Distributor2 2.5

Christophe Benoit, Xavier Juvigny, Stephanie Peron, Pascal Raud
- Onera -

1 Distributor2: block distribution module

1.1 Preamble

This module provides functions to distribute blocks on a given number of processors. At the end of the process, each block will have a number corresponding to the processor it must be affected to for a balanced computation, depending on given criterias. This module doesn't perform splitting (see the Transform module for that).

This module is part of Cassiopee, a free open-source pre- and post-processor for CFD simulations.

To use the module with the Converter array interface, you must import it as:

```
import Distributor2 as D2
```

To use the module with the pyTree interface, you must import it as:

```
import Distributor2.PyTree as D2
```

1.2 Automatic load balance

D2.distribute: distribute automatically the blocks amongst N processors.
With the array interface, where A is a list of blocks:

- prescribed is a list of blocks that are forced to be on a given processor. prescribed[2] = 0 means that block 2 MUST be affected to processor 0.
- perfo is a tuple or a tuple list for each processor. Each tuple describes the relative weight of solver CPU time regarding the communication speed and latence (solverWeight, latenceWeight, comSpeedWeight).
- weight is a list of weight for each block indicating the relative cost for solving each block.
- com is a ixj matrix describing the volume of points exchanged between bloc i and bloc j.
Algorithm can be chosen in: 'gradient', 'genetic', 'fast'.

The function output is a stats dictionary. stat['distrib'] is a vector describing the attributed processor for each block, stats['meanPtsPerProc'] is the mean number of points per proc, stats['varMin'] is the minimum variation of number of points, stats['varMax'] is the maximum variation of number of points, stats['varRMS'] is the mean variation of number of points, stats['nptsCom'] is the number of points exchanged between processors for communication, stats['comRatio'] is the ratio of points exchanged between processors in this configuration divided by the total number of points needed in communications, stats['adaptation'] is the value of the optimized function:

```
stats = D2.distribute(A, NProc, prescribed=[], perfo=[], weight=[], com=[], algorithm='gradient', nghost=0)
```

With the pyTree interface, the user-defined node .Solver#Param/proc is updated with the attributed processor number.
If useCom=0, only the grid number of points is taken into account.
If useCom='all', matching and overlap communications are taken into account.
If useCom='match', only match connectivity are taken into account.
if useCom='overlap', only overlap connectivity are taken into account.
if useCom='bbox', overlap between zone bbox is taken into account.

When using distributed trees, prescribed must be a dictionary containing the zones names as key, and the prescribed proc as value. weight is also a dictionary where the keys are the zone names and the weight as the value. It is not mandatory to assign a weight to all the zones of the pyTree. Default value is assumed 1, only different weight values can be assigned to zones. t can be either a skeleton or a loaded skeleton pyTree for useCom=0 or useCom='match', but must be a loaded skeleton tree only for the other settings:

```
t, stats = D2.distribute(t, NProc, prescribed=, perfo=[], weight=, useCom='all', algorithm='gradient')
```

(See: distribute.py) (See: distributePT.py)

1.3 Various operations

D2.addProcNode: add a "proc" node to all zones of A with given value:

```
B = D2.addProcNode(A, 12)
```

(See: addProcNodePT.py)

D2.getProc: get the proc node of a zone or a list of zones:

```
proc = D2.getProc(a) .or. [proc1,proc2,...] = D2.getProc(A)
```

(See: getProcPT.py)

D2.getProcList: return procList where procList[proc] is a list of zone names attributed to the proc processor:

```
procList = D2.getProcList(A, NProc=None)
```

(See: getProcListPT.py)

D2.copyDistribution: copy the distribution of B to A matching zones by their name:

A = D2.copyDistribution(A, B)

(See: copyDistributionPT.py)

D2.redispatch: redispatch a tree where a new distribution is defined in the node 'proc':

B = D2.redispatch(A)

(See: redispatchPT.py)

1.4 Example files

Example file: distribute.py

```
# - distribute (array) -
import Generator as G
import Distributor2 as D2
import numpy

# Distribution sans communication entre blocs
N = 11
arrays = []
for i in xrange(N):
    a = G.cart( (0,0,0), (1,1,1), (10+i, 10, 10) )
    arrays.append(a)
out = D2.distribute(arrays, NProc=5); print out

# Distribution avec des perfos differentes pour chaque proc
out = D2.distribute(arrays, NProc=3, perfo=[(1,0,0), (1.2,0,0), (0.2,0,0)]); print out

# Distribution avec forcage du bloc 0 sur le proc 1, du bloc 2 sur le proc 3
# -1 signifie que le bloc est a equilibrer
prescribed = [-1 for x in xrange(N)]
prescribed[0] = 1; prescribed[2] = 3
out = D2.distribute(arrays, NProc=5, prescribed=prescribed); print out

# Distribution avec communications entre blocs, perfos identique pour tous
# les procs
volCom = numpy.zeros( (N, N), numpy.int32 )
volCom[0,1] = 100; # Le bloc 0 echange 100 pts avec le bloc 1
out = D2.distribute(arrays, NProc=5, com=volCom, perfo=(1,0.,0.1)); print out

# Distribution avec des solveurs differents pour les blocs (le solveur est 2
# fois plus couteux pour les bloc 2 et 4)
out = D2.distribute(arrays, weight=[1,2,1,2,1,1,1,1,1,1,1], NProc=3); print out
```

