Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Performance Assessment of Pipelined Conjugate Gradient method in Alya

Pedro Ojeda-May[a], Jerry Eriksson[a], Guillaume Houzeaux[b] and Ricard Borrell[b*]

[a]*High Performance Computing Center North (HPC2N), MIT Huset, Umeå Universitet, 90187 Umeå, Sweden*
[b]*Barcelona Supercomputing Center, C/Jordi Girona 29, 08034-Barcelona, Spain*

Abstract

Currently, one of the trending topics in High Performance Computing is related to exascale computing. Although the hardware is not yet available, the software community is working on developing and updating codes, which can efficiently use exascale architectures when they become available. Alya is one of the codes that are being developed towards exascale computing. It is part of the simulation packages of the Unified European Applications Benchmark Suite (UEABS) and Accelerators Benchmark Suite of PRACE and thus complies with the highest standards in HPC. Even though Alya has proven its scalability for up to hundreds of thousands of CPU-cores, there are some expensive routines that could affect its performance on exascale architectures. One of these routines is the conjugate gradient (CG) algorithm. CG is relevant because it is called at each time step in order to solve a linear system of equations. The bottleneck in CG is the large number of collective communications calls. In particular, the preconditioned CG (PCG) already implemented in Alya utilises two collective communications. In the present work, we developed and implemented a pipelined version of the PCG (PPCG) algorithm which allows us to half the number of collectives. Then, we took advantage of non-blocking MPI communications to reduce the waiting time during message exchange even further. The resulting implementation was analysed in detail by using Extrae/Paraver profiling tools. The PPCG implementation was tested by studying the flow around a 3D sphere. Several tests were performed using a different number of processes/workloads to attest the strong and weak scaling of the implemented algorithms. This work has been developed in the context of the preparatory access program of PRACE, simulations were run on the MareNostrum 4 (MN4) supercomputer at Barcelona Supercomputing Center (BSC).

Keywords: Alya, Pipelined Conjugate Gradient, Extrae, Paraver, profiling, exascale

1 Introduction

One of the current trends in High Performance Computing is the development of hardware capable of executing operations in the exascale range (10^{18} FLOPS). This is reflected in the broad range of topics related to exascale computing covered in the PRACE Annual Report 2018 [1]. The software community is working in parallel to develop or update codes that could efficiently run on exascale hardware when it becomes available on the market. This would increase the application ecosystem in the exascale range. One of these applications is Alya [2], a high performance computational mechanics code developed at the Barcelona Supercomputing Center. Alya aims at solving multi-scale, multi-physics, and multi-phase problems using a modular approach whereby a simulation can grow in complexity depending on the desired level of description by adding subsequent modules (models). The physics solvable with Alya include solid mechanics, chemistry, particle transport, heat transfer, turbulence modeling, and biomechanics, among others, deeming both compressible and incompressible flows. It aims at massively parallel supercomputers; its parallelization includes both the MPI and OpenMP frameworks, as well as heterogeneous programming models including graphic accelerators. Alya is one of the twelve simulation codes of the Unified European Applications Benchmark Suite and Accelerators Benchmark Suite of PRACE and thus complies with the highest standards in HPC.

* Corresponding author. E-mail address: ricard.borrell@bsc.es

Although Alya already scales over a large number of cores (>100K) [2], it employs some algorithms that could be improved in order to make it more efficient when running on highly scalable exascale machines in the near future. One of these algorithms, the preconditioned conjugate gradient (PCG), is responsible for solving a system of linear equations. It is one of the most computationally expensive parts in a simulation that is repeated at each time step of the simulation. Its potential performance at the extreme scale is handicapped by the collective communications employed. In the present work, we focus on improving the efficiency of PCG algorithm by reducing the number of collective communications. For this purpose we have considerd the pipelined version of PCG (PPCG).

We implemented the PPCG algorithm in the Alya code and we tested the implementation by studying the flow on a 3D sphere case. In order to attest the performance of this algorithm, we ran strong and weak scaling simulations over a relatively large number of cores (up to 7680). Hardware counters were used to track the threads during the simulation and data was collected in trace files with an instrumented Alya code using the Extrae profiling tool. Further post-processing and analysis was conducted with the Paraver graphical trace analyser [3].

