Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

OpenMP optimisation of the eXtended Discrete Element Method (XDEM)

Pedro Ojeda-May[a], Jerry Eriksson[a], Alban Rousset[b], Xavier Besseron[b], Abdoul Wahid Mainassara Chekaraou[b], Bernhard Peters[b]

[a]*High Performance Computing Center North (HPC2N), MIT Huset, Umeå Universitet, 90187 Umeå, Sweden*
[b]*Faculty of Science, Technology and Medicine (FSTM), University of Luxembourg, 1359 Luxembourg*

Abstract

The eXtended Discrete Element Method (XDEM) is an extension of the regular Discrete Element Method (DEM) which is a software for simulating the dynamics of granular material. XDEM extends the regular DEM method by adding features where both micro and macroscopic observables can be computed simultaneously by coupling different time and length scales. In this sense XDEM belongs the category of multi-scale/multi-physics applications which can be used in realistic simulations. In this whitepaper, we detail the different optimisations done during the preparatory PRACE project to overcome known bottlenecks in the OpenMP implementation of XDEM. We analysed the Conversion, Dynamic, and the combined Dynamics-Conversion modules with Extrae/Paraver and Intel VTune profiling tools in order to find the most expensive functions. The proposed code modifications improved the performance of XDEM by ~17% for the computational expensive Dynamics-Conversion combined modules (with 48 cores, full node). Our analysis was performed in the Marenostrum 4 (MN4) PRACE infrastructure at Barcelona Supercomputing Center (BSC).

1 Introduction

The eXtended Discrete Element Method (XDEM) is an extension of the regular Discrete Element Method (DEM) with the additional feature that both micro and macroscopic observables can be computed simultaneously by coupling different time and length scales [1-2]. Thus, this software fell in the category of multi-scale/multi-physics applications which can be used in large realistic simulations such as combustion of materials and drug design. The different multi-physics components in XDEM are organised in a modular layout, i.e. Conversion, Dynamics, and Computational Fluid Dynamics modules.

XDEM has already a good support for High Performance Computing architectures where it has shown to scale with more than 500 cores. The high level of parallelism in XDEM is achieved by using several paradigms such as, distributed memory (MPI), shared memory (OpenMP) and a hybrid of them.

In the present project, we targeted the OpenMP implementation which is one of the major bottlenecks in a typical simulation especially for the Conversion module. This was detected previously by the developers using the timings from different modules. In order to achieve this target, several steps were proposed:

- Initial profiling analysis of the OpenMP implementation to identify the bottlenecks. The profiling tools were Extrae/Paraver and Intel VTune.
- Optimise the following modules in four phases
- Optimisation of Conversion module (phase I)
- Optimisation of Dynamic module (phase II)
- Optimisation of Conversion and Dynamic modules (phase III)
- Optimisation of Conversion, Dynamic and CFD modules (phase IV)

Some of the phases overlap because there are several functions that are used in more than one phase.

2 Test case

In the present work we studied a biomass 3D example (see Fig. 1) [2]. This example is relevant due to the global warming of our planet which is pushing us to find other sources of renewable and alternative energy. Biomass as a renewable carbon-based energy source is a sustainable alternative for generating power and therefore continues to grow in popularity to reduce fossil fuel consumption for environmental and economic benefits. Numerical simulations are therefore used in order to anticipate and improve the efficiency and optimisation of harmful gas emission.

The model we propose to predict is the entire biomass process. This process is very challenging because it involves multi-scale, multi-phase and multi-species phenomena including bed motion, turbulence, chemical reactions and heat radiation. The fuel bed behaviour including its motion and different conversions are solved with XDEM (XDEM Dynamics and XDEM Conversion). This model contains 350,000 particles and 8470 CDF cells, and it requires to simulate 400 seconds to reach a steady state in the furnace.

