SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE

PRACE-2IP

PRACE Second Implementation Project

Grant Agreement Number: RI-283493

D8.1.4
Plan for Community Code Refactoring
Final

Version: 1.0
Author(s): Claudio Gheller, Will Sawyer (CSCS)
Date: 24.02.2012

4 Material Science

4.1 ABINIT

4.1.1 Overview

ABINIT [34] is a package, available under the GNU General Public Licence (GPL), whose main program allows one to find from first principles the total energy, charge density, electronic structure and miscellaneous properties of systems made of electrons and nuclei (molecules and periodic solids) using pseudo-potentials and a plane-wave or wavelet basis. The basic theories implemented in ABINIT are Density-Functional Theory (DFT), Density-Functional Perturbation Theory (DFPT), Many-Body Perturbation Theory (MBPT: the GW approximation and Bethe-Salpeter equation), and Time-Dependent Density Functional Theory (TDDFT).

Historically, ABINIT uses plane-waves to describe the electronic wave functions; in recent years, a development of wave functions utilising a wavelet basis has been introduced (for the ground state calculations). The implementation of wavelets has been achieved in a library named "BigDFT". This library is an inseparable part of the ABINIT project.

ABINIT parallelisation is performed using the *MPI library* for the current stable version. In the last version, several time-consuming code sections of the ground-state part have been ported to GPU (beta stage); also several sections of the excited-state part have been parallelised using the OpenMP shared memory scheme.

The “Performance analysis” [3] and “Performance improvements exploration” [4] phases were divided in three sections: **1-**ground-state calculations using plane waves, **2-**ground-state calculations using wavelets, **3-**Excited states calculations. In this “Plan for code refactoring” phase, we have added a new section: **4-**Linear response calculations.

4.1.2 Plan for code refactoring: ground–state calculations using plane waves

During the performance analysis phase [3], we identified the critical parts of the code; then during the performance improvements exploration phase [4] we investigated three promising approaches that could substantially improve the parallelism in ABINIT:

- Introduce activation (choice) thresholds for the use of a parallel eigensolver
- Improve load balancing (“band” and “plane wave” distributions)
- Introduce shared memory parallelism level using OpenMP

These three approaches have been tested by writing prototype codes or by modifying some selected small sections of ABINIT. We report here the results of these tests and collate obtained information to build a simplified performance model for the ground-state part.

In the following, all mentioned tests have been performed on *TGCC-CURIE PRACE* French supercomputer (using large or hybrid CPU/GPU nodes). They all have been performed on a 107 gold atoms simulation cell (a gold vacancy in a 108 atoms cell).

1- Parallel eigensolver activation thresholds

Concerning the introduction of activation thresholds for selected code sections, we have tested the procedure for the eigenvalue problem in the (small) wave functions subspace. When executing this code section without distributing the work load, it becomes rapidly time consuming. We have tested the possibility to do this diagonalisation 1-using *MPI* (*ScaLAPACK*), 2-using GPU (*MAGMA*). In both cases we find that it is not profitable to parallelise if the work load is too small (i.e., if the studied physical system is too small).

Figure 22 shows how the CPU time needed to diagonalise a square matrix of size 512 (resp. 1024, 2048) evolves with the number of CPU cores using different versions of *ScaLAPACK*. We can deduce from these results that, if the matrix is too small, the use of *ScaLAPACK* is not profitable : the yellow curve (matrix of size 512) shows an increase above 16 cores although it is not the case fort the blue curve(matrix of size 2048); it also appears that, even if the code runs over a large number of CPU cores, *ScaLAPACK* should be called (several times) for a smaller number of cores (between 15 and 20).

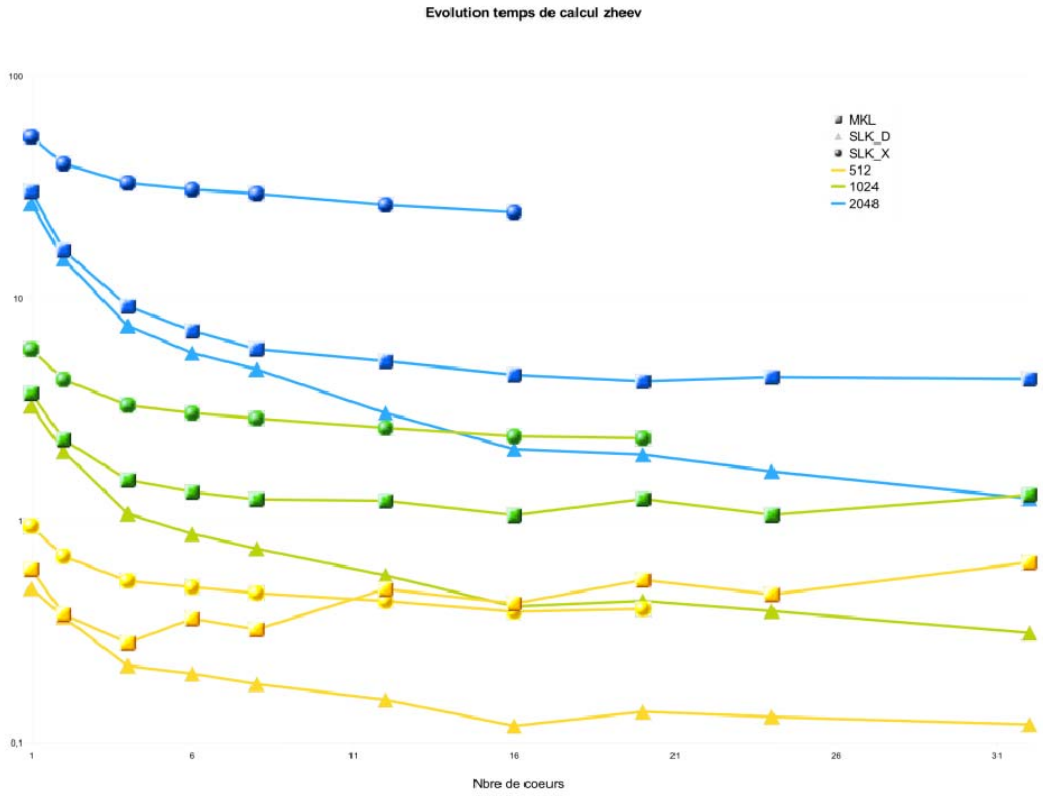


Figure 22: CPU time (sec.) per process needed by a single call to ZHEEV *ScaLAPACK* routine with respect to number of MPI process for several sizes of square matrix (512, 1024, 2048) and several *ScaLAPACK* implementations

As concerns the use of the *MAGMA* GPU eigensolver, we find that (on the *TGCC-CURIE* supercomputer) we do not take advantage of the Graphics Processing Unit if the matrix has a size lower than 128 (double precision). It is obviously related to the communication/computation ratio.

For the 107 gold atoms test case executed on *TGCC-CURIE* we can deduce a *ScaLAPACK* activation threshold and a maximum number of CPU cores usable for *ScaLAPACK* as well as a *MAGMA* activation threshold. Of course, these activation thresholds have to be adapted to the computer architecture. The goal is to write some small threshold automatic determination

routines that could be called at the beginning of each run (only if *ScaLAPACK* and/or *MAGMA* use is requested).

2- Bands and plane waves charge balancing

As written in deliverable D8.1.3 [4], we also have tested the possibility to improve the load balancing (which is sometimes highly unfavourable: some cores can have a load 1.75 times larger than others).

We found that bands can be much better distributed among processors than in the current ABINIT implementation. Only a minor modification at the level of the distribution routine is required, and we plan to do it. With this modification each CPU core will treat exactly the same number of bands as the others or only one band more. The expected improvement depends on the physical system under investigation.

We also tested methods to better distribute the plane wave vectors. The charge imbalance (in the current version) is due to the fact that plane wave vectors have to be distributed according to one of their coordinates (*x*, *y* or *z*) to be usable by the FFT routines. If one cuts the plane wave space among the *z*-axis and apply the plane wave selection rule (*norm<cut-off^2*) one necessarily obtains a different number of plane waves per *z* layer. We have written a prototype code to test different plane wave distributions in a parallel FFT call (using several FFT versions). It appears that the distribution among one axis is mandatory (parallel FFTs cannot run without this distribution), but it also appears that we could have different *z*-layer thickness per process. By adopting the latter solution it may be possible to balance the work load.

We plan to introduce such a variable *z*-layer thickness for the FFT distribution in ABINIT. But this necessarily implies a full refactoring of each code section where the plane wave vectors are distributed, and this affects a lot of routines in addition to the FFTs.

3- Hybrid MPI/OpenMP parallelisation

In the present ABINIT official version, only distributed-memory parallelism is used for electronic ground-state calculations. After having distributed the work load over *MPI* processes several code sections remain time consuming (i.e., linear algebra in iterative eigensolver *LOBPCG*). Apart from diagonalisation, two sections of code appear to be bottlenecks; they are mainly due to communications (*MPI_REDUCE* on the full band/plane-wave communicator). This can be seen in the Figure 23 below (the corresponding sections are yellow and red). These two sections could take benefit from a *shared-memory* parallelisation scheme. Of course, communications will not disappear: 1-intra-node bandwidth will necessarily be a limitation; 2-distribution of work load will not be possible over a large number of CPU cores (as in the *MPI* case); we necessarily will have to use a hybrid scheme, mixing *MPI* and *OpenMP* parallelism.

