ELSA Performance Analysis

Xavier Saez and José María Cela

Barcelona Supercomputing Center

Technical Report TR/CASE-08-1

2008

ELSA Performance Analysis

Xavier Saez[1] **and José María Cela**[2]

[1] *Computer Application and Science Engineering, Barcelona Supercomputing Center, Edifici Nexus, Campus Nord UPC, c/ Gran Capità, 2-4, 08034 Barcelona, Spain.*
E-mail: xavier.saez@bsc.es, *Web page:* http://www.bsc.es

[2] *Computer Application and Science Engineering, Barcelona Supercomputing Center, Edifici Nexus, Campus Nord UPC, c/ Gran Capità, 2-4, 08034 Barcelona, Spain.*
E-mail: josem.cela@bsc.es, *Web page:* http://www.bsc.es

Abstract. The objective of this document is to present the performance problems detected in the elsA program in parallel environments of a great number of processors and the proposed solutions. The two main problems detected were the communication scheduling and the load balancing. The proposed communication scheduling, based in the graph colouring problem, has maximized the number of communications done concurrently and significantly has reduced the communication phase time. With respect to load balancing, a new clustering algorithm based in METIS was developed to assign blocks to processes with a negligible improvement. So future developments will require deep modifications of the elsA source codes.

Key words. elsA, performance, communications scheduling, load balancing, graph colouring

1. Introduction

ElsA [1, 2, 3] is a software dedicated to numerical simulation of single-species laminar or turbulent compressible flows, on 3D block-structured grids.

The equations to be solved are the Navier-Stokes (NS) equations, in which turbulence is modeled via a statistical approach. By carrying out the averaging operation upon the NS equations, one obtains the Reynolds Average Navier-Stokes (RNAS) equations. Finally, these equations are expressed in the general Arbitrary Lagrangian-Eulerian (ALE) formulation, so that arbitrary grid motions can be taken into account.

The parallelization of elsA is based on a parallel (sequential) iteration-by-subdomain method. The variable which ensure the coupling between the subdomains is the total flux (pressure+convective+diffusive), which consists of the terms of the momentum equations that have been integrated by parts. The algorithm is:

- Impose initial conditions
- Do while not convergence:
  - Advance in time
  - Solve the n subdomains simultaneously
  - Exchange the fluxes on the subdomains interfaces

This iteration-by-subdomain domain decomposition method can be viewed as an interactive method for solving the Schur complement $Au = n$ of the subdomains, that is the unknown on the interfaces u. Each iteration of this iterative method is a Richardson iteration for the system:

$$u^{k+1} = P^{-1}(b - Au^k) + u^k \tag{1}$$

The preconditioner P depends on the data that are exchanged between the interfaces to achieve the coupling between the subdomains. Let us mention the Dirichlet/Neumann and Robin/Robin methods. In any case, the condition number of P goes like $O(h^{-1}H^{-1})$, where h is the element size and H is the subdomain size.

2. Initial situation

The elsA program had scalability problems in parallel environments of a great number of processors.

Our work was concentrated in obtain traces of the parallel execution of elsA in order to analyze the parallel performance and propose a solution. This task was performed using *PARAVER* tool [4].

Two main problems were detected:

- Communication scheduling
- Load balancing

3. Communication scheduling

The fluxes exchanges between neighbour blocks are performed using MPI blocking communications primitives [5]. Several blocks can be mapped on a single MPI process. There is only MPI communications associated with boundaries between blocks mapped on different MPI processes. There are two different kinds of MPI communications:

1. *MPI_sendreceive* primitive is used for boundaries where the meshes fit perfectly in both sides (in both blocks).
2. *MPI_broadcast* primitive is used in boundaries where the meshes do not fit perfectly. Moreover this kind of boundaries could be between more than 2 blocks. Then, the broadcast is done between the MPI processes associated with this boundary.

The figure 1 shows an example of both kinds of boundaries.

The communications are organized by boundaries and they are not grouped in a single exchange. This means that if 2 blocks have as common boundary through 3 non connected surfaces, there will be 3 exchanges (`MPI_sendreceive`) between these blocks.

The cost of the communication phase depends on three factors:

- the kind of boundary connections between blocks (MPI primitive and message length)
- the scheduling to perform the exchanges (number of steps to complete all the communications)
- the mapping of block to processes (total amount of communications)

The boundaries depend on the mesh generation. The exchanges scheduling is a scheduling optimization problem. The blocks-to-processes mapping is a clustering optimization problem.

