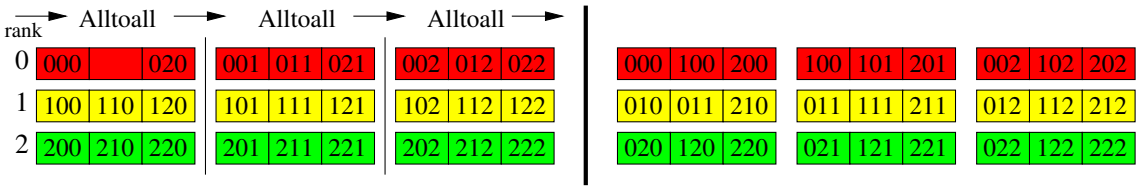

Figure 3: Memory layout for three concurrent Alltoall operations on the same data.

Full Backward Transformation The changes to the backward transformation are similar as in the forward transformation. A big difference is though that the computation time is much smaller and the communication of a single plane is only overlapped with the computation of a 1d transformation of this plane (as opposed to a 2d transformation in the forward transform). The packing function is exchanged with `scramble_htor` and the unpacking function with `mpiswitch_htor`.

Zero padding transformations Both, the forward and backward transformation with zero padding are modified in a similar manner. The modifications are slightly more complicated due to the padding and the different (non block-wise) distribution scheme, but follow the same principles.

4 Microbenchmark Results

Two different sets of benchmarks have been run on all implementations. A detailed benchmark provides insight into the scaling behavior of the different parts of the FFT, while another microbenchmark measures the detailed communication overheads and the general scaling (from an application's perspective) of the implementation.

All benchmarks have been run on the "odin" cluster at Indiana University. The system consists of 128 Opteron nodes. Each node is equipped with 2 2Ghz 270 dual core Opteron processors and 4 GB RAM. The system is connected with an InfiniBand™ network. All the tests have been run over InfiniBand™. The secondary network is Gigabit Ethernet. It is expected that the performance gain for Ethernet is much higher than for InfiniBand™ (cf. benchmarks in [11], ran on the same system).

4.1 Detailed Benchmark

This benchmark structures the implementations described in Section 2 in different logical blocks (they are mainly identical for all transformations in one direction, regardless if a full or partial transformation is performed). We examine the detailed scaling of the forward transformation in the following. The detailed timing can be enabled for the microbenchmark if the code is compiled with the preprocessor-flag `TIMING`.

The implementations have been split into the following blocks:

allocate time needed to allocate all the temporary arrays and calculate the FFT parameters (subroutine `ctrig`)

nd3proc time needed to perform the loop over all z-planes of the local processor (transforms in x and y direction). This step includes the communication initiation time (call to NBC_IALLTOALL) in case of non-blocking communication.

unmpiswitch time needed to pack the communication array. This is a part of the time nd3proc. It has been included to analyze effects of the changed memory layout.

communication time needed to perform the communication. This is the complete communication overhead (call to MPI_ALLTOALL) in the blocking case and the NBC_WAIT overhead in the non-blocking case.

md2proc time needed to perform the last transformation in z-direction

unscramble subset of md2proc, indicates the time to unpack the communication array. It has been included to study the effects of the changed memory layout.

deallocate time for array deallocations

All the benchmarks have been run on the odin cluster with 1 CPU/node and the indicated times are an average of three program runs.

4.2 Detailed timing results for the full forward transformation

This section provides the timing results for the forward transformation of a 128^3 cube for different node numbers (4, 8 and 16). This allows to determine the parallel scaling and to assess the implementation quality. The original code has been highly cache-optimized so that every small change to the memory access pattern may have significant consequences. The benchmarks have been run with fftplanes=2 (two planes are accumulated into a single non-blocking communication) to have the same pipeline-depth as the zero-padding implementation.

