Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Optimizing GPAW

Jussi Enkovaara[a,*], Martti Louhivuori[a], Petar Jovanovic[b], Vladimir Slavnic[b], Mikael Rännar[c]

[a]CSC - IT Center for Science, P.O. Box 405 FI-02101 Espoo Finland
[b]Scientific Computing Laboratory, Institute of Physics Belgrade, Pregrevica 118, 11080 Belgrade, Serbia
[c]Department of Computing Science, Umea University, SE-901 87 Umea, Sweden

Abstract

GPAW is a versatile software package for first-principles simulations of nanostructures utilizing density-functional theory and time-dependent density-functional theory. Even though GPAW is already used for massively parallel calculations in several supercomputer systems, some performance bottlenecks still exist. First, the implementation based on the Python programming language introduces an I/O bottleneck during initialization which becomes serious when using thousands of CPU cores. Second, the current linear response time-dependent density-functional theory implementation contains a large matrix, which is replicated on all CPUs. When reaching for larger and larger systems, memory runs out due to the replication. In this report, we discuss the work done on resolving these bottlenecks. In addition, we have also worked on optimization aspects that are directed more to the future usage. As the number of cores in multicore CPUs is still increasing, an hybrid parallelization combining shared memory and distributed memory parallelization is becoming appealing. We have experimented with hybrid OpenMP/MPI and report here the initial results. GPAW also performs large dense matrix diagonalizations with the ScaLAPACK library. Due to limitations in ScaLAPACK these diagonalizations are expected to become a bottleneck in the future, which has led us to investigate alternatives for the ScaLAPACK.

1. Introduction

Electronic structure simulations have become an important tool in various fields of materials science. Advances in software packages and supercomputer infrastructures have allowed accurate *ab initio* simulations of complex systems consisting of several thousands atoms. One of the most successful computational frameworks in electronic structure calculations has been the density-functional theory (DFT) which provides a good trade-off between accuracy and numerical efficiency. First-principles simulations within DFT are a major consumer of supercomputing resources in several computing centres.

GPAW[1, 2] is one of the several software packages implementing the density-functional theory. GPAW utilizes the projector augmented wave (PAW) method. The PAW approximation offers good description over the whole periodic table of elements, and especially for first row and transition metal elements it provides typically better description than the more conventional norm-conserving or ultrasoft pseudopotential approaches. The discretization of the equations can be done with two complementary approach, either with a finite-difference method using uniform real-space grids or with localized atomic orbital basis. The real-space approach is more relevant for the petascale applications as it offers good parallelization possibilities. To our knowledge, GPAW is currently the only publicly available code implementing the PAW method with real-space grids.

In addition to standard density-functional theory, GPAW implements also the time-dependent density functional theory (TDDFT) which can be used for studies of excited state properties such as optical spectra, which are beyond the realm of standard DFT. Two different forms of time-dependent density functional are implemented, the real-time formalism (RT-TDDFT) and the linear response formalism (LR-TDDFT). Utilizing the PAW method within TDDFT is also relatively unique feature of GPAW.

GPAW is already used for massively parallel calculations in several research groups in Europe and in the USA. There are, however, still some bottlenecks in extremely large scale calculations. We report here some work that has been performed on resolving these bottlenecks within the PRACE 1st Implementation Phase project. We report also some experimental optimizations that are targeted more to the future usage of GPAW, as well as the collaboration done with GPAW user and developer community during this work.

[*]Corresponding author.
tel. +358 9 457 2935 fax. +358 9 457 2302 e-mail. jussi.enkovaara@csc.fi

The report is organized as follows: In Sec. 2. we discuss the I/O bottleneck in the initialization phase related to the use of Python programming language, and the approach used for solving it. We give brief introduction to parallelization strategy of GPAW in Sec. 3. as a background to the following sections. In Sec. 4. we describe the memory bottleneck in LR-TDDFT and its resolution, and in Secs. 5. and 6. we discuss the experiments with hybrid OpenMP/MPI implementation and ScaLAPACK alternatives, respectively. The collaboration with GPAW user and developer communities is discussed in Sec. 7., and in Sec. 8. are the final conclusions.

2. I/O bottleneck in Python initialization

2.1. Overview

GPAW is implemented using a combination of Python and C programming languages. Generally, high level algorithms are implemented in Python, while the actual time-consuming kernels are implemented in C (or imported from high performance libraries). MPI routines are called both from C and from Python. The high level data structures and tools in Python make development and maintaining the code efficient, while the C-kernels and high performance numerical libraries make the execution time still comparable to pure C/Fortran code. The overhead from Python is typically only few percent and GPAW has been able to achieve over 25 % of peak floating point performance in Cray XT5.

