A Performance Analysis of ABINIT on a Cluster System

Torsten Hoefler[1], Rebecca Janisch[2], and Wolfgang Rehm[1]

[1] Chair of Computer Architecture, Chemnitz University of Technology, 09107 Chemnitz, Germany
[2] Electrical Engineering and Information Technology, Chemnitz University of Technology, 09107 Chemnitz, Germany

1 Introduction

1.1 Electronic Structure Calculations

In solid state physics, bonding and electronic structure of a material can be investigated by solving the quantum mechanical (time-independent) Schrödinger equation,

$$\widehat{H}_{\text{tot}}\Phi = E_{\text{tot}}\Phi \quad , \tag{1}$$

in which the Hamilton operator $\widehat{H}_{\text{tot}}$ describes all interactions within the system. The solution Φ, the wavefunction of the system, describes the state of all N electrons and M atomic nuclei, and E_{tot} is the total energy of this state.
Usually, the problem is split by separating the electronic from the ionic part by making use of the Born-Oppenheimer approximation [1]. Next we consider the electrons as independent particles, represented by one-electron wavefunctions ϕ_i. Density functional theory (DFT), based on the work of Hohenberg and Kohn [2] and Kohn and Sham [3], then enables us to represent the total electronic energy of the system by a functional of the electron density $n(\mathbf{r})$:

$$n(\mathbf{r}) = \sum_i |\phi_i|^2 \tag{2}$$

$$\begin{aligned} \rightarrow E = E\left[n(r)\right] &= F[n] + \int V_{\text{ext}}(\mathbf{r})n(\mathbf{r})d\mathbf{r} \\ &= E_{\text{kin}}[n] + E_{\text{H}}[n] + E_{\text{xc}}[n] + \int V_{\text{ext}}(\mathbf{r})n(\mathbf{r})d\mathbf{r} \quad . \end{aligned} \tag{3}$$

Thus the many-body problem is projected onto an effective one-particle problem, resulting in a reduction of the degrees of freedom from $3N$ to 3. The

one-particle Hamiltonian $\hat{H}$ now describes electron i, moving in the effective potential V_{eff} of all other electrons and the nuclei.

$$\begin{aligned} \hat{H}\ \phi_i &= \epsilon_i\ \phi_i \\ \left\{-\tfrac{\hbar^2\Delta}{2m} + V_{\text{eff}}[n(\mathbf{r})]\right\} \phi_i(\mathbf{r}) &= \epsilon_i\ \phi_i(\mathbf{r}), \\ \text{where}\quad V_{\text{eff}}[n(\mathbf{r})] \quad &= \quad V_{\text{eff}}(\mathbf{r}) = V_{\text{H}}(\mathbf{r}) + V_{\text{xc}}(\mathbf{r}) + V_{\text{ext}}(\mathbf{r}) \quad . \end{aligned} \tag{4}$$

In (4), which are part of the so-called Kohn-Sham equations, $-\frac{\hbar^2\Delta}{2m}$ is the operator of the kinetic energy, V_{H} is the Hartree and V_{xc} the exchange-correlation potential. V_{ext} is the external potential, given by the lattice of atomic nuclei. For a more detailed explanation of the different terms see e.g. [4]. The self-consistent solution of the Kohn-Sham equations determines the set of wavefunctions ϕ_i that minimize the energy functional (3). In order to obtain it, a starting density n_{in} is chosen from which the initial potential is constructed. The eigenfunctions of this Hamiltonian are then calculated, and from these a new density n_{out} is obtained. The density for the next step is usually a combination of input and output density. This process is repeated until input and output agree within the limits of the specified convergence criteria.
There are different ways to represent the wavefunction and to model the electron-ion interaction. In this paper we focus on pseudopotential+planewave methods.
If the wavefunction is expanded in plane waves,

$$\phi_i = \sum_{\mathbf{G}} c_{i,\mathbf{k}+\mathbf{G}} e^{i(\mathbf{k}+\mathbf{G})\mathbf{r}} \tag{5}$$

the Kohn-Sham equations assume the form [5]

$$\sum_{\mathbf{G}'} H_{\mathbf{k}+\mathbf{G},\mathbf{k}+\mathbf{G}'} \times c_{i,\mathbf{k}+\mathbf{G}'} = \epsilon_{i,\mathbf{k}} c_{i,\mathbf{k}+\mathbf{G}} \quad , \tag{6}$$

