Available online at www.prace-ri.eu

**Partnership for Advanced Computing in Europe**

# Parallel Mesh Partitioning in Alya

A. Artigues[a]*** and G. Houzeaux[a]*

[a]*Barcelona Supercomputing Center*

***antoni.artigues@bsc.es *guillaume.houzeaux@bsc.es

## Abstract

The *Alya* System is the BSC simulation code for multi-physics problems [1]. It is based on a Variational Multiscale Finite Element Method for unstructured meshes.

Work distribution is achieved by partitioning the original mesh into subdomains (submeshes). This pre-partition step has until now been done in serial by only one process, using the *metis* library [2]. This is a huge bottleneck when larger meshes with millions of elements have to be partitioned. This is due to the data not fitting in the memory of a single computing node and in the cases where the data does fit; *Alya* takes too long in the partitioning step.

In this document we explain the tasks done to design, implement and test a new parallel partitioning algorithm for *Alya*. In this algorithm a subset of the workers, is in charge of partition the mesh in parallel, using the *parmetis* library [3].

Partitioning workers, load consecutive parts of the main mesh, with a parallel space partitioning bin structure [4], capable of obtaining the adjacent boundary elements of their respective submeshes. With this local mesh, each of the partitioning workers is able to create its local element adjacency graph and to partition the mesh.

We have validated our new algorithm using a *Navier-Stokes* problem on a small cube mesh of 1000 elements. Then we performed a scalability test on a 30M element mesh to check if the time to partition the mesh is reduced proportionally with the number of partitioning workers.

We have also done a comparison between *metis* and *parmetis,* the balancing of the element distribution among the domains, to test how the use of many partitioning workers to partition the mesh affects the scalability of *Alya*. We have noticed in these tests that it's better to use fewer partitioning workers to partition the mesh.

Finally we have two sections explaining the results and the future work that has to be done in order to finalise and improve the parallel partition algorithm.

## Introduction

*Alya* solves time-dependent Partial Differential Equations (PDEs) using the Finite Element Method (FEM). The meshes are unstructured and can be hybrid, that is of different types of elements.

Work distribution is achieved by partitioning the original mesh into subdomains (submeshes) that run concurrently by the corresponding MPI processes. Mesh partitioning is carried out using *METIS*, an automatized mesh partitioner.

*Alya* is divided in two types of processes: The master and $N_w$ workers. The master is mainly in charge of pre-process and some of the post-process tasks. The workers are in charge of running concurrently the simulation iterations, by assembling matrices and RHS, and solving the corresponding algebraic system by way of iterative solvers.

In the initial version of *Alya*, the mesh partitioning is done sequentially by the master while the workers wait for receiving their parts of the mesh. The partition job done by the master consists in the following steps:

1. Mesh reading from the hard drive.
2. Initialise mesh properties variables.
3. Create the nodes-elements adjacency graph, for each node we get the elements to which it belongs.
4. Create the adjacency nodes graph, to know the neighbours of each node.
5. Create the adjacency elements graph, to know the neighbours elements of each element. This graph is computed using the two previous graphs defined in points 3 and 4.
6. Call *metis* passing the adjacency elements graph and the number of sub-domains. Obtain as a response an array assigning the subdomain to each element.
7. Compute the communication arrays, needed to exchange data between the workers, based on the result obtained from *metis*.
8. Compute the communication scheduling, needed to determine in which order the workers have to interchange its data in pairs (through `MPI_SendReceive`).
9. Distribute the sub-meshes and corresponding element, boundary and node arrats to their corresponding workers.

After these steps the workers can start to run the simulation time steps.

The main problems with this approach are:

- For larger meshes, that contain millions of elements, the node memory is too small to partition the mesh by the master. Typically, nodes of supercomputers have a small RAM memory.
- For larger meshes the master takes a long time in the partitioning process, described earlier (points 1 to 9). This solution does not scale.

The goal of this project is to parallelize the mesh partitioning represented by steps 1 to 9. The approach consists in performing these steps by a subset of the $N_w$ workers in parallel, and let the master only be in charge of calculating the communication scheduling. Let us denote a *partitioning worker* as a worker, belonging to the subset of the $N_w$ workers, and participating to the parallel mesh partitioning. Thus we have $N_p<=N_w$ workers involved in the partitioning.