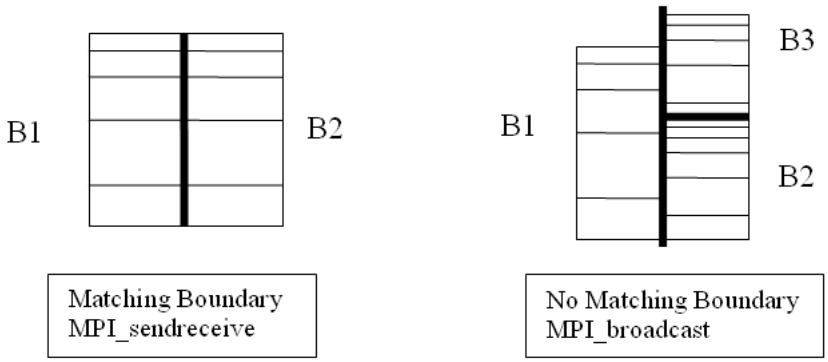


Figure 1. Kinds of boundaries.

3.1. The scheduling problem

The communications to be performed in a time step could be represented by a graph [6, 7]. Each node of the graph is associated with a MPI process. Each edge of the graph is associated with a MPI communication in both directions. The objective of the scheduling algorithm is to eliminate all the edges in the communication graph in the minimum number of steps, with the restriction that in each step we can not eliminate edges going out/in of the same node.

There is a difference in the graph between the `MPI_sendreceive` and the `MPI_broadcast` type communications. For the `MPI_sendreceive` communications the edges in the graph could be scheduled individually. For the `MPI_broadcast` communications, the edges between all the nodes of a single broadcast are special edges. These edges must be scheduled altogether. We call clique the set formed by all the nodes/edges in a single broadcast.

We start with the `MPI_sendreceive` type communications. In those communications the algorithm executed by each MPI process is the following:

```
Loop on my neigbours
    MPI_sendreceive (my_boundary, neigh_boundary)
    Process the received neigh_boundary
End Loop
```

The order in the loop determines the order of the communications. Given a process A, the first neighbour in its loop is the process B. The process A will be blocked until the `MPI_sendreceive` primitive is completed. If any other process (C) wants to communicate with the process A, it will have to wait until the present communication between A and B has finished. So, the process C will be blocked and it will remain in idle time. However, process C could start a communication with other neighbour different from A.

The figure 2 is an example that the current communication schedule in elsA creates too much idle time on the process (e.g. process 99 has to wait blocked until process 28 finishes the previous communications, when process 117 is ready to start communications).Our target will be to organize the exchange operations in such a way that maximum number of exchanges can be done concurrently and also avoiding the blocking of processes.

3.2. Algorithm to find the optimum communication scheduling for `MPI_sendreceive` communications

The new algorithm is based in the graph colouring problem [8]. This problem is known to be NP-complete. Therefore, for a general graph it is not possible to compute an optimal solution in a

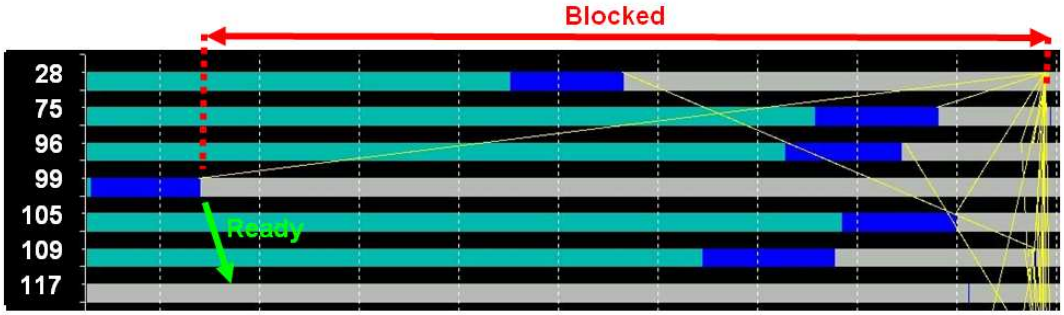


Figure 2. Example of a blocked process that could start a communication.

reasonable amount of time. This fact has led us to develop a heuristic approach. The heuristic algorithm that we apply can provide quasi optimal solutions.

We know that the minimum number of step to complete the communication graph is the maximum degree of any node in the graph (degree is the number of neighbour nodes). Our algorithm provides a scheduling that is optimum or it has one step more that the optimum.

In the following we describe the algorithm:

1. **Build a graph (G) that describes the communications between the processes.**
   Each node represents a process, which has associated a set of blocks in the elsA input mesh. The edges are the boundaries between blocks of different processes. So, an edge between $node_i$ and $node_j$ represents a common boundary between $process_i$ and $process_j$, and its weight represents the number of different boundaries between themselves [7]. There will be an exchange communication in elsA for each common boundary.

