SINTEF A121 - Unrestricted

REPORT

GRIDDLER and VISTA with CGNS interface

Karstein Sørli and Runar Holdahl

SINTEF ICT
Applied Mathematics

May 2006

SINTEF

SINTEF ICT

Address: NO-7465 Trondheim, NORWAY
Location: Sem Sælandsv 5 NO-7031 Trondheim
Telephone: +47 73 59 30 48
Fax: +47 73 59 43 02

Enterprise No.: NO 948 007 029 MVA

SINTEF REPORT

TITLE

GRIDDLER and VISTA with CGNS interface

AUTHOR(S)

Karstein Sørli and Runar Holdahl

CLIENT(S)

UNINETT/SIGMA AS

REPORT NO.	CLASSIFICATION	CLIENTS REF.	
SINTEF A121	Unrestricted		
CLASS. THIS PAGE	ISBN	PROJECT NO.	NO. OF PAGES/APPENDICES
Unrestricted	82-14-02846-9	90A272	27
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.)	CHECKED BY (NAME, SIGN.)
SINTEF A121.pdf		Bjørnar Pettersen	Karl J. Eidsvik
FILE CODE	DATE	APPROVED BY (NAME, POSITION, SIGN.)	
	2006-05-12	Svein Nordenson, Research Director	

ABSTRACT

(STF90 A06009)

This developer's and user's note has been written to help developers and users of GRIDDLER and VISTA to maintain and use their recent CGNS (CFD General Notation System) interface. The present note contains a description of the first version of GRIDDLER and VISTA with this option. This version has implemented CGNS formatted GRIDDLER output and VISTA input of an arbitrary multi-block structured 3D grid with block connectivity and boundary conditions. In later versions, other options of the CGNS system will be included as well.

The main reason for making a CGNS output format option available to GRIDDLER was the need for making a blocked grid from a GRIDDLER model. Even though GRIDDLER is a multi-block grid generator, implemented with the use of object-oriented programming; until now, the final result has been a global grid that collects the points from all the blocks. A trigger for making this blocked grid option to GRIDDLER was the requirement of the parallel VISTA CFD code - based on the algorithm of domain decomposition - to be able to contain just the grid of its own domain (possibly with overlaps to its neighbouring domains or blocks) and not the whole grid as was required in the first place.

Some examples are given in the appendices.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	Mathematics	Matematikk
GROUP 2	Software, Documentation	Programvare, Dokumentasjon
SELECTED BY AUTHOR	CFD, Grid generation, Data format	CFD, Gridgenerering, Dataformat

GRIDDLER and VISTA with CGNS interface

Developer's and User's Note based on
GRIDDLER Version 1.12, VISTA Version 1.0
and CGNS Version 2.4

Karstein Sørli
Runar Holdahl

SINTEF ICT Applied Mathematics

May 12, 2006

Contents

1 INTRODUCTION

This Developer's and User's Note has been written to help developers and users of GRIDDLER to maintain and use its recent CGNS (CFD General Notation System) output option. The present note contains a description of the first version (1.13) of GRIDDLER with this option. This version has implemented CGNS formatted output of an arbitrary multi-block structured 3D grid with block connectivities and boundary conditions (BCs). In later versions of GRIDDLER other options of the CGNS system will be included.

1.1 What is CGNS?

CGNS originated in 1994 as a joint effort between Boeing and NASA, and has since grown to include many other contributing organizations worldwide. It is an effort to *standardize* CFD input and output, including grid (both structured and unstructured), flow solution, connectivity, BCs, and auxiliary information. CGNS is also easily extensible, and allows for user-inserted-commenting. It employs ADF (Advanced Data Format), a system which creates binary files that are portable across computer platforms. CGNS also includes a second layer of software known as the mid-level library, or API (Application Programming Interface), which eases the implementation of CGNS into existing CFD codes.

In 1999, control of CGNS was completely transferred to a public forum known as the CGNS Steering Committee. This Steering Committee is made up of international representatives from government and private industry. All CGNS software is completely free and open to anyone (open source). The CGNS standard is also the object of an ISO standardization effort for fluid dynamics data [8], for release some time in the early to mid-2000's.

1.2 Why CGNS?

The following is stated by the CGNS organization. *CGNS will eventually eliminate most of the translator programs now necessary when working between machines and between CFD codes. Also, it eventually may allow for the results from one code to be easily restarted using another code. It will hopefully therefore save a lot of time and money. In particular, it is hoped that future grid-generation software will generate grids with all connectivity and BC information included as part of a CGNS database, saving time and avoiding potential costly errors in setting up this information after-the-fact.*

Our reason for making a CGNS output format option available to GRIDDLER was the need for making *a blocked grid* from a GRIDDLER model. Even though GRIDDLER is *a multi-block grid generator*, implemented with the use of object-oriented programming; until now, the final result has been *a global grid* that collects the points from all the blocks. A trigger for making this blocked grid option to GRIDDLER was the requirement of *a parallel CFD code based on the algorithm of domain decomposition* to be able to contain just the grid of its own domain (possibly with overlaps to its neighboring domains or blocks) and not the whole grid as was required in the beginning.

2 Preprocessing for Block Connectivities

Two neighboring blocks have 24 different relative positions. These positions are described by side (6 alternatives) and rotation (4 alternatives). Below is listed a code extract of the new coding in GRIDDLER comprising the determination of these alternatives.

```
for (n=0; n<NumBlocks; n++) {
   // Block on WEST side ?
   surface = b->ob->block3D[n]->WestSurface;
   nd = (int)((b->ob->block3D[n]->WestDir)/1.999999);
   for (nb=0; nb<NumBlocks; nb++) {
      if (nb == n) continue; // cannot be neighbor to itself
      surfacenb = b->ob->block3D[nb]->WestSurface;
      if (surfacenb == surface) {
         b->ob->block3D[n]->NeBlockWest = b->ob->block3D[nb];
         b->ob->block3D[n]->NeBlockWestSurface = 'W';
         nbo = nd - (int)((b->ob->block3D[nb]->WestDir)/1.999999);
         if (nbo < 0) nbo += 4; if (nbo > 3) nbo -= 4;
         b->ob->block3D[n]->NeBlockWestOrient = nbo; // 0,1,2,3
         break;
      }
      surfacenb = b->ob->block3D[nb]->EastSurface;
      if (surfacenb == surface) {
         b->ob->block3D[n]->NeBlockWest = b->ob->block3D[nb];
         b->ob->block3D[n]->NeBlockWestSurface = 'E';
         nbo = nd - (int)((b->ob->block3D[nb]->EastDir)/1.999999);
         if (nbo < 0) nbo += 4; if (nbo > 3) nbo -= 4;
         b->ob->block3D[n]->NeBlockWestOrient = nbo; // 0,1,2,3
         break;
      }
      ...similar for EAST,SOUTH,NORTH,BOTTOM surfaces and finally:
      surfacenb = b->ob->block3D[nb]->TopSurface;
      if (surfacenb == surface) {
         b->ob->block3D[n]->NeBlockWest = b->ob->block3D[nb];
         b->ob->block3D[n]->NeBlockWestSurface = 'T';
         nbo = nd - (int)((b->ob->block3D[nb]->TopDir)/1.999999);
         if (nbo < 0) nbo += 4; if (nbo > 3) nbo -= 4;
         b->ob->block3D[n]->NeBlockWestOrient = nbo; // 0,1,2,3
         break;
      }
   }
   // Similar tests for blocks on EAST,SOUTH,NORTH,BOTTOM,TOP sides
}
```

