Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Memory Optimization for the Octopus Scientific Code

J. Alberdi-Rodriguez[a,b], A. Rubio[a,c], M. Oliveira[d]

A. Charalampidou[e,f], D. Folias[e,f]

[a]Nano-Bio Spectroscopy Group and European Theoretical Spectroscopy Facility (ETSF)
University of the Basque Country UPV/EHU, Donostia, Spain
[b]Department of Computer Architecture and Technology
University of the Basque Country UPV/EHU, Donostia, Spain
[c]Max Planck Institute for the Structure and Dynamics of Matter, Hamburg, Germany
Departamento de Física de Materiales, Centro de Física de Materiales CSIC-UPV/EHU-MPC and DIPC, University of the Basque Country UPV/EHU, Donostia, Spain
Fritz-Haber-Institut Max-Planck-Gesellschaft, Berlin, Germany
[d]Dep. Physique, Université of Liège
[e]Greek Research and Technology Network, Athens, Greece
[f]Scientific Computing Center, Aristotle University of Thessaloniki, Greece

Abstract

Octopus is a software package for density-functional theory (DFT), and its time-dependent (TDDFT) variant. Linear Combination of the Atomic Orbitals (LCAO) is performed previous to the actual DFT run. LCAO is used to get an initial guess of densities, and therefore, to start with the Self Consistent Field (SCF) of the Ground-State (GS). System initialization and LCAO steps consume a large amount of memory and do not demonstrate good performance. In this study, extensive profiling has been performed, in order to identify large matrices and scaling behaviour of initialization and LCAO. Alternative implementations of LCAO in Octopus have been investigated in order to optimize memory usage and performance of LCAO approach. Use of ScaLAPACK library led to significant improvement of memory allocation and performance. Benchmark tests have been performed on MareNostrum III HPC system using various combinations of atomic systems' sizes and numbers of CPU cores.

Introduction

In this project time-dependent density functional theory (TDDFT) formulation of quantum mechanics was used as implemented in the Octopus code. Octopus [1] is a scientific software package for the calculation of electronic properties of matter, released under the GPL license. Octopus is a very efficient code used to study by first principles the properties of the excited states of large biological molecules, complex nanostructures, and solids.

One of the first steps in the calculation of the ground-state for a given system is the construction of an initial guess for the wave function and to build the Hamiltonian matrix previous to the SCF iterations. This process has been implemented in the Octopus code by performing a Linear Combination of the Atomic Orbitals (LCAO). It was proved that this stage had not been efficiently implemented.

This project focuses on optimization of the LCAO implementation to reduce memory cost and execution time. A memory allocation problem has also been observed during the initialization, previous to the Ground-State (GS) Self Consistent Field (SCF) cycle. LCAO consists in a diagonalization of a matrix of size NxN (N is equal to the number of orbitals of the atomic system).

This is a small matrix for a small system, but with a quadratic scalability. Therefore, it becomes a huge matrix for systems with many states.

We solved this problem with a parallel implementation of the LCAO using the external library ScaLAPACK. With this parallel library, large matrices have been distributed among all the MPI processes. Besides, alternative implementations have decreased the execution time of the LCAO step.

Benchmark tests have been performed on Barcelona Supercomputing Center MareNostrum III supercomputer [2], which is based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux Operating System and an Infiniband interconnection.

Porting of Octopus on MareNostrum III

Octopus source code was checked out from SVN repository and was ported on MareNostrum III.

MareNostrum III modules that were used: OpenMPI/1.8.1, INTEL/13.0.1, MKL/11.0.1 and GSL/1.15.

Libraries that were compiled on MareNostrum III (using intel, mkl and cmake modules) in order to enable Octopus latest revision to run efficiently are the following:

1. metis-5.1.0 (cc=icc cxx=icpc)
2. parmetis-4.0.3 (cc=mpicc cxx=mpicxx)
3. fftw-3.3.3
4. pfft-1.0.7
5. libxc-2.0.0 (--enable-shared CC=icc FC=ifort)
6. lapack-3.5.0
7. scalapack-2.0.1

Slightly less memory is consumed during mesh partitioning when using ParMETIS package in comparison to the serial version of METIS. This, is more noticeable with larger numbers of MPI processes. [3]

The following configuration was used for Octopus compilation on MareNostrum III.

Configuration options : max-dim=3 mpi sse2 avxC

C compiler flags : -I$HOME/local/parmetis/openmpi/include -DENABLE_PARMETIS=1 -g -xHost -O3 -m64 -prec-div -shared-intel -sox

Fortran compiler flags : -I$HOME/local/parmetis/openmpi/include -DENABLE_PARMETIS=1 -g -xHost -O3 -m64 -prec-div -shared-intel -sox

Parameters used to run the configuration script before compilation:

```
--with-libxc-prefix --with-libxc-include --disable-gdlib --enable-mpi --enable-newuoa --with-gsl-prefix --with-parmetis --with-pfft-prefix
```

```
LIBS_FFT="-Wl,--start-group -L$PFFT_HOME/lib -L$FFTW_HOME/lib -lpfft -lfftw3 -lfftw3_mpi -lfftw3_threads -Wl,--end-group"
```

In order to compile Octopus with ScaLAPACK, the following MKL libraries were used:

```
MKL_LIBS="-L$MKL_HOME -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_openmpi_lp64 -lpthread -lm"
```

Alternatively, the following configuration parameters were used to employ libraries that were compiled locally:

```
--with-blas=$HOME/SRC/LIBS/BLAS/libblas.a --with-lapack=$HOME/SRC/LIBS/LAPACK/liblapack.a
--with-blacs=$HOME/SRC/LIBS/scalapack/libscalapack.a --with-scalapack=$HOME/SRC/LIBS/scalapack/libscalapack.a
```

Implementations of LCAO

A set of initial Kohn-Sham orbitals are needed as an initial guess, since the solution of the ground-state problem is calculated iteratively. The Linear Combination of Atomic Orbitals (LCAO) approach is used to obtain an initial guess of the wave-functions and densities before the SCF cycle takes place.

a. LAPACK native implementation of LCAO

LCAO is performed by each MPI process using LAPACK library routines for the computations in a replicated way. For small atomic systems, this approach is relatively fast. However, when applied to larger systems, it leads to increased memory requirements in general, as it forces all the processes to allocate Kohn-Sham Hamiltonian and Overlap matrices, each of size NxN, as well as additional matrices. Therefore, this technique does not demonstrate good performance for larger systems due to the quadratic manner in which the size of the matrices increase.