We have tested the feasibility of using an *OpenMP* version of the matrix orthogonalisation on a prototype code. For that purpose we have used the multi-threaded version of the Intel MKL library. On our architecture (*TGCC CURIE* supercomputer) we have found that a speedup of a factor 10 could be reached using a 16-cores node per orthogonalisation/diagonalisation.

We plan to introduce *OpenMP* directives in the whole ground-state part of ABINIT, especially in the linear algebra and matrix algebra sections of the parallel eigensolver.

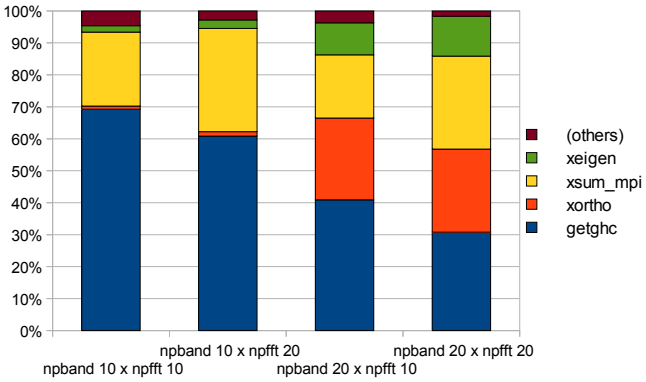


Figure 23: CPU time distribution for the ABINIT parallel eigensolver (standard test case: 107 gold atoms) with respect to the number of "band" cores (npband) and "FFT" cores (npfft).

4- Simplified performance model

We propose here a performance model for a typical *ground-state* calculation with ABINIT. This model includes *MPI* and *OpenMP* parallelisation and deliberately excludes the use of GPU (for simplification purpose). As it is very difficult to conceive a performance model valid for all physical systems, we choose to build such a model for our standard test case (107 gold atoms). This is a typical test case (typical simulation cell size, chemical specie from the middle of the periodic table, non-negligible forces …).

Most of our hypotheses will be drawn from the following strong scaling graph (Figure 24) already presented in 8.1.2 and 8.1.3 deliverables.

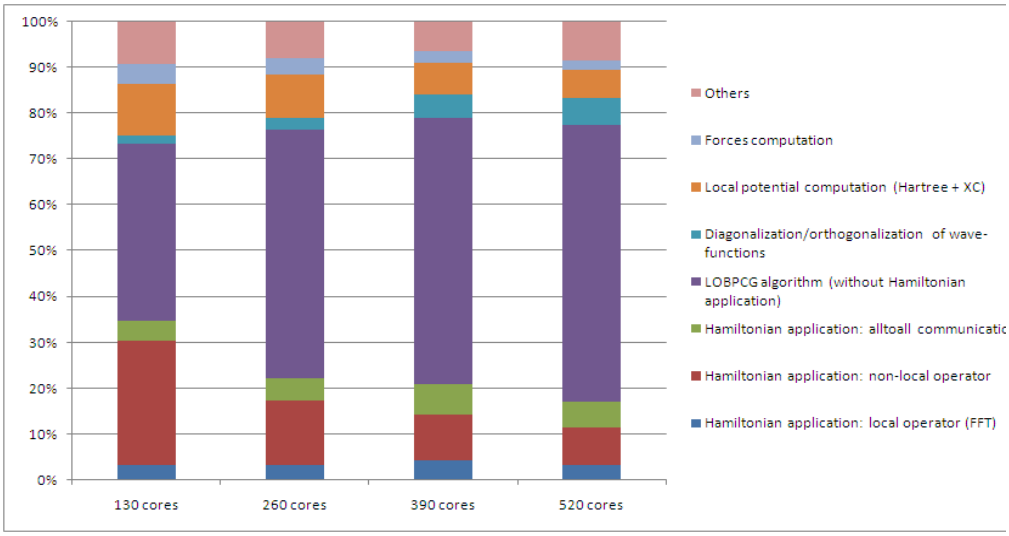


Figure 24: Repartition of CPU time in ABINIT routines varying the number of CPU cores.

For the following we define:

N_{MPI}	= number of *MPI* cores
$T_{TOT,1}$, $T_{TOT,MPI}$	= total wall clock time using 1 (or N_{MPI}) *MPI* processes
$T_{EIGEN,1}$, $T_{EIGEN,MPI}$	= wall clock time using 1 (or N_{MPI}) processes spent in eigensolver (violet in the graph)
$T_{HAM,1}$, $T_{HAM,MPI}$	= wall clock time using 1 (or N_{MPI}) processes spent in Hamiltonian application (blue, green and red in the graph)

$T_{OTHER,1}$, $T_{OTHER,MPI}$ = wall clock time using 1 (or N_{MPI}) processes spent in other routines (~15% of total CPU time, according to above graph)

$S_{OMP,16}$ = *OpenMP* speedup of the kernel on 16 cores (~10)

$S_{EIGEN-MPI}$ = *MPI* speedup of the eigensolver on N_{MPI} cores (0.8 N_{MPI})

$S_{ZEEV-MPI}$ = *MPI* speedup of the ZEEV *ScaLAPACK* routine on N_{MPI} cores

As previously mentioned we can estimate $T_{ZEEV-MPI} \approx 20$ if $N_{MPI} > 20$

Let us consider that we are running ABINIT over a number of *MPI* processes in the range ensuring a linear scaling of the Hamiltonian application (N_{MPI} <500). Let us also consider that the sizes of matrixes are large enough to be above the *ScaLAPACK* activation threshold.

As shown by the previous graph we can estimate the total CPU time needed for a typical run:

$$T_{TOT,MPI} \approx T_{OTHER,MPI} + T_{EIGEN,MPI} + T_{HAM,MPI} \approx (T_{EIGEN,MPI} + T_{HAM,MPI})/0.85$$

As assumed, the Hamiltonian application scales linearly:

$$T_{HAM,MPI} \approx T_{HAM,1} / N_{MPI}$$

Meaning that:

$$T_{TOT,MPI} \approx T_{TOT,1}/N_{MPI} + (T_{EIGEN,MPI} - T_{EIGEN,1}/N_{MPI})/0.85$$

Now, for the eigensolver, let us divide the time in 3 parts (diagonalisation, orthogonalisation and communications) according to the Figure shown in the section 3-:

$$T_{EIGEN,MPI} \approx T_{DIAGO,MPI} + (T_{ORTHO,MPI} + T_{COMM,MPI})$$

The last two times are communications times.

- For the diagonalisation, we can use *ScaLAPACK* (*MPI*) or a multithreaded (*openMP*) version of LAPACK. Assuming that the diagonalization is exclusively made by the processors of a single node, we can have (*ScaLAPACK*):

  $$T_{DIAGO,MPI} = T_{DIAGO,1}/S_{ZEEV-MPI}/S_{EIGEN-MPI}$$

  Or (*openMP*):

  $$T_{DIAGO,MPI} = T_{DIAGO,1}/S_{OMP,16}/S_{EIGEN-MPI}$$

- For the communications, it is difficult to evaluate exactly what could be the performance improvement induced by the use of *openMP*. If we look carefully inside the LOBPCG algorithm we can roughly estimate the amount of communications as inversely proportional to the size of the eigenvalue problem in wave functions subspace. In other words, if the wave functions subspaces (blocks) are large they do not need to be orthogonalized to each other.

If we assume that the diagonalization step can be reduced by one or two order of magnitude using Scalapack or openMP, we can choose to diagonalize blocks of larger size (one single block of maximum size is possible) and thus make the communications disappear. In conclusion, a speed-up of diagonalization as described previously will probably suffice to significantly accelerate the code.

5- Validation procedure

The ABINIT package comes with a collection of automatic tests to verify the correctness of the results (~450 tests). ABINIT has to give exactly the same results according to the sensitivity (in terms of digits) defined for each test.

Then the 107 gold atoms test ("standard" test) will be used to measure the performance improvement.

4.1.3 Plan for code refactoring: ground–state calculations using wavelets (BigDFT)

The main part of the plan is to improve the optimisation of *BigDFT* running it on one node. The idea is to build an automatic generation of code to test different strategies of optimisation for CPU first and then GPU.

BigDFT has more than 25 kernels to optimise and has no hot-spot operations. Optimising these kernels becomes problematic with the increasing numbers of new architectures. Another problem is that optimising a kernel on one core is not the right solution, because the main limitation is the *bandwidth* memory. We need to optimise kernels using all cores of one node or one socket.

1- Hybrid MPI/OpenMP test

The test is the ground-state calculation of a cluster made of 80 bore atoms done on CCRT-Titane (CCRT French centre, Intel processors) on one node (8 cores). There are 120 orbitals. This means that we can use 120 *MPI* processes as an upper limit. It is a rather small system. Figure 25 illustrates the efficiency and the speedup with respect to the number of *MPI* processes and *threads*. We can see that using 2 *threads* slightly changes the efficiency and is almost equivalent to using 2 times more *MPI* processes.