3 Multi-block 3D Structured Grid

This section gives a description of the writing of a multi-block structured grid.

```
// open CGNS file for writing grid points
cg_open(cgfile,MODE_WRITE,&index_file);
// create base
icelldim=3; iphysdim=3;
cg_base_write(index_file,cgbase,icelldim,iphysdim,&index_base);
for (n=0; n<NumBlocks; n++) { // loop over zones
   imax = b->ob->block3D[n]->imax; ni = imax + 1;
   jmax = b->ob->block3D[n]->jmax; nj = jmax + 1;
   kmax = b->ob->block3D[n]->kmax; nk = kmax + 1;
   xx = new double[ni*nj*nk];
   yy = new double[ni*nj*nk];
   zz = new double[ni*nj*nk];
   for (k=0; k<=kmax; ++k) {
      for (j=0; j<=jmax; ++j) {
         for (i=0; i<=imax; ++i) {
            ijk = i + ni*j + ni*nj*k;
            xx[ijk] = b->ob->block3D[n]->x[i][j][k];
            yy[ijk] = b->ob->block3D[n]->y[i][j][k];
            zz[ijk] = b->ob->block3D[n]->z[i][j][k];
         }
      }
   }
   // vertex and cell sizes
   isize[0][0] = ni; isize[1][0] = ni-1;
   isize[0][1] = nj; isize[1][1] = nj-1;
   isize[0][2] = nk; isize[1][2] = nk-1;
   // boundary vertex size (always zero for structured grids)
   isize[2][0] = 0; isize[2][1] = 0; isize[2][2] = 0;
   sprintf(zonename[n],"Zone %d", n+1); /* defines zonename */
   // create zone and write grid coordinates
   cg_zone_write(index_file,index_base,zonename[n],*isize,Structured,
                 &index_zone);
   cg_coord_write(index_file,index_base,index_zone,RealDouble,
                 "CoordinateX",xx,&index_coord);
   cg_coord_write(index_file,index_base,index_zone,RealDouble,
                 "CoordinateY",yy,&index_coord);
   cg_coord_write(index_file,index_base,index_zone,RealDouble,
                 "CoordinateZ",zz,&index_coord);
   delete xx; delete yy; delete zz;
} // end looping zones
cg_close(index_file); // closes CGNS file
```

4 Connectivity Data

This section gives a description of adding multi-block structured grid connectivity data to an existing multi-block grid file. This version of the GRIDDLER interface to CGNS is limited to 1-to-1 nonoverlapping blocks or zones. Later versions are anticipated to include GRIDDLER options for block overlaps. The latter issue is crucial for domain decomposition algorithms in parallel CFD with respect to convergence of the algorithm. In the meantime, the user of GRIDDLER must extract this data from the non-overlapping data that comes out from the present version.

```
// open CGNS file for writing block connectivity data
cg_open(cgfile,MODE_MODIFY,&index_file);
// we know there is only one base
index_base=1;
// get number of zones
cg_nzones(index_file,index_base,&NumZones);
if (NumZones != NumBlocks) {
   printf("Error!  Expects %d zones.  %d read.",NumBlocks,NumZones);
   exit(0);
}
// loop over zones to get zone names and sizes
for (n=0; n<NumZones; ++n) {
   index_zone=n+1;
   cg_zone_read(index_file,index_base,index_zone,zonename[n],isize[0]);
   ilo[n]=1; ihi[n]=isize[0][0];
   jlo[n]=1; jhi[n]=isize[0][1];
   klo[n]=1; khi[n]=isize[0][2];
}
nxt[0]=1; nxt[1]=2; nxt[2]=3; nxt[3]=0;
for (n=0; n<NumZones; ++n) { // loop over zones again
   index_zone=n+1;
   // there should be no existing connectivity info:
   cg_nconns(index_file,index_base,index_zone,&nconns);
   if (nconns != 0) {
      printf("Error!  Expects no interfaces yet.%d read.",nconns);
      exit(0);
   }
   cg_n1to1(index_file,index_base,index_zone,&n1to1);
   if (n1to1 != 0) {
      printf("Error!  Expects no interfaces yet.%d read.",n1to1);
      exit(0);
   }

   // testing all 6 sides in the following
```

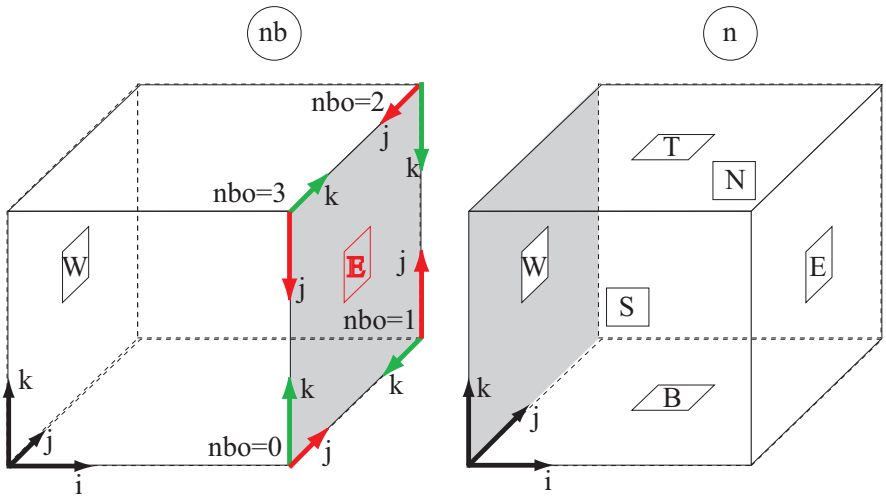

Figure 1: West-to-east (default) connection of neighboring blocks.

```
// neighbor on WEST side ?
if (b->ob->block3D[n]->NeBlockWest != NULL) {
   it[0]=2; it[1]=3; it[2]=-2; it[3]=-3; // for W and E
   nb = b->ob->block3D[n]->NeBlockWest->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ilo[n]; ipnts[1][0]=ilo[n];
   ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
   ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ihi[nb]; ipntsdonor[1][0]=ihi[nb];
   ipntsdonor[0][1]=jlo[nb]; ipntsdonor[1][1]=jhi[nb];
   ipntsdonor[0][2]=klo[nb]; ipntsdonor[1][2]=khi[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockWestOrient;
   switch (b->ob->block3D[n]->NeBlockWestSurface) {
      case 'W': t1 = -1; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'E': t1 = 1; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'S': t2 = -1; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'N': t2 = 1; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'B': t3 = -1; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'T': t3 = 1; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  W)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-W",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

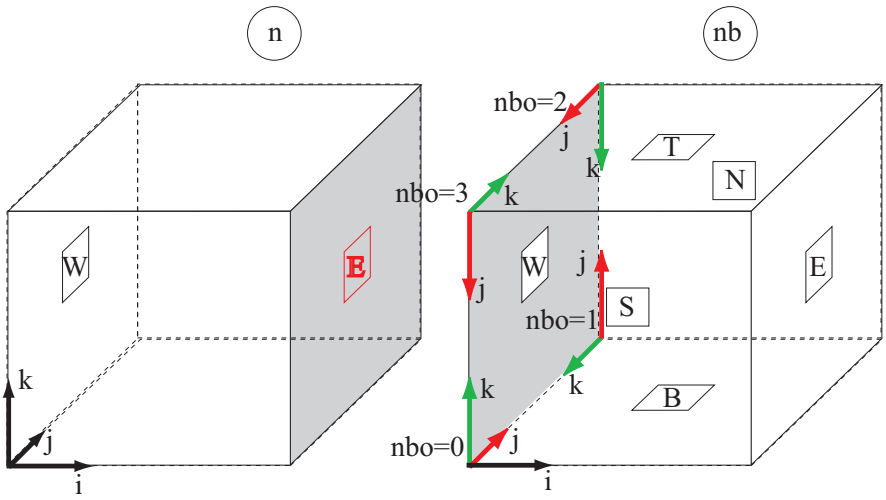

Figure 2: East-to-west (default) connection of neighboring blocks.

