



# **Tutorial for the *elsA* version 3.1 interface**

## **Parallel Usage and Load Balancing**

M. Gazaix

ONERA Châtillon, March 23, 2006



# Contents

## Contents .1

### I Parallel Computation

#### 1. Parallel Computation with elsA

1.1 General description

1.2 MPI execution

1.3 Block allocation to processors

1.4 Differences between Parallel and Sequential mode

1.5 Debugging MPI problems

### II Load Balancing

#### 2. Load balancing

2.1 General description

2.2 Automatic mode

2.3 Manual mode

- 2.4 Main options (1)
- 2.5 Main options (2)
- 2.6 Generation of scripts and splitted files
- 2.7 2nd Step: launch (generated) parallel script
- 2.8 Mapping of blocks to processors
- 2.9 Additional Information (1)
- 2.10 Additional Information (2)
- 2.11 Additional Information (3)

Tutorial for the *e/sA* version 3.1 interface

---

part I

Parallel Computation

---

ONERA

The logo for ONERA, consisting of the word "ONERA" in a bold, sans-serif font. Below the text is a thin blue horizontal line, and below that is a thicker blue curved line that arches over the text.

## Parallel Computation with elsA

### General description

- **MPI paradigm**
- **Coarse-grained parallelism**
  - ▷ each block is allocated to a single processor
  - ▷ each processor can own several blocks
  - ▷ a load balancing tool is available inside *elsA*
- **OpenMP**
  - ▷ Only limited experience
    - ⇒ *Not available in supported versions*

## Parallel Computation with elsA

### MPI execution

- **Specific MPI Parallel executable**

- ▷ Example: `ELSAPROD=pgi_r8_mpi`

- ▷ `$ELSAHOME/Dist/bin/$ELSAPROD/elsA.x`

- `(LD_LIBRARY_PATH=$ELSAHOME/Dist/bin/$ELSAPROD)`

- ▷ However, parallel executable can work with `NPROC=1`

- (and should give identical result)*

- **Launch of MPI application: specific to platform and batch system**

- `mpirun -np 4 elsA.x script.py`

- `prun -n 4 elsA.x script.py`

- `mpich: ...`

## Parallel Computation with elsA

### Block allocation to processors

- **Required for every block object (*not* for mesh, extract, ...)**
- **Two options**
  - ▷ 1. `<block>.proc = procId (0 < procId < NPROC - 1)`
  - ▷ 2. `<cfdpb>.mpi_block2proc = 'script_block2proc'`
    - ⇒ *currently, requires <cfdpb>.automatic\_block\_gen*
    - ⇒ *used by load balancing tool*
- **Consider using environment variable `ELSA_MPI_MODULO_PROC`**
  - ▷ **Often convenient to select NPROC without modifying Python script**
    - Ex: Mapping pre-computed, assuming NPROC=12*
    - ELSA\_MPI\_MODULO\_PROC=2: run with NPROC=6*
    - ELSA\_MPI\_MODULO\_PROC=3: run with NPROC=4*
    - ELSA\_MPI\_MODULO\_PROC=4: run with NPROC=3*
    - ELSA\_MPI\_MODULO\_PROC=6: run with NPROC=2*

## Parallel Computation with elsA

### Differences between Parallel and Sequential mode

- **Standard output (logfile) redirection**

- ▷ `elsA_MPI_Pid_xxxx_N_0, ...`

- ⇒ *Avoid scrambled logfile on some platforms*

- When debugging, look carefully to each `elsA_MPI_xxx` logfiles*

- **Extraction process works identically, but:**

- ▷ `file_position='append'` may fail

- ▷ `var='viscrapp', with loc='node'` does not work

- **Chimera algorithm may give different results**

- not a bug*

## Parallel Computation with elsA

### Debugging MPI problems

- **Can be hard work**
  - ▷ Error messages are burried inside several logfiles
  - ▷ Challenging specially if NPROC large
    - ⇒ *use of batch queuing system, response time can be slow*
- **Parallel executable cannot be run interactively**
  - ▷ Interactive script debugging must be done with the sequential executable
- **If possible, strongly recommended to run single processor, preliminary computation**
  - ▷ `<cfdpb>.coarsen_mesh=2`: → memory divided by 8
  - ▷ mapping of block to processor: ignored if NPROC=1
- **Probably useful also to run the coarsened configuration with the selected NPROC**

Tutorial for the e/sA version 3.1 interface

---

part II  
Load Balancing

---

ONERA

The logo for ONERA, consisting of the word "ONERA" in a bold, sans-serif font. Below the text is a thin blue horizontal line, and below that is a thicker blue curved line that arches over the text.