However, the use of Python introduces a surprising problem in massively parallel calculations with thousands of MPI tasks. When the Python interpreter starts-up, the Python code of all the used modules is read from the disk via so-called import statements. The number of files read can be several hundreds, and all the processes read the same files. For each file to be read there are also several file open/close operations as well as directory operations when searching for the modules. With up to few tens of processes the initialization time is typically few seconds, but as the number of processes increases the file system cannot serve all the processes and the initialization time increases drastically. The problem has been realized at least in the Lustre and the GPFS filesystems, and experiments on BlueGene/P systems in Jülich and Argonne have showed that with 12 racks (49152 processes) the start-up time can as much as 45 minutes!

2.2. Scalable Python

In order to solve the I/O bottleneck in importing Python modules, we have modified Python interpreters **import** mechanism in such a way that only a single process will perform the I/O operations, and MPI is used for broadcasting the data to other processes. In order to accomplish that, special wrapper routines have been created for all the C stdio-functions that are used during Python imports. Fig. 1 shows an example, where only the master process performs the actual call to I/O routine, and uses then MPI for broadcasting the data.

```
int __wrap_fstat(int fildes, struct stat *buf)
{
  int size = sizeof(struct stat);
  if (enabled)
    if (rank == MASTER)
    {
      fstat(fildes, buf);
      MPI_Bcast(buf, size, MPI_BYTE, MASTER, MPI_COMM_WORLD);
    }
    else
      MPI_Bcast(buf, size, MPI_BYTE, MASTER, MPI_COMM_WORLD);
  else
    fstat(fildes, buf);
  return 0;
}
```

Fig. 1. MPI enabled wrapping for C stdio function fstat

The wrapper functions are used via special header file *parallel_stdio.h*, snippet of which is shown Fig. 2.

```
...
#define fgets __wrap_fgets
#define fgetc __wrap_fgetc
#define fstat __wrap_fstat
...
```

Fig. 2. Snippet of header file which redefines the stdio functions so that wrappers can be used instead.

Finally, the special header can be included only at specific points in the Python source files, so that only the parts related to importing modules use the wrapper functions. Fig. 3 shows example of the file *import.c*. After including the *parallel_stdio.h*, all remaining the stdio calls will use the wrapper functions instead of original the functions. Our implementation is currently available under the GPL license from a Gitorius repository [5].

```
...
#include "importdl.h"
#ifdef ENABLE_MPI
#include "parallel_stdio.h"
#endif
...
```

Fig. 3. Snippet of *import.c* including the special header file.

2.3. Results

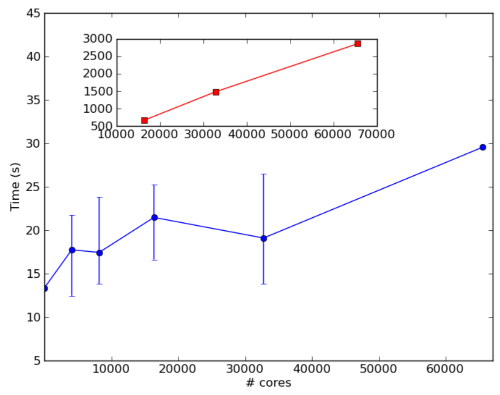
We have investigated the behaviour and performance of our scalable Python with the following simple Python code:

```
from ase import Atoms
from gpaw import GPAW
from gpaw.mpi import rank, world

world.barrier()
if rank == 0:
    print "Init completed"
```

which implicitly imports all the modules needed by GPAW, including the NumPy package [4]. We have inserted timers in the very beginning and in the very end of Python interpreter in order to measure the real time needed for the imports. We have also run the script under the **strace** utility in order to ensure that only a single process is performing I/O calls during the imports.

Fig. 4 shows the time needed for the imports as a function of MPI tasks in BlueGene/P and in the Bullx cluster CURIE. The BlueGene/P results are obtained partly in the JUGENE system in Jülich and in the Intrepid system in Argonne. The BlueGene/P results are obtained from five independent runs (except the last data point which is from a single run). The Bullx results shown here are generally from three runs.