```
// neighbor on EAST side ?
if (b->ob->block3D[n]->NeBlockEast != NULL) {
   it[0]=2; it[1]=3; it[2]=-2; it[3]=-3; // for W and E
   nb = b->ob->block3D[n]->NeBlockEast->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ihi[n]; ipnts[1][0]=ihi[n];
   ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
   ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ilo[nb]; ipntsdonor[1][0]=ilo[nb];
   ipntsdonor[0][1]=jlo[nb]; ipntsdonor[1][1]=jhi[nb];
   ipntsdonor[0][2]=klo[nb]; ipntsdonor[1][2]=khi[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockEastOrient;
   switch (b->ob->block3D[n]->NeBlockEastSurface) {
      case 'E': t1 = -1; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'W': t1 = 1; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'N': t2 = -1; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'S': t2 = 1; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'T': t3 = -1; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'B': t3 = 1; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  E)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-E",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

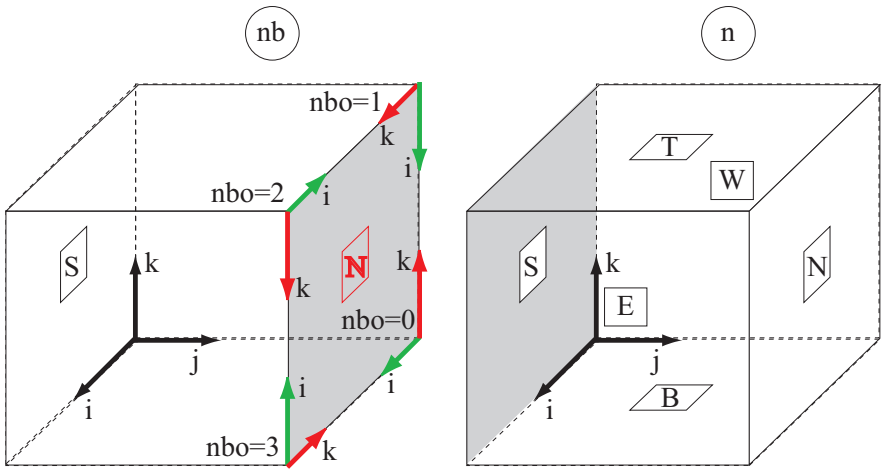

Figure 3: South-to-north (default) connection of neighboring blocks.

```
// neighbor on SOUTH side ?
if (b->ob->block3D[n]->NeBlockSouth != NULL) {
   it[0]=1; it[1]=3; it[2]=-1; it[3]=-3; // for S and N
   nb = b->ob->block3D[n]->NeBlockSouth->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
   ipnts[0][1]=jlo[n]; ipnts[1][1]=jlo[n];
   ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ilo[nb]; ipntsdonor[1][0]=ihi[nb];
   ipntsdonor[0][1]=jhi[nb]; ipntsdonor[1][1]=jhi[nb];
   ipntsdonor[0][2]=klo[nb]; ipntsdonor[1][2]=khi[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockSouthOrient;
   switch (b->ob->block3D[n]->NeBlockSouthSurface) {
      case 'W': t1 = -2; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'E': t1 = 2; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'S': t2 = -2; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'N': t2 = 2; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'B': t3 = -2; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'T': t3 = 2; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  S)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-S",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

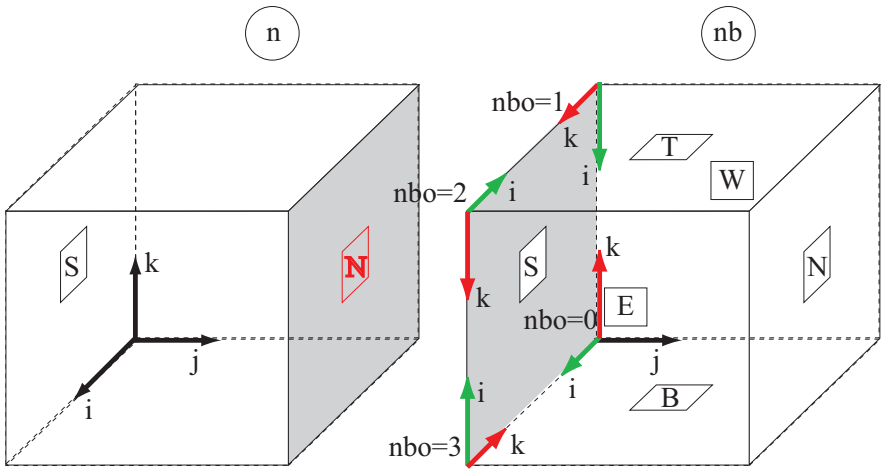

Figure 4: North-to-south (default) connection of neighboring blocks.

```
// neighbor on NORTH side ?
if (b->ob->block3D[n]->NeBlockNorth != NULL) {
   it[0]=1; it[1]=3; it[2]=-1; it[3]=-3; // for S and N
   nb = b->ob->block3D[n]->NeBlockNorth->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
   ipnts[0][1]=jhi[n]; ipnts[1][1]=jhi[n];
   ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ilo[nb]; ipntsdonor[1][0]=ihi[nb];
   ipntsdonor[0][1]=jlo[nb]; ipntsdonor[1][1]=jlo[nb];
   ipntsdonor[0][2]=klo[nb]; ipntsdonor[1][2]=khi[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockNorthOrient;
   switch (b->ob->block3D[n]->NeBlockNorthSurface) {
      case 'E': t1 = -2; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'W': t1 = 2; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'N': t2 = -2; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'S': t2 = 2; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'T': t3 = -2; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'B': t3 = 2; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  N)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-N",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

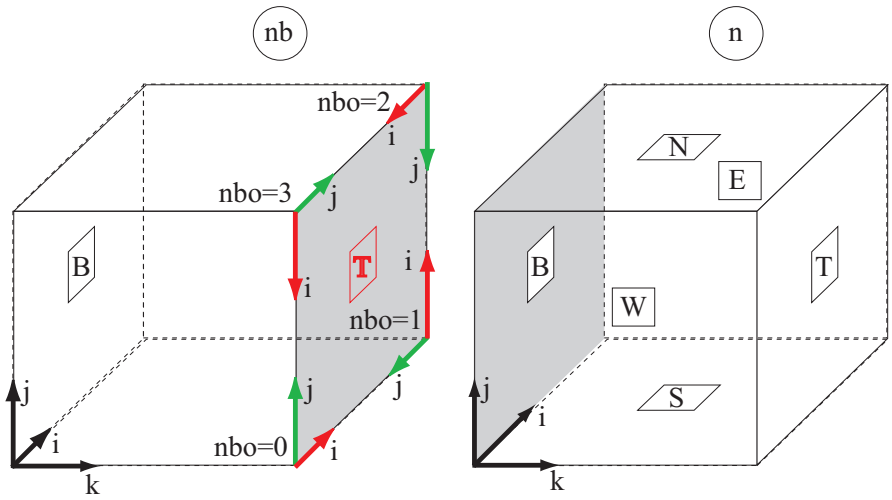

Figure 5: Bottom-to-top (default) connection of neighboring blocks.

```
// neighbor on BOTTOM side ?
if (b->ob->block3D[n]->NeBlockBottom != NULL) {
   it[0]=1; it[1]=2; it[2]=-1; it[3]=-2; // for B and T
   nb = b->ob->block3D[n]->NeBlockBottom->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
   ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
   ipnts[0][2]=klo[n]; ipnts[1][2]=klo[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ilo[nb]; ipntsdonor[1][0]=ihi[nb];
   ipntsdonor[0][1]=jlo[nb]; ipntsdonor[1][1]=jhi[nb];
   ipntsdonor[0][2]=khi[nb]; ipntsdonor[1][2]=khi[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockBottomOrient;
   switch (b->ob->block3D[n]->NeBlockBottomSurface) {
      case 'W': t1 = -3; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'E': t1 = 3; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'S': t2 = -3; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'N': t2 = 3; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'B': t3 = -3; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'T': t3 = 3; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  B)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-B",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

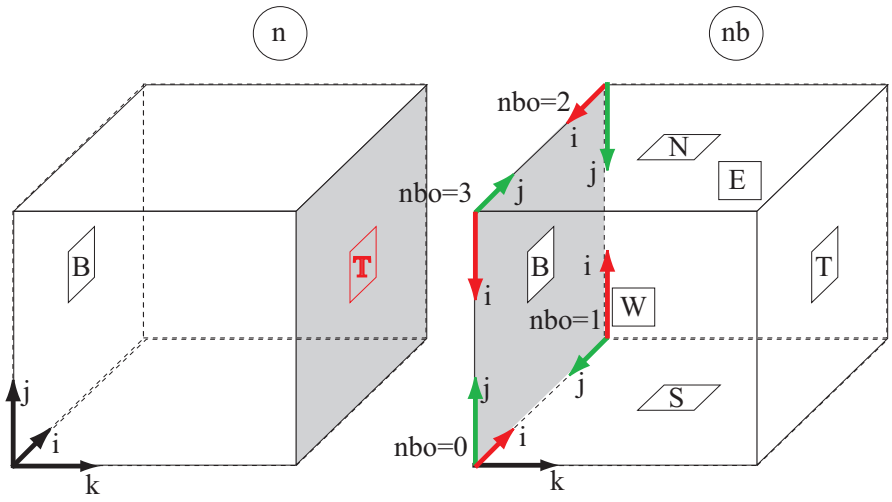

Figure 6: Top-to-bottom (default) connection of neighboring blocks.

