parpp3d++

A parallel C++ version of the *pp3d* module in FEATFLOW

User guide

v1.1

Author: Sven H.M. Buijssen

UNIVERSITY OF DORTMUND

INSTITUTE FOR APPLIED MATHEMATICS AND NUMERICS

Contents

List of Tables

List of Figures

1 Up to the starting line

What is parpp3d++? *parpp3d++* is a parallel 3D code for the solution of incompressible nonstationary Navier–Stokes equations. It is an adaptation, i.e. parallel implementation, of the existing sequential solver `pp3d` from the FEATFLOW package and, as such, applies the same numerical methods. See [17, 5] for mathematical details.

parpp3d++ is not capable of solving 2D flow problems. According efforts are not ventured either on the basis of the FEAT package. It will not be before the completion of the currently developed, new FEASTFLOW package that parallel 2D simulations will be potentiated [6, 7]. For details about its release date see our website `www.featflow.de`.

Supported platforms: *parpp3d++* has been successfully compiled, deployed and tested in almost every UNIX environment. It does not run, though, on Windows® 95/98/ME/NT/2K/XP, MacOS® nor OS/2®. Considering the fact that the aim of *parpp3d++* is High Performance Computing, especially using the tremendous computing power of multi-processor workstations, clusters or supercomputers, this restriction to UNIX flavours is quite a matter of course. Thus, to run *parpp3d++*, you need to have a UNIX system.

Prerequisities: For the impatient:

- 30 - 70 MB disk space for sources and object files
- more than 10 MB disk space for visualisation files
- C/C++ compiler (with STL support)
- MPI environment (headers + libraries)

The program has only very basic prerequisities.The most important thing is disk space. You will need 20 MB for the compressed and unpacked sources. Depending on your compiler and compile flags (whether or not you are including debug information etc.) an additional amount of up to 50 MB is needed for object files, libraries and the linked program. During run time, large quantities of storage space will be consumed if you choose to have visualisation output. Prescribing small time steps or a high grid density (i.e. high multigrid output level) hundreds of megabytes, even gigabytes, can easily be stored on your hard drive in a single program run.

Further on, you will need a reasonably featured C as well as a C++ compiler (with at least basic STL support). *parpp3d++* has been well-tested with GNU (2.9x, 3.x and 4.0), Intel, PGI and KAI Compilers on Linux, Sun, SGI and Digital Compilers on their respective platforms, too. Given the fact that the code has already been ported to a vast number of platforms it should not pose signifcant problems porting it to yet another architecture, compiler or MPI environment. The settings in `Makefile.inc` can serve as a template then.

Finally, you will need an MPI 1.x or above environment (including headers and libraries).

parpp3d++ ships with pre-defined settings (stored in `Makefile.inc`) for the cases shown in table 1. See section 2.1 for how to invoke these settings.

architecture	compiler	MPI environment
Alpha	Compaq C/C++ 6.x GNU C/C++ 2.95.x - 4.0.2	Digital MPI LAM/MPI 7.x MPICH 1.1 Digital MPI
Hitachi SR8000	KAI C/C++	Hitachi MPI2
Linux (Pentium3, Pentium4, Athlon, AthlonXP, Nocona, Opteron)	GNU C/C++ 2.95.x - 4.0.2 Intel C/C++ 8.0 PathScale C/C++ PGI C/C++ 6.0	LAM/MPI 7.x TopSpin MPI
SUN	GNU C/C++ 2.95.x - 4.0.2	LAM/MPI 7.x

Table 1: *parpp3d++* ships with settings for these combinations of architectures, compilers and MPI environments

Contact: parpp3d@featflow.de

2 Installation

Unlike most modern free software, *parpp3d++* does not provide a comfortable configure script which determines system type and installed software and automatically generates a taylor-made Makefile. But this does not mean that you are on the point of embarking upon big trouble if you have decided to try out *parpp3d++*. In most cases you will have to make only few changes to `Makefile.inc`. Mainly, these will consist of setting up your desired C/C++ compiler as well as the path to your MPI library and MPI header files.

2.1 Configuration

Download `parpp3d++.1.x.x.tar.gz` from our website and unpack it to a directory of your choice. A new subdirectory called `parpp3d++` will be created, containing the source code as well as some exemplary parameter files and corresponding grids.

We will first describe how to set up the Makefile for *parpp3d++*. Change to the newly created directory, invoke your favorite editor and open `Makefile.inc`.

```
% cd parpp3d++
% vi Makefile.inc
```

This file contains all those settings that you will have to alter most likely. It is quite improbable that you will encounter the need to edit the actual Makefiles yourself. If you plan to incorporate own modules into *parpp3d++*, however, you will have to update the dependencies in the Makefiles involved.

As recent versions of FEATFLOW and FEAST, *parpp3d++* uses internally a script called `guess_id` to retrieve information on the architecture it is tried to compile on. On a AMD AthlonXP the output of

```
% bin/guess_id
```

is, for instance,

```
pc-athlonxp-linux32
```

on a IBM p690 it would be

```
ibm-powerpc_power4-aix
```

For every ID determined by `guess_id` a default choice of compiler and MPI environment is defined in the top section of `Makefile.inc`.

The remainder of `Makefile.inc` consists of sections describing compiler and MPI settings to use during compilation. Each section is uniquely identified by a string consisting of 5 tokens connected with dashes. These tokens describe architecture, CPU, operating system, MPI environment and compiler. The string is refered to as *build target id*.

You will, for instance, find sections named `pc-athlon-linux32-lammpi-gnu`, `pc-athlonxp-linux32-lammpi-gnu`, `pc-nocona-linux32-lammpi-intel`, `sun4u-sparcv8-sunos-lammpi-gnu` etc.

This way you can keep a single `Makefile.inc` for all your different compute platforms and simply activate the appropriate settings by either "magic" or by prescribing the build target ID you prefer. The first ("magic") mechanism is triggered by simply invoking

```
% make
```

on the command line. `guess_id` will determine architecture, CPU and operating system whereupon the default compiler and MPI environment for this triple will be selected. For `pc-opteron-linux64` this would, for instance, be `pc-opteron-linux64-lammpi-gnu`. The latter can be done by either hard-coding it in `Makefile` [1] (the one in the top-most directory of your *parpp3d++* installation) or by prescribing the build target ID directly on the command line:

```
% make ID=pc-opteron-linux64-lammpi-pgi
```

If the default settings do not meet your system or you want to add more build target ids, you will have to modify `Makefile.inc`. Each section describing the settings for one build target ID defines the following variables:

- `CPP` – indicating the path to the C++ compiler

[1] There is line commented out directly before `Makefile.inc` gets included in `Makefile` that indicates how this works.

- CC – indicating the path to the C compiler
- CFLAGSCPP – optimisation switches for the C++ compiler
- CFLAGSC – optimisation switches for the C compiler
- LD – indicating the path to the linker
- LDFLAGS – switches to be passed to the linker
- INC – additional include paths to look for header files
- BUILDLIB – libraries to be built[2]
- LIBDIR – list of directories to be searched for libraries specified with -l.
- LIBS – list of libraries to be search when linking
- AR – program to create, modify and extract from archives
- RANLIB – program to generate an index to the contents of an archive

If you are familiar with compiling code yourself you will have no problems choosing appropriate values here. If you are unsure, ask your system administrator or local "unix guru".

If you have a look in Makefile.inc you will notice that CFLAGSCPP and CFLAGSC are always defined twice. Either one will be used depending on the value of OPT, which stands for optimisation (yes/no). The variable OPT can either be defined in Makefile or directly on the command line, as with the build target id ID.

It is always a good idea when you are trying to compile *parpp3d++* for the first time on a new platform, to start with

```
% make OPT=NO
```

Once you have compiled *parpp3d++* successfully, add code optimisation, re-compile by issuing

```
% make clean; make
```

and have an extensive coffee break. :-)

Compilation with full optimisation will take at least minutes, if not hours.[3]

[2] On a typical system set this to metis umfpack. If your system has metis or umfpack installed system-wide, try to use those libraries and omit them here.

[3] Well, I have seen both and it will, obviously, strongly depend on the power of your machine.

Program-specific compile opions: There are a few compile options which are specific to *parpp3d++* and which can be set by adding them to CFLAGSCPP in Makefile.inc.

- If you want to have **run time statistics** about how much time has been spent in matrix assemblation, prolongation and restriction, communication routines etc., add -DCLOCK_MEASURE.
- If you want **Boussinesq model** added to the Navier–Stokes equations and export this data into the visualisation output files, add -DINCLUDE_TEMPERATURE.
- If you want **vorticity information** included into the visualisation output files, add -DINCLUDE_VORTICITY.
- If you would like to see details about the **stop criterions** applied within nonlinear and linear **solver** steps, specify -DMG_DEBUG.
- If your MPI environment incorporates **MPI-C++-Bindings** and you experience **severe compile errors**, try specifying -DMPIPP_H
- If your compiler complains about an error in function OutOfMemory in module CProcessApp.cc, specify -DNO_OUT_OF_MEMORY_HANDLER.[4]

2.2 Compilation

Having set up Makefile.inc according to your system configuration, type

```
% make
```

This should build the executable ./parpp3d++.[5]

If you need help on available built options, invoke

```
% make help
```

If you want to run a quick test, take one of the sample parameter files from

[4] OutOfMemory handles the out-of-memory execption anywhere in the program. Specifying -DNO_OUT_OF_MEMORY_HANDLER disables this feature. Consequence: If your problem size gets too big, *parpp3d++* will just terminate without an eligible error message.