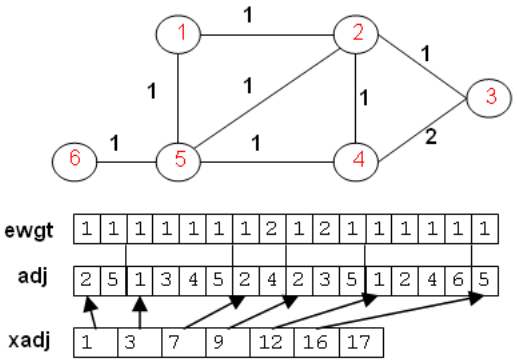


Figure 3. The graph (G) with the communications between the processes.

2. **Build the dual graph (DG) from the previous graph (G).**
   In this new graph, figure 4, the nodes represent the edges of the original graph (G) and its weight is the weight of these edges. The nodes are connected whenever the edges in the original graph (G) shares a common node. So, each node represents a communication between processes.

3. **Colouring the dual graph (DG) using the minimum colours.**
   It is an assignment of colours to the nodes of the graph, with the condition that there is not two adjacent nodes assigned to the same colour. Here, "adjacent" means sharing the same edge. The "colour" is a positive integer number (e.g. 0, 1, 2...) that represents the step to perform

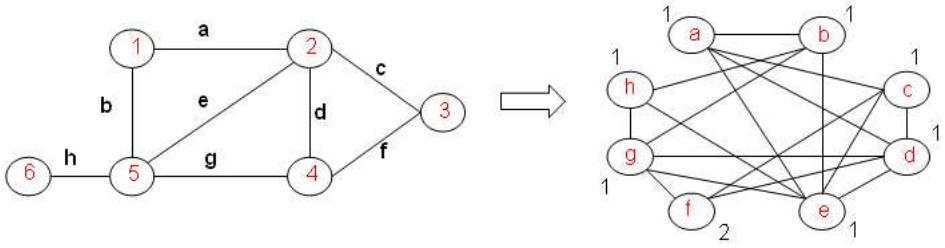
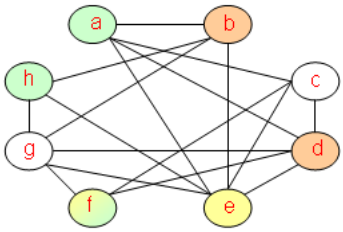


Figure 4. The dual graph (DG) from graph (G).

the communication, colour=0 means first step, colour=1 means second step, and so on. The chosen colour has to be the smallest number that no neighbour node already owns.

The problem of finding a minimum colouring of a graph is NP-hard. The heuristic to decide the order to assign colours to nodes is the degree of the nodes (in other words, the number of edges incident to each node) multiplied by its weight. The node with maximum heuristic is the first to choose colour, and it will choose as many colours as its weight. The idea of this heuristic is to schedule as soon as possible the potential bottlenecks of the graph. So, it is easier to assign firstly the nodes with higher degree because they need more colours and there are more possibilities to have conflicts with colours of other adjacent nodes. Moreover, the treatment of the nodes with lower degree, at the end, will get to fill easier the holes, because they will have fewer possibilities to have conflicts with the colours of other nodes.

The maximum degree of the graph limits the minimum number of colours in the graph. In the figure 5, the degree of nodes 5 and 2 is 4, so it is needed at least 4 colours to colour the entire graph. In this case, the algorithm gets an optimum solution, because it only uses 4 colours.



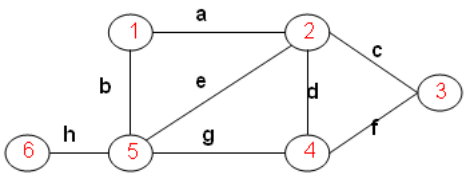
Node	Degree * weight	Order	Invalid color	Chosen color
a	4*1 = 4	5	0, 2	1
b	4*1 = 4	6	0, 1, 3	2
c	4*1 = 4	7	0, 1, 2	3
d	5*1 = 5	3	0, 1	2
e	6*1 = 6	1	.	0
f	3*2 = 6	2	.	0,1
g	5*1 =5	4	0, 1, 2	3
h	3*1 = 3	8	0, 2, 3	1

Figure 5. Colouring the dual graph (DG).

4. **Fill the communication scheduling.**
Now, each node of the dual graph DG (this node represents a `MPI_sendreceive`) has colours assigned, and each colour is the step assigned to perform a communication. We represented these assignments in a table. Each row of the table shows the pairs of MPI processes performing a communication in the same time step.