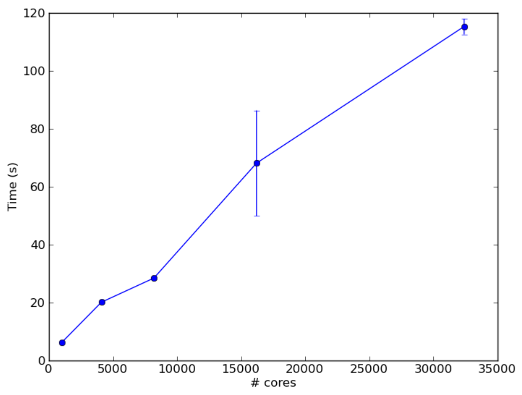


Fig. 4. Left: Time to perform imports in BlueGene/P. Inset shows the time with ordinary Python without the modifications. Right: Time to perform imports in the Bullx CURIE.

The figure shows clearly the improvement in import time in BlueGene/P. With scalable Python the import time increases only slightly with the number of CPU cores (theoretically the increase should be log(N) from the MPI_Bcast), while without modifications the import time increases more or less linearly. Furthermore, with 65536 CPU cores (*i.e.* 16 racks) the import time decreases from ∼2900 s. to ∼30 s, *i.e.* by a factor of nearly 100.

In CURIE, the performance is not improved as much as in BlueGene/P, and the for reason this is currently unclear. One obvious difference is in the file system, in Blue/Gene/P the GPFS filesystem is used, while in CURIE the filesystem is Lustre. However, experiments in other Lustre systems, e.g. Cray XT5 at CSC, Finland, have shown similar performance improvements as in BlueGene/P (however with more limited number of CPU cores).

3. Parallelization strategy in GPAW

In this section we describe the basic parallelization strategy in GPAW as a background material for forthcoming optimization chapters.

The basic data structure in GPAW (in the real-space mode) is five-dimensional wave function array, where the first dimension is for spin and k-point index, second is the electronic state index, and the last three ones are the spatial indices of the real-space grid. Parallelization is possible over all the indices. The spin/k-point parallelization is trivial, however, it has limited usage in large scale calculations. The domain decomposition over the real-space grids uses nearest-neighbour exchange type communication which provides better scalability than the fast fourier transforms utilized in the plane-wave based code packages.

The parallelization over electronic states is trivial in the real-time time-dependent density functional theory mode, where time-propagation can be carried out independently for each electronic state, and communication over the electronic state index is limited to reductions when summing for the electron density.

In ground state mode (standard DFT) the parallelization over electronic states is non-trivial due to need for subspace diagonalization and orthonormalization. In subspace diagonalization and orthonormalization a $N \times N$ dense matrix, where N is the total number of electronic states, is constructed and diagonalized or Cholesky decomposed and inverted. When parallelizing over electronic states this matrix construction (essentially a matrix-matrix multiplication) requires information about every electronic state in every process. This is carried out by using a pipeline with exchange communication in one dimensional cyclic chain. When permitted by the underlying hardware and MPI implementation, matrix-matrix products and communication (over electronic states) are overlapped. After the matrices are constructed, the diagonalization and Cholesky factorization are done by ScaLAPACK. Currently, the maximum matrix dimensions have been few thousands, and only a subset of all the processes is used for ScaLAPACK.

In linear-response time-dependent density functional theory the so called Ω-matrix is constructed in a electron-hole pair basis. The matrix elements can be calculated independently, thus very efficient parallelization over electron-hole pairs is possible.

4. Time-dependent density-functional theory

Recently, a memory bottleneck has been realized in linear response TDDFT calculations. Currently, the Ω-matrix is replicated over all the MPI tasks, and in very large scale calculations there will not be enough memory to store the matrix on all processes.

During this project, the implementation has been changed in such a way that each process stores only the part of the Ω-matrix it has calculated. This leads to a one dimensional row wise decomposition of the matrix. The matrix construction is typically much heavier operation than the diagonalization, and therefore the construction and diagonalization are often done in separate steps. This can be accomplished by writing the matrix into a file. As each process contains part of the matrix the HDF5 library is utilized for writing the matrix in parallel.

The actual diagonalization can be performed with ScaLAPACK, and the matrix can be either read from the file in parallel with HDF5, or in case the matrix construction and diagonalization are done within the same run, BLACS routines are used to transfer the row-wise distribution to a two dimensional distribution which is more appropriate.

Measurements of memory usage show that memory need decreases as expected, and that by increasing the number of CPUs for the electron-hole parallelization the memory requirement per CPU can now be kept constant.

5. Hybrid OpenMP/MPI implementation

Even though the current pure MPI implementation of GPAW performs well, it is expected that ever increasing number of cores within a CPU can make a hybrid MPI/shared memory implementation advantageous in the near future. Especially, parallelization over electronic states involves communication of large quantities of data (up to tens of MB), and utilizing high performance threaded BLAS routines could speed up the calculations.