Each partitioning worker reads a consecutive part of the mesh from the disk and contributes to the partition. *Metis* has been substituted by *Parmetis*, a parallel version of *Metis* working with MPI. *Parmetis* uses a Distributed CSR format as the input of the element graph. Each partitioning worker has to create its part of the elements adjacency graphs in the *Parmetis* format. The main difficulty is that the elements of the local graphs have to be numbered using the original global numbering.

We have designed the parallelisation process of the partitioning in different phases:

1. Parallelise the steps from 3 to 6. Corresponding to the *metis* partitioning part.

2. Parallelise the steps from 1 to 2. Corresponding to the loading mesh part.
3. Parallelise the steps 7 and 9. Corresponding to the computation and distribution of the communication arrays.

In this project we have implemented and validated the most complex part of the parallel partitioning, phase 1. The workflow that has been implemented is (see Figure 1):

1. (Master) Mesh reading from the hard drive.
2. (Master) Initialise mesh properties variables.
3. (Master) Distributes a consecutive part of the mesh to a partitioning workers subset in charge of the partitioning.
4. (Workers) Each partitioning worker receives a part of the mesh and computes its own elements adjacency graph.
5. (Workers) Each partitioning worker computes the partition of it's part calling the *parmetis* library in parallel. Each partitioning worker obtains a part of the array assigning the subdomains to each element (LEPAR_PAR).
6. (Workers) Each partitioning worker sends to the master his LEPAR_PAR.
7. (Master) Joins all the LEPAR_PAR received from the partitioning workers.
8. (Master) Compute the communication arrays, needed in order to interchange data between the partitioning workers, based on the LEPAR_PAR
9. (Master) Compute the communication scheduling, needed to determine in which order the workers have to interchange its data in pairs.
10. (Master) Distribute the mesh part to its corresponding worker.