with the matrix elements

$$\begin{aligned} H_{\mathbf{k}+\mathbf{G},\mathbf{k}+\mathbf{G}'} = \ & \frac{\hbar^2}{2m}|\mathbf{k}+\mathbf{G}|^2\delta_{\mathbf{G}\mathbf{G}'} \\ & + V_{\text{H}}(\mathbf{G}-\mathbf{G}') + V_{\text{xc}}(\mathbf{G}-\mathbf{G}') + V_{\text{ext}}(\mathbf{G}-\mathbf{G}') \quad . \end{aligned} \tag{7}$$

In this form the matrix of the kinetic energy is diagonal, and the different potentials can be described in terms of their Fourier transforms. Equation (6) can be solved independently for each k-point on the mesh that samples the first Brillouin zone. In principle this can be done by conventional matrix diagonalization techniques. However, the cost of these methods increases with the third power of the number of basis states, and the memory required to store the Hamiltonian matrix increases as the square of the same number. The number of plane waves in the basis is determined by the choice of the cutoff energy $E_{\text{cut}} = \hbar^2/2m|\mathbf{k}+\mathbf{G}_{\text{cut}}|^2$ and is typically of the order of 100 per atom, if

norm-conserving pseudopotentials are used. Therefore alternative techniques have been developed to minimize the Kohn-Sham energy functional (3), e.g. by conjugate gradient (CG) methods (for an introduction to this method see e.g. [6]). In a band-by-band CG scheme one eigenvalue (band) $\epsilon_{i,\mathbf{k}}$ is obtained at a time, and the corresponding eigenvector is orthogonalized with respect to the previously obtained ones.

1.2 The ABINIT Code

ABINIT [7] is an open source code for *ab initio* electronic structure calculations based on the DFT described in Sect. 1.1. The code is the object of an ongoing open software project of the Université Catholique de Louvain, Corning Incorporated, and other contributors [8] .

ABINIT mostly aims at solid state research, in the sense that periodic boundary conditions are applied and the majority of the integrals that have to be calculated are represented in reciprocal space (k-space). It currently features the calculation of various electronic ground state properties (total energy, bandstructure, density of states,..), and several structural optimization routines. Furthermore it enables the investigation of electric and magnetic polarization and electronic excitations. Originally a pseudopotential+planewave code, ABINIT for a short time (since version 4.2.x) also features the projector-augmented wave method, but this is still under developement. In the following we refer to the planewave method.
To begin the self-consistency cycle, a starting density is constructed, and a starting potential derived. The eigenvalues and eigenvectors are determined by a band-by-band CG scheme [5], during which the density (i.e. the potential) is kept fixed until the whole set of functions has been obtained. The alternative of updating the density with each new band has been abandoned, to make a simple parallelization of the calculation over the k-points possible. Only at the end of one CG loop is the density updated by the scheme of choice (e.g. simple mixing, or Anderson mixing). For a comparison of different schemes see e.g. [9]). A more detailed description of the DFT implementation in general is given in [10].

Different levels of parallelization are implemented. The most efficient parallelization is the distribution of the k-points that are used to sample the Brillouin zone on different processors. Unfortunately the necessary number of k-points decreases with increasing system size, so the scaling with the number of atoms is rather unfavourable. One can partially make up for this by distributing the work related to different states (or bands) within a given k-point. Since the number of states increases with the system size, the overall scaling with number of atoms improves. For example a blocked conjugate gradient algorithm can be used to optimize the wavefunctions, which provides the possibility to parallelize over the states within one block. Instead of a single

eigenstate, as in the band-by-band scheme, `nbdblock` states are determined at the same time, where `nbdblock` is the number of bands in one block. Of course this leads to a small increase in the time that is needed to orthogonalize the eigenvectors with respect to those obtained previously. Furthermore, to guarantee convergence, a too high value for `nbdblock` should not be chosen. The ABINIT manual advises $\texttt{nbdblock} \leq 4$ as a meaningful choice.
Both methods, k-point parallelization and parallelization over bands, are implemented using the MPI library. A third possibility of parallelization is given by the distribution of the work related to different wavefunction coefficients, which is realized with OpenMP compiler directives. It is used for example in the parallelization of the FFT algorithm, but this feature is still under developement.