b. *LAPACK alternative implementation of LCAO*

Only the root MPI process allocates memory for Hamiltonian and Overlap matrices and performs LCAO computations.

```
if (mpi_grp_is_root(mpi_world)) then
      SAFE_ALLOCATE(hamiltonian(1:this%norbs, 1:this%norbs))
      SAFE_ALLOCATE(overlap(1:this%norbs, 1:this%norbs))
end if
```

After the computation is finished, the computed eigenvalues are distributed to all the MPI processes using MPI_Bcast:

```
call MPI_Bcast(eval(1), size(eval), R_MPITYPE, 0, gr%mesh%mpi_grp%comm, mpi_err)
```

c. ScaLAPACK alternative implementation of LCAO

Scalapack parallel library has been used to perform LCAO computations. Kohn-Sham Hamiltonian and Overlap matrices are distributed among the MPI processes.

```
 ! The size of the distributed matrix in each node
this%lsize(1) = max(1, numroc(this%norbs, nbl, st%dom_st_proc_grid%myrow, 0, st%dom_st_proc_grid%nprow))
this%lsize(2) = max(1, numroc(this%norbs, nbl, st%dom_st_proc_grid%mycol, 0, st%dom_st_proc_grid%npcol))
--
SAFE_ALLOCATE(hamiltonian(1:this%lsize(1), 1:this%lsize(2)))
SAFE_ALLOCATE(overlap(1:this%lsize(1), 1:this%lsize(2)))
```

The Call Graph in Figure 1 has been acquired using the Valgrind profiling tool Callgrind, which records the call history among functions, and has been visualized using KCachegrind tool. The graph demonstrates calls to ScaLAPACK routines by root MPI process, when the ScaLAPACK alternative LCAO implementation is applied.

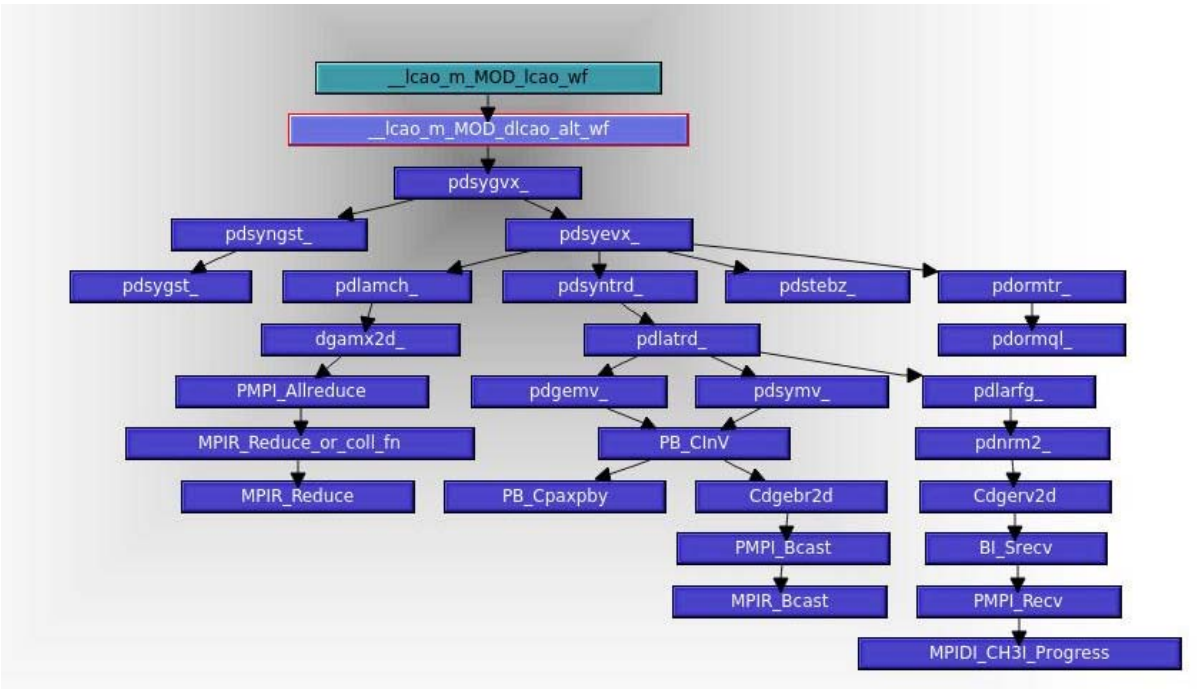


Figure 1: Call Graph produced using Valgrind Callgrind tool and KCachegrind for visualization.

Use cases

In this study, in order to acquire useful information on memory usage and performance of LCAO implementations, benchmark tests were performed on different numbers of chlorophyll molecules (Figure 2). The following test cases were used:

a. ***1 chlorophyll*** molecule - ***149 atoms*** which corresponds to ***374 orbitals***

b. ***2 chlorophyll*** molecules - ***247 atoms*** which correspond to ***643 orbitals***

c. ***4 chlorophyll*** molecules - **460 atoms** which correspond to ***1213 orbitals***

d. ***3 chlorophyll*** molecules - ***569 atoms*** which correspond to ***1382 orbitals***

e. ***14 chlorophyll*** molecules - ***2025 atoms*** which correspond to ***5121 orbitals***

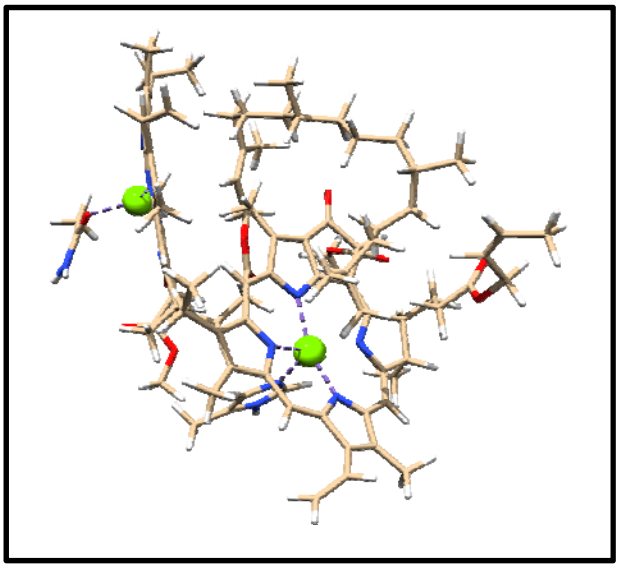


Figure 2: Visualization of 2 chlorophyll molecules with Chimera UCSF tool.