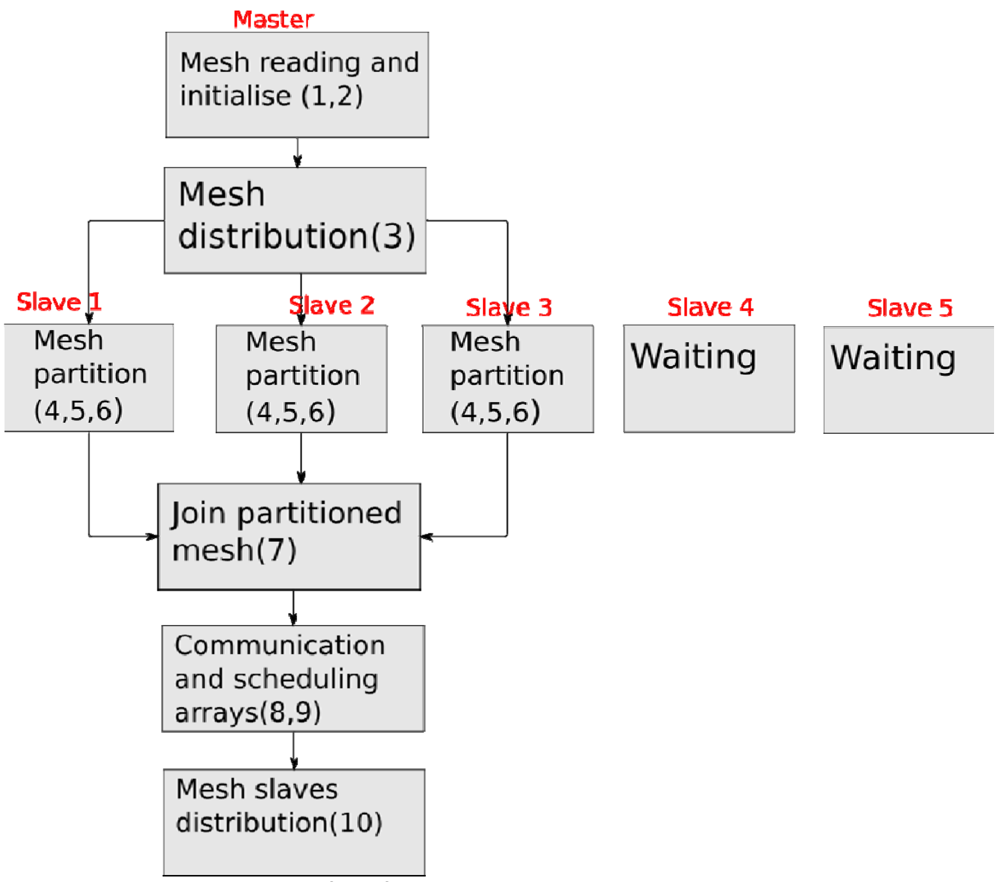

Figure 1. *Alya* workflow for 6 mpi processes with 3 partitioning workers

## Design and implementation

### Parallel partition parameters

The parallel partition process has two main parameters:

- **kfl_paral_parmes**: Number of mesh partitioning workers.
- **kfl_paral_proc_node**: An integer indicating how the partitioning workers have to be distributed among the nodes. If *k* is the value of *kfl_paral_proc_node* then the processes selected to partition the mesh will be 1, 1+*k*, 1+*k*+*k*, and so on, where this number is the mpi rank of the process. For example if the parameter has a value of 3, the workers in charge to partition the mesh will be the 1, 4, 7, 10,… This allows us to distribute the partitioning workers in different nodes.

*Alya* is launched in *N* mpi processes , process *0* is the master and the other *N*-1 processes are the workers. Only a subset of the *N*-1 workers will be in charge of partitioning the mesh, the parameter *kfl_paral_parmes* defines the number of partitioning workers. This number has to be as small as possible in order to obtain the best partition from *parmetis*.

The *kfl_paral_proc_node* determines the distribution of the partitioning workers among the supercomputer nodes. If the Ram memory of one node is not enough to partition the mesh by the workers we can distribute the partitioning workers among different nodes in order to have more memory available.

For example: In *MareNostrum III* [6] each node has 16 cores and 32 GB of RAM, processes from 0 to 15 are pinned in the first node, processes from 16 and 31 are pinned in the second node, and so on. If we partition a mesh with workers 1 and 2 we only will have 32 GB of RAM, but if we partition a mesh with workers 1 and 16 we will have 64GB of RAM to do the job. In this case the *kfl_paral_proc_node* = 15.

### Parallel partition workflow

Each partitioning worker receives from the master a consecutive part of the mesh. It consists in two type of information: The elements to node connectivity (*LNODS*) and the node coordinates (*COORD*).

Each type of information is loaded independently on each of the partitioning workers; because of that one partitioning worker can have a node in the connectivity that has no corresponding coordinate data.

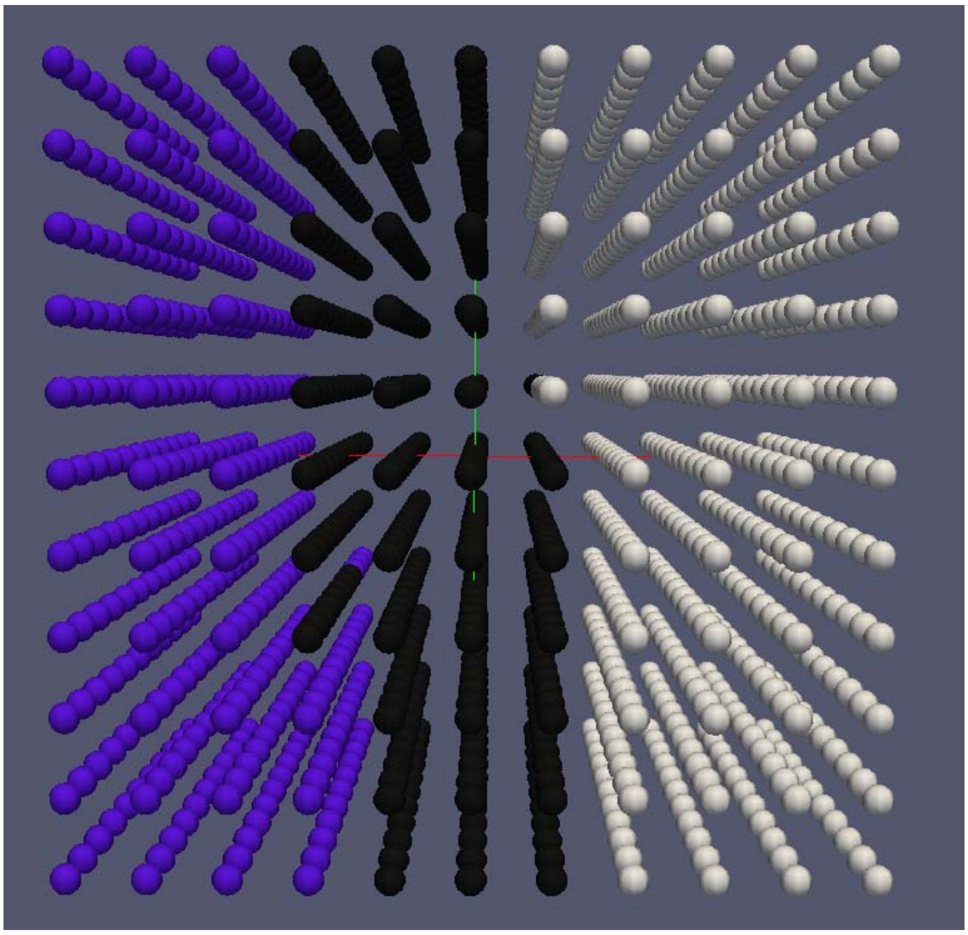
Figure 2. Mesh distribution between 3 workers, each colour corresponds to the mesh part received by one worker.