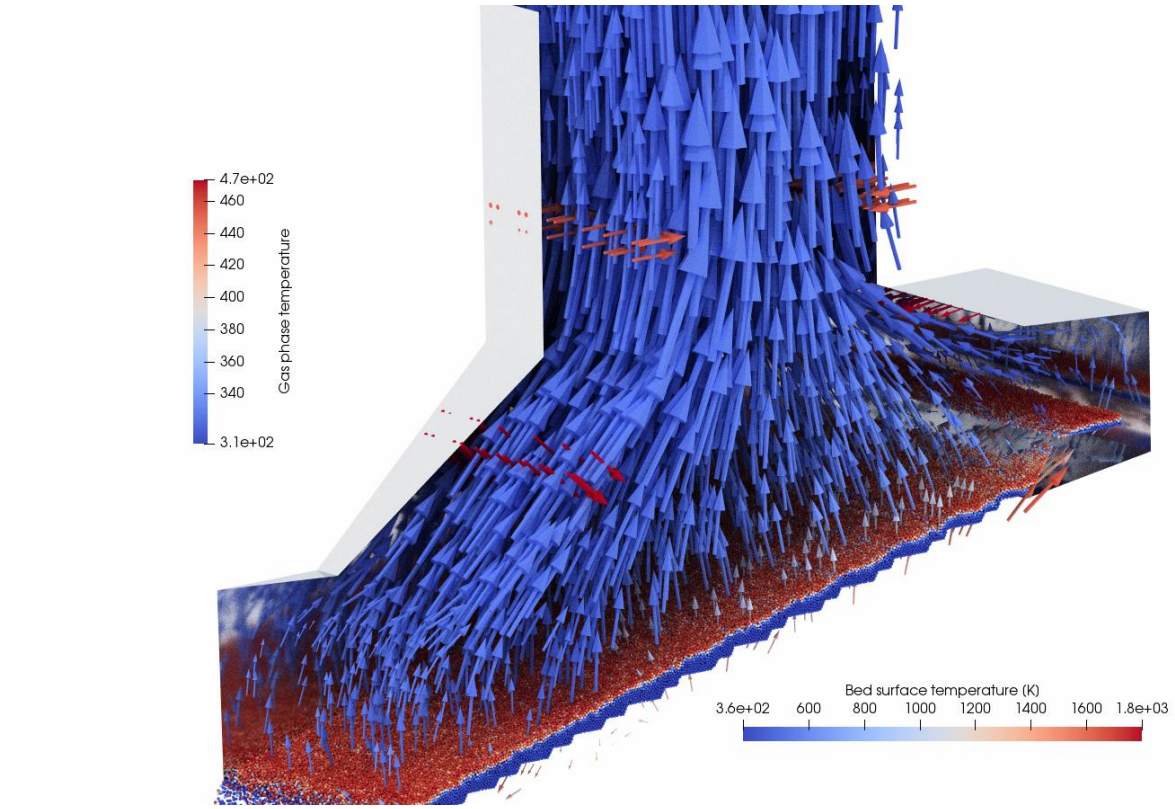

Figure 1. Biomass3D test case showing the fuel bed and gas phase, temperature scales are provided in the boxes

3 Code optimisations

For the present study, only the OpenMP threaded version of XDEM [1] was benchmarked and optimised. The MPI implementation would be the target for a future PRACE project, this implementation was also considered in the past in other initiatives [3]. The benchmarking was done at the commits **819a18f** and **fab6795** for the master and PRACE branches, respectively in the private GitLab repository of XDEM[2]. Results for benchmarking are provided in Figs. 4-9 and accompanying tables.

Several optimisations were done in the present project. We used Intel VTune (v. 2019)[4] and Extrae (v. 3.7.1)[5] profiling tools to investigate the performance of the most expensive routines. We created a branch (po-prace) at the level of commit **5fe8f60a** and made the first commit **8da7d30** for the initial modifications.

An analysis using Extrae/Paraver showed that code efficiency, measured by the useful IPC metrics, could be hindered by the extensive use of OpenMP *locks*, see Fig. 2 a) and b). We also found considerably large waiting time regions by monitoring IPCs in a) (in black). A full node with 48 cores was used for this analysis. Instead of using locks, we explored the possibility of using OpenMP *atomic* directives but the data structures involved were complex and that would require extensive code modifications. As an alternative, we used OpenMP *critical* regions

but the performance of the code decreased considerably (-50%). Although the use of locks is expensive, they seem to be necessary for the simulations (communication with the development team).

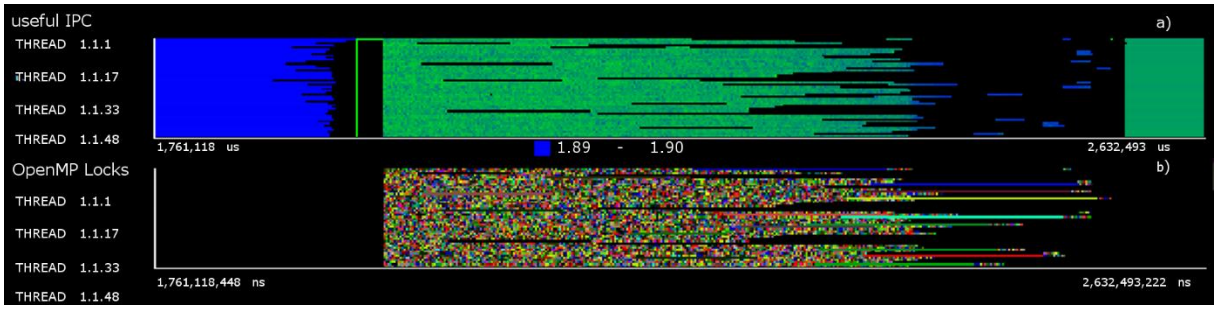

Figure 2. Initial profiling analysis with Extrae/Paraver showing the useful Instructions per Clock cycle (IPC) (a) and the OpenMP locks (b) for a cut region of the collected trace. In (a), blue colour represents high while green low IPC. There were also large waiting time regions in black. In (b) the addresses of memory for the locks are displayed with different colours.