   For example, in the figure 6 we see at step 0 there are two simultaneous communications: one between processes 2 and 5 and other one between processes 3 and 4. On the other side, an empty cell in the table means that this process has not any communication at this time step. For example, process 1 does not perform any communications in steps 0 and 3.

   Finally, the table is written in a file for a post-processing.



Step	Nodes
0	e, f
1	a, f, h
2	b, d
3	c, g

Step	Process					
	1	2	3	4	5	6
0		5	4	3	2	
1	2	1	4	3	6	5
2	5	4		2	1	
3		3	2	5	4	

Figure 6. Communication scheduling.

3.3. Algorithm to find the optimum communication scheduling for `MPI_broadcast` communications

This algorithm is a variation of the previous algorithm, so the steps are briefly explained. The algorithm is based in the graph colouring problem too:

1. **Build a clique graph (CG) that describes the communications between the cliques.** Each node represents a clique, which is formed by all the nodes/edges in a single broadcast. The weight of the node represents the number of processes contained in the clique. An edge represents that there is a common process in the two cliques.

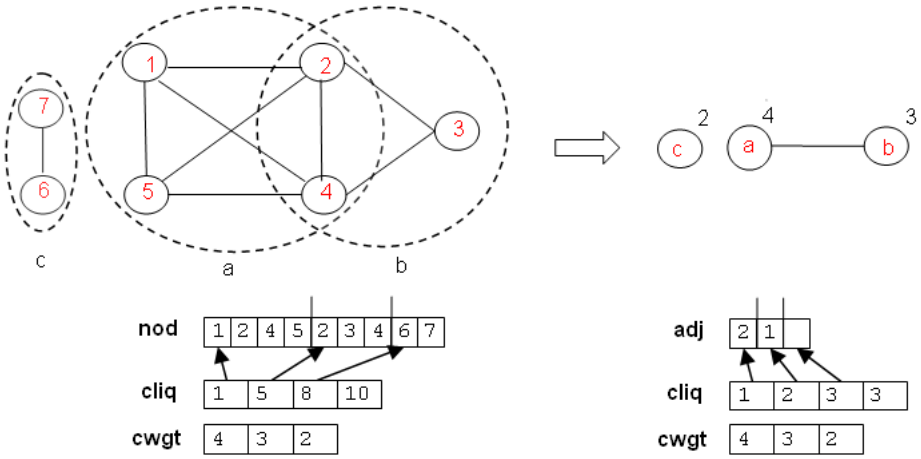


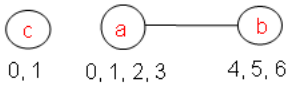
Figure 7. The clique graph (CG) with the communications between cliques.

2. **Colouring the clique graph (CG) using the minimum colours.** It is an assignment of colours to the nodes of the graph, with the condition that there is not two adjacent nodes assigned to the same colour. Here, "adjacent" means sharing the same edge. The "colour" is a positive integer number (e.g. 0, 1, 2...) that represents the step to perform the communication.

   We have chosen the same heuristic than previous case. The heuristic to decide the order to assign colours to nodes is the degree of the nodes (in other words, the number of edges incident

to each node) multiplied by its weight. The node with maximum heuristic is the first to choose colour, and it will choose as many colours as its weight.

In the figure 8, clique a and clique b have common nodes, so it is needed at least 4 colours to colour clique a and 3 colours to colour the clique b. In this case, the algorithm gets an optimum solution, because it only uses 7 colours.



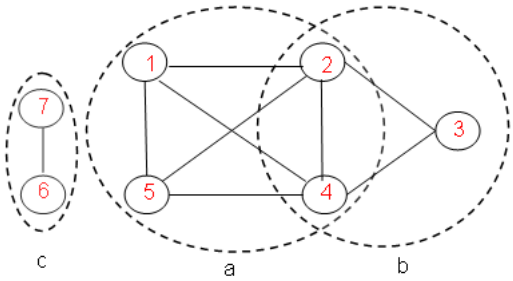
Node	Degree * weight	Order	Invalid color	Chosen color
a	1*4=4	1	.	0,1,2,3
b	1*3=3	2	0,1,2,3	4,5,6
c	0*2=0	3	.	0,1

Figure 8. Colouring the clique graph (CG).

3. **Fill the communication scheduling.**
Now each node of the clique graph CG (this node represents a `MPI_broadcast`) has colours assigned, and each colour is the step assigned to perform a communication. We represented these assignments in a table. Each row of the table shows the MPI processes performing a broadcast communication in the same time step. Data is broadcast from the root process to all other processes of the clique.