The different convolutions – which are the basic operations with wavelets functions – represent the main part of the calculation.

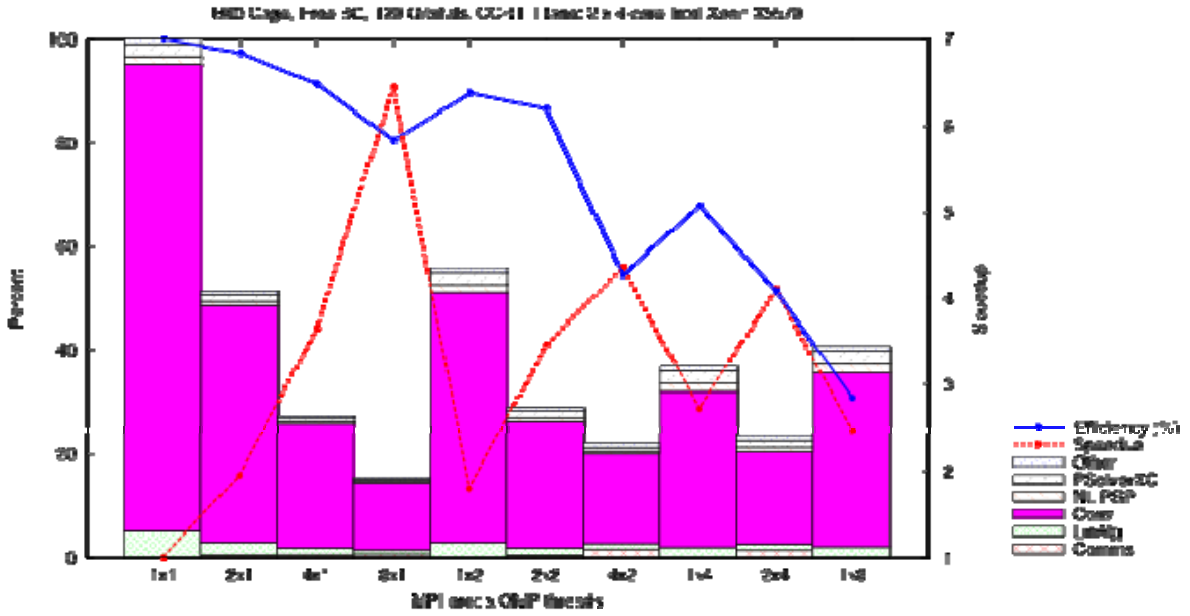


Figure 25: BigDFT on Titane-CCRT supercomputer: efficiency (solid blue curve) and speedup (dashed red line) with respect to the number of *MPI* processes and *threads*.

Now if we use a computer using *AMD* cores (CSCS-Palu, see Figure 26 below), then the results are different: the efficiency strongly decreases when all cores are used (24 cores). But we can use more threads (up to 4) with a small decrease of the efficiency.

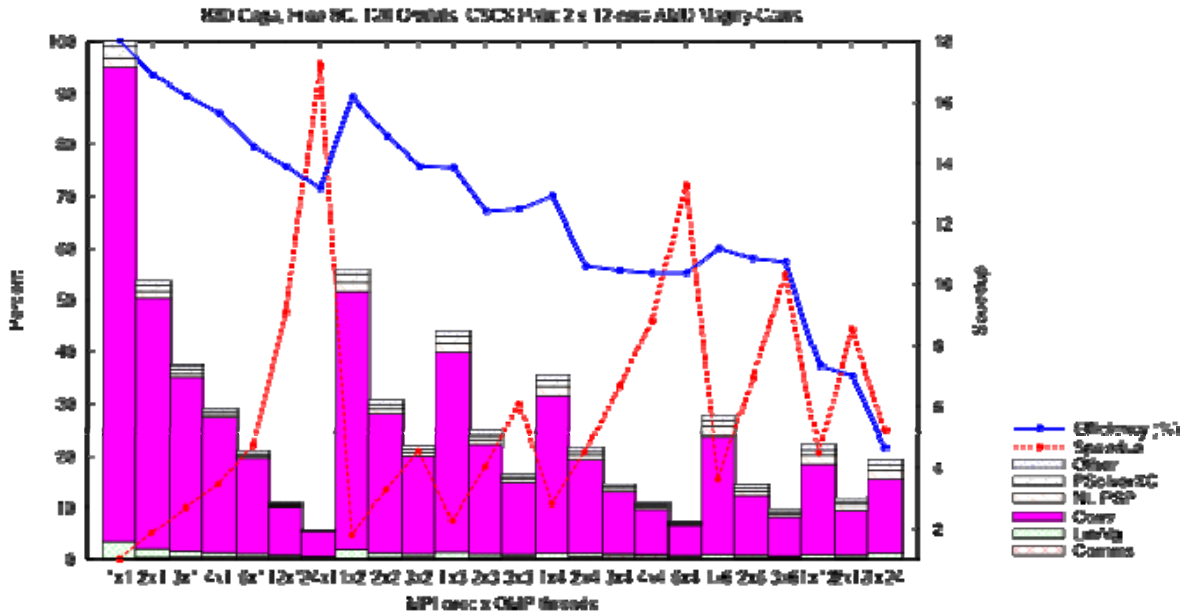


Figure 26: BigDFT on Palu-CSCS supercomputer: efficiency (solid blue curve) and speedup (dashed red line) with respect to the number of *MPI* processes and *threads*.

In Figure 27, we compare one node of both computers (24 AMD cores and and 8 Intel cores) using the best MI+threads configuration (i.e. only MPI processes).. We can see that 16 AMD cores are equivalent to 8 Intel cores.

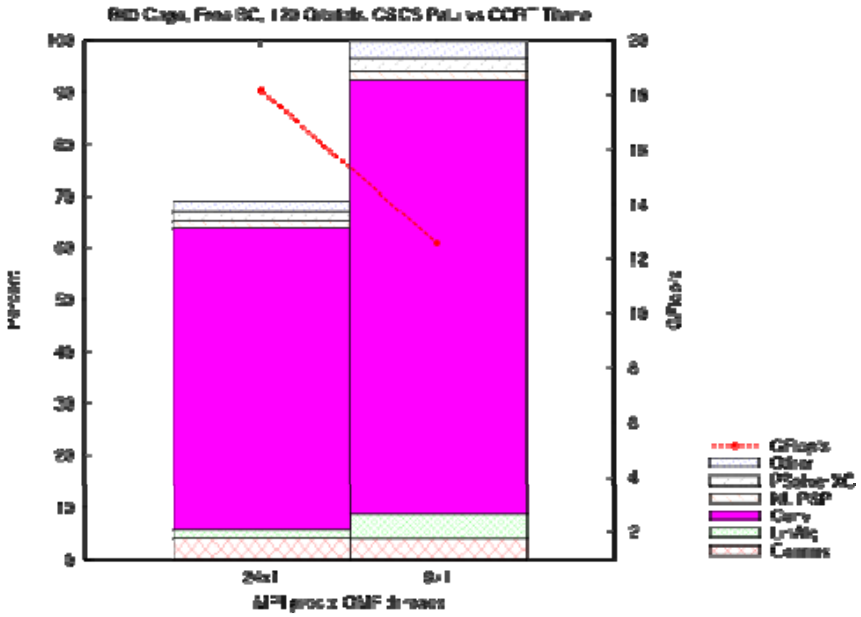


Figure 27: Computer of time spent in convolutions on two different architectures (Titane-CCRT and Palu-CSCS)

2- Hybrid *MPI/OpenMP* and GPU test

If we combine *OpenMP* + *MPI* and GPU usage (*OpenCL* or *CUDA*) then we have different behaviours (see Figure 28). The main conclusion is that the use of GPU always gives a speedup (sometimes small). The best performances are obtained using *OpenCL*, *CUBLAS* for linear algebra and *MPI* processes. Comparing to *CPU+mkl+MPI*, we can have a speedup of 2 by adding only one GPU.

All these tests are very dependent on the considered system, namely the number of atoms and boundary conditions. It is possible to have a speedup of more than 10 when considering periodic boundary conditions with k points.

In conclusion, it is really important to have optimised routines.