A typical SLURM batch script for collecting traces with Extrae looks as follows:

```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --cpus-per-task=48
#SBATCH --time=00:15:00

ml purge
ml gcc/8.1.0
ml EXTRAE/3.7.1
ml openmpi/4.0.1
ml cmake/3.15.4

export DPM_DIR=./xdem-source/data
export OMP_NUM_THREADS=48
export OMP_PROC_BIND=true
export OMP_PLACES=cores

source /apps/BSCTOOLS/extrae/3.7.1/openmpi_4_0_1/etc/extrae.sh
export EXTRAE_CONFIG_FILE=extrae.xml
export LD_PRELOAD=${EXTRAE_HOME}/lib/libomptrace.so

./xdem/build_OMP_relWithDebInfo/XDEM_Simulation_Driver \
./biomass3D/biomass_3D_Conversion_2.h5  --output-disable-all --max 100
```

The initial profiling analysis (see Fig. 3) we obtained from VTune showed that the function ***pow()*** is being called many times and it was the major contributor (~14%) of the total execution time. We inspected the code and detected that this function was used to compute products such as $pow(x, b) = x^b$ (with b integer), in this case, we proposed to use explicit multiplication if $b=2$, and for $b > 2$ we suggested to use additional auxiliary variables to compute partial products. Arithmetic operations with a fractional value of b where substituted with integer operations (**8da7d30**, **fb00d3c**, **2cc1058**):

$$std :: pow(x, 1.5) \rightarrow std :: sqrt(x * x * x)$$

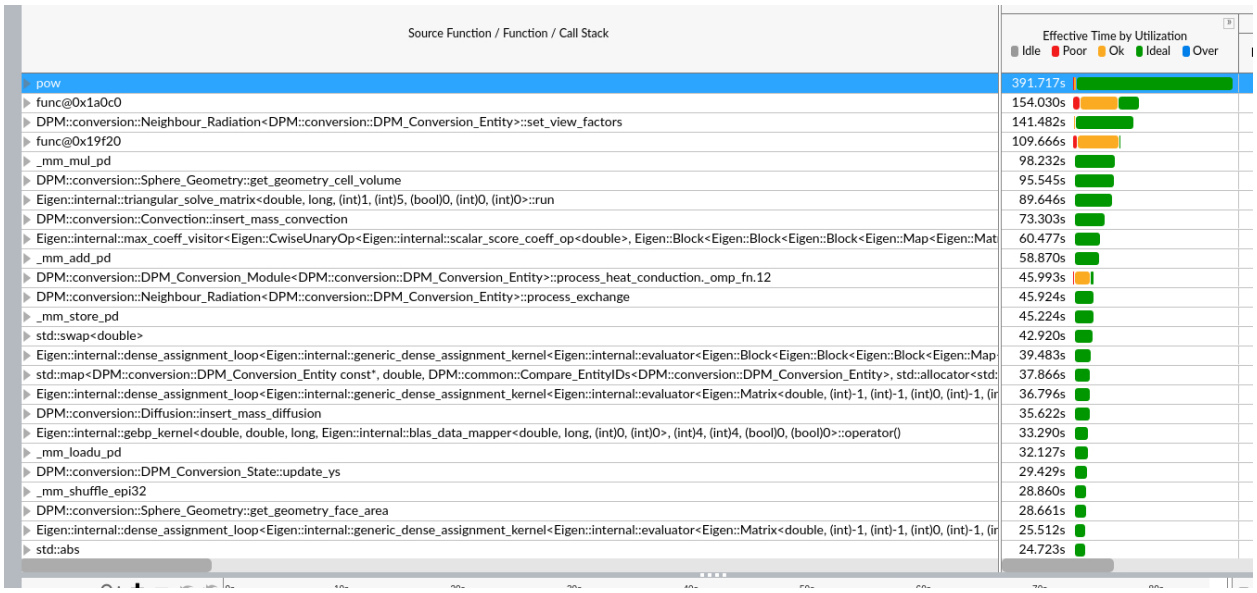

Figure 3. Initial profiling analysis with Intel VTune showing the most expensive function calls

Exponential expressions such as the Arrhenius factor were transformed to logarithmic expressions (see ***calculate_rate()*** function, **f8c0f32, 47e772f**, **e63bf9e**):

$$k_r * std :: exp\left(\frac{-E_a}{k_b T}\right) \rightarrow log k_r - E_a/k_b t$$

We observed a significant reduction in the total simulation time by implementing these arithmetic modifications (~10% of the total time).

Creation and initialisation of structures was taken out of loops when possible, as an example in the function ***set_view_factors()*** (**ce6f97d**), see Schematics 1. It is important to notice that the overheads in the loops was observed with the debug version (compiler flags: -O2 -g) of XDEM where statistics can be collected by profiling tools. A well instructed compiler (through compiler flags) could optimise the use of structures in loops depending on the complexity of them.