Table 1 shows detailed timing results for the full forward transform. The second column gives the absolute time for the block in seconds on 4 processrs. The columns for 8 and 16 processors show the speedup with regards to the 4 processor base-case (ideally 2 and 4 respectively). The second part of the table shows the timings for the non-blocking implementation. This part shows also the relative time with regards to the blocking implementation (ideally 1 or smaller).

Table 1: Full forward transform detailed timing

forw - original implementation			
block	4 procs	8 procs (speedup)	16 procs (speedup)
allocate	0.000023	1.06	1.05
nd3proc	0.037205	1.85	3.62
unmpiswitch	0.005316	2.01	4.15
communication	0.022314	1.67	3.03
md2proc	0.022962	1.95	3.97
unscramble	0.008718	2.42	5.13
deallocate	0.000008	1.00	1.02
forw_htor - non-blocking implementation			
block	4 procs (rel. to orig)	8 procs sp. (rel. to orig)	16 procs sp. (rel. to orig)
allocate	0.000022 (0.95)	1.02 (0.97)	1.04 (0.96)
nd3proc	0.046629 (1.25)	1.72 (1.33)	2.95 (1.53)
unmpiswitch	0.005430 (1.02)	1.92 (1.06)	4.03 (1.05)
communication	0.002268 (0.10)	0.36 (0.46)	0.58 (0.53)
md2proc	0.024388 (1.06)	1.81 (1.14)	3.67 (1.14)
unscramble	0.008783 (1.00)	1.58 (1.53)	3.38 (1.52)
deallocate	0.000015 (1.76)	1.27 (1.38)	1.40 (1.29)

The non-blocking implementation spends more time in the nd3proc loop because all non-blocking communications are initiated and progressed (NBC_TEST) in this loop. However, much time is saved in the communication where the non-blocking implementation performs 50-90% faster than the blocking case. One can also see that the modified `unscramble` routine scales worse than the original version with a growing processor number. This is due to the constant overhead of the calculation of the corrected j3-index (cf. Section 3, the calculation could possibly be optimized if a direct algorithm would be found). The remaining blocks are nearly identical to the original implementation.

4.3 Detailed timing results for the partial forward transformation

Similar to Section 4.2, we discuss the detailed timing results of the zero-padding transformation of a 128^3 cube with a boxcut of 2 in the following. The blocks are similar to the full transform. The main difference is the changed packing and unpacking routines and that less data is communicated. All benchmarks have been run with fftplanes=1 (one plane per non-blocking communication) to guarantee the same pipeline-depth as for the full transform. The detailed results are shown in Table 2.

4.4 Communication and Scaling Benchmark

This benchmark measures the parallel scaling of the transformation and the communication overhead in relation to the number of planes accumulated for communication. The influence of using

Table 2: Partial forward transform detailed timing

forw_wf - original implementation			
block	4 procs (orig.)	8 procs (speedup)	16 procs (speedup)
allocate	0.000019	1.00	0.94
nd3proc	0.040705	1.97	4.13
unmpiswitch	0.001541	1.96	3.92
communication	0.006264	2.12	3.47
md2proc	0.004317	1.78	3.69
unscramble	0.001508	2.02	4.36
deallocate	0.000008	1.33	1.60
forw_wf_htor - non-blocking implementation			
block	4 procs (rel. to orig)	8 procs sp. (rel. to orig)	16 procs sp. (rel. to orig)
allocate	0.000055 (2.89)	2.89 (1.00)	2.89 (1.01)
nd3proc	0.064339 (1.58)	2.78 (1.12)	5.62 (1.16)
unmpiswitch	0.002681 (1.74)	3.19 (1.07)	6.31 (1.08)
communication	0.005237 (0.84)	31.17 (0.06)	7.39 (0.39)
md2proc	0.005115 (1.18)	2.02 (1.04)	4.23 (1.04)
unscramble	0.001771 (1.17)	2.07 (1.15)	4.85 (1.05)
deallocate	0.000018 (2.25)	2.00 (1.50)	2.25 (1.60)

more than one CPU-core per node is also investigated. Ideally, all cores (in our case 4) should be used to perform useful computation or to progress communication. Neither LibNBC nor MPI provides useful asynchonous progress in the case of collective communication. That is why we do not investigate the use of one or more cores as communication co-processor. All benchmarks have been executed with a 128^3 processor grid from 1 to 32 processes (the FFT saturates at this point) and 1 to 4 used processing cores.