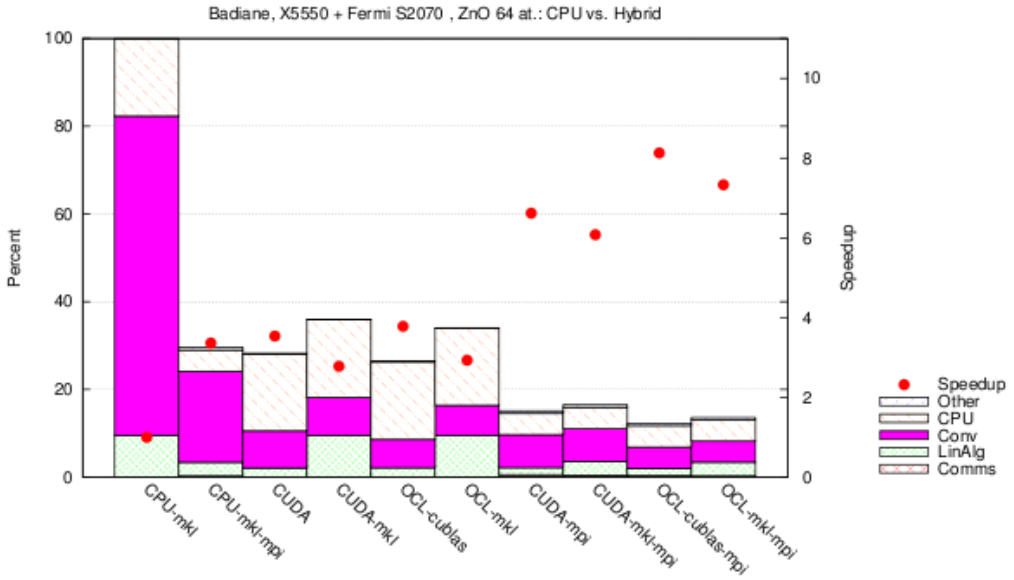


Figure 28: Speedup of BigDFT using GPUs

3- Automatic code generation

In order to simplify the maintenance of the optimised kernels, we tried to find a solution to automatically generate routines with different optimisation strategies: the number of unrolling loops, different patterns for the memory accesses, etc..

We have tested a first version based on the script language Ruby to generate different strategies of optimization. This is an elegant and easy to use language; our first results are promising. The main advantage is the ease to make changes and add new features. On this first example we prove the feasibility of the implementation.

In conclusion, the plan for *BigDFT* refactoring is to implement a complete solution of automatic code generation.

4.1.4 Plan for code refactoring: excited states calculations

The performance analysis done in [4] allowed us to identify the most critical parts of the code. On the basis of these results, we proposed four different modifications that should substantially improve the scalability of the GW code:

- Implementation of a hybrid *MPI-OpenMP* approach
- Use of *ScaLAPACK* routines for the inversion of the dielectric matrix
- Implementation of a new *MPI* distribution scheme of the orbitals in order to improve the load balancing during the computation of the exchange part of the self-energy
- Use of *MPI-IO* routines to read the orbitals and the screening matrix from file

In order to assess the efficiency and the feasibility of these four different approaches, we have developed prototype codes that have been benchmarked using realistic parameters.

Subsequently we report the results of these tests, including an estimate of the parallel efficiency of the new implementation. Finally, a simplified performance model, whose parameters are estimated from the results of these preliminary tests, is presented and discussed.

1- Hybrid *OpenMP-MPI* implementation

We have generalised the *FFTW3* routines used in ABINIT so that the transforms can be executed in parallel with *OpenMP* (OMP) threads.

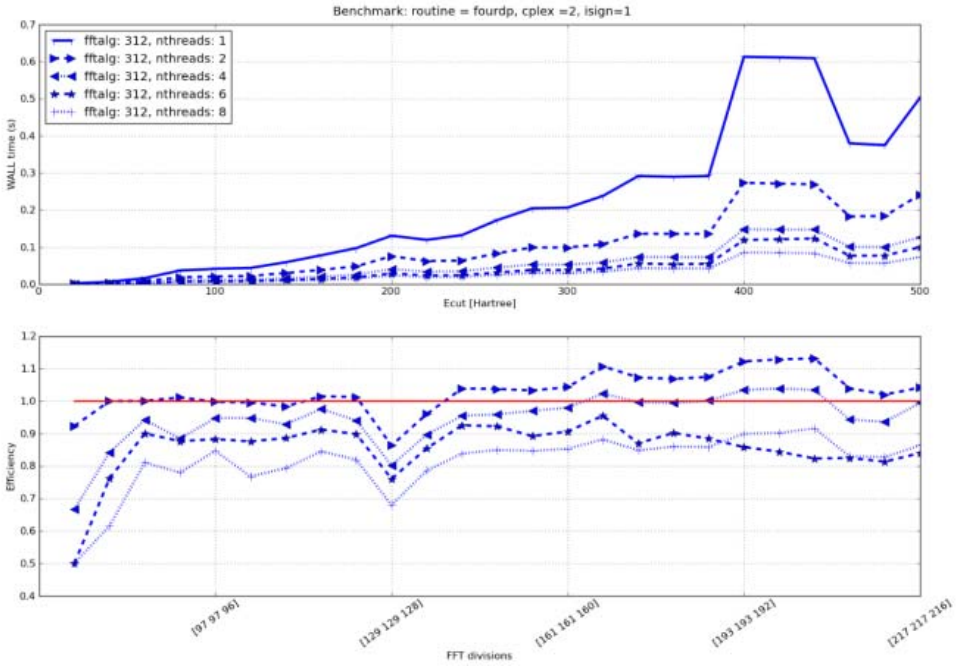
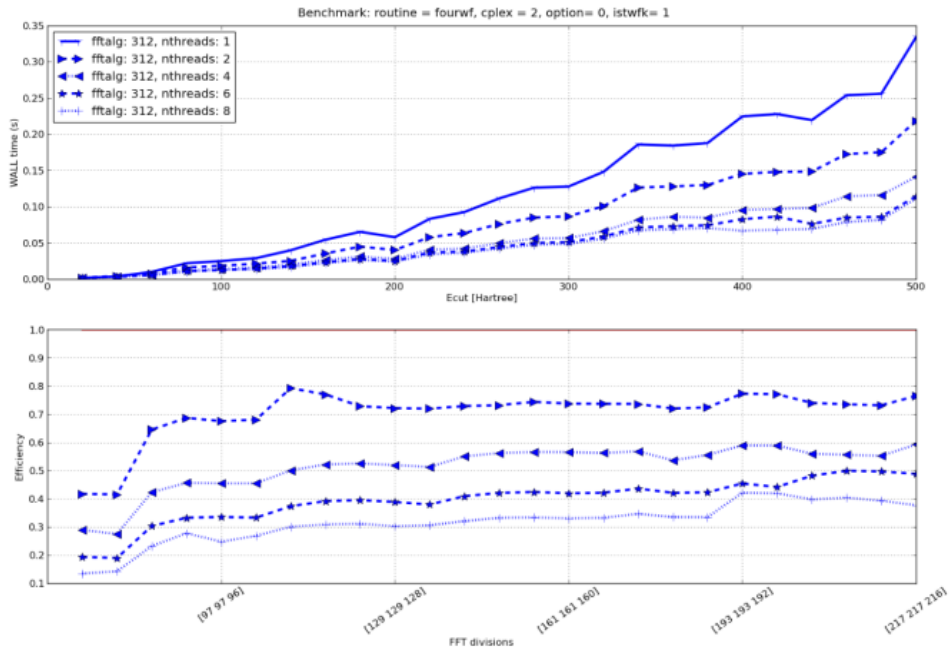


Figure 29 shows the parallel efficiency of the new implementation for different number of threads employed.

The benchmarks are performed using the *FFTW3* interface provided by the *MKL* library.



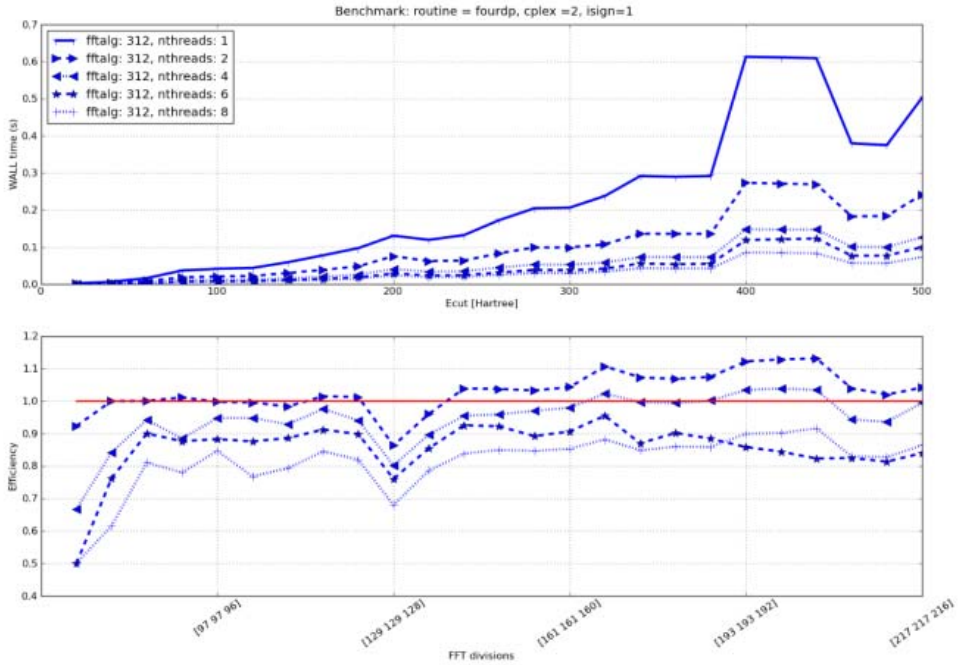


Figure 29: Parallel efficiency of new FFTW implementation in ABINIT-GW using multithreaded FFTW3 library

The threaded version of *fourdp* shows a very good parallel efficiency, whereas the results obtained with *fourwf* are somehow less satisfying. A point worth noting is that *fourwf* transforms the wave functions by employing a pruned FFT to reduce the number of 1D-sub-transforms (about 1/8 of the input Fourier components are non-zero), whereas *fourdp* performs 3D FFT transforms using the standard algorithm for "dense" arrays. For this reason, the sequential version of *fourwf* is faster than *fourdp* but, according to our results, reducing arithmetic cost does not necessarily imply better parallel efficiency when *OMP* is used.