Initial version (Master)	PRACE optimisation

`for( ; it_beg != it_end; ++it_beg){`
`const DPM_ENTITY* particleB;`
`...`
	`const DPM_ENTITY* particleB;`
	`for( ; it_beg != it_end; ++it_beg){`
	`...`

Schematics 1. Simplification of loops in ***set_view_factors()***

We also observed that some checking functions such as ***DPM_ERROR()*** slowed down the code and some of them are now used only in the debug version ***DPM_ASSERT_OR_ABORT_DEBUG()*** (**5401f68**, **79932da**, **2dd0b24**).

In function ***process_heat_conduction()***, an overhead was detected which was traced back to the opening of two consecutive parallel regions. We rearranged both regions so that they could fit into a single parallel region where data initialisation, without dependencies, was performed outside the parallel region (**3b672bf**).

Some functions like ***update_ys()***, made use of complex loops definitions, we transformed these loops into lightweight loops by defining the iterators outside them (**748b46f**, **7d6f7de**), see Schematics 2. As mentioned before for the usage of structures in loops, it is up to the compiler and optimisation flags whether it would optimise or not complex loops definitions, it is not guaranteed (compiler dependent) that the compiler can inline code, for instance.

Initial version (Master)	PRACE optimisation

`for ( DPM::common::Species_ID_set::const_iterator specie_pos =fuel.solid_species_set.begin();`
 	
`specie_pos != fuel.solid_species_set.end(); specie_pos++ )`
 	
`{ ...`
	`DPM::common::Species_ID_set::const_iterator specie_pos;`
	
	`DPM::common::Species_ID_set::const_iterator begin1=fuel.solid_species_set.begin();`
	
	`DPM::common::Species_ID_set::const_iterator endin1=fuel.solid_species_set.end();`
	
	`for ( specie_pos = begin1; specie_pos != endin1; specie_pos++ )`
	
	`{..`

Schematics 2. Simplification of iterator variables in the loops of **update_ys()**

This method proved effective especially when the loops were heavy and the iterators involved complex objects (~1%, using the times for this function only, before and after optimisation). However, we did not observe a performance improvement when the loops involved lightweight computations. This analysis was done with the debug version of XDEM (compiler flags: -O2 -g) , the performance improvement was monitored with the Effective Time by utilisation from VTune.

The functions ***get_grid_size()***, ***cell_grid.size()***, ***cs_vec.size()*** were called many times, for instance in the function ***broadphase()***, in the initial version and they represented about 1% of the total time taken together, using the Effective Time from VTune in the debug version (reference initial time for these routines taken together was 8.6 sec.). In order to avoid these calls, we defined auxiliary variables and saved the value of the ***size()*** functions instead of computing them explicitly (**be0fa87**, **160989f**, **4eaa845**, **3ddb661**, **d6cfa6c**). A similar approach was taken in other parts of the code, for instance in the function ***process_exchange()***, where functions such as ***get_particle_surface_shell_rho_cp()*** were called several times inside loops (**97ab3b5**).

We noticed that some functions are called where an auxiliary variable would make the computation faster. However, the choice of auxiliary variables could make the code less easy to interpret. This could be solved, however, with an appropriate choice of names for the auxiliary variables. Two of the functions that benefited by the use of auxiliary variables were ***insert_heat_conduction()*** *and* ***insert_mass_convection()*** (**be0fa87**, **4eaa845, f299ff7**). For these two functions we observed a performance improvement of 44% and 50%, respectively (initial times were 9 sec. and 60 sec. in the same order).

Because the code modifications we proposed in the last two paragraphs make either small improvements in performance or could affect the readability of the code, the reader could ask if it is worth to implement those modifications into the master branch. If the functions involved in these code modifications were called in a specific part of the code and they were rarely used the answer will certainly be ‘no’. However, these functions are called many times across the entire code (also in modules we did not cover in this study) which suggest that a better coding policy could be used.

4 Results

The profiling of XDEM was performed on Marenostrum 4 infrastructure at BSC using a single node which has 2x Intel Xeon Platinum 8160 chip with 24 cores per socket at 2.1 GHz [6] and 2-way SMT. In the present analysis only physical cores were considered for handling OpenMP threads. XDEM was compiled with the following *Cmake* flags:

```
cmake /gpfs/projects/pr1elo00/pr1elo03/xdem/xdem-source \
-DCMAKE_BUILD_TYPE=RelWithDebInfo \
-DEIGEN3_ROOT=./eigen/install DJEMALLOC_ROOT=./jemalloc-master/install \
-DXDEM_CONFIG=OMP -DHDF5_ROOT=…/hdf5-1.8.19-cpp/install/ \
-DBOOST_ROOT=./boost_1_71_0/install -DBoost_NO_SYSTEM_PATHS=ON \
-DBoost_NO_BOOST_CMAKE=ON -DCMAKE_CXX_FLAGS=-march=native
```