[5] If you prefer a different program name, change the according value APP in Makefile in the top-most directory of your *parpp3d++* installation.

the subdirectory `samples`, rename it to parpp3d++.dat[6] and invoke the MPI run command. Have a look at the documentation of your MPI environment for the correct syntax. In most cases, however, it will be a statement similar to

```
% mpirun -c 2 ./parpp3d++
```

or

```
% mpirun -np 2 ./parpp3d++
```

Some MPI implementations require that you fire up a MPI daemon first[7]. For LAM/MPI, for instance, you have to invoke

```
% lamboot
```

first. For anything special (like running the same application on multiple hosts etc.) this will not suffice, so you then should check the LAM documentation [12] or see the appendix of [5]. The same holds if you use other MPI implementations.

[6]The executable assumes that the parameter file has the same basename as the executable and bears a '.dat' suffix. In case you have changed the value `APP` in `Makefile` or renamed the executable to, e.g., '3dsim', you ought to have a parameter file called '3dsim.dat'.

[7]For the popular LAM/MPI that's the case.

3 Syntax of parameter files

In order to perform a 3D flow simulation with *parpp3d++*, you need to traverse five to six steps:

- Download the program source files,
- compile the sources,
- set up a working parameter file,
- create a grid (triangulation),
- prescribe inflow and outflow conditions (hard-coded) and, possibly,
- perform boundary projections (hard-coded)

The first two steps have been dealt with in previous sections, now we will discuss the syntax of the program's parameter file. If you are familiar with the parameter files belonging to the sequential programs from the FEATFLOW package, you will assert that the options are quite acquainted.[8] The subdirectory `samples` contains a few ready-to-run configurations, e.g. flow through a unit cube, flow through a channel around an obstacle, lid-driven cavity, flow within a steel mould.

As with its sequential predecessors from the FEATFLOW package the main purpose of the parameter file is to control the numerical solution process: time stepping schemes to be applied, stabilisation methods, stop criterions, number of multigrid levels etc. But it is used as well to prescribe path and file name of the grid file, Reynolds number, restart information and – this is new in parallel – partitioning information.

On startup, the executable searches the current directory for a parameter file with the same name as the executable with a `.dat` suffix added. So, if you choose to keep the default program name, a parameter file called `parpp3d++.dat` will be searched for.

The parameter file is assumed to be an plain ASCII file. Options are specified one per line. White spaces are ignored as well as everything behind the standard C++ comment delimiter (//). The **order** in which key words are

[8]The key words are just slightly less cryptic! See chapter 10.1 for equivalent key words.

specified **does matter**, unfortunately. But, the length of strings does not matter. This is especially true for path and file names.[9]

Let's have a look at an exemplary parameter file: `unitcube.dat` from the subdirectory `samples` (see previous page). To increase readability for humans, a three-columns format has been used here. The first column contains the option value, either numeric or string. The second column specifies the programs variable (which is, in fact, a kind of key word) this value is assigned to. The last column has some explanatory comments on the key word.

The values in the first column are mandatory, the rest is - being comments - optional. So, if you prefer a more Spartanic parameter file, just omit all comments. Because everything apart from the options in the first column is placed in comments, the first five lines of `unitcube.dat` in figure 1 could be equivalently reduced to as less as

```
1
0
Grids/UnitCube/Triaq512
0
1
```

which is, obviously, a nightmare to maintain.

On the next pages, you will find a complete reference list of key words. You don't need to know exactly what each item does. The most vital key words are as follows: `EpsEqu`, `Func`, `GMVSolutionFile`, `GridFile`, `MaxTimeIterations`, `NFine`, `OutputBaseDir`, `Partition*`, `Restart*`, `TEnd`.

Complete list of parameter file key words:

`AMaxP` *float value*
: Sets an upper bound for the adaptively chosen relaxation parameter used to update pressure solution in multigrid's prolongation step.

`AMaxU` *float value*
: Sets an upper bound for the adaptively chosen relaxation parameter used to update velocity solution in multigrid's prolongation step.

[9] A note for those of you who know the sequential FEATFLOW programs: There, you have a limit of 15 characters for path and file names. This definitely had to change, had it not?

```
// Value          // program variable     // description                                               //
// ------------------------------------------------------------------------------------------------------ //
        1         // TestLoops            // number of subsequent configurations in this file

// --- Simulation and grid ---------------------------------------------------------------------------- //
        0         // Func                 // simulation type:
                                          // 0: unit cube
                                          // 2: ASMO
                                          // 3: driven cavity (pursuant to Deville et al.)
                                          // 4: channel flow with two consecutive cylinders
                                          // 5: channel flow with one cylinder (DFG Benchmark 3D-2Z)
                                          // 8: Cast flow (ABB project)
                                          // 9: BMBF project with Agar
Grids/UnitCube/Triaq512
                  // GridFile             // file name of coarse grid
// --- Restart section -------------------------------------------------------------------------------- //
        0         // Restart              // restart flag:
                                          // 0: no restart
                                          // 1: restart on same level
                                          // 2: restart coarser level
        1         // RestartITE           // starting number for iterations and files
./comp/           // RestartBaseDir       // base directory for solution files
  none.r1         // RestartSolFile       // prefix for the file family that actually
                                          // contain the data for your restart
       10         // SolFileFrequency     // write solution file every xx iteration
        3         // SolFileNumber        // number of different solution files to keep
restart.lev3      // SolFilePrefix        // prefix for solution files (for restart purposes)
// --- Partition section ------------------------------------------------------------------------------ //
        1         // PartitionTool        // third party partition tool:
                                          // 0: party
                                          // 1: metis - variant 1
                                          // 2: metis - variant 2
        0         // PartitionCR          // switch whether to
                                          // 0: create or to
                                          // 1: read partition information
./comp/           // PartitionBaseDir     // base directory for partition files
benchmark         // PartitionFile        // partition file

// --- equation parameters and some for discretization ----------------------------------------------- //
    1.0e3         // EpsEqu               // viscosity parameter 1/nu
                                          // - 1.0e3 for simulation type 4 & 5
                                          // - 1.0e5 for simulation type 2
                                          // - 3.2e3 for simulation type 3
                                          // - 1.0e6 for simulation type 8
        3         // NFine                // maximum mg-level
        7         // ICUB                 // cubature formula for calculation of matrices
        7         // ICUB RHS             // cubature formula for calculation of right hand side
        0         // Lump                 // toggle mass matrix lumping (0: yes, 1: no)
      1.0         // UpSam                // Samarski upwind parameter

// --- parameters for solving burgers equation -------------------------------------------------------- //
        2         // MinFixpItU           // min. number of fixpoint iterations (nonlinear)
        5         // MaxFixpItU           // max. number of fixpoint iterations (nonlinear)
      1.0         // OmgIni               // initial relaxation parameter (nonlinear)
```

(continued on next page)

Figure 1: Sample parameter file `samples/unitcube.dat`

```
        1          // MinIItU               // min. number of it. in inner (linear) iteration
        5          // MaxIItU               // max. number of it. in inner (linear) iteration
     1e-1          // EpsUChange            // limit for U-changes
     1e-8          // EpsUDefect            // limit for U-defects
     1e-1          // DampUMG               // limit for U-defect improvement
      0.0          // AMinU                 // lower limit for optimal U-alpha
      2.0          // AMaxU                 // upper limit for optimal U-alpha
        1          // PreSteps_burg         // number of presmoothing steps solving burgers-eq.
        1          // PostSteps_burg        // number of postsmoothing steps solving burgers-eq.
        3          // Smoother_burg         // type of smoother for burgers-equation:
                                            // 1: jacobi / 2: SOR / 3: ILU / 4: CG
      1.0          // MGOmega_burg          // omega value used in smoothing process
        2          // Solver_burg           // exact solver on coarse grid:
                                            // 1: jacobi / 2: SOR / 3: GS / 4: CG (ILU)
      100          // SolverMaxIt_burg      // max. number of iterations in coarse grid solver
                                            // approved values:
                                            // jacobi: 200 / SOR,GS,CG: 100
        0          // Cycle_burg            // multigrid cycle used: 0:F  1:V  2:W

// --- parameters for solving pressure poisson equation ---------------------------------------- //
        2          // MinIItP               // min. number of it. in inner (linear) iteration
      100          // MaxIItP               // max. number of it. in inner (linear) iteration
    1e+10          // EpsPChange            // limit for P-changes
     1e-8          // EpsDivergence         // limit for divergence(u)
                                            // (is equal to P-defects * current time step)
     1e-1          // DampPMG               // limit for P-defect improvement
      0.0          // AMinP                 // lower limit for optimal P-alpha
      2.0          // AMaxP                 // upper limit for optimal P-alpha
        2          // PreSteps_press        // number of presmoothing steps solving pressure-eq.
        2          // PostSteps_press       // number of postsmoothing steps solving pressure-eq.
        1          // Smoother_press        // type of smoother for pressure-equation:
                                            // 1: jacobi / 2: SOR / 3: ILU
      1.0          // MGOmega_press         // omega value used in smoothing process
        3          // SolverType_press      // type of multigrid solver used:
                                            // 1: conventional multigrid
                                            // 2: cg method preconditioned with one
                                            //    additive multigrid step
                                            // 3: cg method preconditioned with one
                                            //    multiplicative multigrid step
        4          // Solver_press          // exact solver on coarse grid:
                                            // 1: jacobi / 2: SOR / 3: GS / 4: CG (ILU)
       10          // SolverMaxIt_press     // max. number of iterations in coarse grid solver
                                            // approved values:
                                            // jacobi: 200 / SOR: 500 / GS: 50 / CG: 100
        0          // Cycle_press           // multigrid cycle used: 0-F  1-V  2-W
        2          // ProlType_press        // type of pressure prolongation:
                                            // 1: constant prolongation
                                            // 2: linear prolongation
// --- Time iteration parameter ------------------------------------------------------------------ //
       -4          // Method                // time step scheme used:
                                            //  1: Chorin (1st order)
                                            //  2: VanKan (2nd order)
                                            // -x: VanKan with |x| initial Chorin steps
     1000          // MaxTimeIterations     // maximum number of iterations in time
      5.0          // TEnd                  // endpoint in time
```