```
// neighbor on TOP side ?
if (b->ob->block3D[n]->NeBlockTop != NULL) {
   it[0]=1; it[1]=2; it[2]=-1; it[3]=-2; // for B and T
   nb = b->ob->block3D[n]->NeBlockTop->Number;
   strcpy(donorname,zonename[nb]);
   // lower and upper points of receiver range
   ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
   ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
   ipnts[0][2]=khi[n]; ipnts[1][2]=khi[n];
   // lower and upper points of donor range
   ipntsdonor[0][0]=ilo[nb]; ipntsdonor[1][0]=ihi[nb];
   ipntsdonor[0][1]=jlo[nb]; ipntsdonor[1][1]=jhi[nb];
   ipntsdonor[0][2]=klo[nb]; ipntsdonor[1][2]=klo[nb];
   // set up Transform
   nbo = b->ob->block3D[n]->NeBlockTopOrient;
   switch (b->ob->block3D[n]->NeBlockTopSurface) {
      case 'E': t1 = -3; t2 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'W': t1 = 3; t3 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'N': t2 = -3; t3 = it[nxt[nbo]]; t1 = it[nbo]; break;
      case 'S': t2 = 3; t1 = it[nxt[nbo]]; t3 = it[nbo]; break;
      case 'T': t3 = -3; t1 = it[nxt[nbo]]; t2 = it[nbo]; break;
      case 'B': t3 = 3; t2 = it[nxt[nbo]]; t1 = it[nbo]; break;
   }
   itranfrm[0]=t1; itranfrm[1]=t2; itranfrm[2]=t3;
   // write 1-to-1 info (username is side of block, i.e.  T)
   cg_1to1_write(index_file,index_base,index_zone,"Interface-T",
         donorname,ipnts[0],ipntsdonor[0],itranfrm,&index_conn);
}
```

```
} // end looping blocks
cg_close(index_file); // closes CGNS file
```

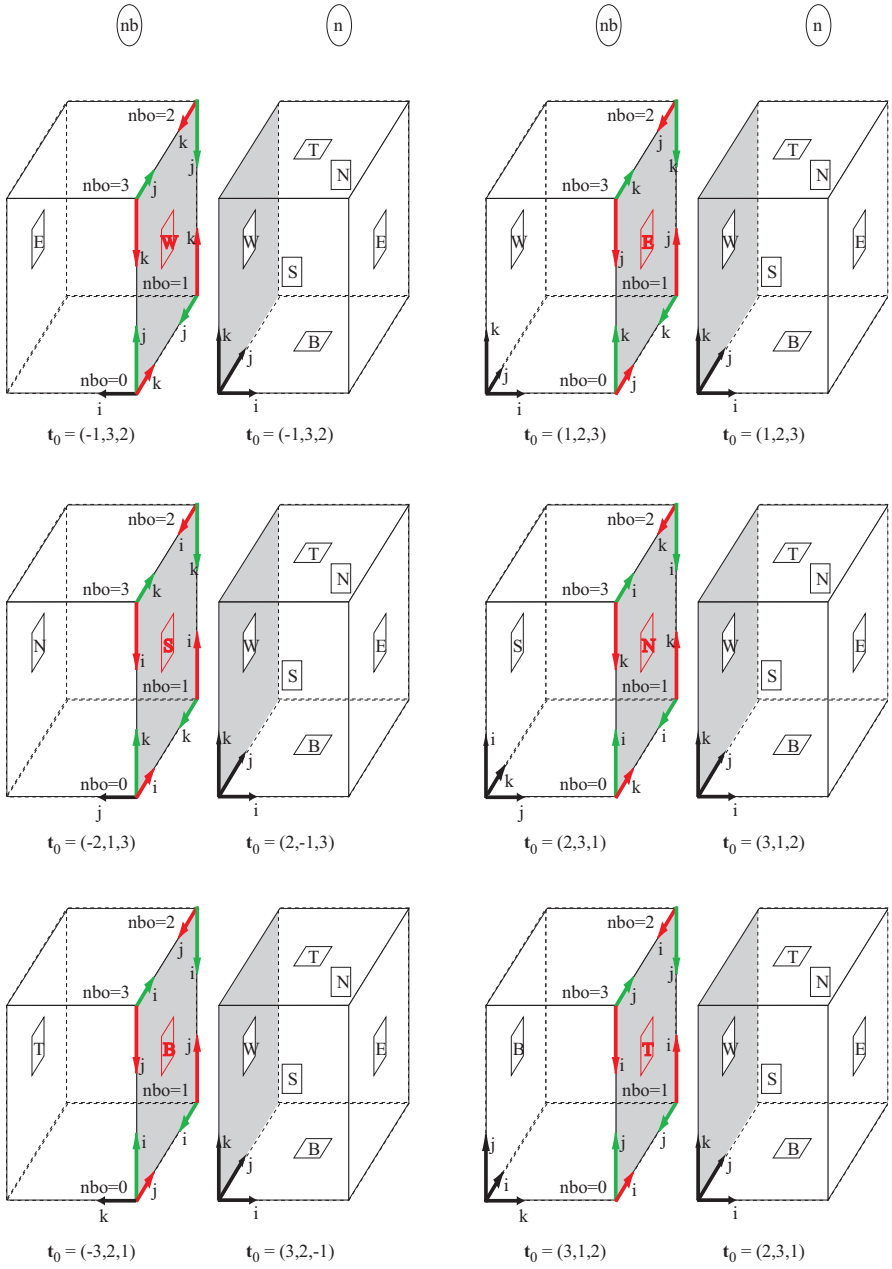

Figure 7: Connections to the West side of block.

5 Boundary Conditions

Note that this version has implemented 3 different cases of boundary conditions. In using GRIDDLER these are imposed by the following commands:

INLET: sbc surfno1 surfno2 name 3 **1** 0 0 w e s n/

OUTLET: sbc surfno1 surfno2 name 3 **2** 0 0 w e s n/

WALL: sbc surfno1 surfno2 name 3 **3** 0 0 w e s n/

where codes are given in boldface, i.e., 1 means inlet, 2 means outlet and 3 means wall BC conditions. Note that the two zeroes after the BC code are dummy codes. The final w, e, s, and n are the usual boundary surface bounding curves, where the values 1 and 0 indicates that the points on the bounding curve are to be included and not included, respectively. Other options are available for the latter issues.