and using the following modules for compilation: gcc/8.1.0, EXTRAE/3.7.1, openmpi/4.0.1, and cmake/3.15.4. The compiling option `march=native` allows to achieve a ~10% performance improvement. A typical SLURM batch script for getting traces with VTune looks as follows, notice that for XDEM thread binding is essential to achieve a good performance of ~15% with appropriate OpenMP binding instructions (`OMP_PROC_BIND=true`, `OMP_PLACES=cores`).

```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --cpus-per-task=48
#SBATCH --time=00:15:00

ml purge
export DPM_DIR=./xdem-source/data
export OMP_NUM_THREADS=48
export OMP_PROC_BIND=true
export OMP_PLACES=cores

ml intel
ml vtune_amplifier/2019.4

amplxe-cl -collect hotspots -r ./biomass3D/profile  -- \
./xdem/build_OMP_relWithDebInfo/XDEM_Simulation_Driver \
./biomass3D/biomass_3D_Conversion_2.h5  --output-disable-all --max 100
```

Dynamics Module

For this module 1000 steps of simulation were done. The times for the runs are plotted in Fig. 4 for varying number of cores (in log-log scale), and the speedups are plotted in Fig. 5. Only a slight performance gain was observed for this module because the functions involved were not computationally expensive (the ones we approached in this study) but they involved memory management. A higher performance for this module would require a larger data structure reorganisation.

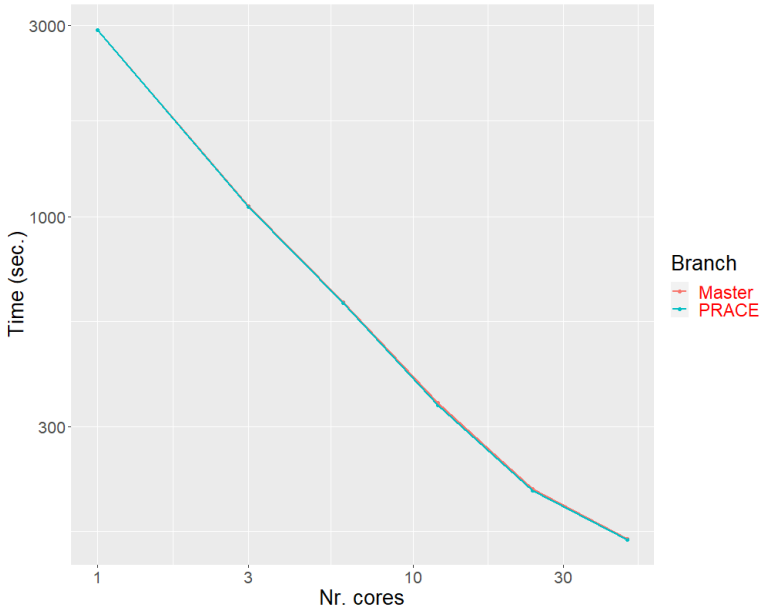

Figure 4. Timing for the Dynamics module as a function of the number of cores in log-log scale.

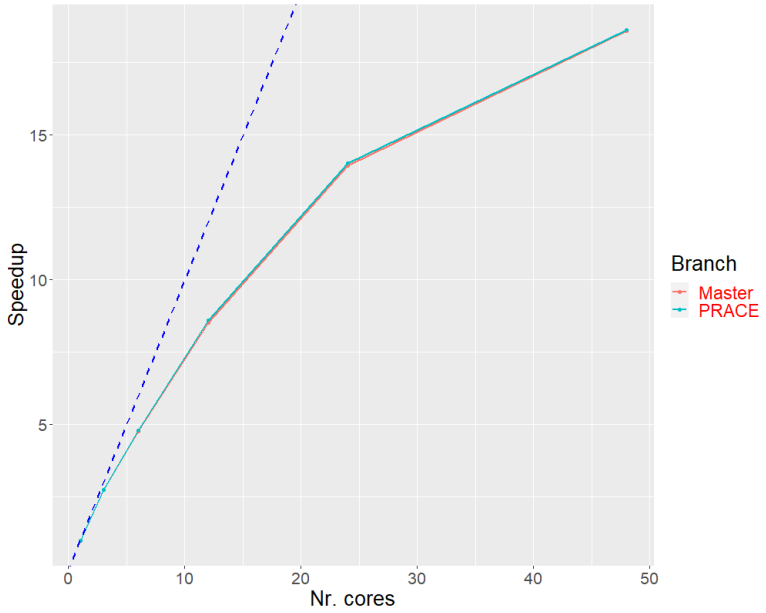

Figure 5. Speedup regarding the Master for the Dynamics module as a function of the number of cores. Perfect scaling behaviour is shown with dashed-blue line.