(continued on next page)

Figure 1: Sample parameter file `samples/unitcube.dat` (continued)

```
     1e-4          // EpsNS                 // lower limit for time derivative
 0.001000          // DtStart               // time step to start with (not macro time step!)
        1          // TStepControlITE       // number of iterations between next time step control
    0.001          // DtMin                 // minimum time step
      4.0          // DtMax                 // maximum time step
      1.0          // TInitPhase            // duration of start phase
    0.001          // EPSADI                // parameter for time error limit in start phase
   0.0001          // EPSADL                // parameter for time error limit after start phase

// --- Output format, location and frequency ---------------------------------------------------- //
./postprocess/     // OutputBaseDir         // base directory for all output files
        0          // AVSOutputLevel        // level of refinement that is used for avs output
                                            // ('0' means no avs output)
benchmark.grid.13  // AVSGridFile           // path and prefix for avs grid output file name
benchmark.13       // AVSSolutionFile       // path and prefix for avs output files
      0.0          // DtAVS                 // time difference for avs output
        3          // GMVOutputLevel        // level of refinement that is used for gmv output
                                            // ('0' means no gmv output)
benchmark.grid.13  // GMVGridFile           // path and prefix for gmv grid output file name
benchmark.13       // GMVSolutionFile       // path and prefix for gmv output files
      0.1          // DtGMV                 // time difference for gmv output

// --- Debug section ----------------------------------------------------------------------------- //
        0          // Debug                 // toggle "function call tracing"
```

Figure 1: Sample parameter file `samples/unitcube.dat` (continued)

`AMinP` *float value*
: Sets a lower bound for the adaptively chosen relaxation parameter used to update pressure solution in multigrid's prolongation step.

`AMinU` *float value*
: Sets a lower bound for the adaptively chosen relaxation parameter used to update velocity solution in multigrid's prolongation step.

`AVSGridFile` *string*
: Specifies the file name that will contain the grid on level `AVSOutputLevel` in AVS format.

`AVSOutputLevel` *integer value*
: Sets the grid density (grid refinement level) for the visualisation output in AVS format.
Specifying numbers bigger than 5 will easily lead to an exceeded disk quota or hard disk capacity. The critical value is depending on the number of elements of your coarse grid. A level leading to roughly 50.000 to 100.000 elements is sufficient for most visualisation purposes.
Values bigger than `NFine` are automatically reduced to the value given for `NFine`.

`AVSSolutionFile` *string*
Specifies the base file name of visualisation output on level `AVSOutputLevel` in AVS format. Information on the current time step and the process number will be appended as well as a suffix `.avs` (see section 8.3).
Example: Specifying `unitcube` will lead to files named `unitcube.t###.p###.avs`

`Cycle_burg` *integer value*
Specifies the type of multigrid cycle to be used for solving linearised Burgers equations.
`0` enables F-cycle, `1` uses V-cycle, `2` invokes W-cycle.

`Cycle_press` *integer value*
Specifies the type of multigrid cycle to be used for solving Pressure Poisson equations.
`0` enables F-cycle, `1` uses V-cycle, `2` invokes W-cycle.

`DampPMG` *float value*
Specifies the minimal factor the initial pressure defect has to be reduced. The same remark applies as for `EpsPChange`.

`DampUMG` *float value*
Specifies the minimal factor the initial velocity defect has to be reduced. `DampUMG` is one of the stop criterions applied in fixpoint iteration to solve Burgers equation.

`Debug` *integer value*
A switch for debugging purposes only.
If the program terminates unexpectedly, set this option to `1`. You will get a message each time the program enters and leaves important functions. This way you can roughly determine the erroneous routine.

`DtAVS` *float value*
Sets the minimum amount of time between to subsequent output files in AVS format. Please read section 8.3 (page 20) about the time difference in subsequent visualisation output files.
If you want a visualisation file for every single time step, set this value to `0.0`.

`DtGMV` *float value*
Sets the minimum amount of time between to subsequent output files in GMV format. Please read section 8.3 (page 20) about the time difference in subsequent visualisation output files.
If you want a visualisation file for every single time step, set this value to `0.0`.

`DtMax` *float value*
Sets an upper bound for the time step size.

`DtMin` *float value*
Sets a lower bound for the time step size.

`DtStart` *float value*
Sets the initial time step.
By setting `TStepControlITE` to an exceptional high value, you can use `DtStart` even to prescribe a global uniform time step.
Example: Set `TStepControlITE` to `10000` and you will have equidistant time steps of size `DtStart`.

`EPSADI` *float value*
Sets the tolerable error for the time derivative during start phase.

`EPSADL` *float value*
Sets the tolerable error for the time derivative after start phase.

`EpsDivergence` *float value*
Specifies an upper limit for the divergence of the velocity solution.
The reason for this parameter being part of the pressure parameter section is quite easy to explain: Keep in mind that in our discrete projection approach the pressure defect can be controlled by `EpsDivergence` divided by the current time step (see [17], page 66) and vice versa.

`EpsEqu` *float value*
Specifies the reciprocal of the viscosity parameter ϵ in the Navier–Stokes–equations.
Warning: This value is **not just the viscosity parameter** for the medium you are simulating. So, there is more to do than simply looking this value up in a table. Instead, you will need characteristic velocity U and characteristic length L for your configuration as well as the Reynolds number Re. Then calculate the viscosity yourself using the formula

$$\texttt{EpsEqu} = U \cdot L / Re$$

This is because, in science, one usually uses metre as standard unit, but it would be inappropriate to express dimensions and distances (and in

a way viscosity) in metres if simulating a configuration that is several orders of magnitude larger or smaller than 1 metre.
So, instead of using kinematic viscosity's value for ϵ in the Navier–Stokes–equations directly, calculate ϵ by applying the formula above according to the units that hold for your grid dimensions and inflow conditions.

`EpsPChange` *float value*
: Specifies the minimum relative change between initial and acceptable pressure solution.
This parameter has hardly any effect on solution accuracy or iteration count if `EpsDivergence` is set reasonably, i.e. to something like 10^{-6} – 10^{-10}.

`EpsNS` *float value*
: Specifies the lower limit for the time derivate of the velocity solution.
In case it turns out that the simulation has a stationary limit, this value controls when to terminate the program.

`EpsUChange` *float value*
: Specifies the minimum relative change between initial and acceptable velocity solution.
`EpsUChange` is one of the stop criterions applied in fixpoint iteration to solve Burgers equation.

`EpsUDefect` *float value*
: Specifies the maximum tolerable value for the velocity defect, measured in l2 norm.
`EpsUDefect` is used as stop criterion in fixpoint iteration to solve the non-linear Burgers equation in every time step as well as in multigrid routines to solve linearised Burgers equations.

`Func` *integer value*
: If you choose to incorporate all your different flow configurations in a single version of *parpp3d++* and not to have multiple version of the program lingering around – one for each different flow configuration –, this multiswitch activates the hard-coded inflow and outflow conditions as well as additional configuration-specific tasks like boundary projections, computation of lift and drag values, pressure differences etc.
We will learn later which parts of the program have to be at least adjusted to create a new flow configuration (see chapter 5 - 7).

`GMVGridFile` *string*
: Specifies the file name that will contain the grid on level `GMVOutputLevel` in GMV format.

`GMVOutputLevel` *integer value*
: Sets the grid density (grid refinement level) for the visualisation output in GMV format.
Specifying numbers bigger than 5 will easily lead to an exceeded disk quota or hard disk capacity. The critical value is depending on the number of elements of your coarse grid. A level leading to roughly 50.000 to 100.000 elements is sufficient for most visualisation purposes.
Values bigger than `NFine` are automatically reduced to the value given for `NFine`.

`GMVSolutionFile` *string*
: Specifies the base file name of visualisation output on level `AVSOutputLevel` in GMV format. Information on the current time step and the process number will be appended as well as a suffix `.gmv` (see section 8.3).
Example: Specifying `unitcube` will lead to files named `unitcube.t###.p###.gmv`

`GridFile` *string*
: Path and file name of the triangulation file to be used.
If not absolute, the path will be treated as relative to the programs path.

`ICUB RHS` *integer value*
: Specifies the internal (FEAT) number for the cubature formula to be used to calculate right hand side vectors.
The same remarks apply as for *ICUB*.

`ICUB` *integer value*
: Specifies the internal (FEAT) number for the cubature formula to be used to assemble matrices.
See the FEATFLOW manual, if you want to learn more about possible settings. Unless you really know what you are doing, leave this value unchanged.

`Lump` *integer value (boolean)*
: Switch to control whether lumping of mass matrix should be enabled (0) or disabled (1).

`MaxFixpItU` *integer value*
: Sets an upper limit for the number of fixpoint iterations to be performed to solve nonlinear Burgers equation in each time step.
Values greater than 10 should be prevented.