These parallelization methods, which are based on the underlying physics of the calculation, are useful only for a finite number of CPUs (a fact, that is not a special property of ABINIT, but common to all electronic structure codes). In a practical calculation, the required number of k-points, `nkpt`, for a specific geometry is determined by convergence tests. To decrease the computational effort, the k-point parallelization is then the first method of choice. The best speedup is achieved if the number of k-points is an integer multiple of the number of CPUs:

$$\texttt{nkpt} = n \times N_{\mathrm{CPU}} \quad \text{with} \quad n \in \mathbb{N}. \tag{8}$$

Ideally, $n = 1$.
If the number of available CPUs is larger than the number of k-points needed for the calculation, the speedup saturates. In this case, the additional parallelization over bands can improve the performance of ABINIT, if

$$N_{\mathrm{CPU}} = \texttt{nbdblock} \times \texttt{nkpt} \quad . \tag{9}$$

In principle the parallelization scheme also works for $N_{\mathrm{CPU}} = \texttt{nbdblock}$, which results in a parallelization over bands only. However, this is rather inefficient, as will be seen below.

1.3 Related Work

The biggest challenge after programming a parallel application is to optimize it according to a given parallel architecture. The first step of each optimization process is the performance and scalability measurement which is often called benchmarking. There are methods based on theoretical simulation [11, 12] and methods based on benchmarking [13, 14]. There are also studies which try to explain bottlenecks for scalability [15] or studies which compare different parallel systems [16]. In the following we analyze parallel efficiency and scalability of the application ABINIT on a cluster system and describe the results with the knowledge gained about the application. We also present several simple ideas to improve the scaling and performance of the parallel application.

2 Benchmark Methodology

The parallel benchmark runs have been conducted on two different cluster systems. The first one is a local cluster at the University of Chemnitz which consists of 8 Dual Xeon 2.4GHz systems with 2GB of main memory per CPU. The nodes are interconnected with Fast Ethernet. We used MPICH2 1.0.2p1 [17] with the ch_p4 (TCP) device as the MPI communication library. The source code of ABINIT 4.5.2 was compiled with the Intel Fortran Compiler 8.1 (Build 20050520Z). The relevant entries of the `makefile_macros` are shown in the following:

```
FC=ifort
COMMON_FFLAGS=-FR -w -tpp7 -axW -ip -cpp
FFLAGS=$(COMMON_FFLAGS) -O3
FFLAGS_Src_2psp =$(COMMON_FFLAGS) -O0
FFLAGS_Src_3iovars =$(COMMON_FFLAGS) -O0
FFLAGS_Src_9drive =$(COMMON_FFLAGS) -O0
FFLAGS_LIBS=-O3 -w
FLINK=-static
```

Listing 1. Relevant `makefile_macros` entries for the Intel Compiler

The -O3 optimization had to be disabled for several directories, because of endless compiling. For the serial runs we also compiled ABINIT with the open source g95 Fortran compiler, with the following relevant flags:

```
FC=g95
FFLAGS=-O3 -march=pentium4 -mfpmath=sse -mmmx \
           -msse -msse2
FLINK=-static
```

Listing 2. Relevant `makefile_macros` entries for the g95 Compiler

The second system is a Cray Opteron Cluster (strider) of the High Performance Computing Center Stuttgart (HLRS), consisting of 256 2 GHz AMD Opteron CPUs with 2GB of main memory per CPU. The nodes are interconnected with a Myrinet 2000 network. ABINIT has been compiled with the 64-bit PGI Fortran compiler (version 5.0). On strider the MPI is implemented as a port of MPICH (version 1.2.6) over GM (version 2.0.8).

```
FC=pgf90
FFLAGS=-tp=k8-64 -Mextend -Mfree -O4
FFLAGS_LIBS = -O4
LDFLAGS=-Bstatic -aarchive
```

Listing 3. Relevant `makefile_macros` entries for the pgi Compiler

2.1 The Input File

All sequential and parallelized benchmarks have been executed with an essentially identical input file which defines a (hexagonal) unit-cell of Si_3N_4 (two formula units). We used 56 bands and a planewave energy cut-off of 30 Hartree (resulting in $\approx$ 7700 planewaves). A Monkhorst-Pack k-point mesh [18] was used to sample the first Brillouin zone. The number of k-points along the axes of the mesh was changed with the `ngkpt` parameter as shown in Table 1.
To use parallelization over bands, we switched from the default band-by-band wavefunction optimisation algorithm to the blocked conjugate gradient algorithm (`wfoptalg` was changed to 1) and chose numbers of bands per block > 1 (`nbdblock`) according to (9) in section 1.2.