For example, in the figure 9 we see at step 0 there are two simultaneous communications: one broadcast from process 1 to processes 2, 4 and 5 and other broadcast from process 6 to process 7. On the other side, an empty cell in the table means that this process has not any communication at this time step. For example, process 3 does not perform any communications between steps 0 and 3.

Finally, the table is written in a file for a post-processing.



Step	Nodes
0	a, c
1	a, c
2	a
3	a
4	b
5	b
6	b

Step	Process						
	1	2	3	4	5	6	7
0	1	1		1	1	6	6
1	2	2		2	2	7	7
2	4	4		4	4		
3	5	5		5	5		
4		2	2	2			
5		3	3	3			
6		4	4	4			

Figure 9. Communication scheduling.

3.4. Software developed

The new scheduling algorithms are external to elsA code. These algorithms could be run in a pre-processing phase. The following programs have been developed:

build.py Python program [9].
It extracts the block-to-processor assignments and the boundaries definitions from elsA python script. Then, it generates the input file for schedule program.

schedule Fortran program. Sources: `entorn.F`, `grafs.F` and `schedule.F`
It computes the new communication scheduling. The input is a graph which describes the communications between processes. The output is the communication order for each process.

build2.py Python program.
It modifies the elsA python script to add the new communication scheduling.

elsA_stub.py Python program.
It contains a collection of redefined elsA classes. These classes make possible to process the elsA file script with our new functionalities.

runShed.sh Bash shell script.
It runs the previous programs in the correct order.

```
# Call build.py to get input graph
python ./build.py elsAin.epy > tmp.in

# Execute the scheduling program
./schedule <tmp.in >tmp.out

# Call build2.py to create new python file
python ./build2.py elsAin.epy <tmp.out >newin.epy
```

3.5. Results

The benchmarks used are:

	Benchmark A	**Benchmark B**
Number of cells:	13,998,784	10,332,096
Number of blocks:	265	766
Largest block size (cells):	121,841	48,909
Smallest block size (cells):	1,053	729

The figure 10 shows the number of steps to complete all the communications with the original elsA scheduling and the new scheduling. This figure is obtained from benchmark A. We can observe that the new scheduling reduces by a factor of 10 the number of steps.

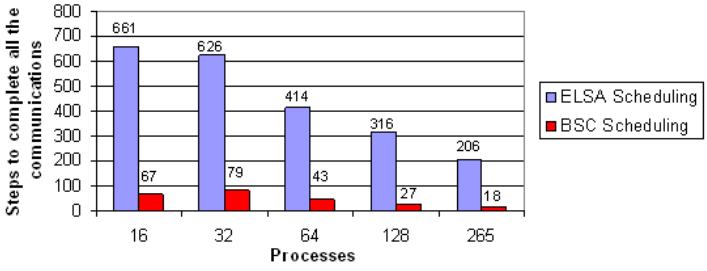


Figure 10. The number of steps to complete all the communications.

The first consequence is the reduction of the time spent for communication phase. The impact of this improvement in the total time depends on the ratio communications vs. computation. The figure 11 shows the time spent in one elsA iteration: in computation, communication and both. This figure is obtained from the benchmark B with 128 processes using *PARAVER* [4] to measure the length of each phase inside a time step. We can observe that with the new scheduling the communication time has been reduced by a factor of 2.67, and the total time has been reduced by a factor of 1.71.

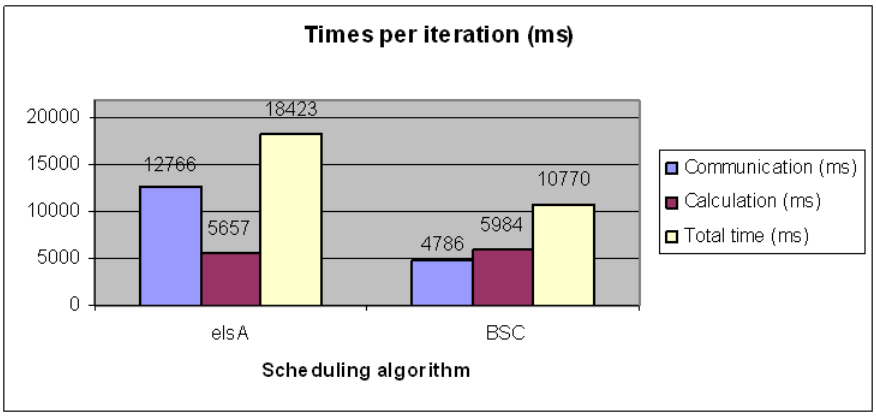


Figure 11. Time spent in one elsA iteration.