```
// open CGNS file for writing BC data
cg_open(cgfile,MODE_MODIFY,&index_file);
// we know there is only one base
index_base=1;
// get number of zones
cg_nzones(index_file,index_base,&NumZones);
if (NumZones != NumBlocks) {
   printf("Error!  Expects %d zones.  %d read.", NumBlocks, NumZones);
   exit(0);
}
// loop over zones to get zone sizes and names
for (n=0; n<NumZones; ++n) {
   index_zone=n+1;
   cg_zone_read(index_file,index_base,index_zone,zonename[n],isize[0]);
   ilo[n]=1; ihi[n]=isize[0][0];
   jlo[n]=1; jhi[n]=isize[0][1];
   klo[n]=1; khi[n]=isize[0][2];
}
// loop over zones again
for (n=0; n<NumZones; ++n) {
   index_zone=n+1;
   if (b->ob->block3D[n]->WestSurface->BoundarySurface) {
      // lower and upper points of range
      ipnts[0][0]=ilo[n]; ipnts[1][0]=ilo[n];
      ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
      ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
      if (b->ob->block3D[n]->WestSurface->bc[0].value1 == 1)
         cg_boco_write(index_file,index_base,index_zone,"BC-W",BCInflow,
                       PointRange,2,ipnts[0],&index_bc);
```

```
    else if (b->ob->block3D[n]->WestSurface->bc[0].value1 == 2)
      cg_boco_write(index_file,index_base,index_zone,"BC-W",BCOutflow,
                    PointRange,2,ipnts[0],&index_bc);
    else if (b->ob->block3D[n]->WestSurface->bc[0].value1 == 3)
      cg_boco_write(index_file,index_base,index_zone,"BC-W",BCWall,
                    PointRange,2,ipnts[0],&index_bc);
  }
  if (b->ob->block3D[n]->EastSurface->BoundarySurface) {
    // lower and upper points of range
    ipnts[0][0]=ihi[n]; ipnts[1][0]=ihi[n];
    ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
    ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
    if (b->ob->block3D[n]->EastSurface->bc[0].value1 == 1)
      cg_boco_write(index_file,index_base,index_zone,"BC-E",BCInflow,
                    PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->EastSurface->bc[0].value1 == 2)
      cg_boco_write(index_file,index_base,index_zone,"BC-E",BCOutflow,
                    PointRange,2,ipnts[0],&index_bc);
    else if (b->ob->block3D[n]->EastSurface->bc[0].value1 == 3)
      cg_boco_write(index_file,index_base,index_zone,"BC-E",BCWall,
                    PointRange,2,ipnts[0],&index_bc);
  }
  if (b->ob->block3D[n]->SouthSurface->BoundarySurface) {
    // lower and upper points of range
    ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
    ipnts[0][1]=jlo[n]; ipnts[1][1]=jlo[n];
    ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
    if (b->ob->block3D[n]->SouthSurface->bc[0].value1 == 1)
      cg_boco_write(index_file,index_base,index_zone,"BC-S",BCInflow,
                    PointRange,2,ipnts[0],&index_bc);
    else if (b->ob->block3D[n]->SouthSurface->bc[0].value1 == 2)
      cg_boco_write(index_file,index_base,index_zone,"BC-S",BCOutflow,
                    PointRange,2,ipnts[0],&index_bc);
    else if (b->ob->block3D[n]->SouthSurface->bc[0].value1 == 3)
      cg_boco_write(index_file,index_base,index_zone,"BC-S",BCWall,
                    PointRange,2,ipnts[0],&index_bc);
  }
  if (b->ob->block3D[n]->NorthSurface->BoundarySurface) {
    // lower and upper points of range
    ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
    ipnts[0][1]=jhi[n]; ipnts[1][1]=jhi[n];
    ipnts[0][2]=klo[n]; ipnts[1][2]=khi[n];
    if (b->ob->block3D[n]->NorthSurface->bc[0].value1 == 1)
      cg_boco_write(index_file,index_base,index_zone,"BC-N",BCInflow,
                    PointRange,2,ipnts[0],&index_bc);
```

```
      else if (b->ob->block3D[n]->NorthSurface->bc[0].value1 == 2)
         cg_boco_write(index_file,index_base,index_zone,"BC-N",BCOutflow,
                       PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->NorthSurface->bc[0].value1 == 3)
         cg_boco_write(index_file,index_base,index_zone,"BC-N",BCWall,
                       PointRange,2,ipnts[0],&index_bc);
   }
   if (b->ob->block3D[n]->BottomSurface->BoundarySurface) {
      // lower and upper points of range
      ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
      ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
      ipnts[0][2]=klo[n]; ipnts[1][2]=klo[n];
      if (b->ob->block3D[n]->BottomSurface->bc[0].value1 == 1)
         cg_boco_write(index_file,index_base,index_zone,"BC-B",BCInflow,
                       PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->BottomSurface->bc[0].value1 == 2)
         cg_boco_write(index_file,index_base,index_zone,"BC-B",BCOutflow,
                       PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->BottomSurface->bc[0].value1 == 3)
         cg_boco_write(index_file,index_base,index_zone,"BC-B",BCWall,
                       PointRange,2,ipnts[0],&index_bc);
   }
   if (b->ob->block3D[n]->TopSurface->BoundarySurface) {
      // lower and upper points of range
      ipnts[0][0]=ilo[n]; ipnts[1][0]=ihi[n];
      ipnts[0][1]=jlo[n]; ipnts[1][1]=jhi[n];
      ipnts[0][2]=khi[n]; ipnts[1][2]=khi[n];
      if (b->ob->block3D[n]->TopSurface->bc[0].value1 == 1)
         cg_boco_write(index_file,index_base,index_zone,"BC-T",BCInflow,
                       PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->TopSurface->bc[0].value1 == 2)
         cg_boco_write(index_file,index_base,index_zone,"BC-T",BCOutflow,
                       PointRange,2,ipnts[0],&index_bc);
      else if (b->ob->block3D[n]->TopSurface->bc[0].value1 == 3)
         cg_boco_write(index_file,index_base,index_zone,"BC-T",BCWall,
                       PointRange,2,ipnts[0],&index_bc);
   }
} // end looping blocks

cg_close(index_file); // closes CGNS file
```

6 Multi-block Structured Grids in Diffpack

Recall that the GRIDDLER software possess basic geometrical elements that enable the modeling of complex 3D domains by composing multiple blocks. However, at the final stage of the gridding a global mesh is created, with a single global numbering of nodes and elements.

One of the motivations for extending GRIDDLER to create multi-block structured grids was the current limitations of the Diffpack library [4] in handling large grid models for parallel simulations. In particular, these problems have become evident when running the inhouse CFD code Vista for 3D cases with several million elements. The bottleneck in the computations has been related to be the way the global grid is partitioned into subgrids suitable for parallel processing. In Diffpack the graph and mesh partitioning software METIS [5] is used for this purpose. METIS can partition general unstructured finite element meshes into any number of submeshes with almost the same number of elements. The partitioning algorithm is also computationally efficient. However, the problem is that Diffpack applies METIS sequentially on all processors. This leads to an excessive use of memory during the partitioning phase since the memory requirements for each processor scale proportional to the size of the global grid. These shortcommings have limited the number of processors that can be efficiently used for the Vista code. The largest simulations that have been run up to now has included up to 128 processors on the SGI Origin supercomputer at NTNU. This has required about 256 Gb of RAM, of which most is only needed during the partitioning of the mesh.

We believe that the use of multi-block structured grids is an efficient way of overcomming the current problems in the Vista code. The multi-block output from GRIDDLER contains information about each grid block and the block connectivity. This is all the information required to set up the parallel communication in Diffpack. As a first approach we associate a single block to each processor. Each processor can then read the grid block and connectivity data from file before setting up the parallel communication. In this way the memory requirements will scale like the size of the local grid block. Another advantage of this approach is that we can control the shape of the subgrids through the gridgeneration, which in turn can improve the performance of the domain decomposition methods used to solve the linear systems in the code. METIS has previously given unfavorable partitionings in some cases, especially in the vicinity of boundary layers where the mesh has a strong grading.

6.1 Classes for block-structured grids

First we have to implement classes to store multi-block meshes in Diffpack. Most of the finite element functionality in Diffpack assumes that the mesh is stored in the class GridFE, which handles general unstructured finite element meshes. We have therefore chosen to implement the class GridFE2 for storing structured grids as a subclass of GridFE, although it should have been the other way around from an object-oriented point of view. The header file for the class GridFE2 is listed below. The extra functionality is related to

the vectors nodedim and elmdim which gives the number of nodes and elements in each spatial direction. It is assumed that a natural numbering of the nodes is used.