This paper is organised as follows, in the Theory section we provide the mathematical background required for the algorithms employed in this work as well as their application in a standard physical problem of incompressible flows using the Navier-Stokes equations. In the section Implementation we provide details on how the PPCG algorithm was deployed implemented in Alya. In section Preliminary results we describe the main findings of the project with our current implementation. Finally, in the section Summary and future work we provide a set of summarising points of the work together with suggestions which can be used for further development of the PPCG algorithm.

2 Theory

2.1 Pipelined preconditioned conjugate gradient PPCG algorithm

Conjugate gradient method aims at solving the linear equations system:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

Here, $\mathbf{A}$ (nxn) is a symmetric ($\mathbf{A}^{\mathrm{T}} = \mathbf{A}$) and positive definite ($\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} > 0$) matrix. The idea is to find a basis set of conjugate vectors $\mathbf{p}_{\mathrm{i}}$ ($\forall \mathbf{p}_{\mathrm{j}}\mathbf{A}\mathbf{p}_{\mathrm{i}} = 0$) to express the solution of Eq. 1, as a linear combination of basis vectors:

$$\mathbf{x}^{*} = \sum_{\mathrm{i}=1}^{\mathrm{n}} \alpha_{\mathrm{i}}\mathbf{p}_{\mathrm{i}} \tag{2}$$

where the α_{i}'s coefficients in the expansion are obtained by using the basis set from the i-Krylov subspace, $K_{\mathrm{i}}(\mathbf{A}, \mathbf{r}_0) = \mathrm{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{\mathrm{i}-1}\mathbf{r}_0\}$, through a normalization procedure,

$$\alpha_{\mathrm{i}} = \frac{\mathbf{r}_{\mathrm{i}}^{\mathrm{T}}\mathbf{r}_{\mathrm{i}}}{\mathbf{p}_{\mathrm{i}}^{\mathrm{T}}\mathbf{A}\mathbf{p}_{\mathrm{i}}} \tag{3}$$

while the $\mathbf{p}_{\mathrm{i}}$'s vectors are obtained through a Gram-Schmidt orthogonalisation process,

$$\mathbf{p}_{\mathrm{i}} = \mathbf{r}_{\mathrm{i}} + \beta_{\mathrm{i}}\mathbf{p}_{\mathrm{i}-1} \tag{4}$$

$$\beta_{\mathrm{i}} = \frac{\mathbf{r}_{\mathrm{i}}^{\mathrm{T}}\mathbf{r}_{\mathrm{i}}}{\mathbf{r}_{\mathrm{i}-1}^{\mathrm{T}}\mathbf{r}_{\mathrm{i}-1}} \tag{5}$$

We refer to Eqs. 2-5 as the plain CG method. It has been found that for sparse systems a faster and more stable alternative to the plain CG algorithm is the so-called preconditioned conjugate gradient (PCG). In PCG one solves the equation:

$$\mathbf{M}^{-1}(\mathbf{A}\mathbf{x}) = \mathbf{M}^{-1}\mathbf{b} \tag{6}$$

instead of Eq. 1, with M not being singular. The structure of PCG algorithm is shown in Algorithm 1. PCG is the standard algorithm in Alya for solving the system of equations (1). As the reader can notice in Algorithm 1, PCG requires two collective communications (steps 7 and 13) and two point-to-point communications (steps 6 and 12) which are time expensive due to process synchronization. Although this is not a problem for executions requiring a small number of processes, it becomes the bottleneck when a large number of processes is employed. In order to avoid this communication overhead, we implemented a pipelined version of the PCG algorithm called PPCG. Here, we split and reordered the set equations (3-5) into linear terms which can be gathered in a single block and could then be communicated collectively, see the details of the implementation in the next section. As the reader can observe in Eq. 5 (step 14 of Algorithm 1), the calculation of β_{i} requires the values of r and u at two different

time steps. In the pipelined version of the algorithm, the idea is to split this step into several intermediate steps ("pipes") by using additional variables (γ,δ).