These use cases are real systems that could be studied in another context. Moreover, these systems were chosen to give an idea of the memory usage from medium to large sizes. Clearly, memory is not an issue in smaller systems than those presented here, as long as they can fit in any commodity PC.

Memory usage

Valgrind Massif heap profiler [4] has been used for memory profiling and the results have been visualized using Massif Visualizer. Figure 3 demonstrates heap memory usage for 64 MPI processes and 1382 orbitals. The greatest amount of memory is allocated during execution of states_allocate_wfns subroutine for dpsi pointer to wavefunctions or states.

```
SAFE_ALLOCATE(st%dpsi(1:np_part, 1:st%d%dim, st1:st2, k1:k2))
```

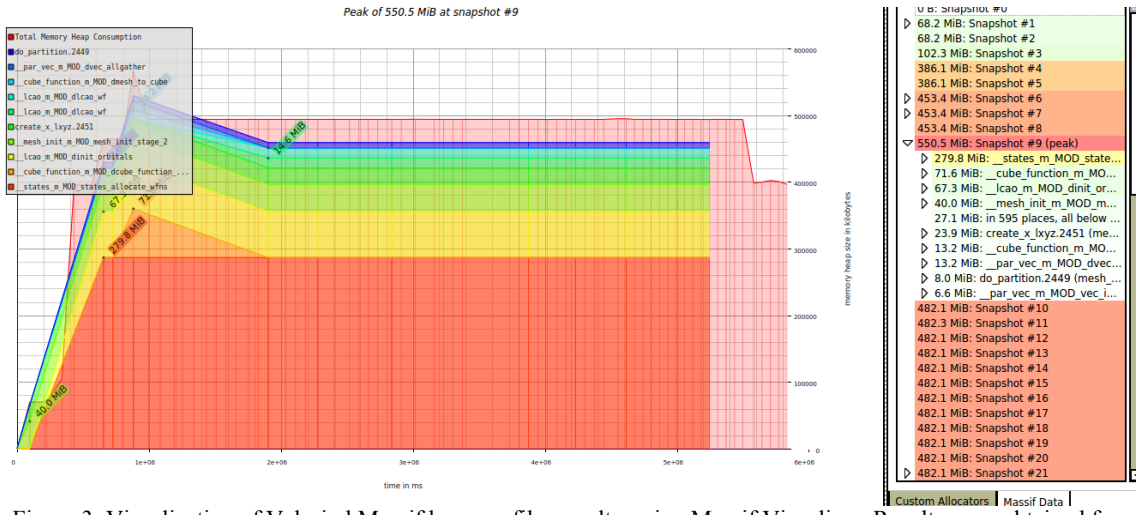


Figure 3: Visualization of Valgrind Massif heap profiler results, using Massif Visualizer. Results were obtained from Octopus execution with 64 processes on 1382 orbitals use case using LCAO native implementation. (Peak of 550.5 MiB)

Extensive profiling of Octopus memory allocation has been performed using the Octopus ad-hoc profiler, which proved to be a very useful tool, due to its capability to provide very detailed information on memory usage and performance [5].

We can demonstrate that the ad-hoc profiler is good enough at estimating the memory usage. Assuming that the result of the Valgrind Massif profiler is acceptable, Octopus shows only a small discrepancy (which can also be justified). This time, the studied use case is only one chlorophyll molecule (a). It consists of 603,076 grid points (spacing 0.2□, radius 4□) and 188 atomic states. Every mesh point is represented with 16 bytes complex number, resulting to a size of 9.2 MiB (9,649,216 bytes). In total, 1.69 GiB (1814052608 bytes) are needed to store all 188 states. The simulation mesh is split in 4 domains (for MPI > 4). Runs with power of 2 MPI processes are performed (no OpenMP parallelization is used). The actual measurement is applied in a time-dependent run, iterating 30 times and restarting from a previous ground-state run. The results are shown in Figure 4:

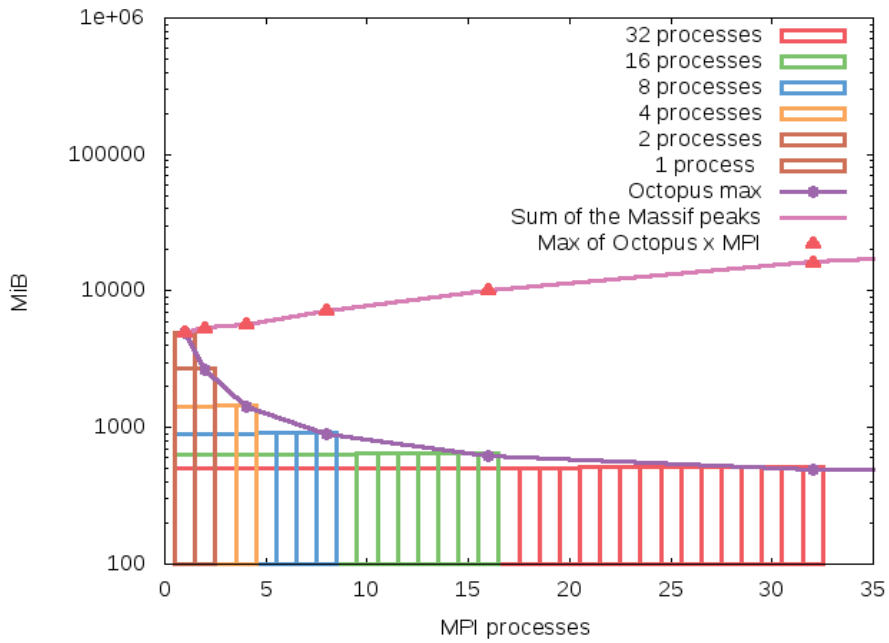


Figure 4: Memory usage of Octopus code measured by its ad-hoc tool and with Massif profiler. Octopus measures only the usage in the root MPI process, while Massif measures in all process. Both results are in agreement.

Massif: the bars of the graphic show the peak memory usage detected by the Massif tool for each of the processes. Despite a small variation, all the processes use the same amount of memory. Octopus: the code prints the maximum memory usage for the main process. These results (circle sphere plots) agree with the peak usage detected by Massif, with a tiny underestimation because the ad-hoc profiler cannot take into consideration memory allocated by the external libraries. Multiplying the maximum memory usage detected by Octopus times the number of used processes (red triangles), the results agree with the sum of the individual Massif memory allocation.