Each partitioning worker has to create the distributed CSR format graph of its mesh part, which is the input needed by *parmetis*. To do this the partitioning worker needs to know the adjacent elements to its boundary elements, in its part of the mesh.

We have to determine which elements one worker will send and receive from, in order to obtain the adjacent boundary elements. For this we have used the computational geometry bin data structure approach.

In order to implement this algorithm each partitioning worker needs all the coordinates of his *LNODS*. Each partitioning worker doesn't have all the coordinate values of his *LNODS*, this is because he receives only a part of the *COORD* data, and some coordinate values of his *LNODS* could be read by another partitioning worker. Because of that the first step is to interchange the node coordinates in *COORD* between the partitioning workers.

Each worker sends to the other workers the identifiers of the coordinates that he needs (with *mpi_allgatherv*), with this received list each worker send to the others the coordinate values that he has in its COORD in pairs (with *mpi_send* and *mpi_recv*).

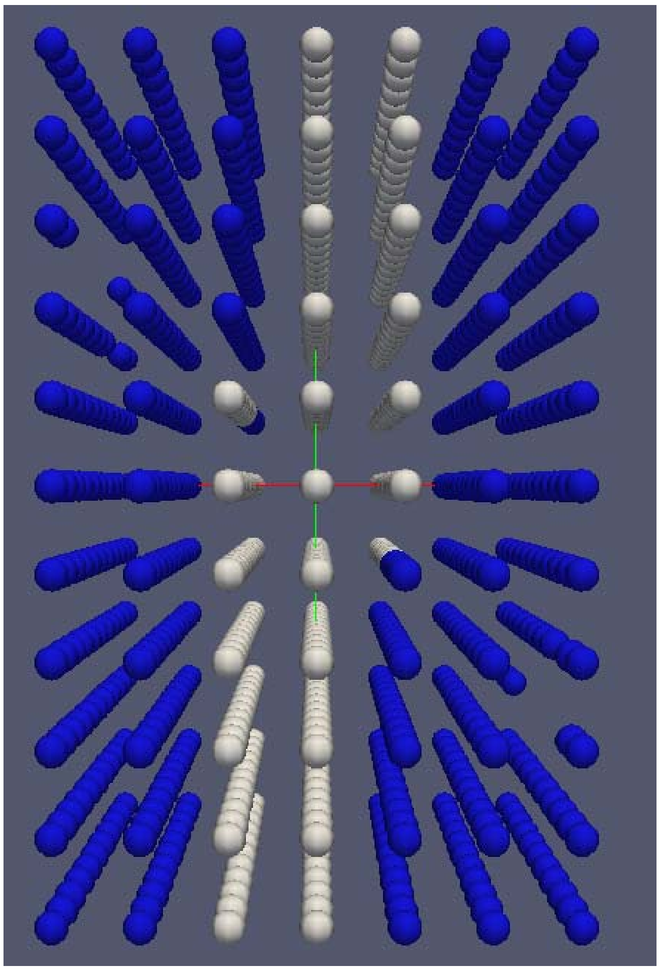
Figure 3. Element redistribution, in white are the worker's own elements, in blue the elements received from other workers