The Python based implementation poses some challenges for threading. Whereas message passing can be used transparently both on the Python and on the C level, the global interpreter lock in CPython limits the thread based parallelization to the C-extensions only. Currently, we have implemented OpenMP based parallelization for the most time-consuming C-functions in GPAW. In most routines the OpenMP parallelization is performed over the electronic state index, only when solving the Poisson equation threading is performed over the real-space grid. Furthermore, a threaded BLAS library is used for dense linear algebra operations. In some routines MPI calls are made within the OpenMP parallel regions, therefore a MPI implementation supporting the MPI_THREAD_MULTIPLE thread safety mode is required.

5.1. Results

Preliminary hybrid version of the code has been tested on CURIE (Bullx, Intel Nehalem, Infiniband QDR). It was compiled using the Intel compiler v12.1.7.256 and linked against the ScaLAPACK and MKL libraries. Profiling and tracing was done using TAU profiling and tracing suite[6] and using MPE MPI tracing library[7]. Code was

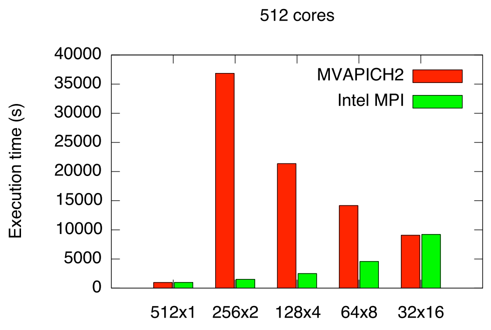
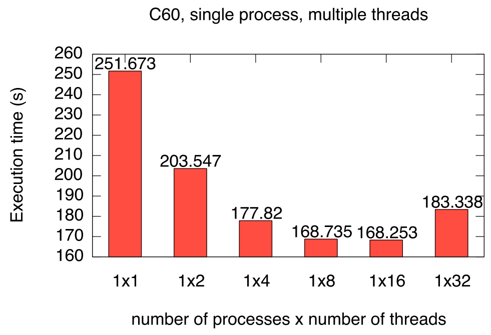


Fig. 5. Execution times with 512 cores with MVAPICH2 and IntelMPI library and varying process-thread count configurations.



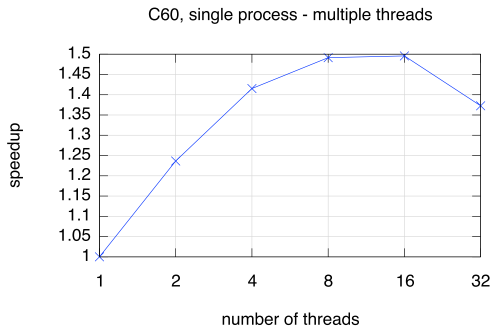


Fig. 6. Left: execution time with one process and increasing number of threads; Right: speedup with increasing number of threads.

tested with two MPI libraries – MVAPICH2 and IntelMPI. These MPI libraries were chosen over default libraries provided by Bull on CURIE (bullxmpi and bullmpi) because hybrid GPAW requires MPI_THREAD_MULTIPLE thread safety level (where multiple threads may call MPI at any time), which is not enabled by default in most MPI implementations due to performance issues. Other MPI implementations were considered, among them OpenMPI v1.4.5 and 1.5 beta, but they had problems with CURIE's resource manager (SLURM) that caused erroneous behaviour. These issues were also confirmed by CURIE's support team. The only library that worked successfully in the provided environment was MVAPICH2 v1.8, built with "–enable-threads=multiple –with-pmi=slurm –with-pm=no" as configuration parameters. Also, it was found that at the runtime, only MPI installation's *lib* directory should be included in $LD_LIBRARY_PATH, while *bin* directory should not be included in the $PATH environment variable. Otherwise, SLURM's process manager will not be invoked. While waiting for Bull's implementation of MPI that supports MPI_THREAD_MULTIPLE, Intel MPI with this feature has been provided by CURIE's staff, and it was used for all GPAW tests (through modules system).

Testing was performed on a 702 atom Si cluster dataset for larger runs up to 1024 cores, and on a C_{60} molecule dataset for smaller tests up to 32 cores. Testing results have shown that the performance of the hybrid version suffers when threads are used along with MPI, as can be seen in Fig. 5. The figure shows also that using different MPI implementations can have significant impact on the execution time. However, none of the runs with MPI processes and threads outperformed the pure MPI run that had only one thread per process. Fig. 6 shows that there are performance gains when only OpenMP threads are used, i.e. without MPI.