`MaxIItP` *integer value*
: Sets an upper limit for the number of multigrid iterations to be performed to solve Pressure Poisson problems.

`MaxIItU` *integer value*
: Sets an upper limit for the number of multigrid iterations to be performed to solve linearised Burgers problems.

`MaxTimeIterations` *integer value*
: Specifies the maximum number of (macro) time steps.
This value is one of the main normal program termination controls. If either the endpoint in time or the maximum number of time steps is reached, the program will terminate. So, be especially careful when setting this value. It will be quite annoying and in most cases even rather expensive in terms of waste of CPU and quota, if your simulation terminates unmeantly because of a poorly chosen maximum number of time iterations.

`Method` *integer value*
: Specifies the time stepping scheme.
`1` will use the Chorin scheme (first order), `2` will invoke Van Kan scheme (second order).
Negative values are possible, too. In this case, the absolute value gives the number of initial time steps using Chorin's method, afterwards the simulation continues with applying Van Kan's scheme. This procedere is especially useful if no restart information is available and a simulation has to be started from scratch. Usually 4-5 initial Chorin steps give a sufficient approximation to continue with Van Kan's scheme.
If a restart is done, the value is ignored and Van Kan's scheme is used.

`MGOmega_burg` *float value*
: Sets the relaxation parameter ω to be used by the smoothing algorithm in multigrid to solve linearised Burgers equations.
See [17] for a detailed study on the influence of this relaxation parameter for different smoothing algorithms. Simplistically speaking, a setting of `0.8` for Jacobi method, `1.3` for SOR method, `0.9` for ILU and `1.3` for CG method should show satisying results in most cases.

`MGOmega_press` *float value*
: Sets the relaxation parameter ω to be used by the smoothing algorithm in multigrid to solve Pressure Poisson equations.
See [17] for a detailed study on the influence of this relaxation parameter for different smoothing algorithms. Simplistically speaking, a setting of `0.8` for Jacobi method, `1.3` for SOR method and `0.9` for ILU should show satisying results in most cases.

`MinFixpItU` *integer value*
: Specifies the minimum number of fixpoint iterations to be performed to solve nonlinear Burgers equation in each time step.

`MinIItP` *integer value*
: Specifies the minimum number of multigrid iterations to be performed to solve Pressure Poisson problems.

`MinIItU` *integer value*
: Specifies the minimum number of multigrid iterations to be performed to solve linearised Burgers problems.

`NFine` *integer value*
: Specifies the highest multigrid level number.
A value of `1` means the coarse grid is **not** refined, `2` means the coarse grid is refined once and so forth. (Note: Each increment will lead to an increase in problem size by a factor of 8.)

`OmgIni` *float value*
: Specifies the initial value for the relaxation parameter ω in nonlinear iteration.
A value of 1.0 should do in most cases.

`OutputBaseDir` *string*
: Specifies the path where visualisation output files are stored. A trailing slash can be omitted.

`PartitionBaseDir` *string*
: Specifies the path where to look for partition information files and where to write them to. A trailing slash can be omitted.

`PartitionCR` *integer value (boolean)*
: Specifies whether partition information should be generated or read from disk.
If set to `0`, partition information will be generated and written to disk

using PartitionBaseDir and PartitionFile.
If set to 1, the value of PartitionTool will be ignored and partition information is read from disk using PartitionBaseDir and PartitionFile.

PartitionFile *string*
Specifies the file name that contains partition information.

PartitionTool *integer value*
Specifies which algorithm to be used for partitioning the coarse grid.
0 for the (non-deterministic) PARTy [14] algorithm,
1 for the (deterministic) METIS [11] algorithm PartGraphRecursive,
2 for the (deterministic) METIS algorithm PartGraphVKway.
The PARTy algorithm relies on a given number of processes that is a power of 2. As does METIS it uses graph theory to distribute coarse grid's elements onto the different processes as uniformly as possible. There is no guaranty, though, that the resulting PARTy partition will be identical each time you invoke the library. Every partition will be valid, but most of the time you will end up with a bunch of different partitions, not a single one. This especially holds for coarse grids with several dozens or even hundreds of elements.
Contrary to this, both algorithms from the METIS library will generate definite partitions if applied under the same conditions.[10] Unlike PARTy both algorithms work for all given (positive) number of processes.
See also section 9.1 and 9.2.

PostSteps_burg *integer value*
Specifies the number of post-smoothing steps used in multigrid to solve linearised Burgers equations.

PostSteps_press *integer value*
Specifies the number of post-smoothing steps used in multigrid to solve Pressure Poisson equations.

PreSteps_burg *integer value*
Specifies the number of pre-smoothing steps used in multigrid to solve linearised Burgers equations.

PreSteps_press *integer value*
Specifies the number of pre-smoothing steps used in multigrid to solve Pressure Poisson equations.

[10]This explains the above terms "deterministic" and "non-deterministic".

ProlType_press *integer value*
Sets the prolongation method in multigrid for solving for Pressure Poisson equations.
1 enables constant prolongation, 2 means linear prolongation. Linear prolongation for a Q_0 ansatz means interpolating the values in the midpoints of the elements of the coarser grid to the vertices, prolongating them (linear) to the finer grid and re-interpolating the new values to the elements' midpoints.

RestartBaseDir *string*
Specifies the path where to look for solution files to use for restart initialisation. A trailing slash can be omitted.

RestartITE *integer value*
If continuing a simulation, it is rather convenient to initialise the iteration counter appropriately by adjusting this value.

RestartSolFile *string*
Specifies the basename of the solution files to use for restart initialisation.
Remark: These files can be arbitrarily exchanged between different platforms. Just ensure that all platforms use the same partition (for instance by reading from the same file specified as PartitionFile). This implicitly means that the same number of processes is used, too.
Example: You have got a set of solution files from a 4–processor–run named c3d0.r3.p000.sol, c3d0.r3.p001.sol, c3d0.r3.p002.sol and c3d0.r3.p003.sol. The value for RestartSolFile would be c3d0.r3
See also section 8.2.

Restart *integer value*
This switch indicates whether you want to start from scratch (0) or with an (approximate) solution from a previous run.
You can continue a simulation with a solution from the same grid refinement level (1) or use a solution that is **one** level coarser (2). In this case, it will automatically be prolongated to the current grid refinement level.

Smoother_burg *integer value*
Selects the smoother to be used in multigrid to solve linearised Burgers equations.
1 means Jacobi method, 2 utilizes SOR method, 3 stands for ILU method, 4 will invoke CG method.

`Smoother_press` *integer value*
: Selects the smoother to be used in multigrid to solve Pressure Poisson equations.
1 means Jacobi method, 2 utilizes SOR method, 3 stands for ILU method.

`SolFileFrequency` *integer value*
: Specifies how many time step iterations have to be done before the next solution file will be written.

`SolFileNumber` *integer value*
: Specifies how many different solutions should be kept.
If `SolFileNumber` is reached, the first set of files will be overwritten. Usually, a number between 1 and 5 should do. Larger values should be chosen very carefully: Having large grid densities you can easily end up with an exceeded disk quota or hard disk capacity.

`SolFilePrefix` *string*
: Specifies the basename for solution files to be written to the directory `RestartBaseDir`.

`SolverMaxIt_burg` *integer value*
: Sets an upper limit for the number of iterations performed by the coarse grid solver used in multigrid to solve linearised Burgers equations.

`SolverMaxIt_press` *integer value*
: Sets an upper limit for the number of iterations performed by the coarse grid solver used in multigrid to solve Pressure Poisson equations.

`SolverType_press` *integer value*
: Controls which solver scheme is used to solve Pressure Poisson equations.
1 means ordinary multigrid method, 2 utilises CG method with exactly one iteration of additive multigrid method as preconditioning step, 3 stands for CG method with multiplicative multigrid preconditioner.
For low degrees of parallelism you will hardly notice significant run time differences from either setting. The performance depends on the mean aspect ratio of your grid as well as the number of processors. Best results are generally achieved with multiplicative preconditioned CG method.

`Solver_burg` *integer value*
: Specifies the solver scheme to be used on the coarse grid when solving linearised Burgers equations with multigrid method.
1 means Jacobi method, 2 will utilise SOR method, 3 stands for ILU method, 4 will invoke CG method with ILU pre-conditioning.

`Solver_press` *integer value*
: Specifies the solver scheme to be used on the coarse grid when solving Pressure Poisson equations.
1 means Jacobi method, 2 will utilise SOR method, 3 stands for ILU method, 4 will invoke CG method with ILU pre-conditioning.

`TEnd` *float value*
: Specifies the endpoint in time.
Conjointly with `EpsNS` and `MaxTimeIterations` this option controls when to end the program.

`TestLoops` *integer value*
: The number of configuration sets within the current file.
Most data processing centers give access to their parallel computing facilities via a queuing mechanism. You have to enqueue your job, specifying the minimum and/or maximum resources your job wil need. Depending on attendant circumstances you will have to wait hours or days before your job is given the permission to run.
By increasing `TestLoops` you will be able to test different configurations within a single enqueued job. This comes in quite handy whenever you are investigating the influence of varying settings for smoothings steps or stop criterions.

`TInitPhase` *float value*
: Specifies the duration of the start phase.
Due to the projection method we apply, the solution of the Navier-Stokes equation gained is not accurate for the very first time steps. A small initialisation phase, a start phase, is needed in order to tune the flow. During this phase, a weaker time error limit can and usually will be used.

`TStepControlITE` *integer value*
: Gives the number of (macro) time steps which have to be performed before adaptove time step control is invoked.