In summary, each partitioning worker has to do the following steps:

1. Get the elements connectivity and coordinates part from the master.
2. Redistribute the coordinates between the partitioning workers.
3. Calculate a local numeration for the worker nodes and coordinates. With this new numeration the coordinate identifiers will be consecutive and we can store them in a smaller array.
4. Calculate a list with the mesh boundary elements.
5. Create the parallel bin structure between all the partitioning workers.
6. Redistribute elements among the partitioning workers using the bin structure and the boundary elements list.
7. Renumber and reorder the local connectivity elements.
8. Partition the mesh with the local connectivity and coordinate information
    a. Create the local nodes-elements graph, to know what elements are in each node.
    b. Create the local adjacency nodes graph, to know the neighbours of each node.
    c. Create the local adjacency elements graph, to know the neighbours elements of each element
    d. Call *parmetis* passing the local adjacency elements graph and the number of subdomains. Obtain as a response a partition array giving the domain of each element of the local mesh.
9. Send the partition array to the master.

## Tests

### Validation

We have executed a small well tested *Navier-Stokes* simulation in order to validate our implementation. We have checked that the serial and parallel partitioning produces the same results.

We have run the simulation with 16 mpi processes, three workers have been used to partition a 1000 elements cube mesh. Figure 4 shows the comparison between the parallel and serial partition. The *paraver* tool has been used for this purpose [5]. As you can see the workers 2, 3, 4 are in charge to partition the mesh while the others are waiting. Between the partitioning workers the interchange data corresponds to the steps 1, 2, 5, 6, 8, and 9 of the algorithm.

With only three partitioning workers the parallel partition runs faster than the serial partition, in the figure 4 is shown that all the partitioning workers are running the time steps while in the serial run the partitioning workers are still waiting to receive his mesh part.

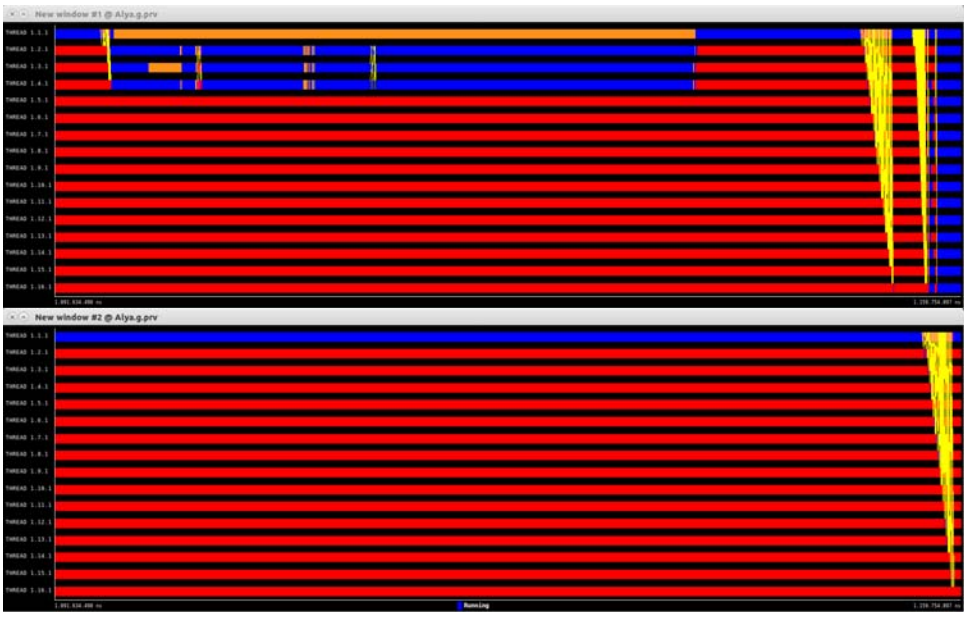

Figure 4. At the top the parallel partition trace, at the bottom the serial partition trace

We also have validated the test with a larger case, running successfully 10 time steps with a 30 million mesh partitioned in 4095 domains with 20 partitioning workers.