The main suspected causes of such performance loss is synchronization and locking strategies internal to MPI library's implementation. MPI_THREAD_MULTIPLE level of thread safety implies issues and performance trade-offs, some of which are discussed in Ref. [8]. Fig. 7 shows that hybrid run (right-hand side) has much more dense communication clusters, and significantly more time spent in MPI_Wait(), which is shown by red rectangles.

The obtained results suggest that reliance on MPI_THREAD_MULTIPLE should be abandoned, and MPI communication should be moved out of those segments of code that run in OpenMP threads. This would avoid thread synchronization issues in MPI libraries. In particular, calls to functions for exchange of boundary conditions *bc_unpack1()* and *bc_unpack2()*, should be made outside of OpenMP parallel regions.

Some tests have been made also in BlueGene/P and BlueGene/Q systems using the same 702 atom Si cluster dataset. The code was compiled with the IBM XL compiler, and threaded ESSL library was used for dense linear algebra. Fig. 8 shows performance with different parallelization options.

In BlueGene/P there is no benefit from threading even though the penalty is not as large as in CURIE. On the other hand, on BlueGene/Q the best performance (with a small margin) is obtained with two threads.

These first experiments with hybrid OpenMP/MPI implementation indicate that large performance gains

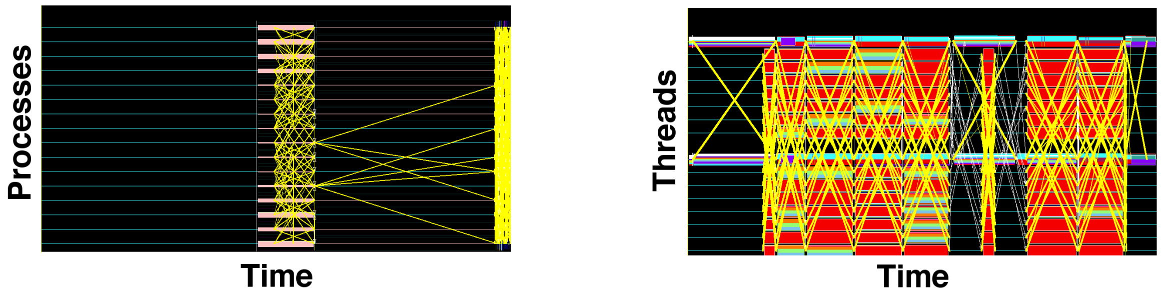


Fig. 7. MPI communication trace on C_{60} molecule data produced by MPE, run on 16 cores. Left: pure MPI run with 16 processes; Right: hybrid MPI+OpenMP run with 2 processes and 8 threads.

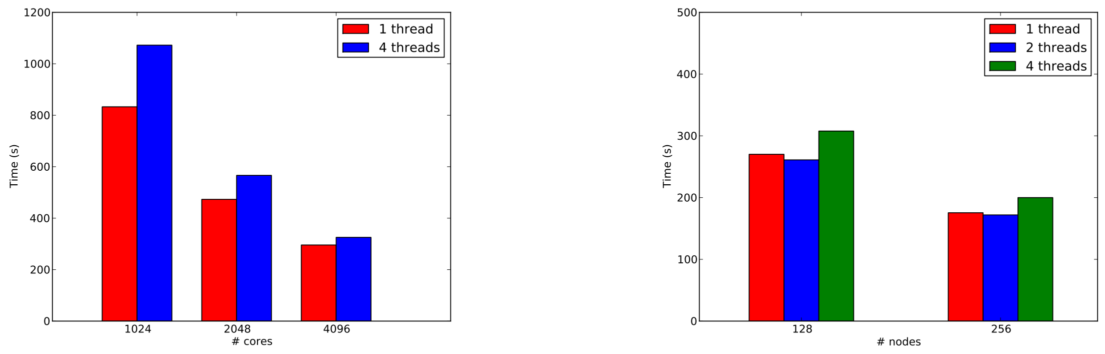


Fig. 8. Left: Performance in BlueGene/P with different threading options. Right: Performance in BlueGene/Q.

are not to be expected from threading. However, some performance improvements could be possible to obtain with further optimization.

6. Parallel dense matrix diagonalizations

Large dense matrix diagonalizations are expected to become a bottleneck in the future and the expectations are that when the number of electrons increases, the routines used from ScaLAPACK will not scale on par with the other parts of the code. We have investigated two alternatives to the current ScaLAPACK based implementation.