4.4.1 The role of the `fftplanes` parameter

The fftplanes parameter determines the number of planes that are collected to a single communication operation. The current implementation requires that the third dimension of the FFT-box (the large in case of a full transform and the small in case of a zero-padded transform) is divisible by fftplanes. The fftplanes f parameter influences two other parameters, namely the size of the communicated data and the depth of the pipeline. The whole communicated data size per processor s_{bl} (blocking case) for the 128^3 case can be calculated by $s_{bl} = \frac{128^3}{p} \cdot 2 \cdot 8\ bytes$ (complex transform). In the non-blocking case, the communication is splitted into multiple smaller pipelined communications. The number of those communications, i.e., the pipeline depth d, is determined by the fftplanes parameter: $d = 128/f$. The size of every non-blocking communication s_{nb} is determined by $s_{nb} = s_{bl}/128 \cdot f = s_{bl}/d = \frac{128^2}{p} \cdot f \cdot 2 \cdot 8\ bytes$.

NBC deliver the optimal performance when the communicated data size is large and there is sufficient time to overlap. The time to overlap t for every single element α_i the pipeline is the time that the remaining items $\alpha_{i+1} \dots \alpha_{128}$ need to compute. Assuming that a single plane needs the

time λ to compute, the element i ($\forall\ 0 < i \leq 128$) has $t = \lambda \cdot (128 - i)$ to overlap the communication (in case $f = 1$). For the general case, (variable f), $t(f) = f\lambda \cdot (128/f - j)$ ($\forall\ 0 < j \leq 128/f$). This means that the last plane has no time to overlap and the size of this plane is directly proportional to f.

The communicated data size is also directly proportional to f. This makes it not trivial to choose the optimal f value for a given problem. However, we tested different scenarios and found that the size dominates the communication overhead gain clearly. Figure 4 shows how the datasize influences the possible gain of overlap in an all-to-all communication for a communicator size of 16.

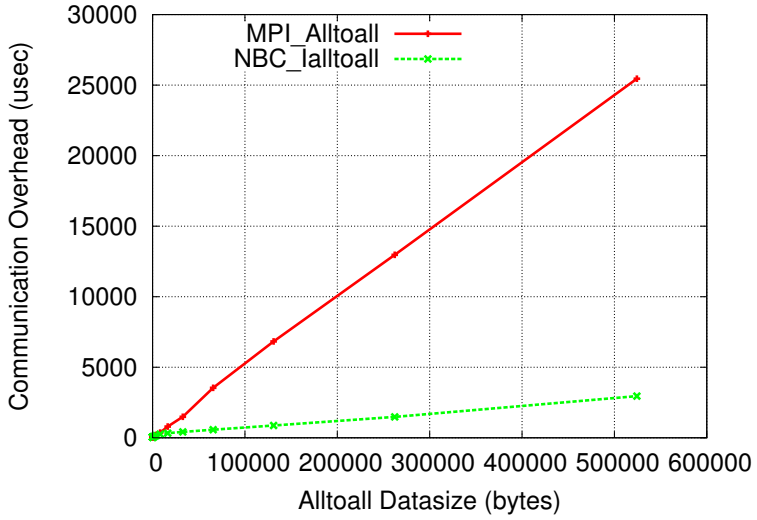

Figure 4: Communication Overhead for blocking MPI_ALLTOALL and non-blocking NBC_IALLTOALL with maximum possible overlap for a communicator with 16 processes (1 process per node)

Results for the full transform with 1 CPU/node and different fftplanes are shown in Figure 5. One can see that a bigger fftplanes parameter benefits the scaling if there are still enough communication operations for the pipeline ($d = 128/f$).