The Diagrams in Figures 5-7 and 9-11 demonstrate results from analysis performed using the Octopus ad-hoc profiler. It has been observed for all use cases that the largest memory allocation occurs on states variable (dpsi) when a few MPI processes are used. However, as it is displayed in Figure 5, this specific matrix is efficiently distributed when the number of MPI processes increases, due to a proper states parallelization. The result is that the memory allocated per process for dpsi decreases and becomes even lower than memory allocated for other matrices, as it can be observed in Figure 6.

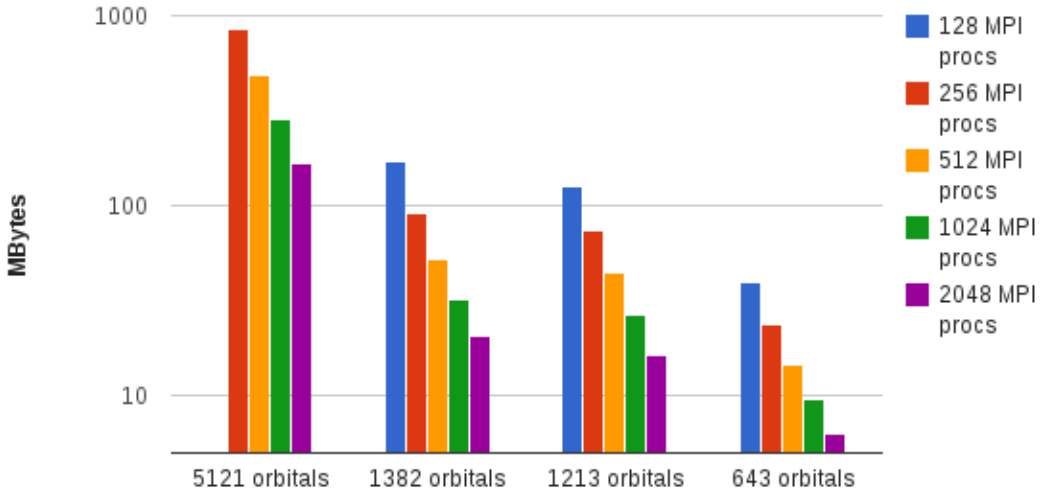


Figure 5: Memory allocation for *dpsi* pointer to wave functions with varying number of orbitals and number of MPI processes (logarithmic scale)

It has to be noted that MareNostrum III provides 2896 nodes with 1800 MB of RAM per task. Due to this limitation we were not able to run the native LCAO implementation for the 5121 orbitals use case with 128 processes, as a larger amount of memory was needed. However, MareNostrum recently upgraded the memory installed in a number of nodes, providing 128 nodes (2048 cores) which offer 3812 MB per task and (64 nodes) 1024 cores which offer 7812 MB per task. These upgraded nodes allow analyses of use cases that require a larger amount of memory.

An overview of the largest variables in terms of memory allocation is displayed in Figure 6, and Table 1. The results refer to tests that were performed using 2048 MPI processes for all use cases. When large systems are analyzed, it can be shown that increasing the number of MPI processes leads to a decrease of memory allocated for the *dpsi* matrix, while the Hamiltonian and Overlap matrices now allocate the largest amount of memory.

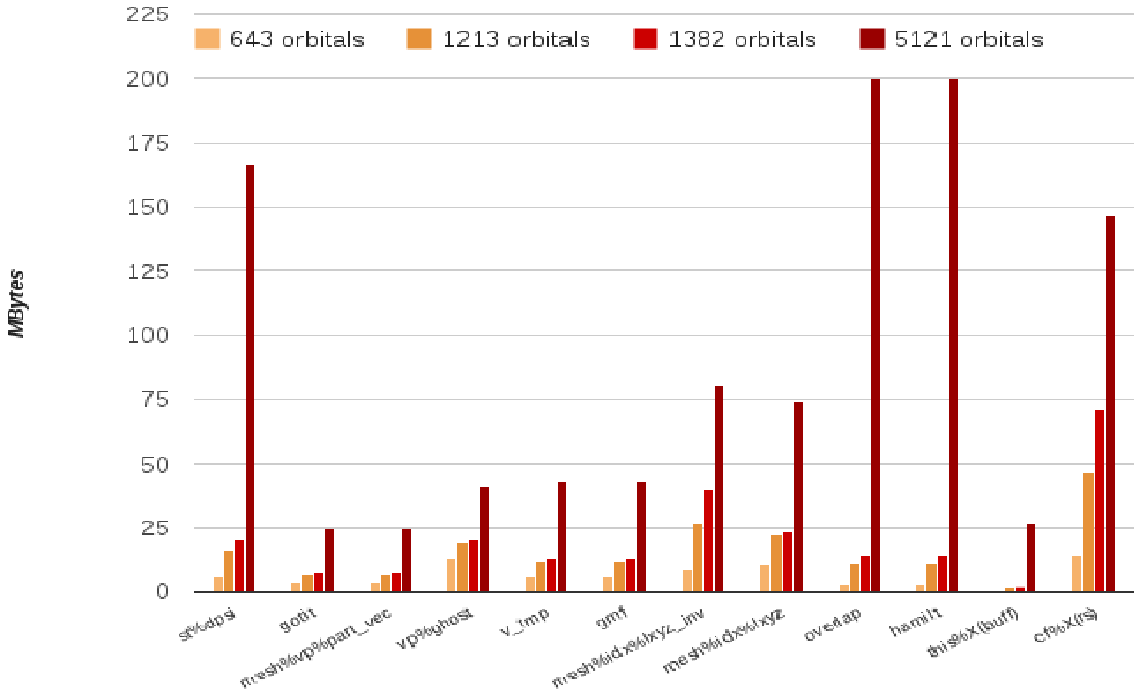


Figure 6: Largest allocated variables overview for varying number of orbitals, using 2048 MPI processes.

Table 1: Largest allocated variables overview for varying number of orbitals, using 2048 MPI processes. For each variable the corresponding source file is displayed where the memory allocation function has been called.

largest variables allocated	source file	643 orbitals (MB)	1213 orbitals (MB)	1382 orbitals (MB)	5121 orbitals (MB)
cf%X(rs)	cube_function_inc.F90	14.592	47.258	71.571	147.073
this%X(buff)	lcao_inc.F90	0.431	1.756	2.109	26.427
hamilt	lcao_inc.F90	3.154	11.226	14.572	200.078
overlap	lcao_inc.F90	3.154	11.226	14.572	200.078
mesh%idx%lxyz	mesh_init.F90	10.997	22.267	23.861	74.595
mesh%idx%lxyz_inv	mesh_init.F90	8.81	26.907	40.029	80.478
gmf	cube_function_inc.F90	5.984	12.188	13.16	42.789
v_tmp	par_vec_inc.F90	5.984	12.188	13.16	42.789
vp%ghost	par_vec.F90	13.297	19.603	20.506	40.975
mesh%vp%part_vec	mesh_init.F90	3.666	7.422	7.954	24.865
gotit	mesh_partition.F90	3.666	7.422	7.954	24.865
st%dpsi	states.F90	6.336	16.538	20.467	166.169

The important increase of memory allocation for Hamiltonian and Overlap matrices, as larger systems are studied is displayed in Figure 7. It is clear that increasing the number of processes when employing native LCAO implementation does not affect the matrices' size. Memory allocation increases in quadratic manner as the number of orbitals of the system increases.