6.1. *Elemental*

Elemental[10] is a C++ framework for distributed-memory dense linear algebra that strives to be fast and convenient. It has been primarily developed by Jack Poulson, The Institute for Computational Engineering and Sciences, University of Texas at Austin. Just like ScaLAPACK and PLAPACK, primary goal of Elemental is in extending BLAS and LAPACK-like functionality into distributed-memory environments. It is designed to be a modern extension of the communication insights of PLAPACK to elemental distributions. While ScaLAPACK linked algorithmic and distribution block sizes, and PLAPACK introduced a mild inscalability through the use of only a single vector distribution, Elementals simplification is to fix the distribution block sizes at one. One of the major benefits compared to the element-wise distribution of matrices approach is the much more convenient handling of submatrices, relative to block distribution schemes.

6.2. *ELPA*

ELPA (Eigenvalue soLvers for Petaflop Applications)[9] is an Open Source Fortran-based computational library for the parallel solution of symmetric or Hermitian, standard or generalized eigenvalue problems. ELPA is a set of Fortran subroutines (module) that can be compiled as a separate library or together with an application of choice. Once compiled, ELPA library routines can be linked to from C, C++, Fortran etc. code alike. ELPA works as a drop-in enhancement for ScaLAPACK-based infrastructures. Thus, ELPA is not independent of this infrastructure, but rather builds on it. Necessary prerequisite libraries for ELPA (often already provided by HPC vendors) include: BLAS, LAPACK, BLACS, and ScaLAPACK. ELPA is an MPI-only implementation, i.e., no hybrid parallelization (MPI/OpenMP) is assumed. It will thus work both in a single-node, shared memory environment, as well as large clusters of separate nodes. Preliminary hybrid version of ELPA library is being tested by the developers and first promising performance results have been achieved.

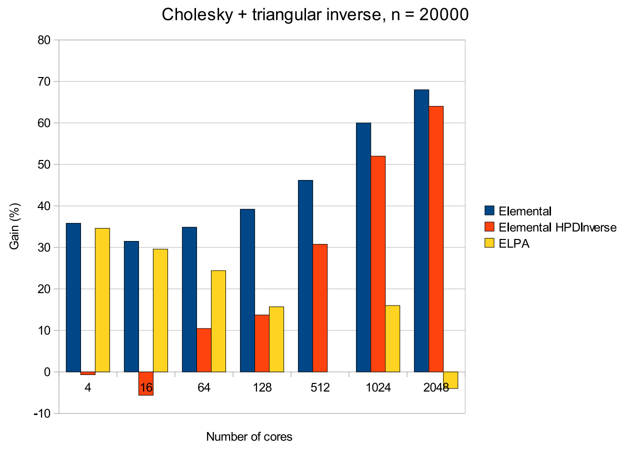


Fig. 9. Gain of the routines from the tested libraries over the reference ScaLAPACK routines (in percents).

For eigenvalue problems ELPA has both a one-stage and a two-stage solver, see [9] for details. For the two-stage solver ELPA has isolated some important linear algebra kernels and put them in a separate file which then can be hand-tuned for optimum performance on special architectures. The library comes with a generic version of the kernels that runs on every platform but is optimized for the Intel SSE instruction set. Included is also a file with optimized ELPA kernels for the BlueGene/P architecture that contains assembler instructions for the BlueGene/P processor which IBM's xlf Fortran compiler can handle.

6.3. Results

We have looked for numerical libraries that contain routines that in the future could be an alternative to the currently used routines from ScaLAPACK. After an initial look at some potential candidates, we chose two libraries to test and in this stage of the process it was decided that it would be enough to do a stand alone test of the routines, i.e., not incorporate the investigated routines into real application.

The testing has been done on up to 2048 nodes of the Tier-0 machine CURIE in France which is an x86 cluster (S6010 bullx nodes with 4 eight-core Intel Nehalem-EX X7560 @ 2.26 GHz, InfiniBand QDR Full Fat Tree network). The size of the square matrices tested is 20000 and testing shows that there is often no gain in using more that 1024 cores, since the wallclock time increases when going from 1024 to 2048 cores for many cases. Everything was compiled using the Intel compiler suit and the test programs were written in F90 and C++, for the ELPA and Elemental routines, respectively. The tested libraries were compiled "as is", i.e., using the included Makefiles and for ScaLAPACK the installed library was used. Testing also included some experimenting with different block sizes and processor grids.

GPAW uses several ScaLAPACK routines and at the moment the default routine for diagonalization is `pdsyevd` (`pzheevd` for complex cases) and `pzpotrf` + `pztrtri` for Cholesky.