4.4.2 Performance gain due to the use of NBC

The performance gain due to the use of NBC in the parallel FFT is shown in Figure 6 for the full transform and in Figure 7 for the zero-padded transform. The ideal fftplanes parameter is assumed (result from multiple experiments). The gain itself varies highly and the implementation seems to have an anomality at 16 processes in both cases. This is subject to further research.

The gain in overall performance is mostly due to a reduced communication overhead. The communication overheads for a full and zero-padded transformation, utilizing all available processing

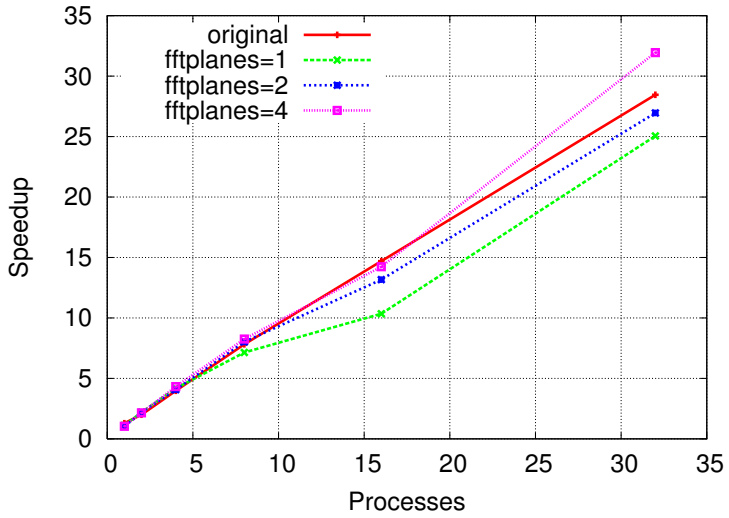

Figure 5: Influence of the fftplanes parameter on the scaling of the parallel FFT

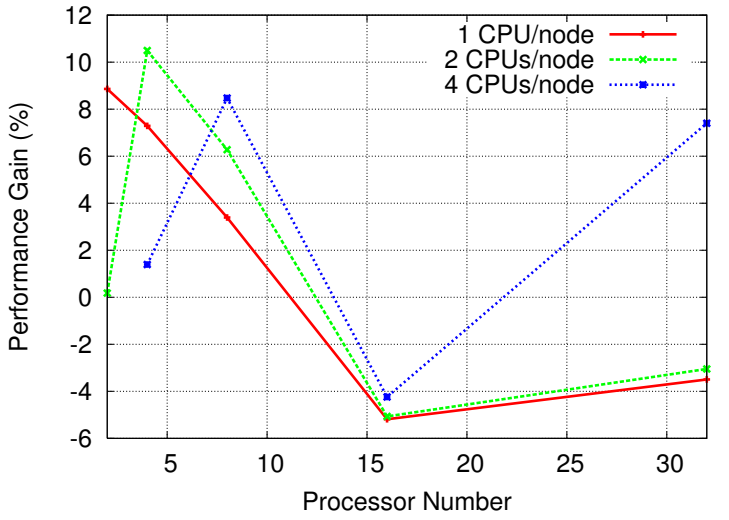

Figure 6: Performance gain due to the use of NBC for the full transformation (the ideal fftplanes parameter was chosen)