We also estimate the time step duration from a direct measure on elsA (figure 12). We measure the total time for 50 and 10 time steps with the two test cases using 128 processes. We calculate the average time per iteration. The gotten speedup with the new scheduling algorithm is 1.82 (for benchmark A) and 1.70 (for benchmark B).

	Benchmark A			Benchmark B		
Scheduling	**10 iter. time (s)**	**50 iter. time (s)**	**1 iter. time (s)**	**10 iter. time (s)**	**50 iter. time (s)**	**1 iter. time (s)**
elsA	886	1687	20,03	591	1320	18,23
BSC	801	1241	11,00	522	952	10,75
Speedup			1,82			1,70

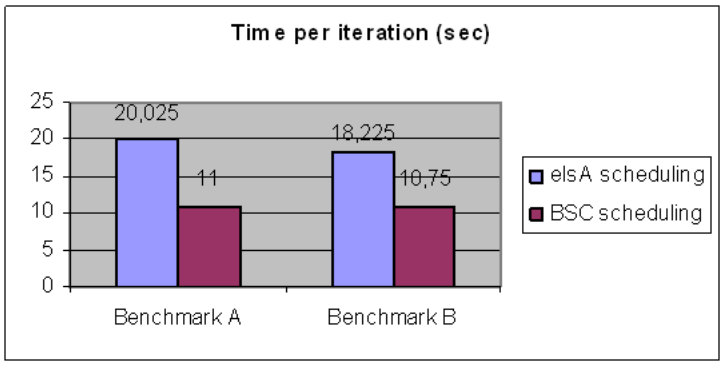


Figure 12. Time spent per iteration.

Finally, we visualize the results of the benchmarks with *PARAVER* in order to show graphically the real impact of our communication scheduling in the global performance of elsA.

The figure 13 is the visualization for the benchmark B using 128 processes. The two pictures have the same time scale. The upper picture represents the execution of 1 time step iteration with the original elsA scheduling algorithm. The lower picture represents the execution of 1 time step iteration with the new BSC scheduling algorithm.

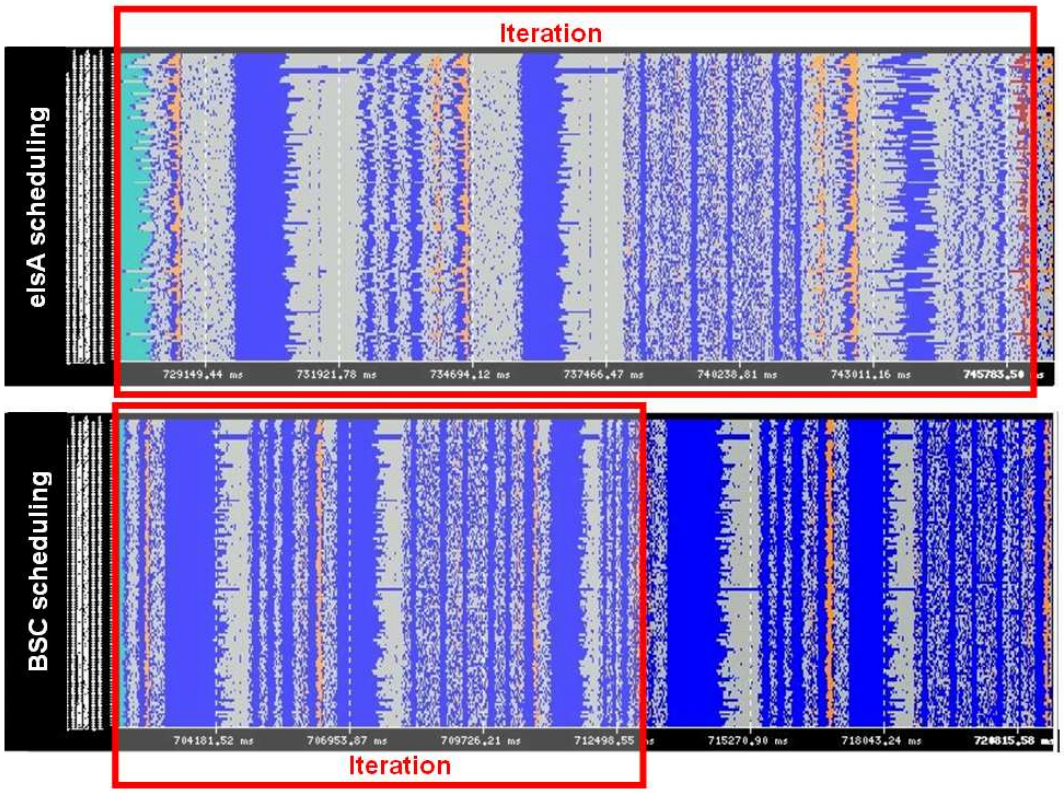


Figure 13. Trace of an elsA execution.