6.3.3. Cholesky

For the Cholesky factorization and the inverse of the triangular matrix (ScaLAPACKs `pzpotrf` + `pztrtri`) there exists corresponding routines in ELPA (`cholesky_complex` and `invert_trm_complex`) that are used in the tests. Using Elemental, there exists two ways to solve this problem. First there is the normal way of using a Cholesky factorization (`Cholesky`) and then a triangular inverse (`TriangularInverse`). The second alternative is a routine (`HPDInverse`) that performs an in-place inversion using a custom algorithm for the inversion of a Hermitian positive-definite matrix.

The results of the testing with these three alternatives are shown in Figures 9 – 10 where they are compared to the routines from ScaLAPACK. The Elemental alternatives are generally performing better that ScaLAPACK, especially when the number of cores increases. For 1024 and 2048 cores the gain is more than 50% compared to ScaLAPACK, but we should note that the execution time is quite small. The fastest execution time is 8 seconds for 2048 cores using the normal Elemental routine, compared to ScaLAPACK's 25 seconds. The scaling of ELPA is worse than that of ScaLAPACK and at 2048 cores ELPA is actually slightly slower than ScaLAPACK.

6.3.3. Diagonalization

For the computation of eigenvalues and eigenvectors ELPA has two different routines available (`solve_evp_real` and `solve_evp_real_2stage`). In contrast, Elemental provides just one routine (`HermitianEig`), but with tuning parameters you can get three different approaches to the reduction to a tridiagonal matrix. The normal approach, which is a pipelined algorithm designed for general (rectangular) process grids, and two square grid approaches. The latter redistribute the matrix so that it is owned by a perfect square number of processes,

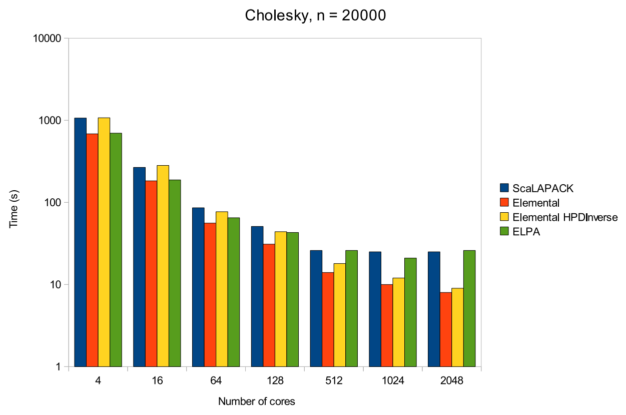


Fig. 10. Times for the routines from the tested libraries and the reference ScaLAPACK routines (in seconds).

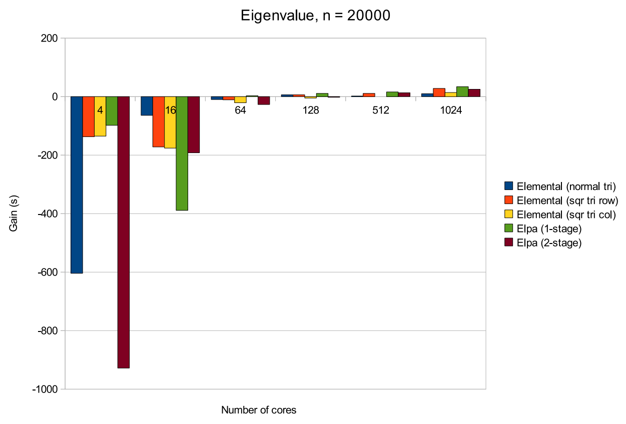


Fig. 11. Gain of the routines from the tested libraries over the reference ScaLAPACK routines (in seconds).

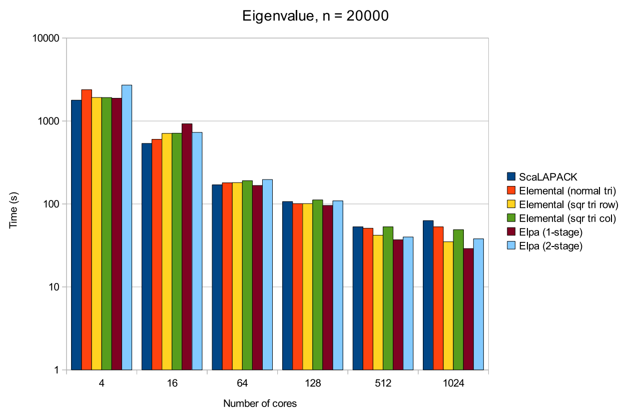


Fig. 12. Times for the routines from the tested libraries and the reference ScaLAPACK routines (in seconds).

perform a fast reduction to tridiagonal form, and redistribute the data back to the original process grid. The square approach can further be split into a row or column subgrid ordering of the processors.