Example file: distributePT.py

```
# - distribute (pyTree) -
import Generator.PyTree as G
import Distributor2.PyTree as D2
import Converter.PyTree as C
import Connector.PyTree as X

N = 11
t = C.newPyTree(['Base'])
pos = 0
for i in xrange(N):
```

```
    a = G.cart((pos,0,0), (1,1,1), (10+i, 10, 10))
    pos += 10 + i - 1
    t[2][1][2].append(a)
t = X.connectMatch(t)

# Distribute on 3 processors
t, stats = D2.distribute(t, 3)
C.convertPyTree2File(t, 'out.cgns')
```

Example file: addProcNodePT.py

```
# - addProcNode (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Distributor2.PyTree as D2

a = G.cart((0,0,0), (1,1,1), (10,10,10))
a = D2.addProcNode(a, 12)
C.convertPyTree2File(a, 'out.cgns')
```

Example file: getProcPT.py

```
# - getProc (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Distributor2.PyTree as D2

a = G.cart((0,0,0), (1,1,1), (10,10,10))
a = D2.addProcNode(a, 12)
proc = D2.getProc(a); print proc
```

Example file: getProcListPT.py

```
# - getProcList (pyTree) -
import Generator.PyTree as G
import Distributor2.PyTree as D2
import Converter.PyTree as C
import Connector.PyTree as X

N = 11
t = C.newPyTree(['Base'])
pos = 0
for i in xrange(N):
    a = G.cart((pos,0,0), (1,1,1), (10+i, 10, 10))
    pos += 10 + i - 1
    t[2][1][2].append(a)

t = X.connectMatch(t)
t, stats = D2.distribute(t, 3)

procList = D2.getProcList(t)
print procList
```

Example file: copyDistributionPT.py

```
# - copyDistribution (pyTree) -
import Converter.PyTree as C
import Distributor2.PyTree as D2
import Converter.Internal as Internal
import Generator.PyTree as G

# Case
```

```
N = 11
t = C.newPyTree(['Base'])
pos = 0
for i in xrange(N):
    a = G.cart((pos,0,0), (1,1,1), (10+i, 10, 10))
    a[0] = 'cart%d'%i
    pos += 10 + i - 1
    D2._addProcNode(a, i)
    t[2][1][2].append(a)

t2 = C.newPyTree(['Base'])
for i in xrange(N):
    a = G.cart((pos,0,0), (1,1,1), (10+i, 10, 10))
    a[0] = 'cart%d'%i
    pos += 10 + i - 1
    t2[2][1][2].append(a)
t2 = D2.copyDistribution(t2, t)
C.convertPyTree2File(t2, 'out.cgns')
```

Example file: redispatchPT.py

```
# - redispatch (pyTree) -
import Converter.PyTree as C
import Distributor2.PyTree as D2
import Distributor2.Mpi as D2mpi
import Converter.Mpi as Cmpi
import Transform.PyTree as T
import Connector.PyTree as X
import Converter.Internal as Internal
import Generator.PyTree as G

# Case
N = 11
t = C.newPyTree(['Base'])
pos = 0
for i in xrange(N):
    a = G.cart((pos,0,0), (1,1,1), (10+i, 10, 10))
    pos += 10 + i - 1
    t[2][1][2].append(a)
t = X.connectMatch(t)
if Cmpi.rank == 0: C.convertPyTree2File(t, 'in.cgns')
Cmpi.barrier()

# lecture du squelette
a = Cmpi.convertFile2SkeletonTree('in.cgns')

# equilibrage 1
(a, dic) = D2.distribute(a, NProc=Cmpi.size, algorithm='fast', useCom=0)

# load des zones locales dans le squelette
a = Cmpi.readZones(a, 'in.cgns', rank=Cmpi.rank)

# equilibrage 2 (a partir d'un squelette charge)
(a, dic) = D2.distribute(a, NProc=Cmpi.size, algorithm='gradient1',
                         useCom='match')

a = D2mpi.redispatch(a)

# force toutes les zones sur 0
zones = Internal.getNodesFromType(a, 'Zone_t')
for z in zones:
```

```
    nodes = Internal.getNodesFromName(z, 'proc')
    Internal.setValue(nodes[0], 0)

a = D2mpi.redispatch(a)

# Reconstruit l'arbre complet a l'ecriture
Cmpi.convertPyTree2File(a, 'out.cgns')
```