`nkpt`	`ngkpt`
2 2 2	2
4 2 2	4
4 4 2	8
4 4 4	16

Table 1. Number of k-points along the axes of the Monkhorst-Pack mesh, `ngkpt`, and resulting total number of k-points in the calculation, `nkpt`.

3 Benchmark Results

3.1 Sequential Analysis

Due to the fact that a calculation which is done on a single processor in the k-point parallelization is exactly the same as in the sequential case, a sequential analysis can be used to analyze the behaviour of the calculation itself.

Call Graph

The call graph of a program shows all functions which are called during a program run. We used the `gprof` utility from the gcc toolchain and the program `cgprof` to generate a call graph. ABINIT called 300 different functions during our calculation. Thus, the full callgraph is much too complicated and we present only a short extract with all functions that use more than 4% of the total application runtime (Fig. 1).

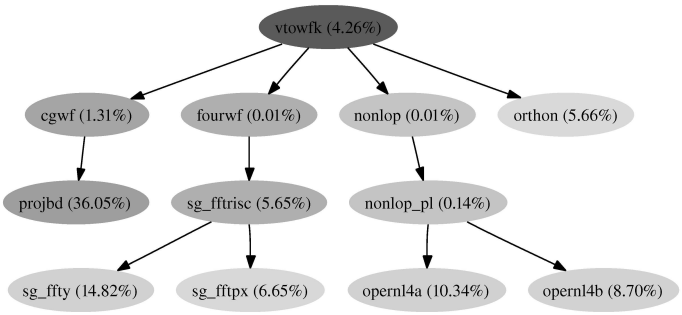

Fig. 1. The partial callgraph

The percentage of the runtime of the functions is given in the diagram, and the darkness of the nodes indicates the percentage of the subtree of these nodes (please keep in mind that not all functions are plotted). About 97% of the runtime of the application is spent in a subtree of `vtowfk` which computes the density at a given k-point with a conjugate gradient minimization method. The 8 most time consuming functions need more than 92% of the application runtime (they are called more than once). The most time demanding function is `projbd` which orthogonalizes a state vector against the ones obtained previously in the band-by-band optimization procedure.

Impact of the Compiler

Compilation of the source files can be done with various compilers. We compared the open-source g95 compiler with the commercial Intel Fortran Compiler 8.1 (abbreviated with ifort). The Intel compiler is able to auto-parallelize the code (cmp. OpenMP), this feature has also been tested on our dual Xeon processors. The benchmarks have been conducted three times and the mean value is displayed. The results of our calculations are shown in Table 2.

Compiler/Features	Runtime (s)
ifort	625.07
ifort -parallel	643.19
g95	847.73

Table 2. Comparison of different compilers

This shows clearly that the Intel Compiler generates much faster code than the g95. However, the g95 compiler is currently under development and there is a lot of potential for optimizations. The auto-parallelization feature of the ifort is also not beneficial, this could be due to the thread spawning overhead at small loops.

Impact of the BLAS Library

Mathematical libraries such as the BLAS Library are used to provide an abstraction of different algebraic operations. These operations are implemented in so called math libraries which are often architecture specific. Many of them are highly optimized and can accelerate the code by a significant factor (as compared to "normal programming"). ABINIT offers the possibility to exchange the internal math implementations with architecture-optimized variants. A comparison in runtime (all libraries compiled with the g95 compiler) is shown in Table 3.

BLAS Library	Runtime (s)
internal	847.73
Intel-MKL	845.62
AMD-ACML	840.56
goto BLAS	860.60
Atlas	844.67

Table 3. Comparison of different mathematical libraries.

The speedup due to an exchange of the math library is negligible. One reason could be that the mathematical libraries are not efficiently implemented and do not offer a significant improvement in comparison to the reference implementation (internal). However, this seems unlikely since all the libraries are highly optimized and several were tested. A more likely explanation is that the libraries are not used very often in the code (calls and execution do not consume much time compared to the total runtime). To investigate this we analyzed the callgraph with respect to calls to math-library functions, and found that indeed all calls to math libraries such as `zaxpy`, `zswap`, `zscal`, ... make less than 2% of the application runtime (with the internal math library). Thus, the speedup of the whole application cannot exceed 2% even if the math libraries are improved.