The results of the testing of these five versions are shown in Figs. 11 – 12 where they are compared to the ScaLAPACK routine `pdsyevd`. In the figures one can see that when we have a relatively low number of cores (< 100) ScaLAPACK is doing better than both ELPA and Elemental, but when the number of cores increases the alternatives perform better. For 1024 cores the times for some of the alternatives are 40-50% better than the corresponding routine from ScaLAPACK. In seconds this corresponds to around 30 seconds faster for the ELPA (1-stage) and the Elemental (square column ordering) solvers compared to ScaLAPACK (`pdsyevd`).

6.3.3. Recommendation

Incorporating ELPA into GPAW should not be too hard since it uses the same framework as the currently used ScaLAPACK. For Elemental it is a bit more difficult since it uses a different distribution and C++ classes.

Both libraries are relatively new and under development, which indicates that it might be a good idea to keep them under surveillance for a possible future incorporation. For ELPA, the idea of a separate file for a small set of hand-tuned kernels may in the future lead to a large collection of architecture dependent files for the available architectures, which could improve performance even further.

7. Collaboration with scientific communities

GPAW is currently developed and used in several research groups in Europe and in the USA. During this project there has been an active collaboration both with the user and the developer communities.

The main way of collaboration has been via the mailing lists of GPAW users and developers. There has been also some developer teleconferences. Collaboration has also been done through visits to research groups in Finland, Denmark, and in the USA, and through participation in research workshops.

The work performed within this project has enabled three proposals to be made for PRACE production access, one in each of the PRACE 3^{rd}, 4^{th}, and 5^{th} calls. The proposal in PRACE 3^{rd} call was accepted and it is currently running.

8. Conclusions

Significant progress has been made during this project in optimizing GPAW. The initialization bottleneck due to Python's import mechanism has been largely solved, and the implementation is availalbe on a public code repository. The work performed in this subproject benefits not only GPAW, but all the researchers who wish to use Python in massively parallel environments.

The memory bottleneck in linear response TDDFT part of the code has been solved by distributing the relevant large matrix. This work makes it possible to study larger system sizes than previously, for example, the optical spectra in large gold clusters can now be extended to plasmonic range as proposed in the application in PRACE 4^{th} call.

The work on hybrid OpenMP/MPI implementation and ScaLAPACK alternatives has not resulted in direct improvements in the performance of GPAW. However, this work has been important for future development and optimization of the code.

All the modifications made to the GPAW code during the project are published via GPAW's main source code repository [2], and are thus available to all GPAW users.

There has been active collaboration with the users and developers of GPAW during the whole project, and work performed in the project has made three proposals to PRACE regular access possible.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources CURIE at CEA, France and JUGENE at Jülich, Germany, as well as the PARADOX system at IPB, Serbia.

References

1. J. Enkovaara, C. Rostgaard, J. J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. A. Hansen, H. H. Kristoffersen, M. Kuisma, A. H. Larsen, L. Lehtovaara, M. Ljungberg, O. Lopez-Acevedo, P. G. Moses, J. Ojanen, T. Olsen, V. Petzold, N. A. Romero, J. Stausholm-Møller, M. Strange, G. A. Tritsaris, M. Vanin, M. Walter, B. Hammer, H. Häkkinen, G. K. H. Madsen, R. M. Nieminen, J. K. Nørskov, M. Puska, T. T. Rantala, J. Schiøtz, K. S. Thygesen, K. W. Jacobsen, Electronic structure calculations with gpaw: a real-space implementation of the projector augmented-wave method, Journal of Physics: Condensed Matter 22 (25) (2010) 253202. `http://stacks.iop.org/0953-8984/22/i=25/a=253202`
2. https://wiki.fysik.dtu.dk/gpaw.

3. T. E. Oliphant, Guide to NumPy, Provo, UT (Mar. 2006). `http://www.tramy.us/`
4. http://numpy.scipy.org/.
5. http://gitorious.org/scalable-python
6. TAU, Tunining and Analysis Utilities, `http://tau.uoregon.edu/`
7. MPE, MPI Parallel Environment, `http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm#MPE`
8. W. Gropp, R. Thakur, Issues in Developing a Thread-Safe MPI Implementation
9. T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems, Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations, Parallel Computing 37, 783-794 (2011).
10. J. Poulson, B. Marker, J.R. Hammond, N.A. Romero, and R. van de Geijn, Elemental: A new framework for distributed memory dense matrix computations, ACM Trans. Math. Soft. In revision. Available from `http://elemental.googlecode.com/files/Elemental-rev2.pdf`.