The inefficiency of the parallel version of *fourwf* calls for a better understanding of the algorithms employed by *MKL* to parallelise multiple 1D-sub-transforms. In particular, we suspect the presence of false sharing when multiple FFT sub-transforms along the *y*- and the *z*-axis are distributed among the threads. We are presently testing different OMP algorithms in order to improve the efficiency of the threaded version of *fourwf*.

The other kernels that have been parallelised with OMP instructions are:

- The computation of the polarisability with the Hilbert transform method
- The evaluation of the self-energy corrections with the contour deformation technique

These two algorithms are well suited for the *OMP* paradigm since they both involve very CPU-intensive loops over (*nomega*) frequencies and/or the (*npweps*) plane waves used to describe the dielectric function. Figure 30 illustrates the parallel efficiency of these two kernels as function of the number of threads.

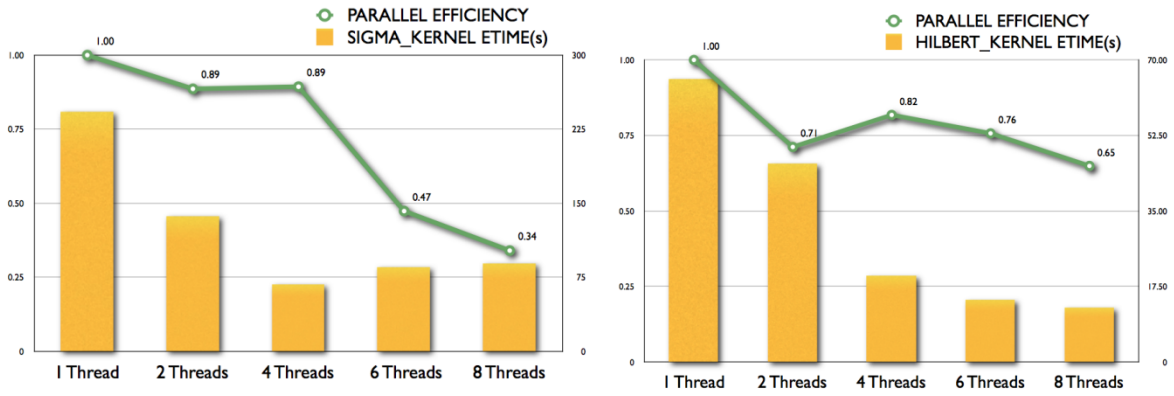


Figure 30: Parallel efficiency polarisability and self-energy kernels in ABINIT-GW using *openMP*

For these tests, we employed *nomega*=70 and *npweps*=531. Note that the parallel efficiency improves when larger values of *npweps* are used (not shown).

2- Computation of the inverse dielectric matrix with ScaLAPACK

The CPU time needed for the inversion of the dielectric matrix scales as *npweps***3, hence this section of the code represents an important bottleneck in the case of systems with large unit cell. For this reason, a new kernel in which the inversion is done in parallel with *ScaLAPACK* routines has been implemented and benchmarked. Figure 31 shows the performance of the new implementation for different dimensions of the dielectric matrix.

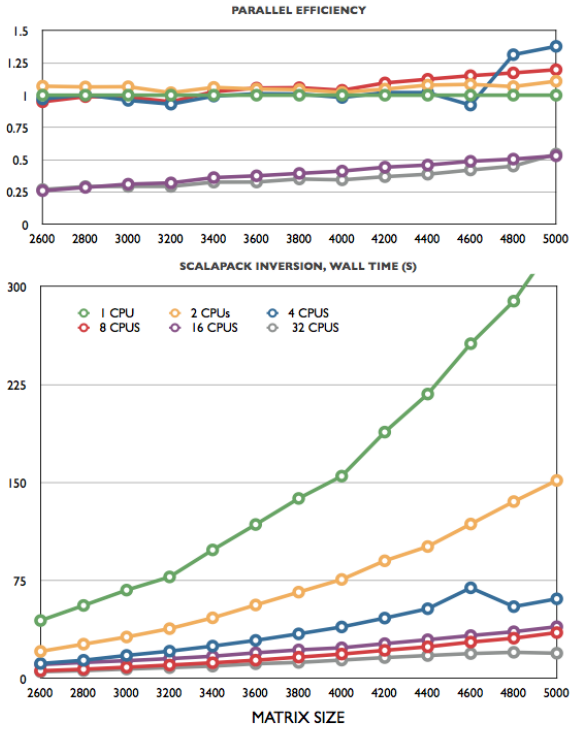


Figure 31: Performance of the new implementation of the inversion of the dieletric matrix using *ScaLAPACK*

As already stressed in the ground-state section, the use of *ScaLAPACK* is beneficial only when the size of the matrix is larger than a threshold value that strongly depends on the architecture, the network, and the library used. Moreover, as shown in the figure, a brute force increase of the number of processors does not necessarily lead to faster computations. The forthcoming version of the GW code will accept a new input variable that specifies the

maximum number of processors to use for the matrix inversion (see also the discussion in section 5 below).

3- New distribution of the orbitals

A new distribution scheme, in which all the occupied states are stored on each node, has been made available in ABINIT. This approach is more memory demanding than the standard algorithm, but it allows one to achieve an almost optimal distribution of the workload during the computation of the exchange part of the self-energy when the number of processors is larger than the number of occupied bands. Preliminary tests (not shown), performed on a relatively small number of processors (<32), confirmed the better scaling of the new implementation.

4- MPI-IO for the reading of the orbitals.

We have added the possibility of reading the orbitals and the screening function using collective *MPI-IO* routines. The new version has been tested on a relatively small number of processors. Additional tests are needed to assess the scalability of this part for large number of processors.

5- Validation procedure

In order to validate the new FFT kernels, we have written a small tool (*fftprof*) that tests the efficiency and the correctness of the different FFT libraries interfaced with ABINIT (*FFTW3*, *MKL*, *Goedecker*'s library). The results produced by *fftprof* can be plotted with a *Python* script so that the user can select the optimal set of parameters (FFT library, number of threads, cache size) for a given FFT mesh.

The hybrid *OpenMP-MPI* implementation can be validated by running the standard GW tests available in the ABINIT test suite.

The parallel inversion can be validated with the new utility tool *LAPACKPROF* that runs selected *ScaLAPACK* routines for a given number of processors. *lapackprof* checks the correctness of the results and the efficiency of *ScaLAPACK* implementation so that one can find an optimal setup (number of processors, *ScaLAPACK* block size) for a given matrix size.

6-Simplified performance model

In what follows, we present a simplified performance model for a typical GW calculation based on norm-conserving pseudo-potentials and plane waves. The polarisability is evaluated for *NOMEGA* frequencies using the Hilbert transform method while the self-energy matrix elements are computed with the contour deformation technique.

This simplified model includes both the *MPI* and the *OpenMP* parallelisation. Most of our assumptions are derived from the strong scaling analysis presented in 8.1.2 [3] and 8.1.3 [4] deliverables, and from the results for the *OpenMP* version presented in the previous sections.

The total wall time of a sequential GW run (screening + sigma) is approximately given by:

T_TOT = T_FFT + T_GW + T_INV + T_OTHER

Where

T_TOT = total wall-clock time

T_FFT = wall-clock time spent in the FFT routines.

T_GW = wall-clock time spent in the GW kernels

T_INV = wall-clock time needed for inverting the dielectric matrix

T_OTHER = remaining portions of the code (MPI communications, IO, etcetera)

To discuss the scaling of parallel implementation, we introduce the following quantities:

NMPI = number of MPI processes

NOMP = number of OMP cores

NOMEGA = number of frequencies in the screened interaction

We use T(NMPI,NOMP) to denote the total wall-clock time of the code executed with NMPI *MPI* processes and NOMP *threads*. According to this notation, the elapsed time of the sequential code will be indicated by T_TOT(1,1).

Since the orbitals are almost equally distributed among the MPI processes, we have:

$$T\_FFT(NMPI,NOMP) = T\_FFT(1,1) \ / \ (NMPI * \sigma\_FFT\_OMP(NOMP))$$

Where σ_FFT_OMP(NOMP) is the *OpenMP* speedup of the FFT transforms with NOMP cores. According to our tests σ_FFTOMP(NOMP) ~ 0.9 NOMP when *fourdp* is employed.