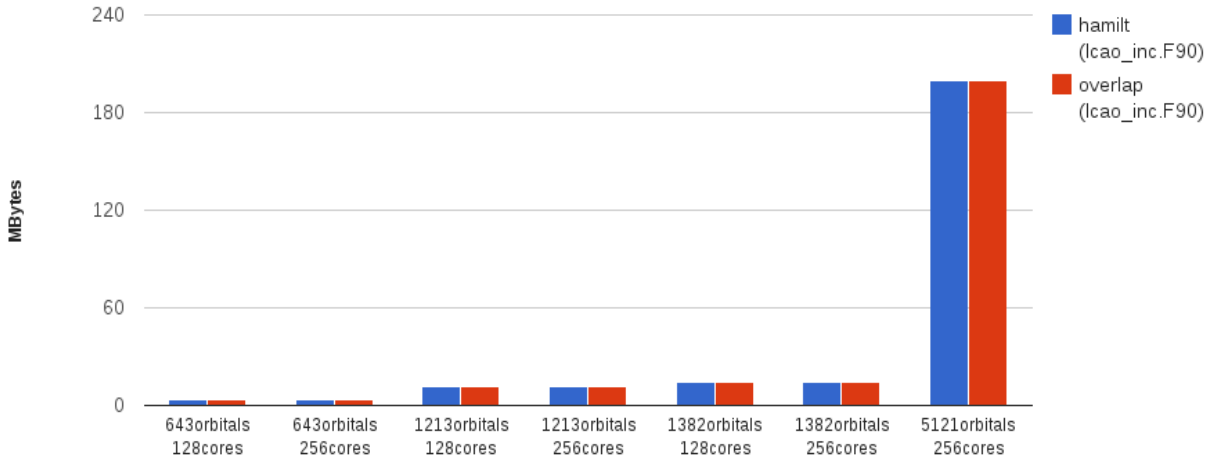


Figure 7 : Size of memory allocated for Hamiltonian and Overlap matrices for different numbers of cores and orbitals.

Memory allocation for Hamiltonian and Overlap matrices is significantly decreased when employing LCAO ScaLAPACK alternative implementation. Matrices are distributed over MPI processes, eliminating the memory allocation problem which was introduced with the increase of orbitals. As an example, memory allocation for Hamiltonian and Overlap matrices for 5121 orbitals use case and 128 processes has dropped down to approximately 1.87 MB per process.

Heap memory usage for 64 MPI processes and 1382 orbitals when employing LCAO ScaLAPACK implementation is displayed in Figure 8. Total memory consumption per process has been decreased in comparison to native LCAO implementation. LCAO implementations are compared in terms of memory consumption in Figure 9.

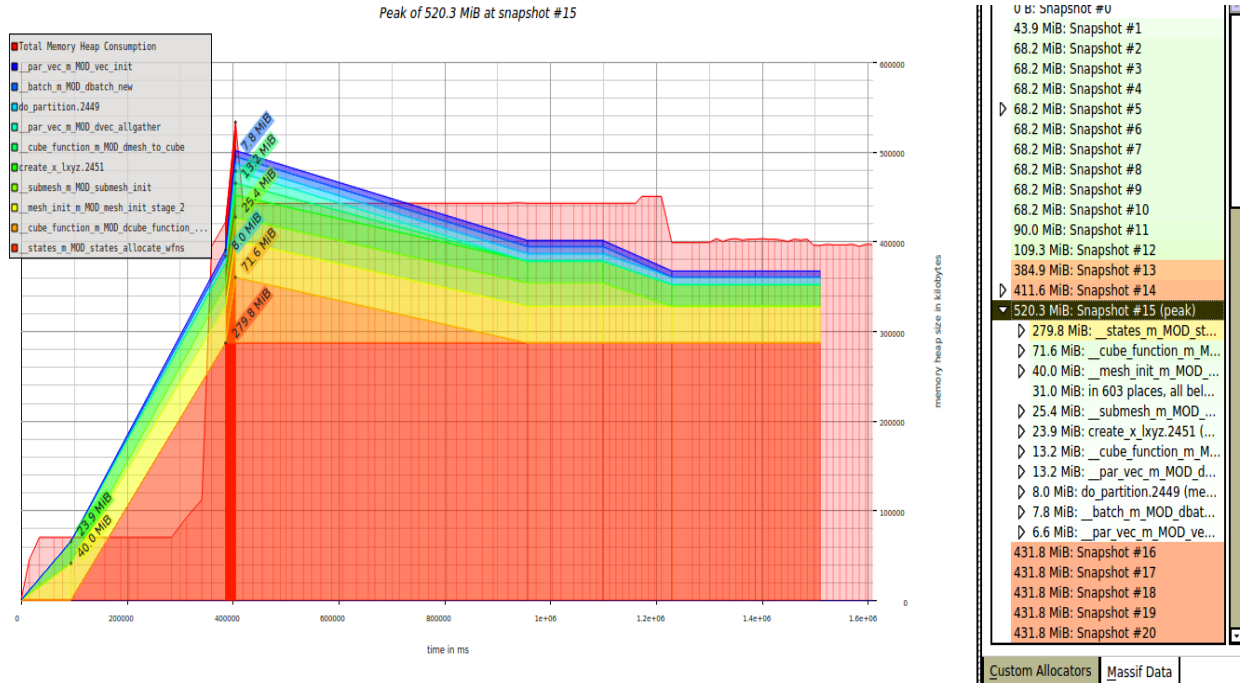


Figure 8: Visualization of Valgrind Massif heap profiler results, using Massif Visualizer. Results were obtained from an Octopus execution with 64 processes on 1382 orbitals use case using LCAO ScaLAPACK implementation. (Peak of 520.3 MiB)

Regarding LCAO native and LCAO ScaLAPACK implementations the memory consumption is

measured per process. However, it should be noted that when applying LCAO LAPACK alternative implementation, only the root MPI process stores the Hamiltonian and Overlap matrices. Therefore, considering this specific implementation, the maximum memory allocation is displayed in Figure 9, which refers only to the root process. It is observed that, although LCAO LAPACK alternative differs from native implementation and allocates less amount of memory when applied in larger systems, this approach is also severely limited by maximum memory per task, due to Hamiltonian and Overlap matrices memory allocation by the root process. On the contrary, the LCAO ScaLAPACK implementation achieved a decrease in memory allocation per process and, as a result, it has been proved to allow the study of larger systems.