2.2 Application to incompressible flows

The Navier-Stokes equations for incompressible flows with a velocity profile **u** reads [4][5]:

$$\partial_t \mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u} = \mathrm{Re}^{-1}\nabla^2\mathbf{u} - \nabla\mathbf{p} + \mathbf{f} \tag{7}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{8}$$

with **p** being the pressure tensor and Re the Reynolds number. The first term in Eq. 7 describes the flow's acceleration, the second term is the convection, and the third term is related to the friction of the media. The contribution of the pressure is given by the fourth term. The last term denoted by **f** describes external forces applied on the system of consideration. Eq. 8 is the continuity equation where time variation of the density is neglected for incompressible flows. The pressure cannot be computed directly due to the lack of a corresponding equation. Thus, the incompressibility of the flow is imposed by Eq. 8. Thereby, Eqs. (7-8) are solved by using the pressure-velocity coupling scheme where Eq. 8 is used as a constrain to obtain the pressure tensor once the velocity has been computed in a fractional time step approach [6][7][8]. Eqs. 7-8 reduce, after discretisation [4][5], to the solution of the set of linear equations of the type (1) with the matrix being equal to the Poisson equation:

$$[\mathbf{Ax}]_k = \sum_{j\in \mathrm{Nb}(k)} \mathbf{A}_{kj}\frac{\mathbf{x}(j)-\mathbf{x}(k)}{\delta n_{jk}} \tag{9}$$

where A is the Laplacian operator and x is the pressure tensor. At this point, a linear solver can be applied to solve the resulting Poisson equation 9. In Alya for instance, this problem has been traditionally solved with the PCG algorithm.

Algorithm 1 PCG	**Algorithm 2 PPCG**
procedure PREC_CG($\mathbf{M}^{-1}, \mathbf{A}, \mathbf{b}, \mathbf{x}_0$)	**procedure** PREC_P_CG($\mathbf{M}^{-1}, \mathbf{A}, \mathbf{b}, \mathbf{x}_0$)
$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$	$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{u}_0 \leftarrow \mathbf{M}^{-1}\mathbf{r}_0$	$\mathbf{u}_0 \leftarrow \mathbf{M}^{-1}\mathbf{r}_0$
$\mathbf{p}_0 \leftarrow \mathbf{u}_0$	$\mathbf{w}_0 \leftarrow \mathbf{A}\mathbf{u}_0$
for i = 0, … , **do**	**for** i = 0, … , **do**
$\mathbf{s}_i \leftarrow \mathbf{A}\mathbf{p}_i \leftarrow$ POINT-TO-POINT-COMM.	START ALL-REDUCE
START ALL-REDUCE	$\gamma_i \leftarrow (\mathbf{r}_i, \mathbf{u}_i)$
$\alpha_i \leftarrow \frac{(\mathbf{r}_i,\mathbf{u}_i)}{(\mathbf{s}_i,\mathbf{p}_i)}$	$\delta_i \leftarrow (\mathbf{w}_i, \mathbf{u}_i)$
END ALL-REDUCE	$\mathbf{m}_i \leftarrow \mathbf{M}^{-1}\mathbf{w}_i \leftarrow$ POINT-TO-POINT COMM.
$\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \alpha_i\mathbf{p}_i$	$\mathbf{v}_i \leftarrow \mathbf{A}\mathbf{m}_i \leftarrow$ POINT-TO-POINT COMM.
$\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \alpha_i\mathbf{s}_i$	END ALL-REDUCE
$\mathbf{u}_{i+1} \leftarrow \mathbf{M}^{-1}\mathbf{r}_{i+1} \leftarrow$ POINT-TO-POINT-COMM.	**if** (i > 0) **then**
START ALL-REDUCE	$\beta_i \leftarrow \gamma_i/\gamma_{i-1}$
$\beta_{i+1} = \frac{(\mathbf{r}_{i+1},\mathbf{u}_{i+1})}{(\mathbf{r}_i,\mathbf{u}_i)}$	$\alpha_i \leftarrow \frac{1}{(\delta_i/\gamma_i)-(\beta_i/\alpha_{i-1})}$
END ALL-REDUCE	**else**
$\mathbf{p}_i \leftarrow \mathbf{u}_{i+1} + \beta_{i+1}\mathbf{p}_i$	$\beta_i \leftarrow 0$
end for	$\alpha_i \leftarrow \frac{\gamma_i}{\delta_i}$
end procedure	**end if**
	$\mathbf{z}_i \leftarrow \mathbf{v}_i + \beta_i\mathbf{z}_{i-1}$
	$\mathbf{q}_i \leftarrow \mathbf{m}_i + \beta_i\mathbf{q}_{i-1}$
	$\mathbf{s}_i \leftarrow \mathbf{w}_i + \beta_i\mathbf{s}_{i-1}$
	$\mathbf{p}_i \leftarrow \mathbf{u}_i + \beta_i\mathbf{p}_{i-1}$
	$\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \alpha_i\mathbf{p}_i$
	$\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \alpha_i\mathbf{s}_i$
	$\mathbf{u}_{i+1} \leftarrow \mathbf{u}_i - \alpha_i\mathbf{q}_i$
	$\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \alpha_i\mathbf{z}_i$
	end for
	end procedure