```
#ifndef GridFE2_h_IS_INCLUDED
#define GridFE2_h_IS_INCLUDED

#include <GridFE.h>
//----------------------- GridFE2 --------------------------------------
/*GridFE2:*/
class GridFE2 : public GridFE
{
 protected:
  bool     structured_mesh;    // is the mesh structured

  Ptv(int) nodedim;            // no of nodes in each direction
  Ptv(int) elmdim;             // no of elements in each direction

 public:
  GridFE2();
  GridFE2(const GridFE2& grid);
  ~GridFE2();

  virtual bool redim
    (
     int nsd,                          // no of space dimensions
     int nno,                          // no of nodes
     int nel,                          // no of elements
     int maxnne,                       // max no of nodes in an element
     int nbind,                        // no of boundary indicators
     int nne,                          // =0: variable no of nodes in elements, or=maxnne
     bool onemat = true,               // one material (true) or several (false)
     bool keep_associated_bfg = false // do not change associated_bfg ptr
    );

  bool isStructured() const          { return structured_mesh; }
  void setStructuredMesh();
  void setUnstructuredMesh();

  int getNodeDim(int dir)     const { return nodedim(dir);    }
  const Ptv(int) getNodeDim() const { return nodedim;         }
  int getElmDim(int dir)      const { return elmdim(dir);     }
  const Ptv(int) getElmDim()  const { return elmdim;          }

  void setNodeDim(int dir, int dim)
    { nodedim(dir) = dim;    }
  void setNodeDim(const Ptv(int)& dim);
  void setElmDim(int dir, int dim)
    { elmdim(dir) = dim;     }
  void setElmDim(const Ptv(int)& dim);
};

#define ClassType GridFE2
#include <Handle.h>
#undef ClassType
```

```
#endif
```

Secondly, we need a class to store the block connectivity information. To this end we have implemented the class NeighborBlock, where the neighbor block information is represented in the same way as in the CGNS standard. The header file for this class is listed below.

```
#ifndef NeighborBlock_h_IS_INCLUDED
#define NeighborBlock_h_IS_INCLUDED

#include <Ptv_int.h>
#include <Ptm_int.h>

//---------------------- NeighborBlock ----------------------------------

/*<NeighborBlock:*/
class NeighborBlock : public virtual HandleId
{
 protected:
  bool has_neighbor;
  int  neighbor_id;
  int  master_id;

  Ptv(int) min_idx;
  Ptv(int) max_idx;

  Ptv(int) min_neighbor_idx;
  Ptv(int) max_neighbor_idx;

  Ptv(int) transform;

 public:
  NeighborBlock(int neighbor_id_,
                int master_id_,
                Ptv(int) min_idx_,
                Ptv(int) max_idx_,
                Ptv(int) min_neighbor_idx_,
                Ptv(int) max_neighbor_idx_,
                Ptv(int) transform_);
  NeighborBlock();

  ~NeighborBlock();

  bool hasNeighbor() const   { return has_neighbor; }
  void hasNeighbor(bool b)   { has_neighbor = b;    }
  int  getNeighborId() const { return neighbor_id;  }
  void setNeighborId(int id) { neighbor_id = id;    }

  int  getMasterId() const { return master_id; }
  void setMasterId(int id) { master_id = id;   }

  Ptv(int) getMinIdx() const { return min_idx; }
  Ptv(int) getMaxIdx() const { return max_idx; }
```

```
  void setMinIdx(Ptv(int) min) { min_idx = min; }
  void setMaxIdx(Ptv(int) max) { max_idx = max; }

  Ptv(int) getMinNeighborIdx() const { return min_neighbor_idx; }
  Ptv(int) getMaxNeighborIdx() const { return max_neighbor_idx; }

  void setMinNeighborIdx(Ptv(int) min) { min_neighbor_idx = min; }
  void setMaxNeighborIdx(Ptv(int) max) { max_neighbor_idx = max; }

  Ptv(int) getTransformVector() const     { return transform; }
  Ptm(int) getTransformMatrix() const;

  void setTransformVector(Ptv(int) t) { transform = t; }

  Ptv(int) findNeighborIdx(Ptv(int) loc_idx) const;

  bool findNeighborSubset(NeighborBlock& neighbor, Ptv(int) min, Ptv(int) max);
  bool findNeighborToNeighbor(NeighborBlock& neighbor);
};
/*>NeighborBlock:*/

#define ClassType NeighborBlock
#include <Handle.h>
#undef ClassType

#endif
```

The classes GridFE2 and NeighborBlock can now be used to defined a class GridFE2Block for block-structured grids. It consists of a vector of structured grids and a vector of vectors containing the neighbor connectivity for each block.

```
#ifndef GridFE2Block_h_IS_INCLUDED
#define GridFE2Block_h_IS_INCLUDED

#include <VecSimplestDef.h>


//---------------------- GridFE2Block ------------------------------

/*GridFE2Block:*/
class GridFE2Block : public HandleId
{
  friend class GridPartBlockCGNS;

 protected:
  int nblock;                                  // no of blocks

  VecSimplest(Handle(GridFE2)) grid_blocks;   // structured grid blocks
  VecSimple(int)               no_neighbor;   // no of neighbors for each block

  VecSimplest(VecSimple(Handle(NeighborBlock))) neighbor_blocks;  // block connectivity

 public:
  GridFE2Block();
```

```
  GridFE2Block(const GridFE2Block& grid);
  ~GridFE2Block();

  int getNoGridBlock()     const { return nblock;         }
  int getNoNeighbor(int i) const { return no_neighbor(i); }

  const GridFE2& getGridBlock(int i) const
    { return grid_blocks(i).getRef(); }
  GridFE2& getGridBlock(int i)
    { return grid_blocks(i).getRef(); }
  const NeighborBlock&  getNeighborInfo(int i, int j) const
    { return neighbor_blocks(i)(j).getRef(); }
  NeighborBlock&  getNeighborInfo(int i, int j)
    { return neighbor_blocks(i)(j).getRef(); }

  void setNoGridBlock(int size);
  void setNoNeighbor (int block, const int size);

  void setGridBlock(int block, GridFE2& grid)
    { grid_blocks(block).detach().rebind(grid); }
  void setNeighborInfo(NeighborBlock& neighbor, int i, int j)
    { neighbor_blocks(i)(j).detach().rebind(neighbor); }

  void copy(const GridFE2Block& grid);
};

#define ClassType GridFE2Block
#include <Handle.h>
#undef ClassType

#endif
```

6.2 Coupling to CGNS

To utilize the CGNS output from GRIDDLER in Diffpack, we have to implement a class CGNS which can read CGNS data into a multi-block structured grid object of type GridFE2Block. Future extensions of the CGNS class should also include routines for storing solution data in CGNS format.

6.3 Parallel communication in Diffpack

The parallel communication in Diffpack is administrated by the class GridPartAdm. This class has functions for computing parallel matrix-vector products, inner-products and norms, which are the linear algebra routines needed to parallelize linear equation solvers. To initialize an object of class GridPartAdm, we need an object of class GridPart which contains the submeshes associated with a given processor and their connectivity with neighboring submeshes. The Diffpack header file for the class GridPart is shown below.

```
#ifndef GridPart_h_IS_INCLUDED
#define GridPart_h_IS_INCLUDED
```

```
#include <GridPart_prm.h>
#include <GridFE.h>
#include <VecSimplest_Handle_GridFE.h>

//--------------------------------------------------------------------------
class GridPart : public HandleId
//--------------------------------------------------------------------------
{
  friend class GridPartAdm;

public:
  Handle(GridPart_prm) part_param;          // input parameters for partitioning
  VecSimplest(Handle(GridFE)) subd_grids;   // subgrids on current processor

  bool overlapping_subgrids;                // if the subgrids are overlapping

  bool matching_grids;                      // subgrids match?
  bool global_nnr_available;                // global node numbering available?
  bool global_enr_available;                // global element numbering available?

  int gno, gne;                             // global number of nodes/elements
  VecSimplest(VecSimple(int)) global_nnrs;  // global node numbers
  VecSimplest(VecSimple(int)) global_enrs;  // global element numbers

  VecSimplest(VecSimple(int)) neighbor_ids; // id for neighboring subgrids

  VecSimple(int) num_ib_nodes;              // number of nodes on interior boundaries
  VecSimplest(VecSimple(int)) ib_node_ids;  // nodes on interior boundaries

public:
  GridPart (const GridPart_prm& pm);
  virtual ~GridPart () {  DPTRACE("GridPart:: ~GridPart"); }

  virtual bool makeSubgrids ()=0;
  virtual bool makeSubgrids (const GridFE& global_grid)=0;

  virtual GridFE& getSubgrid (int i=1)
    { return subd_grids(i)(); }
  virtual VecSimplest(Handle(GridFE))& getAllSubgrids ()
    { return subd_grids; }

  const VecSimplest(VecSimple(int))& getGlobalNnrs () const
    { return global_nnrs; }
  const VecSimplest(VecSimple(int))& getGlobalEnrs () const
    { return global_enrs; }

  static bool initNeighbor (GridFE& grid, MatSimple(int)& elm_neigh);
};