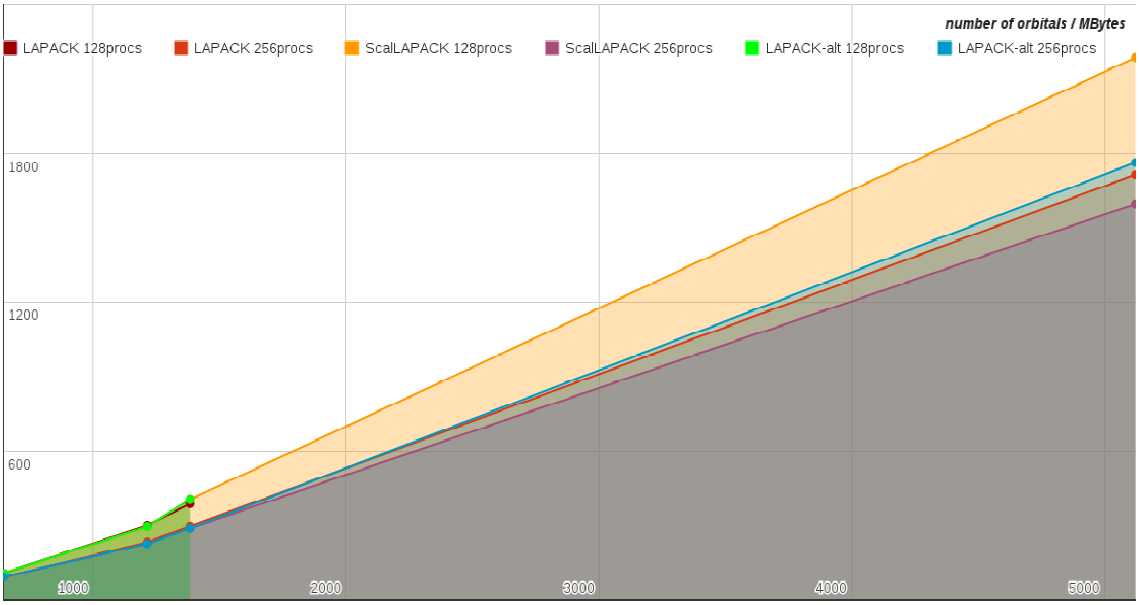


Figure 9: Memory consumption per process for LCAO implementations with varying number of processes and number of orbitals.

As this project aims at optimizing the memory usage in order to make possible running Octopus for many states systems on HPC infrastructures, many of the benchmark tests that were performed used few nodes to extract conclusions from the behaviour of the code when the memory provided is not large enough. This practice intended to simulate the memory allocation effect on bigger use cases in terms of the number of orbitals.

Scaling of the application

Results presented in Table 2 have been acquired from benchmark tests performed on 5121 orbitals use case. Octopus initialization and LCAO steps do not demonstrate good scaling behaviour.

Table 2 : Wall time and speed-up from benchmark tests performed on 5121 orbitals use case.

Number of cores	Hamiltonian / Overlap matrix size (MB)	dpsi matrix size (MB)	Wall clock time (sec)	Speed-up vs the first one	Number of MareNostrum III Nodes
256 LAPACK	200.078	854.85	5216.277	1.000	16
512 LAPACK	200.078	482.655	4938.483	1.056	32
1024 LAPACK	200.078	285.983	4098.483	1.272	64

2048 LAPACK	200.078	166.169	4351.960	1.198	128
alternative implementations					
128 ScaLAPACK	1.875	1472.730	467.293	11.162	8
256 ScaLAPACK	< 1.8	854.85	553.208	9.429	16
256 LAPACK only root MPI process	200.078 (allocated only by root process)	854.85	564.02	9.248	16

Significant performance improvement is observed when applying alternative LCAO in comparison to native implementation. The Diagram in Figure 10 demonstrates the scaling behaviour of the native LCAO implementation in comparison to the parallel implementation using the ScaLAPACK library when the number of orbitals is increased.

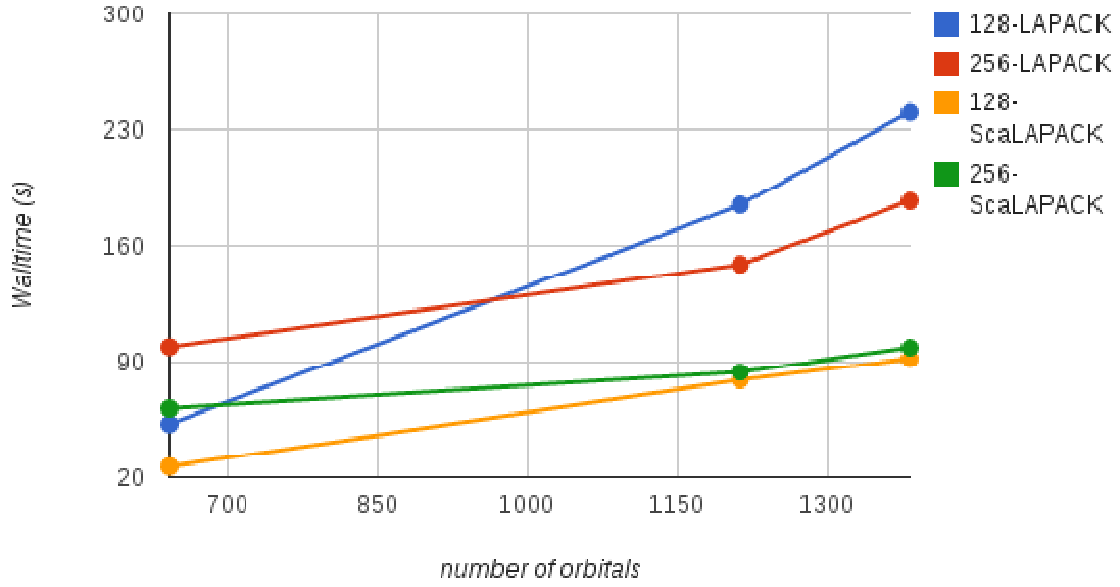


Figure 10: Performance improvement of LCAO ScaLAPACK in comparison to native LCAO implementation.