Conversion Module

For the Conversion module 100 steps of simulation were done. The times for the runs are plotted in Fig. 6, and the speedups in Fig. 7. A better performance gain was observed here because this module involves more functions that do actual computations.

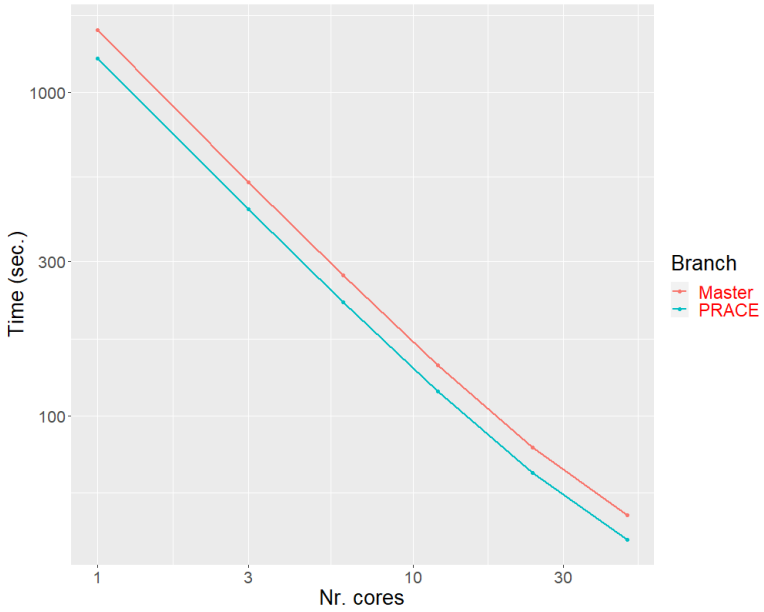

Figure 6. Timing for the Conversion module as a function of the number of cores in log-log scale.

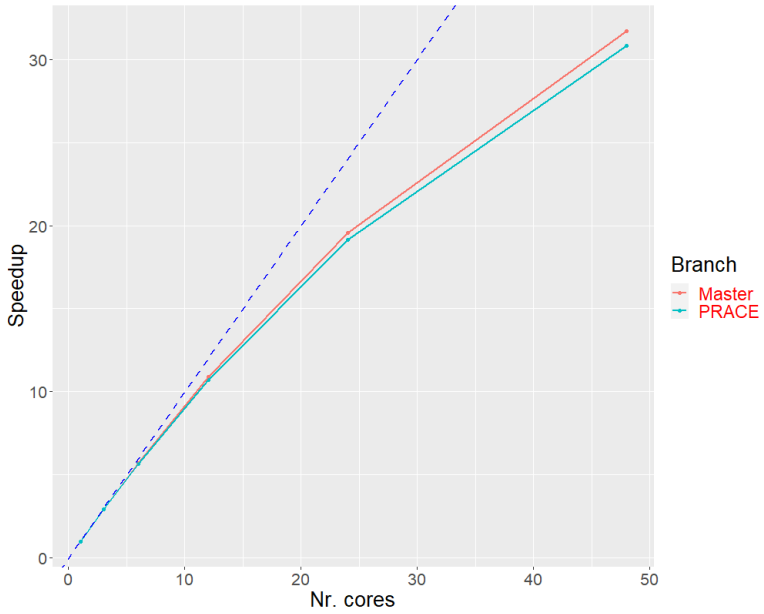

Figure 7. Speedup for the Conversion module as a function of the number of cores. Perfect scaling behaviour is shown with dashed-blue line.

Dynamics-Conversion Modules

For these coupled modules 100 steps of simulation were done. The times for the runs are plotted in Fig. 8 (log-log scale), and the speedups in Fig. 9. The performance gain in this combined simulation was similar to the one of the pure Conversion module.

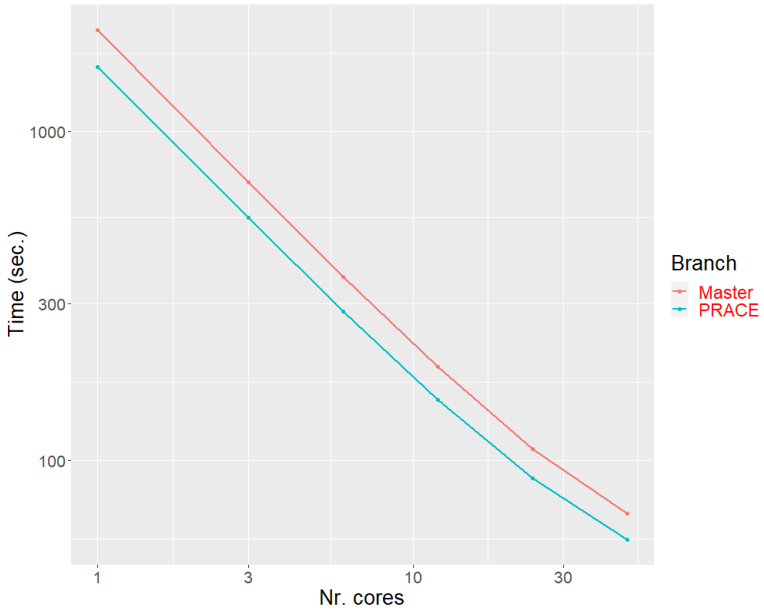

Figure 8. Timing for the Dynamics-Conversion modules as a function of the number of cores in log-log scale.