`UpSam` *float value*
: Specifies the parameter α for weighted Samarskij-Upwinding.
The value usually ranges between 0.1 and 2, see page 10 in [5] or [16].

4 Creating a 3D grid file

The geometry and a first coarse triangulation of the domain is prescribed by a grid file.[11] This file contains the coordinates of all coarse grid's (inner and boundary) nodes and the manner in which they are connected to form hexahedrons which triangulate the domain. In case of curved surfaces, though, the grid file will only contain a rough polynomial approximation of this surface (see remark 1 below).

The file format is called *TRI* format and is described in [10, p. 42f] and [8, p. 239ff].

The first version of this manual listed only two possibilities to create a grid: either manually from scratch or with help of a small tool from the FEATFLOW package called *tr2to3*. In the meantime, *Grid3D* has been released [2, 8], our chair's 2D and 3D grid and geometry editor. Check the documentation section of our website `http://www.featflow.de/` to get a copy of the *Grid3D* manual.

Alternatively, you can use GiD [9] to create a 2D or 3D grid and use the tools described in [1] to convert the data into a valid triangulation file. [1] also describes how to convert grid information stored in the popular DXF format to the TRI format. As almost every commercial grid generation package can export data in DXF format it should be possible to create a TRI file. I have to confess that this preprocessing step is still awkward. We hope to improve this process further in future.

In case your problem is symmetric in z-direction you should first create a 2D grid (in PRM+TRI format) and then use *tr2to3*, a small tool from the FEATFLOW package that extrudes your data.[12] In the early years of FEATFLOW this has been the usual way to create a 3D grid file. See [4] or [3] for documentation on the use of *tr2to3*.

Remark 1: Usually, curved surfaces within your domain are merely approximated in a very rough way by the coarse grid triangulation. The grid refinement algorithm uses the idea of bisection and refines a given grid uniformly. Without additional (hard-coded) node adjustments refined grids will not resolve more details of your geometry than the coarse grid.

[11] All programs from the FEATFLOW package, including *parpp3d++* use the same grid format. So, you can reuse the same grid files for *cc3d*, *pp3d* and *parpp3d++*.

[12] I.e. it creates multiple copies of the 2D grid file, arranges them in layers with differing coordinates with respect to the third dimension and triangulates neighbouring layers (see figure 2)

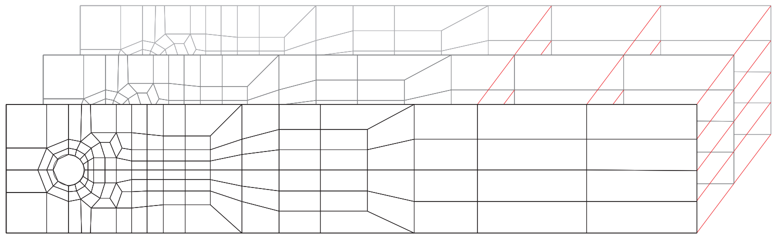

Figure 2: Creating a 3D mesh via extrusion

So, in case of curved surfaces node adjustments or, to state it more precisely, boundary projections are necessary. How to do this is explained in section 6.

Remark 2: Most of the time, a postprocessing step is needed if you have created a 3D grid with *tr2to3*. Most coordinates within the grid are exported by *tr2to3* in floating point notation. Unfortunately, in Fortran – the programming language *tr2to3* is written in – there are two manners to code numbers in floating point notation: with a capital 'E' (for exponent) or 'D' (for exponent with double precision). All sequential programs from the FEATFLOW package are written in Fortran. Thus, they can handle grid files both with 'D' or 'E' syntax. *parpp3d++* is written in C++ and, unfortunately, C++, or more precisely my parser routine, can't handle the 'D' syntax.

This means, if you create a 3D grid with FEATFLOW's *tr2to3*, you will have to open it with your favourite editor and perform a global search and replace operation on all capital 'D' to turn them into 'E' (turning '2.50D0' into '2.50E0').

5 Prescription of inflow and boundary conditions

So far, we have gone through the first four of those six steps listed at the beginning of chapter 3 that are necessary to set up *parpp3d++* for a new simulation. Next on our itinerary is how to prescribe inflow velocities and profiles and how to distinguish between Dirichlet and Neumann boundaries.

5.1 Prescription of inflow

All those forces that propel the flow, whether it be a tangential force at a boundary surface like, for instance, a lid-driven cavity flow or a configuration with a distinct inflow and outflow boundary with a medium running through the domain, are prescribed in method `Solution` of `class ParGrid` in the file `Bound.cc` in the main directory of *parpp3d++*.

The method takes four (floating point number) arguments, the three coordinate values of a (velocity) node as well as the current simulated point in time. Freshly extracted from the distribution tarball, it consists of several dozens of code lines. But this is just because of eight different configurations already been set up. These can be used as a starting point to become familiar with the program and the manner in which inflow is prescribed. In fact, the method consists merely of a single if-clause to distinguish between the three (spatial) components of the inflow profile[13]: inflow in x-, y- and z-direction. The x component is handled first, the y component second and finally the z component. Comment lines within the code clearly indicate each section.

Within each section there is a switch-case-environment to determine inflows for different configurations. It is quite convenient to have, once compiled, a single binary that contains all your recent configurations ready-to-run and be able to switch between them by changing an option in a parameter file (see description of FUNC, page 8). Besides, it happens rather often that the inflow pattern of a new configuration is very similar to one of the already set up inflows and hardly needs more than a copy-and-paste operation.

To add a new configuration insert an additional case statement, calculate the inflow velocity depending on the point in space and time and return this value. Let us assume you want to have inflow in x direction with a mean velocity of 1, a parabolic profile and no inflow in the remaining spatial components. Furthermore, the width and height of the inflow surface (y- and z-direction) is assumed to be 1. Then, the inflow function looks like

$$U(0, y, z, t) = 16yz(1 - y)(1 - z), \qquad V = W = 0$$

The following minimum definition of `ParGrid::Solution` is sufficient:

```
double ParGrid::Solution(double X, double Y, double Z, double T)
{
    if (CoeffRhs == COEFF_RHSV1 || CoeffRhs == COEFF_L2V1) {
        // Prescribe inflow in x direction
        if (X < 1e-8)       // no (X==0) if you can help it!
                            // think of rounding errors
            return 16.0 * Y * (1-Y) * Z * (1-Z);
    }

    return 0.0;
}
```

5.2 Prescription of boundary conditions

Boundary conditions are defined in the same file as inflow conditions and, as with those, the definition is based on coordinates. You can choose to keep the boundary conditions for all of your configurations in a single file, too. Just activate one of them by specifying the appropriate value for FUNC (see page 8) in your parameter file.

So, in your sample file `Bound.cc` from the distribution tarball you will find a method called `SquareGrid_3D::BoundaryCondition`. It takes an unsigned integer as a flag for the boundary conditions to apply (same value as FUNC) as well as four floating point numbers (three coordinates plus point in time) as arguments. Return value is an integer which indicates whether a node (given by its coordinates) belongs for a given point in time to a Dirichlet (1) or Neumann (0) boundary.

Example: Let us start with the simplest possible case: a box of arbitrary width and height, but fixed length of 1. It does not matter here where the actual inflow (sub-)surface is located, just assume that the outflow surface will be the rear end of the box, at x=1. Then, the following definition will do:

[13] Time dependency is treated in each of the spatial branches.

```
int SquareGrid_3D::BoundaryCondition(unsigned int Func, double X,
                                     double Y, double Z, double T)
{
    const int neumann   = 0;
    const int dirichlet = 1;
    double dist = 0.001;

    // avoid '==' statements with floating point coordinates
    if (X > 1.0 - dist)
        return neumann;
    return dirichlet;
}
```

One marginal note on this definition: Any obstacle in the interior of the domain (which, as an obstacle, has obviously Dirichlet boundary condition) is implicitly treated already! You are free, though, to explicitly define boundary conditions for them, too. Have a look at the definition of boundary conditions for the DFG Benchmark 3D-2Z configuration (`Func = 5`), for instance, which consists of a channel flow around a cylinder.

6 Boundary projection

Apart from the creation process of a coarse grid and, sometimes, finding optimal run-time parameters for a configuration[14], boundary projections is the most annoying part of preparing the code for a new simulation. As mentioned earlier, the coarse grid not always already resolves all details of a given geometry. Especially in the case of curved structures within a domain, special precautions have to be taken to guarantee that refined grids approximate a given geometry more thoroughly. This is done by defining an algorithm that projects boundary points (within your grid) to the real boundary (of your domain). To explain the mechanism, let us examine the boundary projections for one of the sample configurations *parpp3d++* ships with: a small device used in chemical engineering to mix different species. The domain consists of a stretched hexagon, extruded in z-direction and additional cylinders at the left and right end. In the interior, nine cylindrical obstacles are to be found (see figure 3). The coarse grid used (`Grids/BMBF_CE/9shifted.round.tri`) approximates all of these curved surfaces polygonally (see figure 4).