3 Implementation

The PCG method is dominated by the sparse matrix vector (SpMV) product. When parallelising this algorithm the major contributors to the overhead are the point-to-point communications. However, at some point, the bottleneck shifts to the dot products, which require a global reduction where the communication is usually performed by 2-by-2 reduction trees. Therefore, it has a cost of O(log2 (P)), where P stands for the number of parallel processes. Eventually, communication time for the dot products will exceed the calculation and become the main bottleneck of the overall iteration.

In Algorithm 1 we present the PCG method for the solution of symmetric and positive definite systems. In blue color we show the operations consisting of a linear combination of vectors, in green color the sparse matrix vector product which requires a point-to-point (`MPI_ISend`, `MPI_IRecv`) communications and in red color the scalar products which require a global reduction (`MPI_Allreduce`).

It is important to notice that the PPCG algorithm is numerically equivalent to the PCG algorithm but the operations are reordered and split to gather the two reduction operations and overlap them with the sparse matrix vector products. The algorithm flow is shown in Algorithm 2. We can observe that the PPCG algorithm has additional operations (steps 19 to 26), thus we expect the algorithm to be slower than PCG when communication is not dominant. In order to hide communications and computations, non-blocking collective communications are required, in particular the `MPI_IReduce` operation, which was one of the novel features of the MPI-3 library.

In the present project we have optimised and validated the implementation of the PPCG algorithm, we have asserted that, discarding round-off errors, the convergence is equivalent to that of the PCG algorithm. In order to obtain a detailed report of routines behaviour (especially MPI calls) at thread-level, Alya was instrumented with the Extrae (v.3.6.1) tool as installed on MN4 supercomputer. The default option to start Extrae at the beginning of an Alya simulation was turned-off to avoid collecting unnecessary information from irrelevant routines. Thus, we placed Extrae calls only on the PCG (file:*cgrpls.f90*) and PPCG (file:*pipelined_CG_rr.f90*) files using the switches:

```
call extrae_restart
        routine of interest (PCG or PPCG)
call extrae_shutdown
```

The following toolchain was used for instrumentation:

```
ml intel/2017.4 impi/2017.4
ml gcc/7.2.0
ml EXTRAE/3.6.1
```

The resulting trace files were post-processed and analysed with the graphical tool analyser Paraver.

4 Preliminary results

We performed different simulations to attest the scaling behaviour of both PCG and PPCG algorithms implemented in this project. Although both algorithms are numerically equivalent, they would perform differently depending on the workloads. In particular, we expect to see some benefits of the PPCG algorithm in the range of values of problem size and number of CPU-cores where the collective communication dominates. This is the reason why, in detriment of the speedup, we have considered rather small problems for our tests. In this way, the communication overhead becomes dominant within the range of CPUs under consideration.

The case used for the scaling tests has been the simulation of the flow around a sphere. A large eddy simulation (LES) has been used and the pressure-velocity coupling has been solved with a fractional step projection method, where only the pressure correction equation is solved implicitly [6][7][8]. Then, both the PCG or PPCG methods have been used to solve the Poisson system in Eq. 9. The elapsed time measurements used in Figure 1 and the tables of Section 3.3, have been obtained by accumulating 100 time-steps, i.e. a hundred executions of the linear solver. Three different meshes have been considered in our tests. The initial mesh (CASE 0) has 3.5 million tetrahedral elements; the subsequent meshes were obtained by division of the elements of this initial mesh. In each

division the number of elements of the mesh is roughly increased by a factor of eight, particularly the meshes obtained have 25.6 M (CASE 1) and 205 M (CASE 2) elements. The measurements for all the tests carried out can be found in Tables S1-S6 in the Appendix.