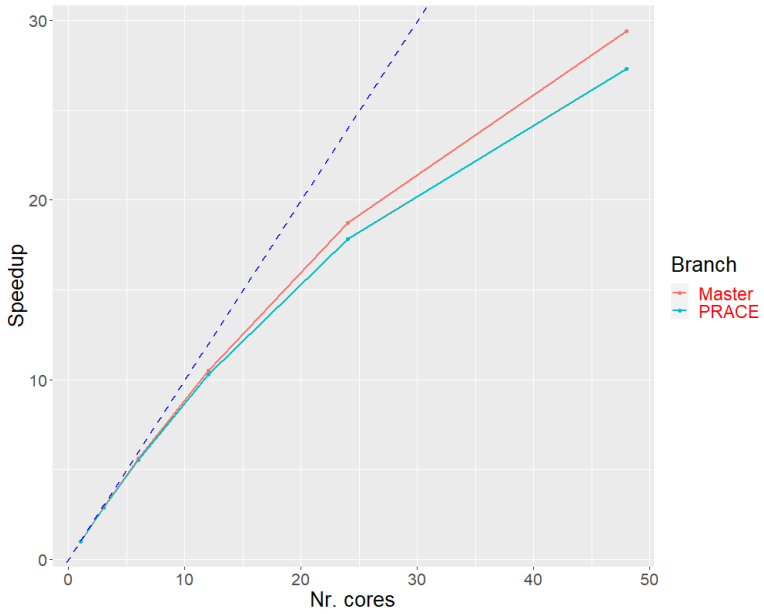

Figure 9. Speedup for the Dynamics-Conversion modules as a function of the number of cores. The perfect scaling behaviour is shown with a dashed blue line.

5 Summary and further work

The Conversion module and the coupled Dynamics-Conversion modules were sped up by ~16% (48 cores). The percentages of the total time for the most expensive loop modules for the master and PRACE branches are shown in Table 1 (using 48 cores, full node). Some of the commits from the PRACE branch have been already merged into the master branch.

Module	Master	PRACE
Broad phase	0.32	0.33
Narrow phase	0.13	0.15
Apply bed surface	0.59	0.64
Apply heat conduction	4.24	4.51
Apply heat radiation	2.95	**2.07**
Integration conversion	66.38	**58.88**

Table 1. Performance evaluation of the different modules of XDEM for 48 cores and for both Master and PRACE branches (in percentages %, wall-clock times for the simulations were 69 sec. and 57.5 sec. respectively).

A list of the optimised functions and the corresponding times (in sec.) can be seen in Table 2.

Function name	Master	PRACE
pow()*	391	40
set_view_factors()	141	20
get_geometry_cell_volume()	70	10
insert_mass_convection()	60	30
process_exchange()	33	24
insert_mass_diffusion()	29	11
get_geometry_face_area()	24	12
update_ys()	23	19
get_geometry_cell_delta()	16	4
apply_dynamics_models()	13	6
get_cell_delta()	9	3

Table 2. Timings (in sec.) for the most representative optimised functions. *In the case of the *pow()* function the statistics disappeared but it was replaced by *exp()* and *log()* times. Thus, the sum of these times was used for *pow()*.

We also explored the following aspects in the code but the results showed either a large fluctuation in the timings or in the worst case the total simulation time increased. There could be multiple reasons for that, for instance the MN4 configuration, XDEM internal data management, thread binding affinities, among others. A more careful analysis using would be desired here and we only mention the following points as possible targets for optimisation in the future:

- guided scheduling (**b2d3384**, **72aa30b**, **349ac9a**). We tried guided scheduling and we observed that for some runs a performance gain of ~15% could be obtained but in other cases a loss of the same magnitude was observed. The reason for this could be the OpenMP locks mechanism that is extensively used in this code but it is necessary and optimal for this type of simulations.

- simplified loops in lightweight functions like *update_Rmgas()* (**805bf9a**) (< ~1% improvement).
- using private variables instead of arrays to keep track of the threads chunk sizes (**24fafc3**, **b482aa1**, **12daace**)
- C++ atomic variables (**f0b5f6c**, **7aa0573**). This approach gave better results only when the number of cores was 48 (~5%). With less than < 24 cores, the code displayed a decrease in performance (-15%).
- OpenMP locks. There are many calls to locks as revealed by the Extrae traces we obtained. These locks are necessary due to the manner in which data is organised. In general, locks create extra overhead. We tried critical regions and atomic directives but we did not observe any performance gain. A different data organisation could help here.
- We detected functions in the Eigen library which used complex data types as arguments such as *conj()*. However, the simulations only require real data types. This is worth it to evaluate because functions like *conj()* are more expensive than plain multiplications of real numbers. We did not profile this part because this is an independent library of Eigen, we only noticed that the function *conj()* was used where it was not necessary.