3.2 Parallel Analysis

Speedup Analysis

Figure 2 shows the speedup versus the number of CPUs on the Xeon Cluster. In all cases except the case of 16 k-points the scaling is almost ideal as long as $N_{CPU} \leq$ `nkpt`. Saturation is observed for $N_{CPU} >$ `nkpt`, only for 16 k-points this occurs already for $N_{CPU} = 8$, due to overheads from MPI barrier synchronizations in combination with process skew on the Xeon Cluster (see section 3.2). The parallelization over bands, which needs intense communication, only leads to negligible additional speedup on this Fast Ethernet network. Figure

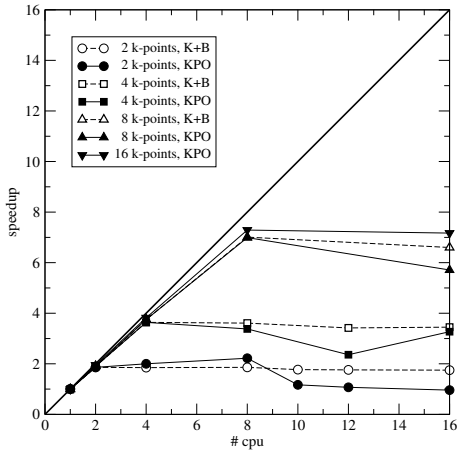

Fig. 2. Speedup vs. number of CPUs on the Xeon Cluster. Parallelization over k-points only (KPO - filled symbols): Except for the case of 16 k-points the scaling is almost ideal as long as $N_{CPU} \leq$ `nkpt` . Saturation is observed for $N_{CPU} >$ `nkpt`, only for 16 k-points this occurs already for $N_{CPU} = 8$. Parallelization over k-points and bands (K+B - open symbols): The number of bands per block at the different data points equals N_{CPU}/`nkpt`. Only negligible additional speedup is observed.

3 shows the speedup vs. the number of CPUs on the Cray Opteron Cluster. For the parallelization over k-points only, an almost ideal speedup is obtained for small numbers (≤ 8) of CPUs. The less than ideal behaviour for larger numbers can be explained by a communication overhead, see section 3.2. For $N_{CPU} >$ `nkpt` the speedup saturates, as expected. In this regime the speedup can be considerably improved (up to 250% in the case of 4 k-points and 16 CPUs) by including the parallelization over bands, as long as the number of bands per block remains reasonably small (≤ 4).

Communication Analysis I: Plain *k*-point Parallelization

We used the MPE environment and Jumpshot [19] to perform a short analysis of the communication behaviour of ABINIT in the different working scenarios on the local Xeon Cluster. This parallelization method is investigated in two scenarios, the almost ideal speedup with 8 processors calculating 8 k-points and less than ideal speedup with 16 processors calculating 16 k-points. The MPI communication scheme of 8 processors calculating 8 k-points is shown in the following diagram. The processors are shown on the ordinate (rank 0-7), and the communication operations are shown for each of them. Each MPI operation corresponds to a different shade of gray. The processing is not depicted (black). The ideal communication diagram would show nothing but

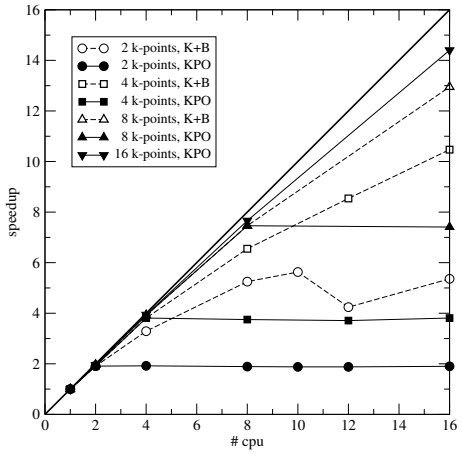

Fig. 3. Speedup vs. number of CPUs on the Cray Opteron Cluster. Parallelization over k-points only (KPO - filled symbols): For $N_{CPU} \leq$ `nkpt` the scaling is almost ideal. Saturation is observed for $N_{CPU} >$ `nkpt`. Parallelization over k-points and bands (K+B - open symbols): The number of bands per block at the different data points equals $N_{CPU}/$ `nkpt`. For reasonable numbers `nbdblock` ≤ 4) the speedup in the formerly saturated region improves considerably.