For each test case we have performed strong scaling tests using a different number of CPU-cores, see Figure 1. In all cases the PPCG algorithm has better scalability than the PCG, and eventually it becomes more efficient than the PCG. We can observe that by increasing both the problems size and the number of CPU-cores, the PPCG outperforms the PCG algorithm earlier (for a larger local problem). For CASE 0 this occurs for the last tests considered with a load per CPU of 4.5K elements, for the CASE 1 the load per CPU is 7K elements, and for CASE 2 from a local load of 35K elements the PPCG becomes more efficient. This is an interesting result, meaning that the larger the scale of the problem considered, the wider becomes the range (of numbers of cores) where the PPCG outperforms the PCG algorithm. A preliminary test of the weak scaling behaviour of both PCG and PPCG algorithms can be seen in Tables S7-S8.

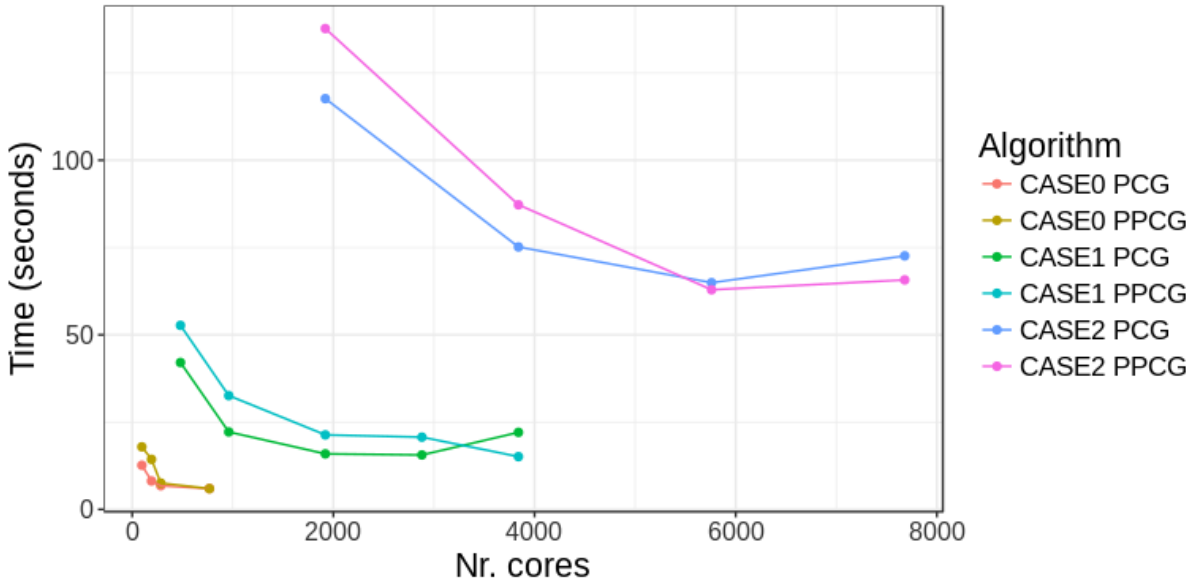

Figure 1: Comparison of the elapsed time for 100 resolutions of the Poisson system considering the PCG and PPCG algorithms for different numbers of CPU-cores. Three different mesh sizes are considered: CASE 0 (3.5M), CASE 1 (25.6M) and CASE 2 (204M).

We have inspected the actual overlapping computation/communication for both algorithms using the collected trace files. In the following description we use the largest traces obtained with 2048 cores. In the following figures, two metrics are presented, the upper panel shows MPI communications and the lower panel shows the Instruction per clock cycle (IPC) metric. The former gives us an idea of the communication overhead while the latter tells us how busy the CPU was when doing useful numerical operations. The codes of colours are explained in the captions of the figures. A zoom out has been applied to Figure 2 to make the explanations clearer.

In the case of the PPCG algorithm, by studying the IPC panel in Figure 2, it is clear that most parts are blue which means that computations are made, and, thus, there are only a few small black parts where no computations are made. During these black parts, `MPI_Waitall` is executed (green in the upper panel). There were some processes involved in this type of communication which lagged behind during the simulation close to the dashed white line. All other MPI calls are non-blocking, and it is very illustrative that computations are overlapped during theses MPI calls.