This study mainly focused on the first steps of Octopus execution: initialization and LCAO. SCF iterations were not extensively performed. The diagrams in Figure 11 aim to demonstrate how wall time is shared between the LCAO step and the system initialization, when Octopus is only executed up to the LCAO step completion. It is clear that the percentage of wall time for these two steps is affected by the number of orbitals of the system and the number of MPI processes. System initialization consumes higher percentage of total wall time when more MPI processes are used. In large systems, however, the LCAO step dominates over the total wall time. As it has been already mentioned, 5121 orbitals use case did not run on MareNostrum III due to larger memory requirements than provided by the system .

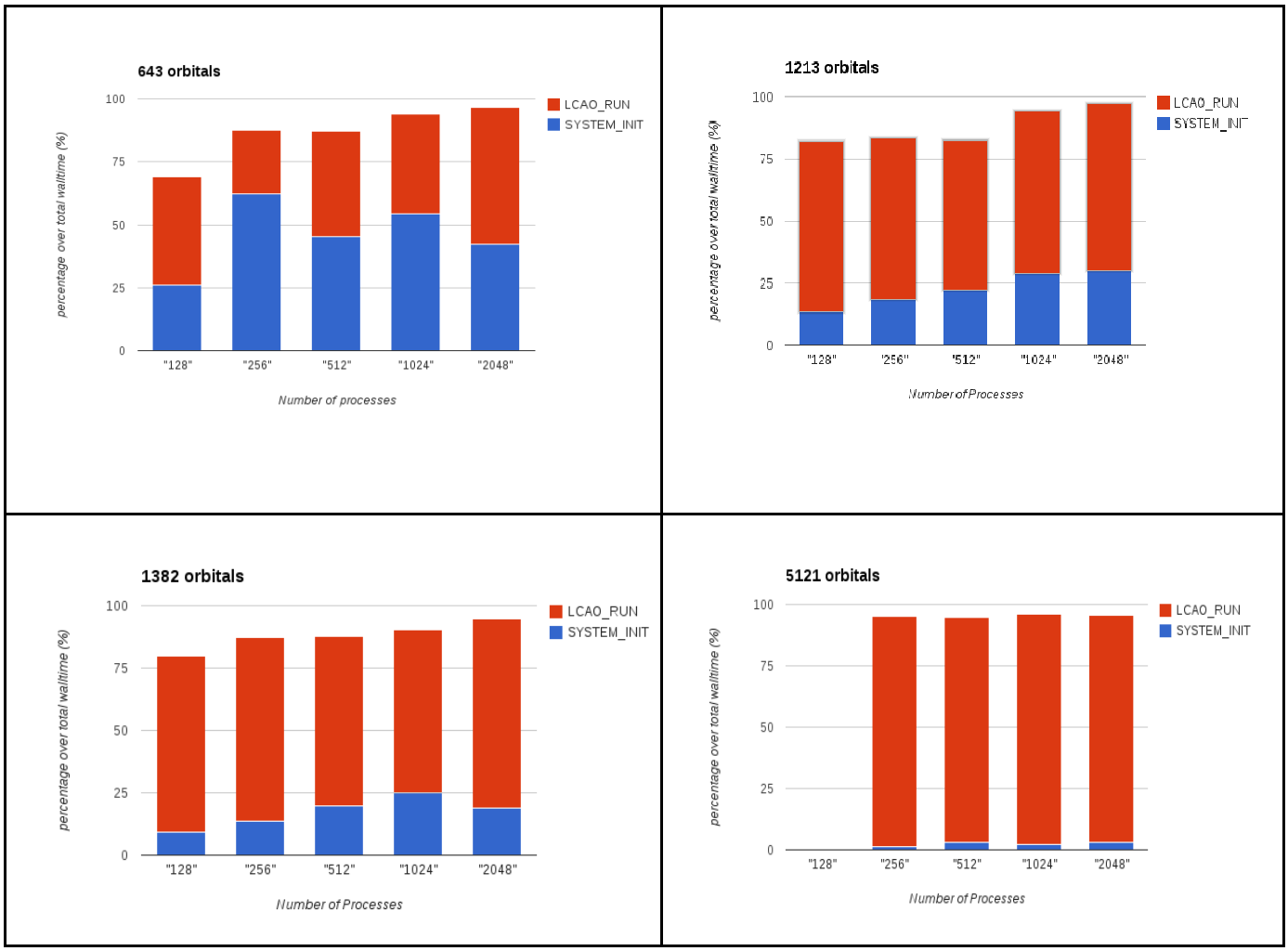
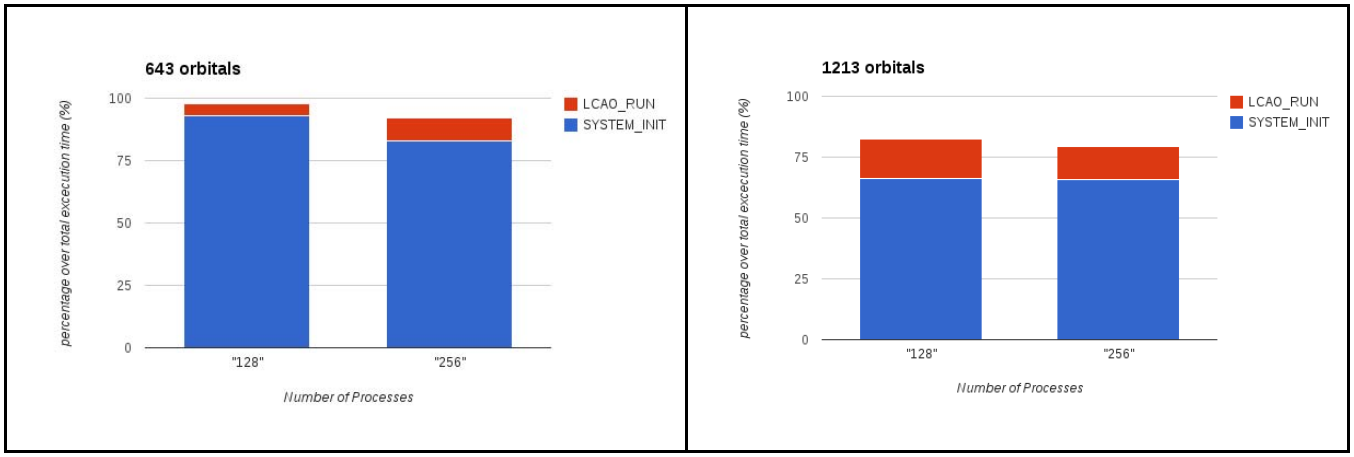


Figure 11: Percentage over total wall time dedicated for LCAO_RUN and SYSTEM_INIT with varying number of orbitals and processes for LCAO implementation.