The pure *MPI* implementation of the GW kernels scales as

$$T\_GW(NMPI,1) = T\_GW(1,1) / NMPI$$

The linear scaling (confirmed by the analysis performed in the precedent deliverables) is due to the fact that the number of calls to the GW kernels is proportional to the number of states that are almost equally distributed among the nodes. For the hybrid *MPI-OpenMP* implementation, one thus obtains:

$$T\_GW(NMPI,NOMP) = T\_GW(1,1)/(NMPI * \sigma\_GWOMP(NOMP))$$

Where σ_GWOMP(NOMP) is the OpenMP speedup of the GW kernels. Our preliminary results indicate that σ_GWOMP(NOMP) ranges between 0.7 NOMP and 0.9 NOMP when NOMP <= 4

The scalability of the algorithm that inverts the dielectric matrix strongly depends on the number of frequencies computed. In the previous implementation, the matrix inversion was distributed over frequencies and then performed in sequential. As a consequence, this section of the code was not scaling anymore when NMPI > NOMEGA. The new version based on *ScaLAPACK* routines presents a much better scaling given by

$$T\_INV(NMPI) = T\_INV(1,1)/(NOMEGA*\sigma\_SLK(NSLK)) \quad (NMPI > NOMEGA)$$

Where σ_SLK(NSLK) is the ScaLAPACK speedup with NSLK processors (the additional level of parallelisation due to OMP has been neglected, for the sake of simplicity). Our tests reveal that, for the typical size of the dielectric matrices used in our applications, σ_SLK(NSLK) ~ NSLK if NSLK < 8.

As stated, T_OTHER includes the wall-clock time spent in the *MPI* sections and in the *IO* routines. The previous tests shown that the GW code is not communication bounded (at least up to NMPI<512) and the introduction of the additional level of parallelism based on OMP will help reduce the number of communications. The use of *MPI-IO* routines should lead to better scaling of the IO sections with respect to the previous implementation based on Fortran IO, in particular when many processors are used.

4.1.5 Plan for code refactoring: linear response calculations

The linear-response part of the ABINIT code plays a specific, but important, role, allowing computing efficiently phonons, electric field responses, etc. However, due to lack of human time, it was not analysed in D8.1.2 [3] or D8.1.3 [4]. Appendix B to the present deliverable fills the gap. It provides a performance analysis, and presents four strategies for the improvement of linear response calculations.

The performance analysis allowed us to identify the most critical parts of the code. For the test case (a barium titanate slab with 29 atoms), it was shown that the most time-consuming parts of the calculation have been parallelised efficiently over k-point and bands. These parts scale linearly, well beyond 256 cores. However, for a smaller number of cores, two bottlenecks appear: the first bottleneck is at the level of the initialisation of the ground state wave functions (reading from file, and spreading the data), the second is at the level of the routines that cannot be parallelised over k-point and bands, typically the treatment of density and potential. Moreover, the amount of memory that is needed for each processor scales as the number of bands times the number of plane waves, that is, quadratically with respect to the size of the system.

On the basis of these results, we propose in Appendix B four different modifications that should substantially improve the scalability of the linear response calculations:

- Remove the IO-related initialisation bottleneck. In particular, use of *MPI-IO* routines to read the ground-state wave functions from file.
- Parallelise several sections that cannot be parallelised over k-point and bands. These sections scale like the number of plane waves and/or the number of FFT points.
- Distribute the ground state wave functions over the bands and/or the plane waves.
- Parallelise over the outer loop on perturbations.

We have made a first assessment of the efficiency and the feasibility of these four different approaches. This work has been started end of November 2011, and will be subject to further refinement. We base our assessment on the analysis of similar approaches implemented in other parts of ABINIT.

1- Remove the IO-related initialisation bottleneck

The initialisation is not satisfactorily parallelised over k-point, and does not take advantage of the band parallelisation. In the ground-state plane-wave part of ABINIT, the reading of wave functions has been parallelised using *MPI-IO* routines. It was observed that this wave function initialisation is very effective, such that the time needed is now negligible with respect to the other parts of the calculation. Although the reading and distribution is more complex in the case of the linear response calculation initialisation, we expect a similar behaviour for the ground-state plane-wave part of ABINIT.

2- Parallelise several sections that are performed sequentially

The operations done in the *fourdp*, *vtorho3* and *vtowfk3* do not have a workload that scales as the number of k-points times the number of bands. Of course, in the sequential mode of linear-response calculations, the associated time is completely negligible. However, as seen in the test, with more than 100 processors, they start to be important.

These sections scale as the number of plane waves (or the number of FFT grid points). They can be parallelised over these quantities. A similar parallelisation has already been done in the ground-state plane-wave as well as the excited state parts of ABINIT. They rely on *OpenMP* or *MPI*. With *OpenMP*, it will be possible, with little coding effort, to decrease the time by a factor of ten (as shown by unitary testing, see section 4.1.4). The *MPI* coding effort would be larger, but is not to be ruled out at this stage.

3- Distribute the ground state wave functions over the bands and/or the plane waves.

Although this task is not impacting the execution time, it will address an important limitation of linear response calculations for larger number of atoms. At present, all the processors treating the same k-point must store a copy of the wave functions for all states for that k-

point. Thus, the memory requirement for one compute core increases with the size of the problem. One should distribute the ground-state wave functions among the processors, and treat the scalar product of ground-state and first-order wavefunctions accordingly. An *OpenMP* solution might be limited, so that *MPI* is to be preferred. The correct analysis of this strategy is to be refined, and actually this must be considered at the same time as the final choice of strategy for task 2 and perhaps even task 1. Again, a similar parallelisation has already been done in the ground-state plane-wave part of ABINIT, and proven effective.

4- Parallelise the outer loop on perturbations

When the bottlenecks addressed by tasks 1-3 will be removed, the possibility to parallelise over perturbations will be open. The number of perturbations can be quite large (on the order of 50 for our test case - however this test case is restricted at present to only one perturbation). The amount of communication is very small. The number of sequential parts with respect to this parallelisation is also very small. However, there is a load balancing problem, described in the appendix. For large systems, non-symmetric (the most time-consuming), more than one order of magnitude improvement of the execution time should be attainable.

5- Simplified performance model

Subsequently we sketch a simplified performance model for the test case presented in Appendix B. We rationalise the available data.

The total wall time of a sequential linear-response (LR) run is approximately given by:

T_TOT = T_INIT + T_WF + T_DENPOT + T_OTHER

Where

T_TOT = total wall-clock time

T_INIT = wall-clock time spent in the initialisation of the run

T_WF = wall-clock time spent in the operations done on the wave functions, after the initialisation

T_DENPOT = wall-clock time spent in the operations done on the density and potential, after the initialisation

T_OTHER = remaining portions of the code

To discuss the scaling of parallel implementation, we introduce the following quantities:

NP_KB = number of MPI processes on which the k-point and band load is distributed

NP_G = number of processes on which the plane waves / FFT load is distributed (could be *OpenMP* or *MPI*)

In the present test case, the communication time is negligible in most sections of the code, except in the initialisation section.

We use T_X(N_KB,N_G) (where X is TOT, INIT, WF or DENPOT), to denote the total wall-clock time of the code executed with NP_KG MPI processes and NP_G MPI processes or OMP threads. According to this notation, the elapsed time of the sequential parts of the code will be indicated by T_X(1,1).

In the present implementation, only the KB parallelisation is used, with:

T_INIT(NP_KB,1) = T_INIT(1,1)* f(NP_KB)

T_WF(NP_KB,1) = T_WF(1,1)/NP_KB

T_DENPOT(NP_KB,1) = T_DENPOT(1,1)

T_OTHER(NP_KB,1) = T_OTHER(1,1)

The time T_WF(1,1) being a very large fraction (more than 99%) of the total time T_TOT(1,1), the speed up can reach about 80 on the test case.

The detrimental factor f(NP_KB) is larger than 1. It is actually close to 1 for a small number of processes NP_KB, until NP_KB increases beyond about 64. The presence of this factor has the aim to indicate roughly the behaviour of the initialisation times, whose scaling is not well characterized at present.

Following strategy 1, that is, using *MPI-IO* to initialize the wave functions, one expect to get rid of the bad scaling behaviour of the communications. The initialisation time can be predicted to change to

T_INIT(NP_KB,1) = T_INIT(1,1)/NP_KB

(without the presence of a detrimental factor f(NP_KB)).

Following strategy 2, the T_DENPOT time can be predicted to change to

T_DENPOT(NP_KB, NP_G) = T_DENPOT(1,1)/ σ_G(NP_G)

Where σ_G(NP_G) is the efficiency of use of the NP_G cores to speed-up the density and potential sections of the code. Note that most of the operations on the density and potentials sections of the code are done in sections in which the wave functions are not present. Thus, without increasing the number of cores, a speed-up of these parts can be realized. In case of *OpenMP*, taking the unitary tests done for the excited state calculations (see section 4.1.4), one can expect a fair speed up, up to 6 or 8, without problem. *MPI* coding will allow more speed-up, but at the expense of a more difficult coding. Still, this would also solve the distribution of array problem.

Explicitly, supposing a maximum number of cores NP, the execution time should be

T_TOT(NP,NP) = T_INIT(1,1)/NP +
T_WF(1,1)/NP +
T_DENPOT(1,1)/ σ_G(NP) +
T_OTHER(1,1)

The first goal of strategy 3 is not to decrease the execution time, but to allow to use more efficiently the memory. Still, this might impact the scaling of different sections of the code.