Blue colour means CPU active. Gray colour means communications phase (`MPI_sendreceive`). Orange colour means MPI collective communications.

We can observe the reduction time in the whole time step iteration with the new scheduling. We can observe that the reduction is due to the reduction of the Gray colour (communication time).

4. Load balancing

In order to reduce the amount of communications, it is also important to improve the mapping of blocks to processes. This problem is connected with the load balancing problem.

The bigger weakness of elsA algorithm from the point of view of parallel execution is related with the multiblock mesh. It is clear that the original mesh must be decomposed in smaller blocks in order to achieve a good computational load balancing. However, when the number of blocks increases, the

amount of computational effort also increase, due the converge problems associated to the iteration-by-subdomain method. The optimum trade off is problem dependent.

Moreover, the load balancing is coupled with the block to process mapping problem. The optimum mapping is a clustering of neighbour blocks in a single process, but this clustering may be not good for load balancing. So, the load balancing tool must to take into account not only the load of the blocks, it must pay attention also to the topology of the block connections. The figures 14 and 15 show the load unbalance in cells and time. The load unbalance is measured using two values:

$$UnbalanceMax = \frac{Max - Min}{Max} \qquad UnbalanceMin = \frac{Max - Min}{Min} \tag{2}$$

The figures 14 and 15 show the unbalance in order to emphasize that there is not a linear relation between cell and time unbalance. This is mainly due to the fact that each MPI process has a different cache miss ratio, because the ordering to access the data is different in each process. This ordering depends on multiple factors: number of blocks in the process, Multigrid ordering used in each block, etc. Moreover, there are other unbalance factors that are different in each process: the operative system preemptions, and those associated with the fluid dynamics simulations nature, for example the shock capturing.

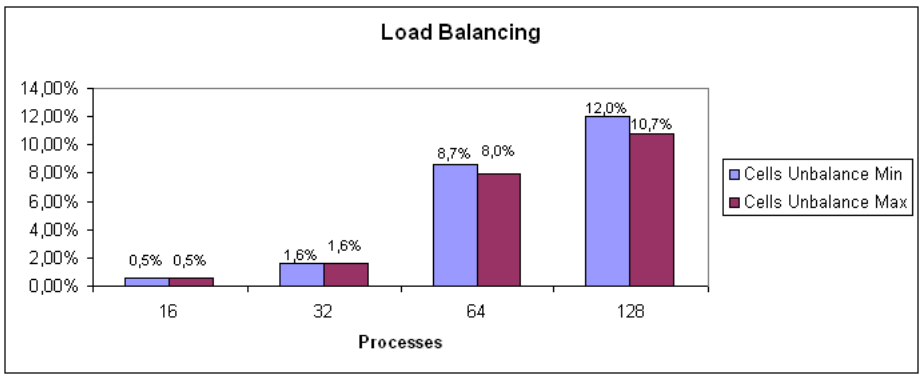


Figure 14. The load balance with cells.

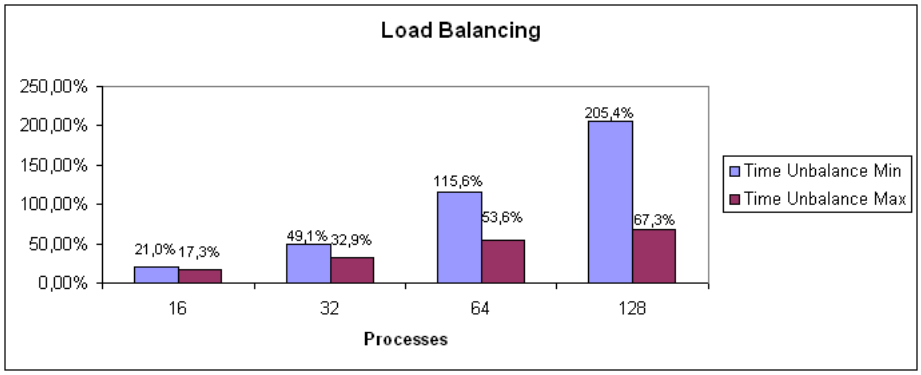


Figure 15. The load balancing with time.