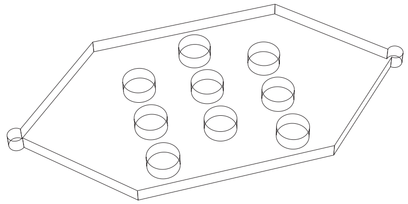

Figure 3: Prototypical device used in chemical engineering

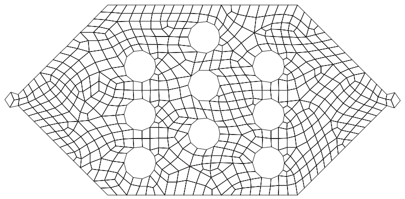

Figure 4: Coarse grid of the geometry (2D cutplane view)

More or less cylindrical structures are created using the definition of the method `Task::BoundaryProjection` listed below which takes no arguments and is called only once for every grid refinement level during the initialising phase of *parpp3d++*. In fact, nothing else is done but defining the midpoint and radius of each cylinder and assuring that each boundary point resides on the edge of its corresponding cylinder. (In this listing, the code lines that actually perform the projection have been omitted for the inner cylinders 2 to 9. They are identical to the projection used for the upper cylinder in the first column.)

[14] In the meaning of resulting in fastest run times.

There are two technical difficulties with this example: Firstly, there is only one boundary surface. So, it's more difficult to identify the cylinders than in 2D. In 2D each would have a different boundary component number. The cylinders must be discriminated to prevent that a boundary point accidentally gets moved to a neighbouring cylinder. So, each cylinder is initially coarsely identified. This is done by selecting boundary points only (first if statement) and enclosing each cylinder in a slightly larger one (subsequent if statements[15]).

The approximation of the cylinder on the coarse grid is contained inside the real cylinder. So, refined boundary points have to be moved outwards to the real boundary if at all. If the distance of a boundary point to the cylinder hull is less than the radius, it has to be moved. This is done in the nested if statements, too.

The second technical issue mentioned above is that this procedure bears difficulties with those boundary points at the intersection of the cylinders with the top and bottom surface. The surrounding cylinders used to identify boundary points belonging to a single cylinder catch inner and boundary points in the interior of the domain. On the top and bottom surface, however, all points are boundary points. So, we have to use another strategy to distinguish between those that have to be moved and those that do not. The solution is to look up the third entry in the list of elements containing that boundary point (`(*ElemVert)(3,IVT)`). This entry is zero if less than three elements meet in that boundary point. This approach, contrariwise, does not work in the interior of the boundary as in the interior in every vertex there is at least one neighbouring element. That explains why you find conditions like

```
if ( fabs(PZ - 0) >  1e-3 && fabs(PZ - 0.1) >  1e-3  && [...]
if ((fabs(PZ - 0) <= 1e-3 || fabs(PZ - 0.1) <= 1e-3) && [...] IEL == 0)
```

in the listing below which might appear confusing at first sight.

[15]We use surrounding cylinders with a radius increased by a factor of 1.5. This factor is chosen rather arbitrarily, it is merely important that it is larger than 1 and not to large to not interfere with a neighbouring cylinder. The increased radius serves as security margin to prevent that rounding errors make us miss a boundary point.

```
void Task::BoundaryProjection()
{
    int    INPR, IEL;
    double PX, PY, PZ, PXM, PYM, RAD;
    double DISTXY = 0;

    for (int IVT=1; IVT <= NumVertices; IVT++) {
        INPR=(*InfoVertEdge)(IVT);
        //  0: inner point
        // >0: point on boundary component with this very number

        if (INPR != 0) {
            // For all boundary points ...
            IEL = (*ElemVert)(3,IVT);
            PX  = (*VertCoord)(1,IVT);
            PY  = (*VertCoord)(2,IVT);
            PZ  = (*VertCoord)(3,IVT);

            // left cylindrical inflow area
            PXM =-2.952;
            PYM = 2.952;
            RAD = 0.25;
            DISTXY = sqrt( pow(PX-PXM, 2) + pow(PY-PYM, 2) );
            if (fabs(PZ - 0) >  1e-3     &&  fabs(PZ - 0.1)   >  1e-3  &&
                         PX <= -2.77522  &&  fabs(PY - 2.952) <= 0.25  &&
                     DISTXY <= RAD) {
                (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
                (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
            }

            if ((fabs(PZ - 0) <= 1e-3     ||  fabs(PZ - 0.1) <= 1e-3)   &&
                          PX <= -2.77522  &&  fabs(PY - 2.952) <= 0.25  &&
                      DISTXY <= RAD       &&  IEL == 0) {
                (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
                (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
            }

            // Upper hole, 1st column
            PXM = 1.0;
            PYM = 4.0;
            RAD = 0.5;
            DISTXY = sqrt( pow(PX-PXM, 2) + pow(PY-PYM, 2) );
            if (fabs(PZ - 0)   >  1e-3       &&  fabs(PZ - 0.1) >  1e-3       &&
                fabs(PX - PXM) <= 1.5 * RAD  &&  fabs(PY - PYM) <= 1.5 * RAD  &&
                        DISTXY <= RAD) {
                    (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
                    (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
            }

            if ((fabs(PZ - 0)   <= 1e-3       ||  fabs(PZ - 0.1) <= 1e-3)      &&
```

```
     fabs(PX - PXM) <= 1.5 * RAD  &&  fabs(PY - PYM) <= 1.5 * RAD  &&
             DISTXY <= RAD        &&  IEL == 0) {
        (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
        (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
}

// mid hole, 1st column
PXM = 1.0;
PYM = 2.5;
RAD = 0.5;
[...]

// lower hole, 1st column
PXM = 1.0;
PYM = 1.0;
RAD = 0.5;
[...]

// Upper hole, 2nd column
PXM = 3.0;
PYM = 4.904;
RAD = 0.5;
[...]

// mid hole, 2nd column
PXM = 3.0;
PYM = 3.404;
RAD = 0.5;
[...]

// lower hole, 2nd column
PXM = 3.0;
PYM = 1.904;
RAD = 0.5;
[...]

// Upper hole, 3rd column
PXM = 5.0;
PYM = 4.0;
RAD = 0.5;
[...]

// mid hole, 3rd column
PXM = 5.0;
PYM = 2.5;
RAD = 0.5;
[...]

// lower hole, 3rd column
PXM = 5.0;
PYM = 1.0;
RAD = 0.5;
[...]

// right cylindrical outflow area
PXM = 8.856;
PYM = 2.952;
RAD = 0.25;
DISTXY = sqrt( pow(PX-PXM, 2) + pow(PY-PYM, 2) );
if (fabs(PZ - 0) >  1e-3      &&  fabs(PZ - 0.1)   >  1e-3  &&
              PX >= 8.67922   &&  fabs(PY - 2.952) <= 0.25  &&
          DISTXY <= RAD) {
    (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
    (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
}

if ((fabs(PZ - 0) <= 1e-3      ||  fabs(PZ - 0.1) <= 1e-3)   &&
               PX >= 8.67922   &&  fabs(PY - 2.952) <= 0.25  &&
           DISTXY <= RAD       &&  IEL == 0) {
    (*VertCoord)(1, IVT) = PXM + RAD/DISTXY * (PX-PXM);
    (*VertCoord)(2, IVT) = PYM + RAD/DISTXY * (PY-PYM);
}
    } // end if INPR condition
  } // end for IVT-loop
  return;
}
```

7 Scattered code changes

Chapter 5 and 6 dealt with changes to the code that are mandatory for every new configuration. Let us now have a closer look at more voluntary exercises.

7.1 Changes to parser routine for parameter files

The file `CProcessApp.cc` in the main directory of *parpp3d++* contains a method called `CProcessApp::ReadData`. This method is a parser routine for parameter files. If you add a new configuration, you should add a new `case` statement within the first `switch` environment. For the user's convenience, the number this configuration has been assigned to (see option FUNC, page 8) and that is used internally by *parpp3d++* can be mapped here to whatever descriptive text you like. For an example investigate the line starting with ¨type of hand-coded simulation¨ in the head section of the sample output file of *parpp3d++* in figure 5.

7.2 Different solver routines for mixed and pure boundary conditions

In most cases, there exists a distinct inflow and an outflow area within your geometry. So, the emerging velocity problem has mixed boundary conditions. This is the normal case. If there is no flow through the geometry, however, the velocity problem bears pure Dirichlet boundary conditions (a driven cavity for instance, pre-set up configuration no. 3). The pressure is not uniquely defined, then – only up to a constant. In order to come to a properly defined problem in such a case, the mean value of the pressure at the outflow boundary is set to zero.

A simple switch statement in the methods `Task::Chorin`, `Task::Fractional` and `Task::CrankNicolson` (file `TimeStep.cc`) selects the solver routines depending on the configuration number FUNC. Search the code for the lines

```
switch(Param->Func)
```

and add a new `case` for your configuration. If your velocity problem has mixed boundary conditions, add it to the first bunch of case statements. They all use one of the solver methods `ParCG`, `ProjMultiDriver` or `ParProjPrecondCG`. If your velocity problem is a pure Dirichlet problem and, thus, your Pressure Poisson problem a pure Neumann problem, add your case to the second group using one of the solver methods `ProjNeumannMultiDriver` or `ProjNeumannMultiDriver`.

7.3 Calculation of drag and lift values

There may be cases where there are some obstacles within the geometry and you might want to calculate drag and lift values for them. This has already been done with *parpp3d++* before, so you do not have to start from scratch for getting this feature.

The procedure is as follows: Edit the method `Task::SetDragLiftInfo` in `Bound.cc` and pinpoint those faces that make up the obstacle. As with inflow prescription and boundary projections a coordinate-oriented approach is used. Three configurations are already set up. Each of these uses a minimal hexahedral box surrounding the obstacle. The computed solution for those boundary faces that lay inside this box will be drawn on to calculate drag and lift values in each time step.