### Partitioning Workers Scalability

In this test we have run the same *Navier-Stokes* simulation, but with a 30 million elements mesh. We have partitioned the mesh in 511 domains using the parallel partition algorithm. We have used 2, 5, 10, 15 and 20 partitioning workers in different runs to achieve this task.

We have measured the time needed to partition the mesh between the following two points (both included):

- The master distributes a consecutive part of the mesh to a workers subset.
- The master receives and joins all the partitioned mesh.

In this test we have measured the speed-up of the algorithm to partition the mesh in parallel when we add more workers to partition the mesh. The speed-up is calculated dividing our base time (252,2) by the time obtained when adding more partition slaves. Our base time is obtained with 2 partition slaves because this is the minimum number of processes required by *parmetis* in order to work.

We have obtained this response times:

| **Number of partitioning workers** | **Response time in seconds** |
|---|---|
| **2** | 252,2 |
| **5** | 118,8 |
| **10** | 76,7 |
| **15** | 62,2 |
| **20** | 57,3 |

Table 1: 30 million mesh partitioning time in 511 domains.

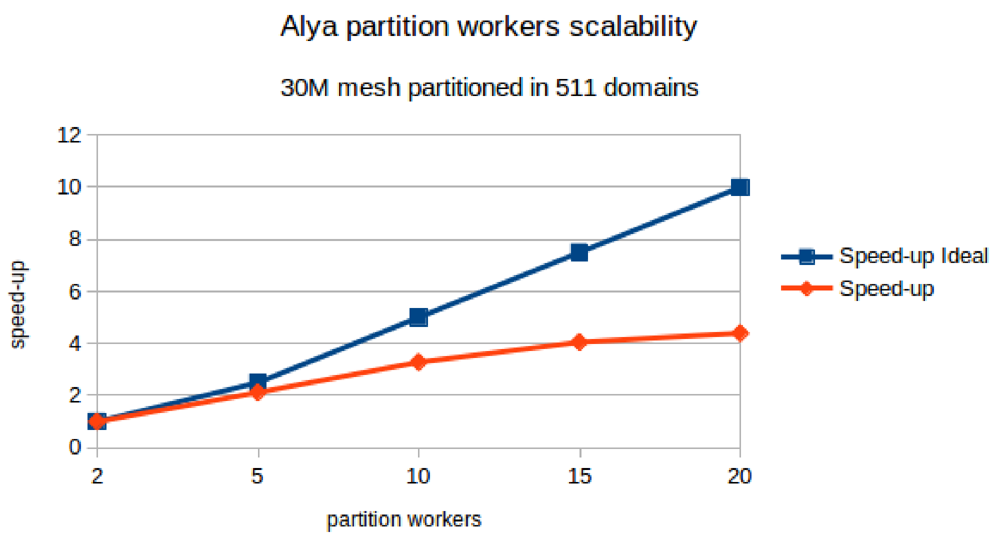

Figure 5. Partition algorithm speed-up when more partitioning workers are used, with a 30 million mesh in 511 domains

The code scales when we use more workers to partition the mesh; in our case a good number of partitioning workers is 15, because with more workers the communication overhead is bigger than the computation time of each worker, as is shown in the figure 5.

Running the same partition with the serial version and *metis* we have obtained a response time of 218,7 seconds, it is faster than the parallel partition in two workers (252,2 seconds) due to communication overhead. But with 5 partitioning workers the response time is 118,8 seconds, that doubles the performance of the serial version.

We also have observed how our code scales in terms of the memory. We launched the simulation with 20 consecutive partitioning workers, that mean that 16 workers are at the same node sharing 32 GB of RAM, in this case the simulation fails because we reached the available memory in one node. However if we launch one partitioning worker per node the simulation runs perfectly, because each worker has 32 GB of RAM available.

## *Parmetis* strong scalability

We have measured the *Alya* scalability running a *Navier-Stokes* simulation with these inputs:

- 30 million elements cube mesh.
- From 127 to 4095 domains.
- Partitioned in parallel with 8 partitioning workers.
- 2 iteration time steps are measured.

Here we want to measure how well *parmetis* is doing a balanced partitioning between the domains. If one of the domains has more assigned elements than the others the worker will have more work than the others, otherwise if one domain has less elements than the others the worker will be idle most of the time.