Note that in this case with just 64 processes the time unbalance Max is around 50%, i.e. the fastest process spends half of the time of the slowest process.

We have test a new clustering algorithm to assign blocks to processes. This algorithm is based in METIS that is known as the best heuristic to solve this NP-complete problem. However, the improvement in the time load balancing is negligible. This means that from the point of view of number of cells assigned to each MPI process the present elsA algorithm is good enough.

The problem is not in the number of cells associated with each MPI process, it is in the cache miss ratio of each MPI process. To modify this cache miss ratio we must modify the orderings in the block grid, but this order affects also the numerical properties of the multigrid algorithm used by elsA. Then this remains as a future improvement which requires deep modifications of elsA internals.

The figure 16 is gotten from an execution of benchmark B using 128 processes. It shows the distribution of instructions per cycle, IPC, (X axis) for each process (Y axis). We identify two regions with different IPC (see yellow lines) caused by different regions of the code. The scattered distribution of the points inside these regions suggests that there is a load unbalance. However, this unbalance comes from the block-to-processor distribution because the pattern in both areas is similar.

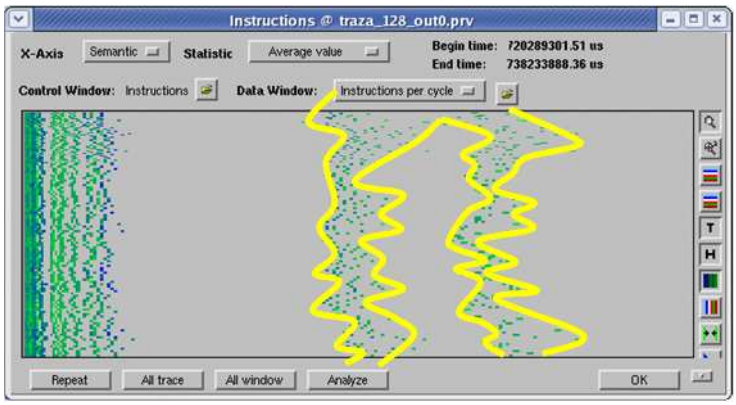


Figure 16. The distribution of IPC (X axis) for each process (Y axis).

The figure 17 shows the instruction latency per process. In this case, we can observe a great dispersion of the length of the instructions (see yellow lines). There is a big unbalance between the processes, and it comes from the algorithm: cache miss ratio, different fluid conditions, S.O. preemptions, etc.

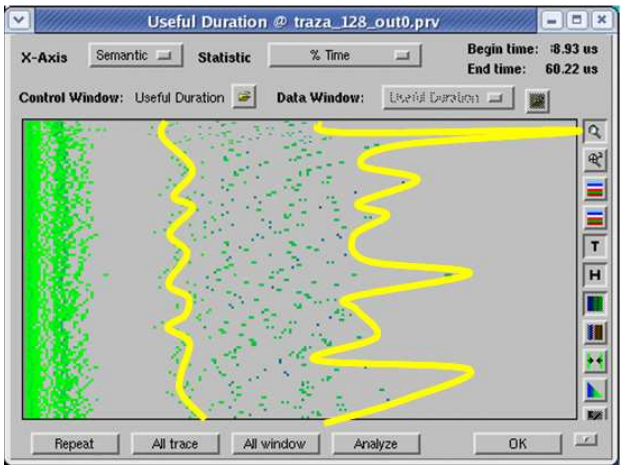


Figure 17. The instruction latency per process.

References

1. M. Lazareff. *User's Reference Manual*. ONERA, 3.1 edition, December 2005.
2. M. Gazaix. *Design and Implementation Tutorial*. ONERA, 1.0 edition, January 2005.
3. M. Gazaix. *Development process tutorial*. ONERA, 1.2 edition, January 2003.
4. Vincent Pillet, Jesús Labarta, Toni Cortés, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *Transputer and occam Developments*, pages 17–32, April 1995. `http://www.bsc.es/plantillaA.php?cat_id=485`.
5. Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.
6. David W. Walker and Steve W. Otto. Redistribution of block-cyclic data distributions using mpi. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.
7. Frédédéric Desprez, Cyril Randriamaro, Jack Dongarra, Antonie Petitet, and Yves Robert. Scheduling block-cyclic array redistribution. *IEEE Trans. Parallel Distrib. Syst.*, 9(2):192–205, 1998.
8. C. Berge. *Graphs and Hypergraphs*. Elsevier Science Ltd, 1985.
9. Guido van Rossum. *Python tutorial*. BeOpen PythonLabs, 2.5.2 edition, February 2008. `http://docs.python.org/tut/`.