a black screen, every MPI operation delays the processing and increases the overhead.
Figure 4 shows the duration of all calls to the MPI library. This gives a rough overview of the parallel performance of the application. The parallelization

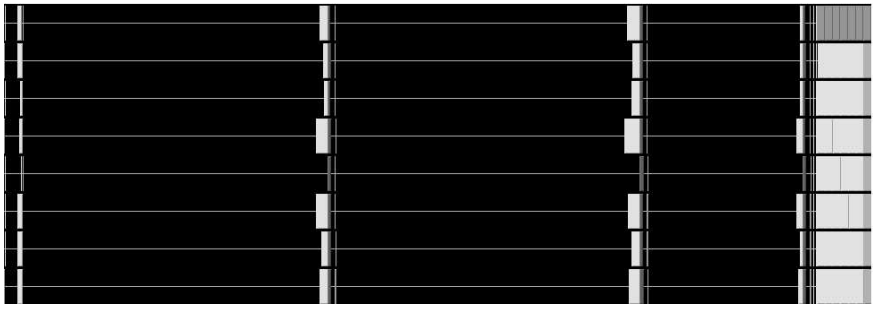

Fig. 4. Visualization of the MPI overhead for k-point parallelization over 8 k-points on 8 Processors. Each MPI operation corresponds to a different shade of gray: `MPI_Barrier` white, `MPI_Bcast` light gray, and `MPI_Allreduce` dark gray. The overhead is negligible and this case is efficiently parallelized.

is very efficient, all processors are computing most of the time, some CPU time is lost during the barrier synchronization. The three self consistent field

(SCF) steps can easily be recognized as the processing time (black) between the `MPI_Barrier` operations. The synchronization is done at the end of each step and afterwards a small amount of data is exchanged via `MPI_Allreduce`, but this does not add much overhead. The parallelization is very efficient and only a small fraction of the time is spent for communication. All processors send their results to rank 0 in the last part, after the last SCF step and synchronization. This is done with many barrier operations and send-receive, and could significantly be enhanced with a single call to `MPI_Gather`.
The scheme for 16 processors calculating 16 k-points, shown in Fig. 5, is nearly identical, but the overhead resulting from barrier synchronization is much higher and decreases the performance. This is due to so called process skew, where the unpredictable and uncoordinated scheduling of operating system processes on the cluster system interrupts the application and introduces a skew between the processes. This skew adds up during the whole application runtime due to the synchronization at the end of each SCF step. Thus, the scaling of ABINIT is limited on our cluster system due to the operating system's service processes. Even if the problem is massively parallel the overhead

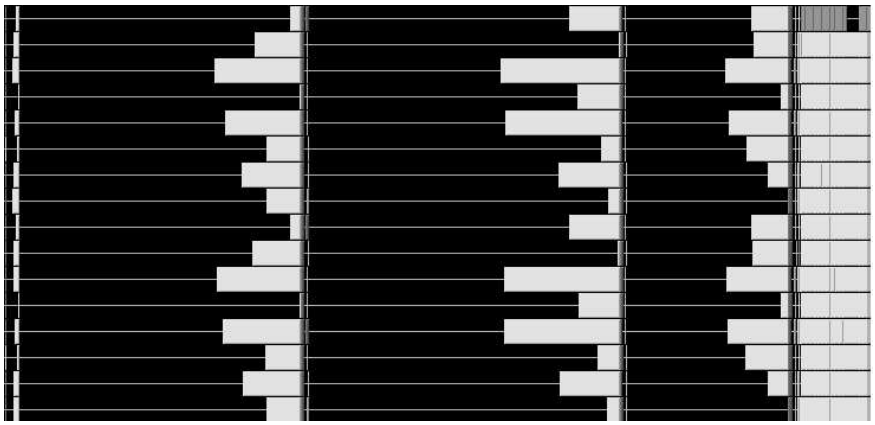

Fig. 5. Visualization of the MPI overhead for k-point parallelization over 16 k-points on 16 Processors. Each MPI operation corresponds to a different shade of gray: `MPI_Barrier` white, `MPI_Bcast` light gray, and `MPI_Allreduce` dark gray. The synchronization overhead is much bigger than in Fig. 4. This is due to the occuring process skew.

is much bigger as for 8 processors due to the barrier synchronization.