In order to avoid these two problems and to have a good scalability and efficiency we need to have exactly the same number of elements in each domain.

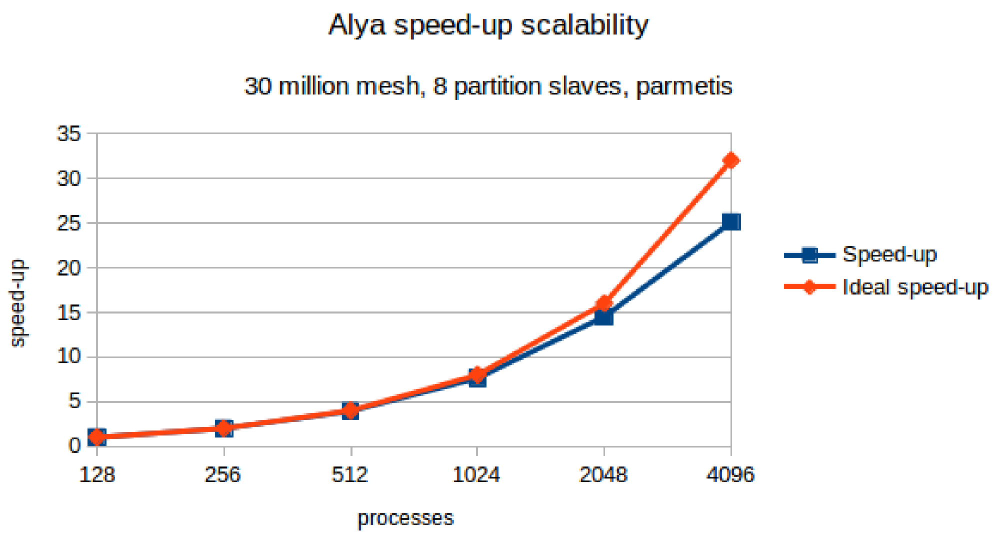

Figure 6. *Alya* scalability with 8 partitioning workers and *parmetis*

As is shown in Figure 6 the *Alya* scalability with *parmetis* is near the ideal, taking into account that with 4096 domains we have only 8000 elements per domain the scalability is quite good.

If we plot the histogram of the elements distribution between the domains we can observe that the major part of the domains have the same number of elements, this plot corresponds to a 4095 domains partition: 3795 domains have about 8000 elements, 282 domains have about 7000 elements and 18 domains have about 6000 elements assigned. We can conclude that *parmetis* has produced a well balanced partitioning.

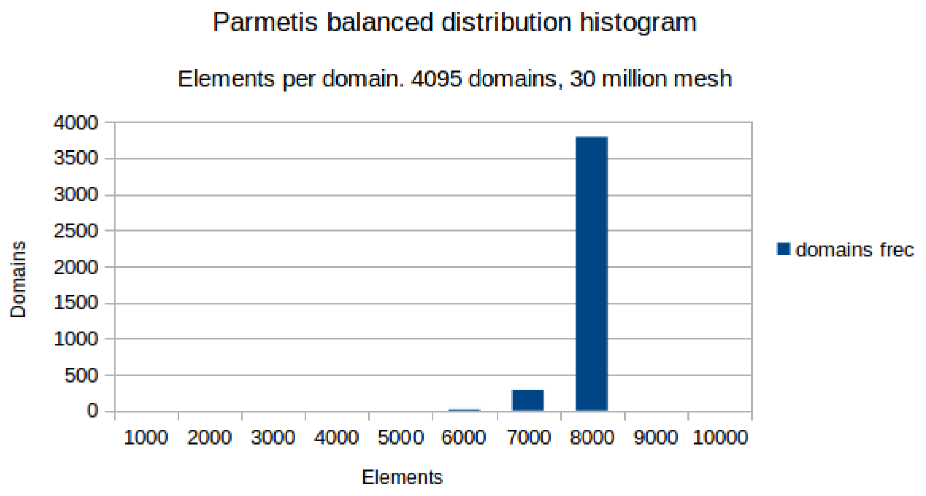

Figure 7. *Parmetis* elements distribution between domains histogram. It shows how elements are distributed among the domains.