## Load balancing

### General description

- **Two step process**
- **1. Run the slightly modified script (in sequential mode)**
  - ▷ the load balancing algorithm selects the "best" partition
  - ▷ production of converted parallel scripts
  - ▷ if splitting occurs, production of splitted files (mesh,...)
- **2. Run the parallel script (generated in 1st step) in parallel mode**

## Load balancing

### Automatic mode

- **Automatic mode** : `<cfdpb>.load_balance(nb_proc)`

- ▷ original script modification:

*replace compute( ) with load\_balance( nb\_proc )*

*<cfdpb>.extract( ) will be ignored*

```
naca = cfdpb()
```

```
...
```

```
# compute "best" partitioning for 2 processors
```

```
naca.load_balance(2)
```

# Load balancing

## Automatic mode

### Output (1)

```
Splitting algorithm called (Option : 'greedy')  
(nbCoarseGrid = 0)
```

```
----- Begin load_balance algorithm -----
```

```
-----  
'greedy' split algorithm (single direction splitting, A. Ytterstrom)
```

```
-----  
Split Algo Info : split (oldId = 0, DIR_I, pos = 129) # --> newId = 1 ( 129 X 33 X 2)  
# --> oldId = 0 ( 129 X 33 X 2)
```

```
-----  
=====  
Number of Processor          =      2  
Total Number of blocks      =      2  
Total Number of cells       =    8192  
Ideal Number of cells / processor =    4096  
Max. Number of cells (proc: 0) =    4096
```

```
Maximum load balance error is 0.00 %  
Theoretical speedup is reduced from 2 to 2.00  
Interprocessor message size : 128
```

# Load balancing

## Automatic mode

### Output (2)

```
-----  
Proc : 0 owns 1 block(s)  
Proc. Number of cells = 4096  
Load balance error = 0.00%  
-----  
Block 0 Nb of cells = 4096 [ 129 33 2]  
-----  
Proc : 1 owns 1 block(s)  
Proc. Number of cells = 4096  
Load balance error = 0.00%  
-----  
Block 1 Nb of cells = 4096 [ 129 33 2]  
-----  
----- End load_balance algorithm -----  
Proc : 0 owns 1 block(s)  
Proc. Number of cells = 4096  
Load balance error = 0.00%  
-----  
Block 0 Nb of cells = 4096 [ 129 33 2]  
-----  
Proc : 1 owns 1 block(s)  
Proc. Number of cells = 4096  
Load balance error = 0.00%  
-----  
Block 1 Nb of cells = 4096 [ 129 33 2]  
-----  
----- End load_balance algorithm -----
```

## Load balancing

### Manual mode

□ Manual mode : `<cfdpb>.split_man()`

```
naca = cfdpb()  
...  
# split block 0 in 2 blocks:  
# old block 0 is resized to : [ 1, 29]  
# new block 1 is           : [29, imax]  
naca.split_man(0, DIR_I, 29)  
  
# compute "best" partitioning for 2 processors  
naca.load_balance(2)
```

# Load balancing

## Manual mode

### Output

```
Split Algo Info : split (oldId = 0, DIR_I, pos = 29) # --> newId = 1 ( 229 X 33 X 2)
# --> oldId = 0 ( 29 X 33 X 2)
```

```
Splitting algorithm called (Option : 'greedy')
(nbCoarseGrid = 0)
```

```
----- Begin load_balance algorithm -----
```

```
-----
Split Algo Info : split (oldId = 1, DIR_I, pos = 129) # --> newId = 2 ( 101 X 33 X 2)
# --> oldId = 1 ( 129 X 33 X 2)
-----
```

```
Proc : 0 owns 1 block(s)
Proc. Number of cells = 4096
Load balance error = 0.00%
```

```
-----
Block 1 Nb of cells = 4096 [ 129 33 2]
```

```
-----
Proc : 1 owns 2 block(s)
Proc. Number of cells = 4096
Load balance error = 0.00%
```

```
-----
Block 2 Nb of cells = 3200 [ 101 33 2]
Block 0 Nb of cells = 896 [ 29 33 2]
```

## Load balancing

### Main options (1)

- `<cfdpb>.load_balance_algo='no_split'`
  - ▷ forbid any block splitting, not usable if  $nb\_block < nb\_proc$ 
    - ⇒ if not set, Manual and Automatic mode can be combined
- `<cfdpb>.split_save_memory=1 (default)`
  - ▷ Useful for massively parallel computations
    - ⇒ load balancing algorithm runs in single processor mode  
(sequential or parallel elsA executable)
  - ▷ Usable only if *trirac* data explicitly specified
- `<cfdpb>.split_grid_size_min`
  - ▷ force minimum grid size (in IJK direction)