Communication Analysis II: Parallelization over Bands

The communication diagram for 8 processors and a calculation with 4 k-points and `nbdblock=2` is shown in Fig. 6. The main MPI operations besides

the `MPI_Barrier` and `MPI_Allreduce` are `MPI_Send` and `MPI_Recv` in this scenario. These operations are called frequently and show a master-slave principle where the block specific data is collected at a master for each block and is processed. The MPI-overhead is significantly higher than for the pure k-point parallelization and the overall performance is heavily dependent on the network performance. Thus the results for band parallelization are rather bad for the cluster equipped with Fast Ethernet, while the results with Myrinet are good.

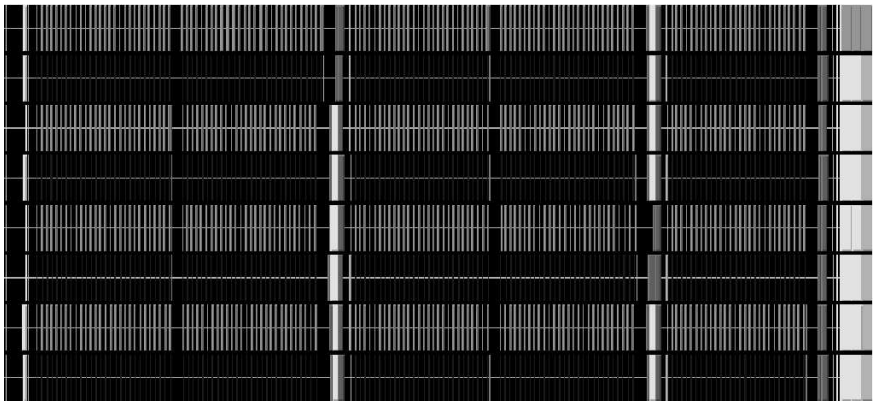

Fig. 6. Visualization of the MPI overhead for k-point and band parallelization over 4 k-points and 2 bands per block on 8 processors. Each MPI operation corresponds to a different shade of gray: `MPI_Barrier` white, `MPI_Bcast` light gray, and `MPI_Allreduce` dark gray. This figure shows the fine grained parallelism for band parallelisation. The overhead is visible but not dominationg the execution time.

The diagram for two k-points calculated on 8 processors is shown in Fig. 7. This shows that the communication overhead outweighs the calculation and the parallelization is rendered senseless. Nearly the whole application runtime is overhead (all but black regions), mainly `MPI_Bcast` and `MPI_Barrier`. Thus, even allowing for the reasons of convergence, mentioned in Sect. 1.2, the band parallelization is effectively limited to cases with a reasonable MPI overhead, i.e. 4 bands per block on this system.

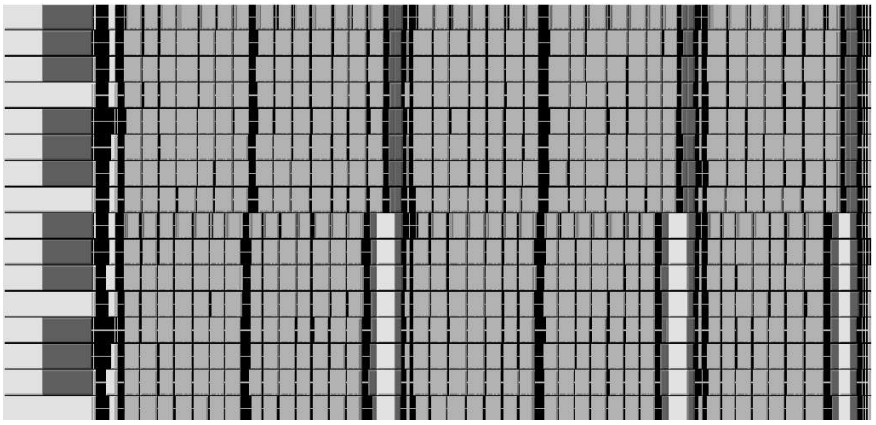

Fig. 7. Visualization of the MPI overhead for k-point and band parallelization over 2 k-points and 8 bands per block on 16 processors. Each MPI operation corresponds to a different shade of gray: `MPI_Barrier` white, `MPI_Bcast` light gray, and `MPI_Allreduce` dark gray. The overhead is clearly dominating the execution and the parallelization is rendered senseless.