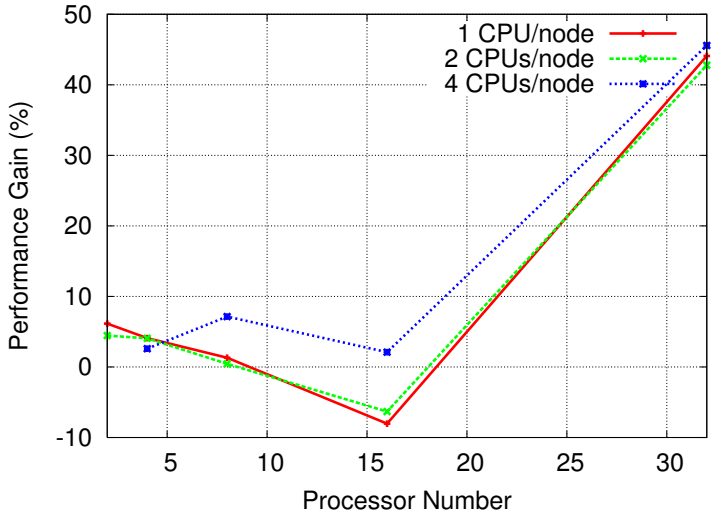

Figure 7: Performance gain due to the use of NBC for the zero-padded transformation (the ideal fftplanes parameter was chosen)

cores per node (this should be the default case), are shown in Figure 8 and Figure 9 respectively.

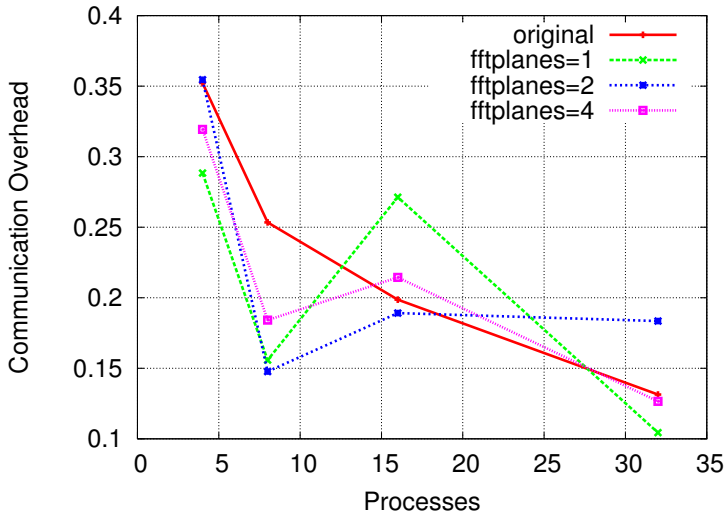

Figure 8: Communication overhead for the full transform

The variation in the overheads is also pretty high. But for every number of processes, there exists at least one fftplanes parameter that has a lower communication overhead than the original version.

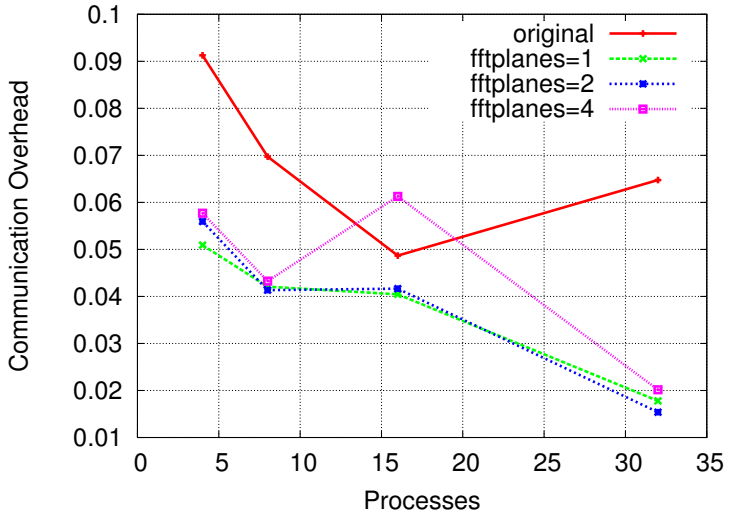

Figure 9: Communication overhead for the zero-padded transform

The increase in the overall costs for the 16 processor case is due to the unfortunate changed memory access pattern that slows the packing/unpacking down (i.e., is not as cache and prefetch friendly as the original pattern).

5 Implementation in ABINIT

The modified functions that have been described in Section 2 have an identical interface to the functions that are used to perform the transformation in ABINIT. The microbenchmark used in Section 4 is only a driver to those routines. The implementation into ABINIT is thus straightforward.