If only drag and lift values for one obstacle are of interest, all that is left to do is printing the calculated values. Modify the section dealing with the printing of drag and lift values in method `Task::DiscreteProjection`, to be found in the file `TimeStep.cc`. Search the code for a line like

```
// Compute drag and lift for some of the
// hard-coded simulations
if (Param->Func == 2 || Param->Func == 4 || [...]
```

If there is more than one obstacle and you want drag and lift values to be calculated for all of them, you need to introduce new data structures (of the same type as `MDragBound[MaxLevel]` and `MElemDragBound[MaxLevel]` and an additional counter (`NumDragBound`) for every obstacle. Scrutinize the code lines for the channel configuration with two consecutive cylinders in method `Task::SetCoeffInfo` in file `Bound.cc` and use them as a guideline.

8 Output files

8.1 Statistical output files

During a normal program run quite a few output files are generated. There will be at least a set of files containing statistics about the calculations being perfomed. Each of the parallel processes has its own output file. Their file names are created according to the following rules: They will consist of the program's file name followed by a `".out.p###"`. Herein, the `'#'` will be replaced by the process numbers, possibly prepended by zeros. So, a 4-node-run with *parpp3d++* will lead to output files named `parpp3d++.out.p000`, `parpp3d++.out.p001`, `parpp3d++.out.p002`, `parpp3d++.out.p003`. Each of them will list the options from the parameter file. Further, each will contain the number of elements, vertices and faces on each grid refinement level the corresponding process has been assigned by the partitioning algorithm. Progress and timing statistics about the assemblation of all matrices needed are included, too. As soon as the initialisation phase has been completed, all but the first process cease printing statistics. Only `parpp3d++.out.p000` shows from this point on information on the program's progress. Figure 5 contains a sample listing:

```
--------------------------------------------------------------------------------
        SIMULATION No.1
--------------------------------------------------------------------------------
        INPUT DATA SECTION
--------------------------------------------------------------------------------
 number of parallel processes          : 4
 type of hard-coded simulation         : project chemical engineering
 coarse grid file                      : Grids/BMBF_CE/9shifted.round.tri
--------------------------------------------------------------------------------
 base directory for restart files      : ./comp/
 write restart file every              : 10 iteration(s)
 number of restart files to keep       : 3
 prefix restart files                  : 9shifted.round.lev3
 partition information is              : created by 3rd party library <party>
 base directory for partition files    : ./comp/
 partition file                        : 9shifted.round.procs4
--------------------------------------------------------------------------------
 viscosity parameter nu, given in 1/nu: 100.0
 maximum mg-level                      : 3
 element type                          : non-parametric, mean value,
                                         rotated tri-linear finite element
 boundary condition                    : 1
 cubature formula for matrix assembl.  : 7
 cubature formula for right hand side  : 7
 enable mass matrix lumping            : yes
 method of stabilisation               : upwind
 samarski upwind parameter             : 1.00
 parameter for velocity computation
 - min. number of fixpoint iterations  : 1
 - max. number of fixpoint iterations  : 2
 - value for opt. omega                : 1.0
```

(continued on next page)

Figure 5: Sample output file of *parpp3d++*

```
 - min. number of multigrid/cg steps   : 1
 - max. number of multigrid/cg steps   : 5
 - limit for changes                   : 1.00e-01
 - limit for defects                   : 1.00e-06
 - limit for defect improvement        : 1.00e-01
 - lower limit for opt. alpha          :-1.0
 - upper limit for opt. alpha          : 1.0
 - number of pre-smoothing steps       : 2
 - number of post-smoothing steps      : 2
 - smoother used                       : ilu
 - omega for smoother                  : 8.00e-01
 - coarse grid solver used             : cg
 - max. iterations coarse grid solver  : 250
 - multigrid cycle used                : f-cycle
 parameter for pressure computation
 - min. number of multigrid/cg steps   : 2
 - max. number of multigrid/cg steps   : 25
 - limit for changes                   : 1.00e+10
 - limit for divergence of velocity    : 1.00e-10
 - limit for defect improvement        : 1.00e-01
 - lower limit for opt. alpha          : 0.0
 - upper limit for opt. alpha          : 1.0
 - number of pre-smoothing steps       : 8
 - number of post-smoothing steps      : 8
 - smoother used                       : ilu
 - omega for smoother                  : 8.00e-01
 - solver scheme used                  : cg method preconditioned with one
                                         multiplicative multigrid step
 - coarse grid solver used             : cg
 - max. iterations coarse grid solver  : 150
 - multigrid cycle used                : f-cycle
 - pressure prolongation is performed  : linear
--------------------------------------------------------------------------------
 projection scheme used                : Van Kan with 4 initial Chorin steps
 number of time iterations             : 1
 simulation length                     : 25.00 sec
 lower limit for time derivate         : 2.00e-04
 time step to start with               : 8.33e-04 sec
 iterations between time step control  : 1
 minimum time step                     : 1.00e-06 sec
 maximum time step                     : 1.00e+00 sec
 duration of starting time             : 0.10 sec
 accuracy for acceptance in start      : 1.00e-03
 accuracy for acceptance after start   : 1.00e-04
--------------------------------------------------------------------------------
 base directory for output files       : ./postprocess/
 write solution in avs format          : no
 write solution in gmv format          : no


--------------------------------------------------------------------------------
        OUTPUT DATA SECTION
--------------------------------------------------------------------------------
 Reading grid file from disk. Done.
 Computing partition information. Done.
 Writing partition information to disk. Done.

 Information on the grid process no. 0 uses in multi grid:

    level | #elements | #vertices | #faces
   -------+-----------+-----------+---------
       1  |      188  |      360  |     708
       2  |     1504  |     2135  |    5088
       3  |    12032  |    14445  |   38400

 Time needed for grid refining:   0h 00m 01.6s
 Time needed for pure refining:   0h 00m 00.6s

 Assembling laplace matrix. Done.
 Assembling mass matrix. Done.
 Calculating projection matrix. Done.

 Time needed for assembling matrices: 8.92576s
```

(continued on next page)

Figure 6: Sample output file of *parpp3d++* (continued)

```
Calculating anisotropy degree:

   level |   ar_max    |   sv_max    |   kv_max          ar_mean    |   sv_mean   |   kv_mean
  -------+------------+------------+------------     ------------+------------+-----------
     1   |  1.39e+01  |  3.04e+00  |  3.04e+00         7.77e+00  |  1.21e+00  |  1.11e+00
     2   |  1.41e+01  |  1.91e+00  |  1.91e+00         7.84e+00  |  1.09e+00  |  1.05e+00
     3   |  1.44e+01  |  1.41e+00  |  1.41e+00         7.86e+00  |  1.04e+00  |  1.02e+00

Calculating anisotropy variation (while calculating volumes):

   level |   av_max    |   av_min    |  av_mean
  -------+------------+------------+------------
     1   |  3.15e+00  |  1.00e+00  |  1.31e+00
     2   |  3.09e+00  |  1.00e+00  |  1.16e+00
     3   |  3.06e+00  |  1.00e+00  |  1.08e+00

Creating structures containing information about
Dirichlet, Neumann and artificial boundaries. Done.

Degrees of freedom for
* burgers equation              :    451488
* pressure poisson equation     :     47360
Total number of unknowns in space:    498848

Elapsed computing time:   0h 00m 15.9s


Computing progress:
* Time iteration no. 1 at 0.000833333 sec:
  Performing step with Chorin method.
  Current macro time step:  8.3333e-04
       |      relative change        |     defect in velocity      |   conv.rate in multigrid    | mg-steps | nonlinear
   it. |    u1       u2       u3     |    u1       u2       u3     |    u1       u2       u3     | u1 u2 u3 | conv.rate
  -----+-----------------------------+-----------------------------+-----------------------------+----------+----------
       |                             | 0.00e+00 0.00e+00 2.64e-06 |                             |          |
    1  | 0.00e+00 0.00e+00 3.53e-01 | 0.00e+00 0.00e+00 2.46e-07 | 0.00e+00 0.00e+00 1.03e-04 |  1  1  1 |  9.31e-02
  (Stop criterion for burgers equation has been fulfilled.)

  Time needed for solving burgers equation:   0h 00m 17.4s

  Projection step using cg method preconditioned with 1 multiplicative multigrid step:

   divergence (l2) | conv.rate in mg | mg-steps
  -----------------+-----------------+----------
       5.53e-11    |     4.01e-01    |     16
  (Stop criterion for pressure poisson equation has been fulfilled.)

  Time needed for solving pressure poisson equation:   0h 00m 13.4s
  Time needed for current time step:   0h 00m 30.9s

  Time needed for current macro time step:   0h 00m 30.9s

  Time derivate of u (l2norm, problem size scaled):  4.6310e+02

  Writing solution (in raw format for restart purposes) to disk. Done.

  Elapsed simulated time:   0h 00m 00.0s  ( 8.3333e-04s)
  Elapsed computing time:   0h 00m 47.2s



-------------------------------------------------------------------------------------------
  Time statistics:
 ==================

Computational time so far:   0h 00m 47.2s

Time needed to solve the non-linear momentum and the pressure poisson problem:

                           |    cpu time     | % of time solving equations | % of total time
---------------------------+----------------+------------------------------+-----------------
momentum equations         |   0h 00m 17.4s |            56.53 %           |      36.92 %
pressure poisson equations |   0h 00m 13.4s |            43.47 %           |      28.40 %
```

(continued on next page)

Figure 7: Sample output file of *parpp3d++* (continued)

```
Time distribution during this computation:

                           |   total time    | comm. time
---------------------------+----------------+-----------
grid refining              |   0h 00m 00.6s |  0.00 %
matrix assemblation        |   0h 00m 22.2s |  0.00 %
equation solving           |   0h 00m 17.1s |    n.a.
other (i/o etc.)           |   0h 00m 07.1s |
 - - - - - - - - - - - - - + - - - - - -+- - - - - - -
matrix-vector multipl.     |   0h 00m 09.6s |  76.04 %
computing norms            |   0h 00m 00.5s |  22.36 %
 - - - - - - - - - - - - - + - - - - - -+- - - - - - -
smoothing                  |   0h 00m 11.1s |  19.32 %
computing defects          |   0h 00m 02.0s |  66.31 %
restriction                |   0h 00m 00.0s |  47.99 %
prolongation               |   0h 00m 00.2s |  6.37 %
solve coarse problem       |   0h 00m 00.0s |  17.57 %

-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
         SIMULATION No.1 COMPLETED
-------------------------------------------------------------------------------
```

Figure 8: Sample output file of *parpp3d++* (continued)

8.2 Restart solution files

At the end of each program run *parpp3d++* automatically stores restart information on your hard disk. Additionally, every few time steps intermediate restart information is saved as well. You can control this frequency be setting SOLFILEFREQUENCY accordingly.