Finally, the strategy 4 addresses a larger demand, for which the test case is to be modified, by using more than one perturbation. Introducing the number of perturbations N_PERT, the time of a run with N_PERT, compared to the time for only 1 perturbation, will be

T_TOT(NPERT) = T_INIT + N_PERT*(T_WF + T_DENPOT + T_OTHER)

This level of perturbation will allow to use a maximal number of NP_PERT * NP processes. The load balancing of this level of parallelisation is to be addressed, though.

6- Validation procedure

In addition to the test case that has been used to make the performance analysis for PRACE-2IP, there is (a) another test case presented in the ABINIT tutorial on the parallelisation of the linear response section of ABINIT, (b) numerous non-regression tests for the linear response case present in the ABINIT automatic test suite (about 100), and (c) even more non-

regression tests for the other aspects of ABINIT. So, as soon as the modifications of the code are committed to the ABINIT worldwide repository, they will be tested and validated.

4.2 Quantum ESPRESSO

4.2.1 Introduction

Quantum ESPRESSO is an integrated suite of computer codes based on density-functional theory, plane waves, and pseudo-potentials. The acronym ESPRESSO stands for opEn Source Package for Research in Electronic Structure, Simulation, and Optimisation.

Two are the main goals of the project: 1) to enable state-of-the-art materials simulations, and 2) to foster methodological innovation in the field of electronic structure and simulations by providing and integrating highly efficient, robust, and user-friendly open source packages containing most recent developments in the field.

The Quantum ESPRESSO distribution offers users a highly portable and uniform installation environment. The web interface, qe-forge, provides to potential developers an integrated development environment, which blurs the line separating development and production codes and engages and nurtures developers by encouraging their software contributions.

Quantum ESPRESSO is freely available under the terms of the GNU General Public License (GPL).

4.2.2 Target a general refactoring of the suite

Quantum ESPRESSO is actually structured as a suite of many packages, each of them is devoted to a particular kind of calculation. The basic are the two DFT engines, PWscf (plane-wave DFT self-consistency calculations) and CP (Car-Parrinello molecular dynamics). Inside the suite different levels of dependencies exist. For instance, some packages such as PHonon, TDDFPT or GIPAW are linked to the same computational kernels used by PWscf and CP, while others, such as PLUMED or YAMBO not.

The overall structure is already organised to take advantage of Fortran 90 modularisation features, and developers already took care of avoiding the replication of similar portions of code in different parts of the suite. Nonetheless, a recent developers meeting[1] with representative computational scientists from CINECA and ICHEC came to the conclusion that it is still necessary to work on this direction to functionally refactoring the deepest part of the distribution.

A refactorisation work should be performed in order to rewrite some of the most used subroutines, belonging to the DFT engine PWscf and the linear response package PHonon, as low level libraries. The creation of such libraries would give a higher level of modularity that could help to maintain most of the packages and to sustain further developments.

A particular case of refactoring needs is the proper implementation of hybrid functionals. The use of Hybrid functionals, i.e., inclusion of a portion of exact-exchange (EXX) inside traditional functionals, is nowadays the better compromise between physical meaning and computational cost. The user community is, indeed, strongly pushing in this direction. At this moment, the possibility of adding an exact-exchange fraction is already present in some packages, PWscf and CP, for some specific calculations. However, it is absolutely necessary to re-engineer and extend it, as many of the advanced features are only available with traditional functionals.

[1] Quantum Espresso developers workshop, Trieste, 24 January 2012

Appendix B. Description of the linear-response methodology of ABINIT, and performance analysis.

B.1 Motivation

The ABINIT code is one of the ETSF codes involved in the PRACE-2IP project from the very beginning. In previous deliverables, a global description of ABINIT, as well as specific descriptions and performance analysis were presented, for the following major methodologies of ABINIT:

- Ground state / plane waves
- Ground state / wavelets
- Excited states (GW calculations)

The linear-response methodology of ABINIT is also quite important. It implements the *Density Functional Perturbation Theory*, for phonon calculations and responses to electric field (among others). In D8.1.2, it was already presented as one of the functional units of ABINIT, in the "Global description section" of ABINIT. However, no specific description neither performance analysis was provided in D8.1.2.

The goal of the present appendix is to provide an update of deliverables 8.1.2 and 8.1.3, with description of the linear-response part of ABINIT, the associated performance analysis, and the list of possible improvements.

B.2 Performances of the linear-response part of ABINIT

Description of the example

This test consists in the computation of the response to only one perturbation (one atomic displacement) at wave vector (0.0, 0.375, 0.0) for a 29 atom slab of barium titanate, using density functional perturbation theory (DFPT - the formalism underlying linear response calculations in ABINIT).

A plane wave basis is used, and many technical details of the calculations are quite similar to the ones for the ground state calculations using plane waves, explained in Section 4.1. In particular, the number of plane waves is determined by the cut-off energy *Ecut*. In the test case, it is chosen to be 20 Ha. The FFT grid is (32x32x270). A converged calculation would better use a large cut-off, e.g. 40 Ha, but the present choice is perfectly appropriate to explore the scaling.

The k-point sampling of the *Brillouin* zone is typical of a production run (8x8x1 grid). The symmetry of the system and perturbation (four spatial symmetry operations are present) will allow to decrease this sampling to one quarter of the *Brillouin* zone, e.g., there will be actually 16 k-points to be treated instead of 64. There are 128 bands.

Structure of a DFPT calculation

Before a DFPT calculation can start, a ground-state calculation must be done (separate parallelisation, see section 4.1.2, to generate the zero-order wave functions, Hamiltonian and eigen-energies. These quantities are denoted:

$\psi^{(0)}$, $\hat{H}^{(0)}$, $\varepsilon_{\alpha}^{(0)}$

In a DFPT calculation, one will start by reading these data from a quite large file (about 0.5 GBytes in the chosen test case). This initialisation is independent of the number of perturbations to consider.

Then, one will consider each perturbation, in turn. The number of perturbations to be considered scales as the number of atoms. At maximum it is three times the number of atoms (in our example, 87), but it is usually decreased by a small factor, thanks to the use of symmetries (in our example, the number of irreducible perturbations is 58).

For each perturbation, characterized by a first-order nuclear potential one has to determine the first-order wave functions self-consistently, with the following iteration loop:

$$\rho_{in}^{(1)}(\vec{r}) \Rightarrow \hat{H}^{(1)} \Rightarrow \left|\psi_\alpha^{(1)}\right\rangle \Rightarrow \rho_{out}^{(1)}(\vec{r})$$

$$\left\{ \begin{array}{l} \hat{H}^{(0)} - \varepsilon_\alpha^{(0)} \left|\psi_\alpha^{(1)}\right\rangle = -P_{unocc} \hat{H}^{(1)} \left|\psi_\alpha^{(0)}\right\rangle \\ \quad \text{with} \quad \hat{H}^{(1)} = \hat{V}_{nucl}^{(1)} + \int \left(\frac{1}{|\vec{r}-\vec{r}'|} + \frac{\delta^2 E_{xc}}{\delta\rho(\vec{r})\delta\rho(\vec{r}')} \right) \rho^{(1)}(\vec{r}')\, d\vec{r}' \\ \qquad \rho^{(1)}(\vec{r}) = \sum_\alpha^{occ} \psi_\alpha^{(1)*}(\vec{r})\psi_\alpha^{(0)}(\vec{r}) + \psi_\alpha^{(0)*}(\vec{r})\psi_\alpha^{(1)}(\vec{r}) \\ \quad \text{under constraint} \quad \left\langle \psi_\beta^{(0)} \middle| \psi_\alpha^{(1)} \right\rangle = 0 \end{array} \right.$$

Schematically, the different steps in a DFPT calculation, relevant to understand the parallelisation, are represented by the following pseudo-code section:

```
(1) Initialize (reading the ground-state quantities)
 (2a) Loop on all the perturbations (up to 3*Natom perturbations)
  (2b) Iterate to reach the self-consistency
(2c) Loop on the electronic wave vectors (#k-points)
       (2d) Loop on the electronic states (#bands)
      [cgwf3] Compute the first-order wave function, for one state at one k-point.
            [accrho3] Accumulate the first-order density, contribution of one
state at one k-point.
       End loop (2d)
     End loop (2c)
              [rhohxc] Perform selected work on the accumulated density
     Decide to finish the self-consistency
  End loop (2b)
End loop (2a)
```

There are different preparatory (resp. analysis) steps before (resp. after) each of the loops.

In the sequential case, by far the largest amount of work is done in the section of the code shown in the box (loops 2c and 2d).

Description of the present implementation of parallelism

The present parallelisation relies on a distribution of the work for different k-point and bands on different cores (loops 2c and 2b). There is comparatively little communication between cores for this work to be done.

For our example, if this work is fully distributed, the number of k points being 16, and the number of bands being 128, the number of cores that can be used at maximum is 2048. Of course, the speed-up will saturate below this value, as will be shown by the tests. Indeed,

some parts are not parallelized at that level (they might be sequential, or only parallelized over the k-points), and there will be communication overheads.