This behaviour changes when LCAO ScaLAPACK alternative is applied (Figure 12). System initialization step consumes the same amount of time as in LCAO native implementation, while wall time for LCAO step decreases significantly, leading to an important decrease of total wall time.



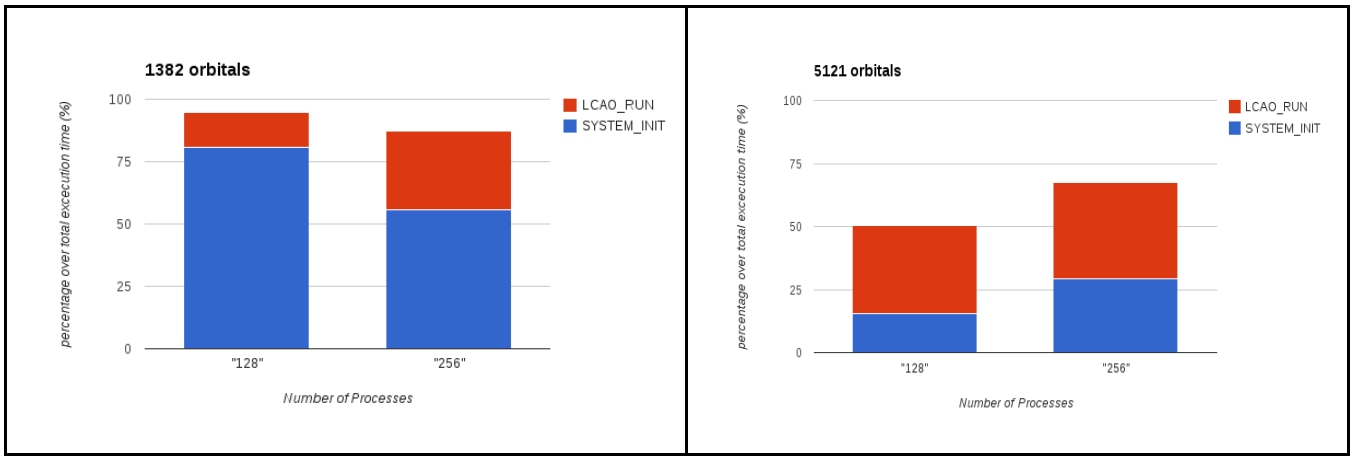


Figure 12: Percentage over total wall time dedicated for LCAO_RUN and SYSTEM_INIT with varying number of orbitals and processes for ScaLAPACK alternative LCAO implementation.

Even though this initialization time is not negligible, this is is only a small percentage in a real run, which could easily last few days.

Remaining time spent, missing from percentages in Figures 11 and 12 is used for calculation of total forces on the ions created by the electrons. SCFCalculateForces variable controls whether the forces on the ions are calculated at the end of a self-consistent iteration. The default value of SCFCalculateForces variable is true, unless the system has only user-defined species. Many initialization steps have to be performed, such as the grid creation and its distribution, creation of a stencil, the selection of the best FFT strategy, and so on, even though these steps do not consume a great amount of time.

Future work

Sparse matrix storage techniques might be considered in the future as an alternative solution in order to study large systems. LAPACK and ScaLAPACK libraries do not provide routines for sparse matrices. As a result a different implementation may be considered. For example, ARPACK library, which provides routines for large sparse matrices computations, could be used.

Table 3 demonstrates the densities of the matrices with varying numbers of orbitals. When number of orbitals or the atomic system is increased, Hamiltonian and Overlap matrices become less dense.

Table 3: Hamiltonian and Overlap matrices densities for 643 and 5121 orbitals

Number of orbitals	Number of stored elements in triangular matrices	Number of non-zero elements : Hamiltonian matrix	Hamiltonian matrix Density (%)	Number of non-zero elements : Overlap matrix	Overlap matrix Density (%)
643	207046	192933	93.18	169536	81.88
5121	13114881	2776861	21.17	1941723	14.80

Conclusions

We have shown that storage requirements on native LCAO implementation for Hamiltonian and Overlap matrices become a problem as the number of orbitals of the system increases. Even greater amount of memory per process is allocated for dpsi states pointer to wave functions matrix. However, benchmark tests have shown that this particular matrix is partitioned properly over the MPI processes. Therefore, the size of allocated memory per process drops down when the number of MPI processes is increased.

The memory allocation obstacle, as it has been demonstrated through the benchmark tests presented in this study, can be overcome when using the parallel alternative implementation of LCAO, enabling Octopus to run for larger systems. Performance optimization has also been achieved. Developments in this project will allow to run bigger simulations and, therefore, more interesting systems.

References

[1] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. Marques, E. Gross, and A. Rubio, "Octopus: a tool for the application of time-dependent density functional theory," *Physica Status Solidi B Basic Research,* vol. 243, pp. 2465–2488, Apr. 2006. http://www.tddft.org/programs/octopus. 17

[2] http://www.bsc.es/marenostrum-support-services/mn3

[3] Joseba Alberdi-Rodriguez. Memory usage in Octopus, December 3, 2014

[4] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. In Workload Characterization, 2006 IEEE International Symposium on, pages 2{2. IEEE, 2006.

[5] Alberdi-Rodriguez, Joseba. "Analysis of performance and scaling of the scientific code Octopus", page 27, September 16, 2010

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-312763.

We acknowledge the Barcelona Supercomputing Center - Centro Nacional de Supercomputación for the computing resources.

This work was partially funded by the University of the Basque Country UPV/EHU under the grant UFI11/45, by the Department of Education, Universities and Research of the Basque Government (IT395--10 research group grant), European Research Council Advanced Grant DYNamo (ERC-2010- AdG-267374), European Commission project CRONOS (Grant number 280879-2 CRONOS CP-FP7), Spanish Grant (FIS2013-46159-C3-1-P) and Grupo Consolidado UPV/EHU del Gobierno Vasco (IT578-13).

JAR acknowledges the grants of the University of the Basque Country UPV/EHU and the grant from the University of Coimbra DRH 3614221, PRACE-3IP project - task WP7.