One of the main goal was to influence the existing code as minimal as possible. All new functions have been added to the existing code-base with the suffix `_htor`. Three additional parameters have been added to the input file:

fftplanes_fourdp the fftplanes parameter for the full transformation `forw,back`

fftplanes_forw_wf the fftplanes parameter for the zero-padded forward transformation

fftplanes_back_wf the fftplanes parameter for the zero-padded backward transformation

All parameters have the default-value 0 that means that the original (blocking) implementation is used to perform the transformation. The non-blocking variant is used if the parameter is larger than 0 and the number is used as the fftplanes parameter for the given transformation (it is decreased until it fulfills the divisibility requirements). Passing a -1 as parameter selects the experimental autotuning feature. Autotuning benchmarks the first runs of the fft with different

fftplane parameters (0 to 8) and selects the fastest one. However, this benchmark is only done once at the beginning and is highly influenced by jitter in the system (the selection is likely to be suboptimal).

6 Conclusions and Future Work

We showed that the performance of a highly cache-optimized fast Fourier transformation can be further improved with the use of non-blocking collective operations. We used a simple pipelining scheme to enable overlap of communication and computation for this case. Furthermore, we investigated the effects of the changes memory access pattern in a detailed speedup analysis. The suboptimal memory access pattern for the pack/unpack operations limit the performance gain of NBC. However, in most cases, an additional decrease in running time could be achieved with the use of NBC.

The influence of the communicated data size has also been investigated. A parameter, that enables the accumulation of multiple planes in a single communication (fftplanes) has been introduced and its influence on the running time has been discussed. The choice of this parameter is non-trivial and hardly predictable (highly system dependent). We recommend, in the case of multiple transformation, to benchmark different parameters and use the fastest one.

References

[1] A. Adelmann, W. P. Petersen A. Bonelli and, and C. W. Ueberhuber. Communication efficiency of parallel 3d ffts. In *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, volume 3402 of *Lecture Notes in Computer Science*, pages 901–907. Springer, 2004.

[2] C. Calvin and F. Desprez. Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines, 1993.

[3] C. E. Cramer and J. A. Board. The development and integration of a distributed 3d fft for a cluster of workstations. In *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta*, volume 4. USENIX Association, 2000.

[4] Anshu Dubey and Daniele Tessera. Redistribution strategies for portable parallel FFT: a case study. *Concurrency and Computation: Practice and Experience*, 13(3):209–220, 2001.

[5] Maria Eleftheriou, Blake G. Fitch, Aleksandr Rayshubskiy, T. J. Christopher Ward, and Robert S. Germain. Performance measurements of the 3d fft on the blue gene/l supercomputer. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 795–803. Springer, 2005.

[6] B. Fang and Y. Deng. Performance of 3d fft on 6d qcdoc torus parallel supercomputer. J. Comp. Phys. Submitted, 2005.

[7] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[9] S. Goedecker, M. Boulet, and T. Deutsch. An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. *Computer Physics Communications*, 154:105–110, August 2003.

[10] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[11] T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In *Recent Advantages in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI User's Group Meeting, Proceedings, LNCS 4192*, pages 374–382. Springer, 9 2006.

[12] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, and W. Rehm. Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University, 08 2006.

[13] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-Blocking Collective Operations. In *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, volume 4331/2006, pages 155–164. Springer Berlin / Heidelberg, 12 2006.

[14] Torsten Hoefler and Andrew Lumsdaine. Design and implementation of the nbc library. Technical report, Indiana University, 2006.

[15] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *In proceedings of IEEE Supercomputing'07*, 2007.

[16] Márcia A. Inda and Rob H. Bisseling. A simple and efficient parallel FFT algorithm using the BSP model. *Parallel Computing*, 27(14):1847–1878, 2001.

[17] Kenneth Moreland and Edward Angel. The fft on a gpu. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.