## Load balancing

### Main options (2)

- `<numerics>.nb_coarse_grid`
  - ▷ Number of coarse levels allowed with splitted configuration
- `<cfdpb>.split_eps`
- `<cfdpb>.load_balance_algo`
  - ▷ `<cfdpb>.load_balance_algo='greedy'` (default)
  - ▷ `<cfdpb>.load_balance_algo='recursive_bisection'`
- `<cfdpb>.script_cfd`
  - ▷ Control name of generated driver script
- `<cfdpb>.script_topology`
  - ▷ Control name of generated topology script
- `<cfdpb>.script_block2proc`
  - ▷ Control name of generated mapping script

## Load balancing

### Generation of scripts and splitted files

```
Pb.set('split_save_memory', 1)
Pb.load_balance(64)

# generate:
# script_cfd_u.py           # main driver script
# script_block2proc.py     # block allocation to processors
# script_bndphys_u.py
# script_topology.py
Pb.print_script()

# default: Mesh_Split/mesh*
Pb.split_init_file()
```

## Load balancing

2nd Step: launch (generated) parallel script

□ `mpirun -np nb_proc elsA.x script_cfd_u.py`

```
naca = cfdpb(name='naca')
...
naca.set_block_creation_mode('automatic')
naca.set('automatic_block_gen', 'db_directory')

naca.set('cfd_mesh_dir',          'Mesh_Split')
naca.set('cfd_init_state',       'stateInf')

...
import script_bndphys_u
import script_topology

#=====
# Block allocation to processors (// MPI)
#=====
if get_nb_proc() > 1:
    naca.set('mpi_block2proc', 'script_block2proc')
```

## Load balancing

### Mapping of blocks to processors

□ `<cfdpb>.mpi_block2proc='script_block2proc'`

Generated File: `script_block2proc.py`

```
# Splitted configuration : 2 blocks
```

```
# Number of processors      :  
nproc = 2
```

```
dict_block2proc = {  
    0 : 0,  
    1 : 1,  
}
```

```
# === End of script script_block2proc ===
```

## Load balancing

### Additional Information (1)

□ **A single driver script can be used in both context**

- ▷ Look to example `$PYTHONPATH/ParelsA.py`
  - ▷ Use command-line option: `-p nb_proc`
  - ▷ or environment variable (Ex: `ELSA_NB_PROC`, `ELSA_MODE`)
  - ▷ Minimize manual script file management
- ⇒ *can be convenient in production environment*

□ **1st step:**

```
export ELSA_MODE=LOAD_BALANCE; export ELSA_NB_PROC=4  
mpirun -np 1 elsA.x driver.py
```

⇒ *creation of splitted config. (topology, mesh ...)*

□ **2nd step:**

```
export ELSA_MODE=COMPUTE  
mpirun -np 4 elsA.x driver.py
```

## Load balancing

### Additional Information (2)

```
import os;      SPROC = os.environ['ELSA_NB_PROC']
import string;  NPROC = string.atoi(SPROC)

ELSA_MODE = os.environ['ELSA_MODE']

if ELSA_MODE == 'LOAD_BALANCE':
    MeshDir = 'Mesh_Unsplitted'
    split_mesh_dir = 'Mesh_Split_' + SPROC
    if not os.access(split_mesh_dir, os.W_OK):
        os.mkdir(split_mesh_dir)
    else:
        command = 'rm -f ' + split_mesh_dir + '/*'
        os.system(command)

    split_topo      = 'script_topo_'      + SPROC
    split_block2proc = 'ascript_block2proc_' + SPROC

else:
    MeshDir = 'Mesh_Split_' + SPROC
    script_topo      = 'script_topo_'      + SPROC
    script_block2proc = 'ascript_block2proc_' + SPROC
```

## Load balancing

### Additional Information (3)

```
naca.setS('cfld_mesh_dir',      MeshDir)
naca.setS('cfld_mesh_dir',      MeshDir)
...
if ELSA_MODE == 'LOAD_BALANCE':
    import script_topo_unsplitted
else:
    __import__(script_topo)
    if get_nb_proc() > 1:
        naca.set('mpi_block2proc', script_block2proc)
    ...

if ELSA_MODE == 'LOAD_BALANCE':
    naca.load_balance(NPROC)
    naca.print_script_topology (split_topo)
    naca.print_script_block2proc(split_block2proc)
    naca.set('split_mesh_dir', split_mesh_dir)
    naca.split_init_file()
else:
    naca.compute()
    naca.extract()
```