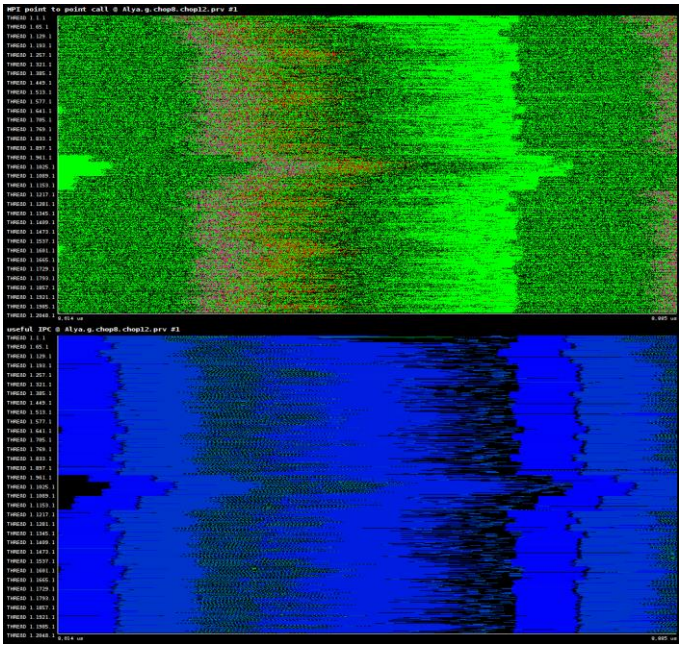

Figure 2: Traces for a PPCG iteration using 2048 CPU-cores. Top: Communication episodes (red: Isend, turquoise: Irecv, gren: Waitall, dark red: IAllreduce) Bottom: IPC (black means no computations).

In contrast, the performance of PCG is significantly different than that of PPCG. In Figure 3, the upper plot shows the MPI calls, where the pink parts are related to a blocking `MPI_Allreduce`. The corresponding lower plot shows that almost no IPC is carried out during this part. This means there are basically no communication/computation overlaps in the CG case. Although we employed blocking routines in this case, we observed that some degree of noisiness in the data which can be attributed to the configuration of MN4.

The overlapping of communications and computations shown in the Figure 2, favours the scalability of the PPCG. In all the scalability test presented we observe that initially the PCG algorithms is faster but at some point the PPCG surpasses it.

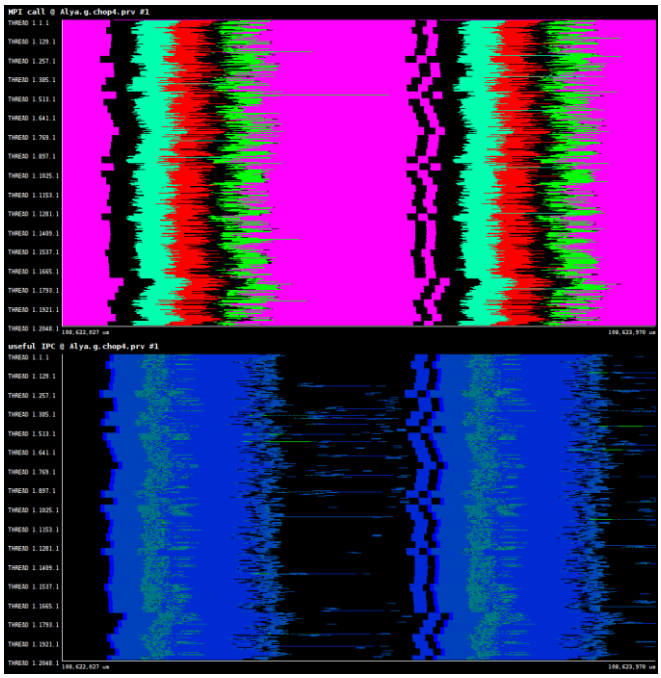

Figure 3: Traces for a PCG iteration using 2048 CPU-cores. Top: Communication episodes (red: Isend, turquoise: Irecv, gren: Waitall, pink: Allreduce) Bottom: IPC (black means no computations).

5 Summary and future work

In the present project we focused on implementing and validating the Pipelined CG solver. The solver code was verified and validated, and its performance was attested using profiling tools and performing scalability tests. Details on all these aspects have been explained in the previous sections.

We used the Extrae profiling tool to collect the traces of processes and Paraver graphical tool to visualize trace files. Extrae collects the information from hardware counters into trace files that can be used to analyse any information that is expressed on its input trace format.