## *Parmetis* and *metis* elements distribution comparison

We have partitioned the same 30 million mesh in 512 domains, using *metis* in serial and *parmetis* in parallel with 20 and 40 partitioning workers. If we plot the histogram of the elements distribution between the domains we can see that *metis* is doing a better job (figure 8), because with *metis* all domains have a similar number of elements, however with *parmetis* there are domains with fewer elements than others.

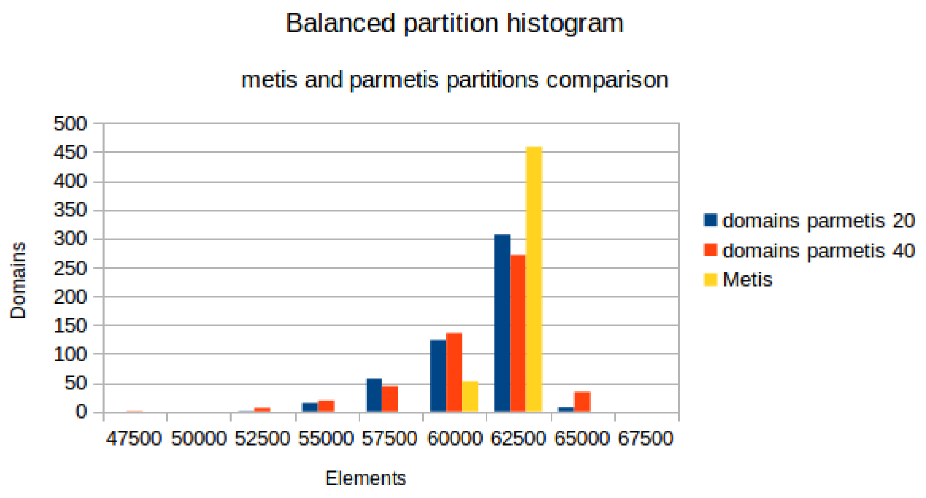

Figure 8. *Parmetis* and metis distribution between domains histogram. It shows how elements are distributed among the domains.

The figure 8 shows that with *metis* all domains have from 60000 to 62500 elements assigned. However, with *parmetis* and 20 partitioning workers the distribution is from 52500 to 65000 elements, and with 40 partitioning workers we obtained a distribution from 47500 to 65000.

That means that if more partitioning workers are used to partition the mesh we obtain less balanced distribution of the elements among the domains.

The conclusion is that we have to assign as few partitioning workers as possible. The minimum number of partitioning workers will come from:

- The needs in terms of RAM memory, if we use more partitioning workers we have more memory available because we can use more nodes in the partition.
- The response time that we need, if we use more partitioning workers we will obtain better response time as you can see in section Partitioning Workers Scalability.

## Summary and conclusions

- We have created and validate a new algorithm to partition *Alya* meshes in parallel, using a workers subset.
- We have checked that our partition algorithm scales in terms of execution time and memory consumption.
- We have obtained a very good *Alya* scalability with the parallel partition algorithm.
- We have concluded that is better to use fewer partitioning workers in order to obtain a balanced elements distribution in the domains.
- We have observed that *metis* obtains better balanced elements distribution than *parmetis*.

## Future work

- Design and implement the parallelization of the mesh loading.
- Design and implement the parallelization of the computation and distribution of the communication arrays.
- Create a hybrid parallel partition algorithm adding *openmp* pragmas to the *Alya* code.
- Test the parallel partition algorithm with bigger meshes and with more domains.

## References

[1] *Alya*: The *Alya* System - Large Scale Computational Mechanics (bsc-cns *Alya* web page)
[2] *Metis*: Serial Graph Partitioning and Fill-reducing Matrix Ordering (*metis* documentation)
[3] *Parmetis*: Parallel Graph Partitioning and Fill-reducing Matrix Ordering (*parmetis* documentation)
[4] Space partitioning: wikipedia article
[5] *Paraver*: Performance Analysis Tools: Details and Intelligence (*paraver* documentation)
[6] *MN3*: MareNostrum III Supercomputer (2013)
[7] G.Houzeaux, R.delaCruz, M.Vazquez: Parallel Uniform Mesh Subdivision in Alya. Available on-line at www.prace-ri.eu

## 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.