4 Conclusions

We have shown that the performance of the application ABINIT on a cluster system depends on different factors, such as compiler and communication network. Other factors which are usually crucial such as different implementations of mathematical functions are less important because the math libraries are rarely used in the critical path for our measurements. The choice of the compiler can decrease the runtime by almost 25%. Note that the promising feature of auto-parallelization is counterproductive. The different math libraries differ in less than 1% of the running time. The influence on the interconnect and parallelization technique is also significant. The embarassingly parallel k-point parallelization hardly needs any communication and is thus almost independent of the communication network. The scalability is limited to 8 on our cluster system due to operating system effects, which introduce process skew during each round. The scalability on the Opteron system is not limited. Thus, in principle this implementation in ABINIT is ideal for small systems demanding a lot of k-points, e.g. metals. For systems demanding large supercells, the communication wise more demanding band parallelization becomes attractive. However, the use of this implementation is only advantageous if a fast interconnect can be used for communication. Fast Ethernet is not suitable for this task.

5 Acknowledgements

The authors would like to thank X. Gonze for helpful comments. R.J. acknowledges the HLRS for a free trial account on the Cray Opteron Cluster strider.

References

1. M. Born and J.R. Oppenheimer. Zur Quantentheorie der Molekeln. *Ann. Physik*, 84:457, 1927.
2. P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864, 1964.
3. W. Kohn and L.J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev*, 140:A1133, 1965.
4. R. M. Martin. *Electronic Structure*. Cambridge University Press, Cambridge, UK, 2004.
5. M.C. Payne, M.P. Teter, D.C. Allan, T.A. Arias, and J.D. Joannopoulos. Iterative minimization techniques for ab initio total-energy calculations - molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64:1045, 1992.
6. J. Nocedal. Theory of algorithms for unconstrained optimization. *Acta Num.*, page 199, 1991.
7. X. Gonze, G.-M. Rignanese, M. Verstraete, J.-M. Beuken, Y. Pouillon, R. Caracas, F. Jollet, M. Torrent, G. Zerah, M. Mikami, P. Ghosez, M. Veithen, J.-Y. Raty, V. Olevano, F. Bruneval, L. Reining, R. Godby, G. Onida, D.R. Hamann, and D.C. Allan. A brief introduction to the ABINIT software package. *Z. Kristallogr.*, 220:558, 2005.
8. http://www.abinit.org/.
9. V. Eyert. A comparative study on methods for convergence acceleration of iterative vector sequences. *J.Comp.Phys.*, 124:271, 1995.
10. X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, Ph. Ghosez, J.-Y. Raty, and D.C. Allan. First-principles computation of material properties: the ABINIT software project. *Comp. Mat. Sci.*, 25:478, 2002.
11. A.D. Malony, V. Mertsiotakis, and A. Quick. Automatic scalability analysis of parallel programs based on modeling techniques. In *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools*, pages 139–158, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
12. R.J. Block, S. Sarukkai, and P. Mehra. Automated performance prediction of message-passing parallel programs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995.
13. M. Courson, A. Mink, G. Marcais, and B. Traverse. An automated benchmarking toolset. In *HPCN Europe*, pages 497–506, 2000.
14. X. Cai, A. M. Bruaset, H. P. Langtangen, G. T. Lines, K. Samuelsson, W. Shen, A. Tveito, and G. Zumbusch. Performance modeling of pde solvers. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations*, volume 33 of *Lecture Notes in Computational Science and Engineering*, chapter 9, pages 361–400. Springer, Berlin, Germany, 2003.

15. A. Grama, A. Gupta, E. Han, and V. Kumar. Parallel algorithm scalability issues in petaflops architectures, 2000.
16. G. Luecke, B. Raffin, and J. Coyle. Comparing the communication performance and scalability of a linux and an nt cluster of pcs.
17. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996.
18. H.J. Monkhorst and J.D. Pack. Special points for brillouin-zone integrations. *Phys. Rev. B*, 13:5188, 1976.
19. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.