#endif
```

Hence, we have to implement a subclass of GridPart to store the partitioning of the mesh imposed by the block-structure. Since the output from GRIDDLER only provides

the connectivity information for the block faces, this class should also retrieve the connectivity data associated with the block vertices and edges. Then the connectivity data must be used to define a global numbering of the nodes and elements. A general implementation of such a class has not been accomplished yet. Instead we have hardcoded the data needed in the GridPart class for a particular example to give a proof of concept. The example is explained in detail in Appendix C.

Appendix A. ADFviewer and CGNSplot

The **ADFviewer** program provides a graphical user interface to view and edit ADF or CGNS files. The **CGNSplot** program displays the mesh, element sets, connectivities and boundary conditions defined in a CGNS file. Other CGNS utilities exist as well. However, here we concentrate on these tools because they give us the needed opportunity to investigate the grid, connectivity and BCs that we are making by the GRIDDLER interface to CGNS.

Appendix B. MULTI-BLOCK MESH EXAMPLE

A 3D example with 11 blocks around a cylinder is given. A solid rendering of this model is shown in Figure 8. The rendering is just a window dump of the CGNSplot window of this model shown in perspective. In Figure 9 the mesh of the same model is shown. Notice the easy identification of each block by different colors.

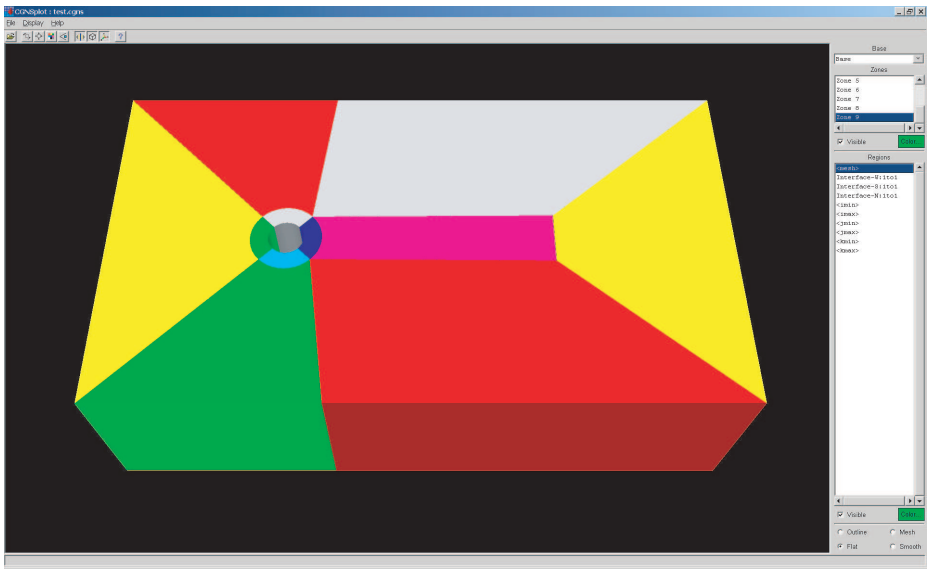

Figure 8: A 3D model composed of 11 blocks around a cylinder.

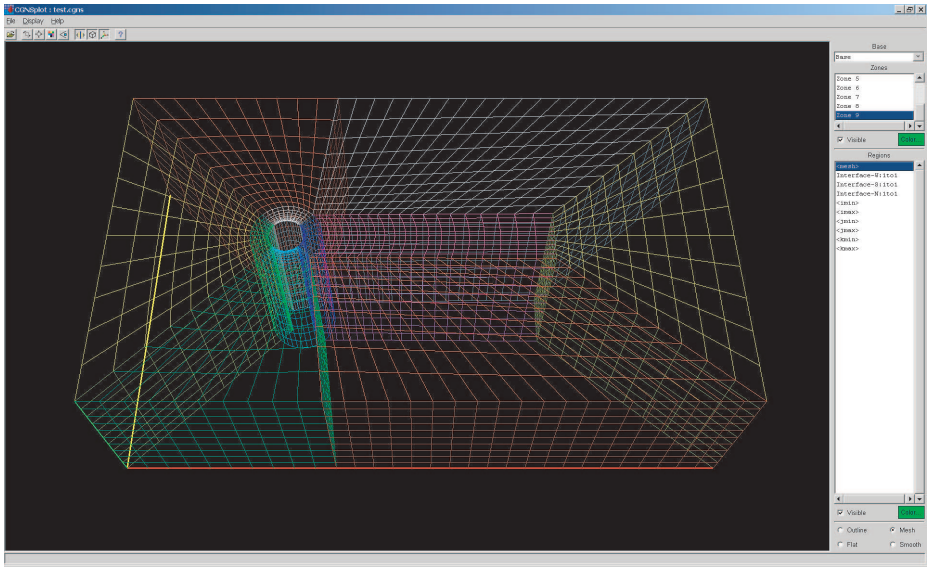

Figure 9: The meshed 3D model of Figure 8.

In Figure 10 is shown a window dump of the ADFviewer window after reading the previous cylinder model. Notice that both the connectivity data and the BC data can be checked by this program and the CGNSplot program.

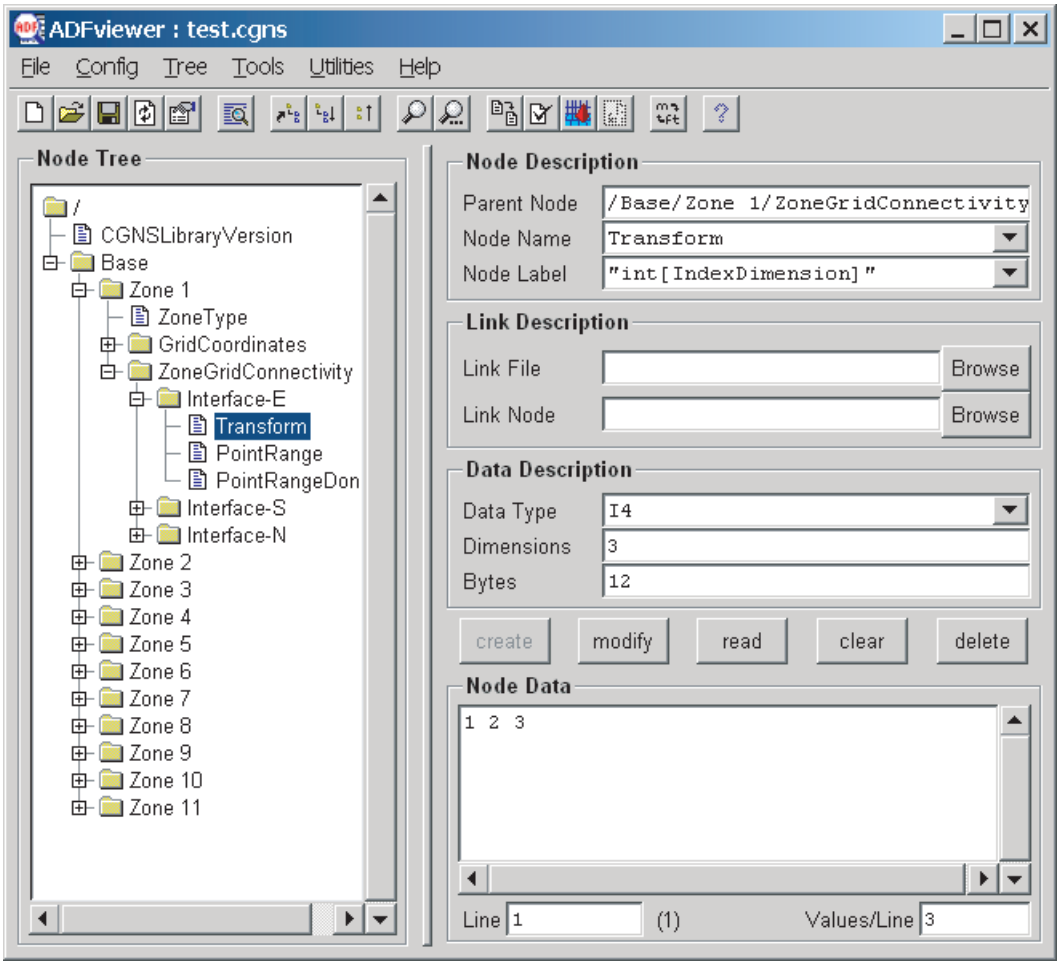

Figure 10: The cylinder model data listed in the ADFviewer.

Appendix C. DIFFPACK INTEGRATION

Here we describe an example we have run in parallel in Diffpack using a simple multi-block structured grid. The geometry is an L-shaped domain consisting of 3 grid blocks as shown in Figure 11. Rendering of the resulting solution field is show as well.

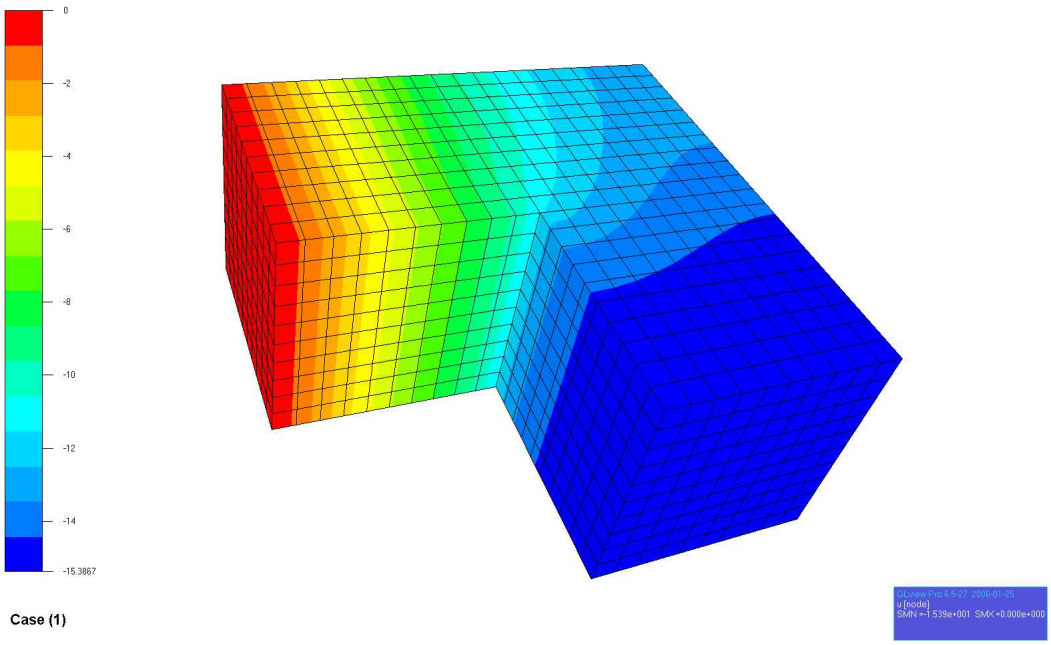

Figure 11: L-shaped domain composed of three structured blocks.

We only want to give a "proof of concept" of the methodology we want to use to extend Diffpack and the Vista code to handle block-structured grids. Thus, we only test that we are able to read the CGNS input file, set up the parallel communication and solve a 3D Poisson problem using iterative solvers. In the example we restrict ourselves to non-overlapping meshes and we use no preconditioner for the linear solvers.

For the test we have used a modified version of the parallel Poisson solver Poisson1 which is part of the Diffpack distribution. To test the block-structured partitioning of the mesh, we have implemented a subclass GridPartBlockCGNSTest of the class GridPart where we have hardcoded the partitioning data for this particular example.