Note: all percentages mention for the functions are computed using the times before and after the proposed optimisation.

References

[1] Checkaraou, A. W. M., Rousset, A., Besseron, X., Varrette, S., & Peters, B. Hybrid MPI+ openMP Implementation of eXtended Discrete Element Method. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 450-457. IEEE (2018)

[2] GitLab repository of XDEM: https://gitlab.uni.lu/LuXDEM/xdem-source/-/wikis/XDEM_installation_guide

[3] https://pop-coe.eu/ (BSC_AR_1)

[4] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html

[5] https://tools.bsc.es/extrae

[6] https://www.bsc.es/marenostrum/marenostrum/technical-information

[7] Mainassara Chekaraou, A. W., Rousset, A., Besseron, X., Peters, B., Galletti, C., Gallo, M. G., & Sansone, F. Detailed Numerical Three-dimensional and Transient Analysis of a Grate Firing Combustion Process by Innovative High Performance Computing (2020).

[8] Peters, B., Baniasadi, M., Baniasadi, M., Besseron, X, Estupinan, A., Mohseni, M., Pozzetti, G., XDEM multi-physics and multi-scale simulation technology: Review of DEM–CFD coupling, methodology and engineering applications, Particuology, **44**, 176-193 (2019).

[9] Peters, B., Baniasadi, M., Baniasadi, M. The eXtended Discrete Element Method (XDEM): An Advanced Approach to Model Blast Furnace. Chapter 7, IntechOpen (2018).

[10] Peters, B., Besseron, X., Dziugys, A., Estupinan, A., Hoffmann, F., Michael, M., Mouhmadi, A., Vogel, F. Die Extended Discrete Element Method (XDEM) für multiphysikalische Anwendungen. ANSYS Conference Rosengarten, Mannheim (2013).

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. We acknowledge the computational resources provided by the High Performance Computing Center North (HPC2N) cluster at SNIC and the PRACE Research Infrastructure resource MareNostrum4 at Barcelona Supercomputing Center (BSC).

6 Appendix

Master branch					
Number of cores	Wall clock time	Speed-up vs the first one	Speed-up vs first one master	Number of Nodes	Number of threads
1	2921.5	1.0	1.0	1	1
3	1061.8	2.7	2.7	1	3
6	612.2	4.7	4.7	1	6
12	343.1	8.5	8.5	1	12
24	209.5	13.9	13.9	1	24
48	157.0	18.6	18.6	1	48
PRACE branch					
1	2915.7	1.0	1.0	1	1
3	1060.2	2.7	2.7	1	3
6	608.2	4.7	4.8	1	6
12	339.1	8.6	8.6	1	12
24	207.8	14.0	14.0	1	24
48	156.5	18.6	18.6	1	48

Table S1. Performance measurements for the Dynamics module only for the Biomass3D test case.

Master branch					
Number of cores	Wall clock time	Speed-up vs the first one	Speed-up vs first one master	Number of Nodes	Number of threads
1	1561.0	1.0	1.0	1	1
3	529.1	2.9	2.9	1	3
6	272.8	5.7	5.7	1	6
12	143.4	10.8	10.8	1	12
24	79.8	19.5	19.5	1	24
48	49.2	31.7	31.7	1	48
PRACE branch					
1	1277.3	1.0	1.2	1	1
3	436.1	2.9	3.6	1	3
6	224.6	5.6	6.9	1	6
12	119.1	10.7	13.1	1	12
24	66.6	19.1	23.4	1	24
48	41.4	30.7	37.6	1	48

Table S2. Performance measurements for the Conversion module for the Biomass3D test case.

Master branch					
Number of cores	Wall clock time	Speed-up vs the first one	Speed-up vs first one Master	Number of Nodes	Number of threads
1	2031.5	1.0	1.0	1	1
3	699.5	2.9	2.9	1	3
6	361.7	5.6	5.6	1	6
12	193.1	10.5	10.5	1	12
24	108.4	18.7	18.7	1	24
48	69.0	29.4	29.4	1	48
PRACE branch					
1	1570.3	1.0	1.3	1	1
3	545.5	2.8	3.7	1	3
6	283.1	5.5	7.1	1	6
12	152.7	10.2	13.3	1	12
24	88.1	17.8	23.0	1	24
48	57.5	27.2	35.3	1	48

Table S3. Performance measurements for the coupled Dynamics-Conversion modules for the Biomass3D test case.