Concerning the data distribution, we note that the most memory consuming quantities are the zero-order and first-order wave functions. The number of elements for these arrays is proportional to the number of plane waves, times the number of bands, times the number of k-points. While both zero-order and first-order wave functions can be distributed over processors that treat different k-points, only the first-order wave functions is presently distributed over processors that treat different states. Thus the processors must store the full array of zero-order wave functions for one k-point, scaling as the number of plane waves times the number of bands. This shortcoming originates from the need to compute the scalar product between zero- and first-order wave functions for different bands, to impose

$$\left\langle \psi_{\beta}^{(0)} \middle| \psi_{\alpha}^{(1)} \right\rangle = 0$$

This data distribution is the simplest one leading to the possibility of benefiting from a combined k-point and band distribution of the work. It might be improved, but this will go with a (limited ?) increase of the communications.

Description of the most CPU demanding routines, in the sequential and parallel cases

The relevant routines for the timing of the scaling are described now.

(A) projbd.F90: computes the projection of the trial first-order wave functions on the orthogonal space to the zero-order wave functions. In the present implementation, the work (a part of cgwf3.F90, inside loop 2d), is distributed over k points and bands.

(B) fourwf.F90 (pot): application of the zero-order local potential to a trial first-order wave function. In the present implementation, the work (a part of cgwf3.F90, inside loop 2d) is distributed over k points and bands.

(C) nonlop.F90 (pot): application of the zero-order non-local potential to a trial first-order wave function. In the present implementation, the work (a part of cgwf3.F90, inside loop 2d) is distributed over k points and bands.

(D) inwffil.F90 and rwwf.F90: reading the ground-state and first-order wave functions from file, and distributing the data to the processing cores. Section (1).

(E) fourwf.F90 (G->r): Fourier transform (reciprocal to real space) of the zero and first-order wave functions, needed to accumulate the first-order change of density (in accrho3.F90, inside loop 2d). This is distributed over k points and bands.

(F) fourdp.F90: Fourier transforms for the density. This operation is done after the loop (2c) and (2d), and does not depend on k-points neither on bands. It is done in sequential in the present implementation.

(G) vtorho3.F90 (synchro): synchronisation of the processors after the loop (2c).

(H) vtowfk3.F90 (contrib): different contributions at the end of the loop (2c), done in sequential.

(I) cgwf3-O(npw) and nonlop.F90(forces): different operations that scale as the number of plane waves, inside cgwf3. In the present implementation, the work (a part of cgwf3.F90, inside loop 2d) is distributed over k points and bands.

(J) vtorho3.F90:MPI: the MPI calls after loop 2c, to synchronize the first-order density on all compute cores.

(K) pspini.F90: the initialisation of the pseudo-potential. It is done in sequential in the present implementation.

Benchmarks results

The following data was gathered on *MareNostrum*, an IBM Powerpc970 cluster with Myrinet network, located at BSC. To increase the amount of memory available only two processes per node were executed.

It turned out that the routines cgwf3-O(npw), nonlop.F90(forces), vtowfk3.F90 (contrib), vtorho3.F90:MPI, and pspini.F90 always take only a minor part of the computation time, so they were not included in the following analysis.

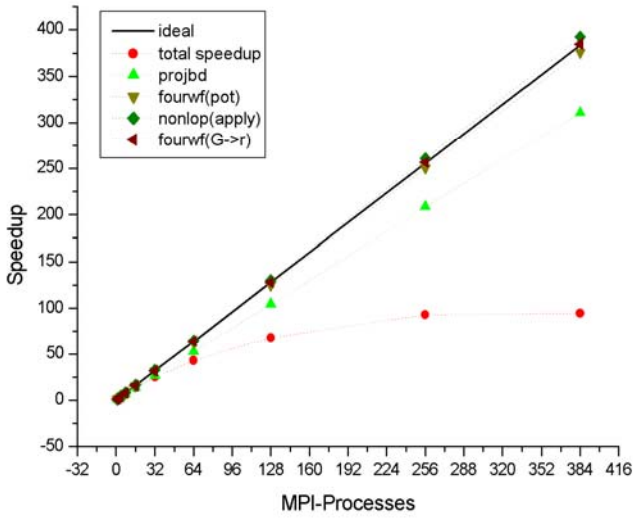


Figure 52: Speedup of the most costly code sections that show good scaling.

The total scaling and the speedup of the most cost important code sections are given in Figure 52. The graphs shows very good scaling of fourwf(pot), nonlop(apply), and fourwf(G->r). The routine projbd scales not as good, but still well. The sequencial parts of the code cause the efficiency of the code to decrease rapidly beyond 64 processes, resulting in a total speedup that converges to a maximum value of about 100 already when using 256 processes.

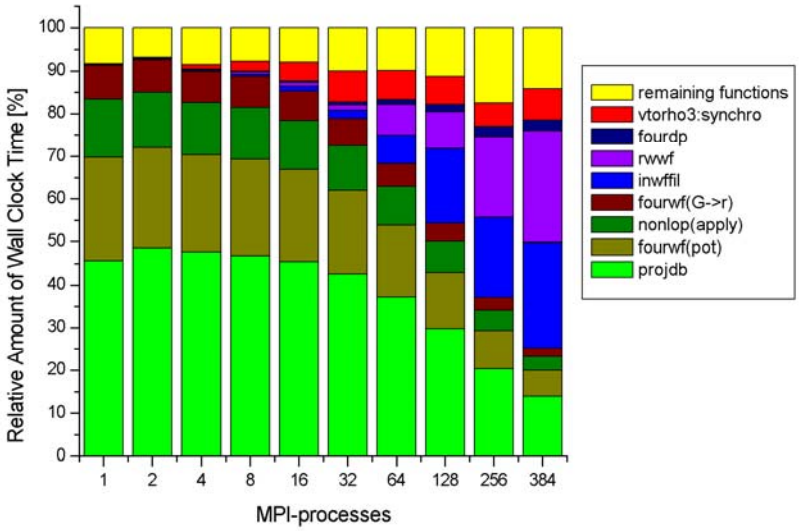


Figure 53: Relative amount of wall clock time for the most costly code sections.

The contributions of the individual code sections is depicted in Figure 53. One can see that the time-fraction of the well scaling routines (projdb, fourwf(pot), nonlop(apply), and fourwf(G->r)) reduces from over 90% for serial execution to 25% when using 384 processes. At the same time just reading the wave functions (rwwf and inwffil) takes over more than 50% of the computation time.

B.3 Performance improvement of the linear-response part of ABINIT.

Different strategies for improvement of the parallelisation can be pursued, even concurrently.

Strategy 1: Remove the IO-related initialisation bottleneck of the present parallelisation.

The major bottleneck seen for scaling this part of ABINIT beyond 64 processors (for the test case described in section B.2 Performances of the linear-response part of ABINIT) lies in the I/O-related initialisation of the wave functions. Indeed, the inwffil.F90 and rwwf.F90 routines take about 10% of the time when 64 compute cores are used for the test case, and they scale badly with the number of cores.

A prototype code is needed to identify whether the bottleneck is specifically due to the reading of the wave functions, or the subsequent distribution of the data already read on one processing core to the different processing cores.

A refactoring of these routines might be needed, involving *MPI-IO*. *MPI-IO* is already used in inwffil.F90 / rwwf.F90 for the ground state calculations, and has been shown to allow large speed ups of the IOs, and the whole test cases.

The performance gains that are expected for such a refactoring of inwffil.F90 and rwwf.F90 are very large, because the present implementation leads to sequential execution. And even a stronger slow down than simple sequential execution.

Strategy 2: The time spent in "fourdp","vtorho3" and "vtowfk3" (for the sections that are not parallelized over k-point and bands), should be examined as well. It is not as large as the one spent in inwffil.F90 and rwwf.F90.

Here, the use of the computing cores in parallel should be made possible thanks to *OpenMP* directives and threads. Unitary tests on the routine "fourdp" have been performed for the

excited state (GW calculation) section of the present D8.1.4 deliverable. It is possible to speed-up these sequential bottlenecks by a factor of about 6 by using 8 cores.

Strategy 3: Supposing the strategies 1 and 2 are successfully implemented, the distribution of the ground-state array should be improved. As remarked in section B.2 Performances of the linear-response part of ABINIT, at present, all the processors treating the same k point must store a copy of the wave functions for all states for that k-point. Thus, the memory requirement for one compute core increases with the size of the problem. One should distribute the ground-state wave functions among the processors, and treat the scalar product between ground-state and first-order wave functions accordingly. An *OpenMP* solution might be limited, so that MPI is to be preferred. The correct analysis of this strategy is to be refined.

Strategy 4: Target an additional parallelisation of the full problem, namely, the loop over perturbations (although only one perturbation was treated in the example case, the real situation implies dealing with 58 perturbations). This loop is labelled 2a in the pseudo-code analysis of section B.2 Performances of the linear-response part of ABINIT. Such a parallelisation has obvious advantages but also drawbacks:

Adv 1: the amount of communication is very low;

Adv 2: the scaling with the size of the system is good (the number of perturbations grows with the number of atoms), hence this level of parallelization can bring easily one order of magnitude more parallelism;

Drawback: the load balancing is not equal among the different perturbations, and only part of the unbalance can be predicted beforehand.

The load balancing should be tackled by constituting several pools of processors (each pool allowing k-point and band parallelisation), each taking in charge one perturbation at a time, according to a "waiting list". A "best" estimation of the unbalance should be done before hand, to tackle first the biggest chunks.