```
#ifndef GridPartBlockCGNSTest_h_IS_INCLUDED
#define GridPartBlockCGNSTest_h_IS_INCLUDED

#include <GridPart.h>
#include <CGNS.h>
#include <DistrProcManager.h>


//-------------------- GridPartBlockCGNSTest ----------------------
```

```
class GridPartBlockCGNSTest : public GridPart
{
 public:
  Handle(GridFE2Block) block_grid;

 public:
  GridPartBlockCGNSTest(const GridPart_prm& pm);
  virtual ~GridPartBlockCGNSTest();

  virtual bool makeSubgrids();
  virtual bool makeSubgrids(const GridFE& global_grid);
};

#endif
```

Here the block-structured mesh on the current processor contains a single block. The member function GridPartBlockCGNSTest::makeSugrids reads the grid block from file using the functionality in the class CGNS, and then initializes the datastructures defined in the Diffpack class GridPart.

The last step in the implementation is to make the new partitioning available to the GridPartAdm objects which administrates the parallel computations in Diffpack. This is done through the following steps.

```
  // We assume that the following objects are defined
  Handle(GridPartAdm)  gp_adm;
  Handle(GridPart_prm) pm;
  Handle(GridPart)     gridpart;

  ...

  // In Poisson1::scan the following modifications are applied
  gp_adm.rebind (new GridPartAdm);
  pm.rebind(new GridPart_prm);
  gridpart.rebind(new GridPartBlockCGNSTest(*pm));
  gp_adm->attachGridPartPrm(*pm);
  gp_adm->attachPartitioner(*gridpart);
  gp_adm->scan (menu);
  gp_adm->prepareSubgrids ();
  grid.rebind (gp_adm->getSubgrid());
```

Here the class GridPart_prm is used to store the menu input for the gridpartitioning, i.e. the name of the CGNS file, the number of grid blocks, if the subgrids overlap, etc. The rest of the initialization of parallel Diffpack is left unchanged.

The test problem we solve is given by

$$\begin{cases} -\triangle u = 1 & \text{in } \Omega, \\ u = 0 & \text{on } \Gamma_{\text{in}}, \\ \dfrac{\partial u}{\partial \boldsymbol{n}} = 0 & \text{on } \partial\Omega \setminus \Gamma_{\text{in}}, \end{cases}$$

where Γ_{in} denotes the west side of the first block. For the solution of the linear systems we have used different Krylov subspace solvers like the conjugate gradient (CG) method, GMRES and BiCGStab. We have verified that the serial and parallel version of the code produce the same numerical solution and that the iteration number is the same.

References

[1] CGNS Project Group, "The CFD General Notation System Standard Interface Data Structures," Version 2.0 beta 2, February 2001; http://www.grc.nasa.gov/www/cgns/sids/index.html

[2] CGNS Project Group, "The CFD General Notation System Advanced Data Format (ADF) User's Guide," April 2001; http://www.grc.nasa.gov/www/cgns/adf/index.html

[3] CGNS Project Group, "The CFD General Notation System SIDS-to-ADF File Mapping Manual," Version 1.2 revision 8, February 2001; http://www.grc.nasa.gov/www/cgns/filemap/index.html

[4] Langtangen, H. P., "Computational Partial Differential Equations: Numerical Methods and Diffpack Programming". Springer Verlag, Heidelberg, 1999.

[5] Karypis, G., and Kumar, V., "METIS, A software package for partitioning graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0". Department of Computer Science/Army HPC Research Center, University of Minnesota, September, 1998.
Available from url: http://www-users.cs.umn.edu/~karypis/metis/metis/download.html

[6] Poirier, D. M. A., Allmaras, S., McCarthy, D. R., Smith, M., and Enomoto, F., "The CGNS System," AIAA Paper 98-3007, June 1998.

[7] CGNS Project Group, "The CFD General Notation System Mid-Level Library," July 2001; http://www.grc.nasa.gov/www/cgns/midlevel/index.html

[8] Poirier, D. M. A., Bush, R. H., Cosner, R. R., Rumsey, C. L., and McCarthy, D. R., "Advances in the CGNS Database Standard for Aerodynamics and CFD," AIAA Paper 2000-0681, January 2000.

[9] Walatka, P. P., Buning, P. G., Pierce, L., Elson, P. A., "PLOT3D User's Guide," NASA TM 101067, March 1990.