The main achievements of this project were:

1) We implemented a new solver, the Pipelined CG, which is designed to overlap the collective reductions with the sparse matrix vector products.
2) The bottleneck that we aimed to overcome are the collective reductions. With the pipelined CG algorithm, the `MPI_Allreduce` instances were substituted by `MPI_IReduce`, moreover all reductions are performed in a single episode.
3) We used profiling tools to attest the performance of the new solver and to ensure that the overlapping of communications and computations is occurring.
4) We performed scalability tests using up to 7680 CPU-cores.

The new PPCG algorithm is numerically equivalent to the PCG algorithm but, by reordering the operations, reductions are grouped and can be overlapped with the SpMV operations. We have implemented the Pipelined CG within the Alya system. We have verified the implementation, validated that the convergence is equivalent to that of the CG algorithm, and finally we have attested the performance by using profiling tools and scalability tests with up to 7680 CPU-cores. We have also compared the performance with that of the CG algorithm to get an impression of the range that this new algorithm can outperform CG. From the experience on the MareNostrum IV supercomputer we know that the Pipelined CG algorithm will be a convenient option for large scale simulations.

References

[1] http://www.prace-ri.eu/ar-2018/
[2] Vázquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Arís, R., Mira, D., Calmet, H., Cucchietti, F., Owen, H., Taha, A., Burness, E. D., Cela, J. M., Valero, M. *Alya: Multiphysics engineering simulation toward exascale*, Journal of Computational Science **14** (2016) 15-27.
[3] https://tools.bsc.es/sites/default/files/documentation/html/extrae/index.html
[4] Oyarzun, G., Borrell, R., Goroberts, A., and Oliva, A. *MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner*, Computers & Fluids, **92**, 244-252 (2014).
[5] Borrell, R., Lehmkuhl, O., Trias, F.X., and Oliva, A. *Parallel direct Poisson solver for discretisations with one Fourier diagonalisable direction*, Journal of Computational Physics, **230**, 4723-4741 (2011).
[6] Tu, J., Yeoh, G., and Liu, C. *Computational fluid dynamics a practical approach*, Elsevier 2nd. Edition.
[7] J. Chorin, *Numerical solution of the Navier–Stokes equations*, Journal of Computational Physics 22 (1968) 745–762.
[8] N.N. Yanenko, *The Method of Fractional Steps*, Springer-Verlag, 1971.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation program (2014-2020) under grant agreement 730913.

6 Appendix

Table S1. Measurements of PCG method for CASE 0

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
96	12.64	1	2	96
192	8.14	1.55	4	192
284	6.78	1.86	8	284
768	5.96	2.12	16	768

Table S2. Measurements of PCG method for CASE 1

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
480	42.07	1	10	480
960	22.18	1.89	20	960
1920	15.91	2.64	40	1920
2880	15.59	2.69	60	2880
3840	22.06	1.90	80	3840

Table S3. Measurements of PCG method for CASE 2

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
1920	117.65	1	40	1920
3840	75.17	1.57	80	3840
5760	64.94	1.81	120	5760
7680	72.62	1.62	7680	7680

Table S4. Measurements of PPCG method for CASE 0

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
96	17.92	1	2	96
192	14.33	1.25	4	192
284	7.54	2.37	8	284
768	5.94	3.01	16	768

Table S5. Measurements of PPCG method for CASE 1

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
480	52.71	1	10	480
960	32.59	1.62	20	960
1920	21.34	2.47	40	1920
2880	20.69	2.54	60	2880
3840	15.11	3.49	80	3840

Table S6. Measurements of PPCG method for CASE 2

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
1920	137.72	1	40	1920
3840	87.24	1.58	80	3840

Nr. of cores	Wall clock time	Speed-up vs the first one	Nr. of Nodes	Nr. of process
5760	62.91	2.19	120	5760
7680	65.73	2.09	160	7680

Table S7. Measurements of PCG method

Nr. of cores	Wall clock time	Elements per process	Parallel efficiency	Nr. of Nodes	Nr. of process
96	12.64	33643	100%	2	96
960	22.18	26686	45%	10	480
5760	64.94	35574	20%	80	5760

Table S8. Measurements of PPCG method

Nr. of cores	Wall clock time	Elements per process	Parallel efficiency	Nr. of Nodes	Nr. of process
96	17.92	33643	100%	2	480
960	32.59	26686	40%	10	480
5760	62.91	35574	28%	80	5760