Each process creates a file where that part of the velocity and pressure solution[16] is stored that resides in the memory space of this process. That means, **restart information can only be successfully used** (for a continuation on the same grid refinement level or a restart on a one level finer grid) **if the same partitioning is guaranteed!**[17]

The restart solution file names are created according to the following rule: They will consist of the string given as SOLFILEPREFIX in the parameter file followed by a `".r#.p###.sol"`. The first `'#'` represents the counter for restart solution files. This counter is incremented every SOLFILEFREQUENCY (macro) time steps, modulo the value given as SOLFILENUMBER.

The trailing three `'#'` are replaced by the process number, possibly prepended by zeros – as with the statistical output files.

[16]Plus point in time and current time step information.

[17]This is the reason why partitioning information is always stored to a file. It has to be stated here, though, that partitions for a fixed number of processes are always identical, if created by the third party library METIS. This, however, holds not for PARTy partitions. The algorithm seems not deterministic, resulting in more or less different partitions at each invokation.

If you want to actually restart from a set of restart solution files – let us assume they are named `restart.lev2.r2.p###.sol` – then just set RESTARTSOLFILE to `restart.lev2.r2`. The program will determine the correct names for each process.

8.3 Visualisation output files

Finally, *parpp3d++* can produce output to be used in a postprocessing step as input data for a visualisation program. *parpp3d++* ships with standard support for the data formats of AVS [15] and GMV [13]. Considering the huge amounts of disk space that can easily be consumed by exporting visualisation data, *parpp3d++* does not export any by default. Unless AVSOUTPUTLEVEL or GMVOUTPUTLEVEL is set to a value greater than zero, no output file in AVS or GMV format, respectively, is created.

If you have opted for any of these two formats (by setting AVSOUTPUTLEVEL or GMVOUTPUTLEVEL to a positive integer), a set of output files is created every DTAVS or DTGMV seconds, respectively. This time difference, however, is not stricty kept. It is just a lower limit for the creation of visualisation output files. The reason is quite simple: A typical time-depending 3D flow simulation requires (at a typical problem size of 10^6 - 10^8 unknowns) at least a hundred, if not several hundred time steps. To not waste any resources, time steps are chosen adaptively and are increased to the limit while maintaining numerical accuracy.[18] As soon as the simulated time exceeds the time difference limit stated by DTAVS or DTGMV, respectively, the next set of visualisation output files is generated.

What does this mean, a set of visualisation output files? Is there no single visualisation output file for each point in time? No, there is not! On most supercomputers, computing time is limited (mostly, to several hours of continously running a program). Transferring all data to a single process and subsequently exporting it to a single file (in each time step) would leave most of the processes idle for an unreasonable amount of time. Instead each process writes his part of the solution to the (local) hard disk – as with statistics and restart solution files.

The merging of the visualisation output files is done in a postprocessing step. Two command line tools are available that will do the job: `mergeavsfiles` and `mergegmvfiles`. They can be downloaded from the same website as *parpp3d++* itself at `http://www.featflow.de/`. Both are quite simple programs that only require a C++ compiler with STL support – very much the same requirements as for *parpp3d++*. You should be able to compile them with minor changes to the Makefile, if any.

[18]In order to accomplish this, implicit time stepping schemes are used. These allow larger time steps than explicit schemes at the same numerical accuracy. (The price are systems of (non-)linear equations which are more difficult to solve.)

The syntax of both program's invocation is identical. The first argument is the base name of a set of visualisation output files. If you have a set of files named `unitcube.t###.p###.avs` just specify `unitcube`.

As second argument they take the starting t-number of your sequence, i.e. the number behind the `.t` and before `.p` of a valid set of files. In most cases this will be 1.[19]

The third argument is the number of different time steps and the last argument tells the number of processes you have used, i.e. the number of fragments an output file consists of after a program run of *parpp3d++*.

Example: You have done a 4-node-run which ended up in visualisation output files at 10 different times:

`unitcube.t001.p000.gmv,`	...	unitcube.t001.p003.gmv,
⋮		⋮
`unitcube.t010.p000.gmv,`	...	unitcube.t010.p003.gmv

To merge these files, invoke the following:

```
% mergegmvfiles unitcube 1 10 4
```

A sequence of files called `unitcube.t001.gmv, ..., unitcube.t010.gmv` will be generated. The input files can then be safely removed.
(Visualisation output files in AVS format are treated in the same way.)

[19]You can start at an arbitrary number of your sequence. Even traversing your sequence reversely is possible.

9 Known bugs

9.1 Not simply connected partitions

parpp3d++ needs simply connected partitions, i.e. if you select two arbitrary elements that have been assigned to the same process, there must always exist a sequence of face-neighbouring elements in between that all reside on the same process.

If the degree of parallelism is very high or, alternatively, the coarse grid consists only of very few elements such that the number of elements that is assigned to each process adds up to 5–6 or less, then there is a (moderate) risk that METIS [11] partitions will **not** be **simply connected**. You will notice some weird convergence problems when *parpp3d++* tries to solve the Burgers problem of the first time step or possibly a dead lock of the program. Use a PARTy [14] partition instead in these cases.

9.2 More cases of inappropriate partitions

Closely connected is another problem with some partitions. Parallel jobs with more than 64 processes may show similar convergence problems when solving Burgers equations. Possibly, you will not encounter any problems within the first few time steps, but, suddenly, (in fact with slightly increasing time step) the linearised subproblems of Burgers type will diverge. In these cases, the coarse grid features too high aspect ratios. It will depend on the partition (mainly the degree of decomposition, i.e. the number of parallel processes, but even on the specific manner of dividing the coarse grid into parallel blocks of elements) whether the core components of the solver engine, namely multi grid's smoothing algorithm and the coarse grid's problem solver, will or will not be able to handle the high aspect ratios. Use better shaped elements instead or experiment with different partitions and varying degree of parallelism in these cases.

9.3 Disadvantage of triangulations with high aspect ratios

Grids with a high mean aspect ratio[20] have additional disadvantages. Not only is there a chance that the partitioning libraries PARTy and METIS will return partitions that are inappropriate for *parpp3d++* (see previous section). Moreover, the multigrid methods used to solve the differing subproblems of high dimensional systems of linear equations highly depend on the ¨smoothing property¨ of the iterative solvers that are applied internally. These solvers of blocksolving type do not appreciate high aspect ratios. With increasing degree of parallelism the iteration counter with deflect more and more before a given accuracy for the solution is reached. Especially the solver engine for Pressure Poisson equations reacts very sensitively on an increasing amount of ¨distorted¨ elements [5]. Stepping from one to 64 processes, for instance, can lead to 5-10 times more mean iteration steps solving the Pressure Poisson equations. Because this part of the simulation accounts for at least half of overall run times, this effect has a significant influence on run times. The numerical deterioration of the solving algorithm with increasing number of parallel processes gives unpleasant parallel efficiencies and gets worse the higher mean aspect ratios are.

So, if possible avoid grids with mean aspect ratios higher than 15–20.

[20] The ratio of element height to width or breadth, respectively.

10 Migrating from *pp3d*

10.1 Equivalent options in parameter files

For those users who are familiar with the sequential programs from the FEATFLOW package and the abbreviated keywords used in their parameter files we supply the (conversion) table 2.

pp3d	*parpp3d++*
AMAXP	AMaxP
AMAXU	AMaxU
AMINP	AMinP
AMINU	AMinU
CMESH	GridFile
CSTART	RestartSolFile
DMPPMG	DampPMG
DMPUD	DampUMG
DTAVS	DtAVS
DTGMV	DtGMV
DTMAX	DtMax
DTMIN	DtMin
EPSADI	EPSADI
EPSADL	EPSADL
EPSNS	EpsNS
EPSP	EpsDivergence
EPSUD	EpsUDefect
EPSUR	EpsUChange
IAVS	AVSOutputLevel
ICYCP	Cycle_press
ICYCU	Cycle_burg
IGMV	GMVOutputLevel
ILMAXP	MaxIItP
ILMAXU	MaxIItU
ILMINP	MinIItP
ILMINU	MinIItU
IMASS	Lump
INLMAX	MaxFixpItU
INLMIN	MinFixpItU
INSAV	SolFileFrequency
INSAVN	SolFileNumber
ISLP	Solver_press
ISLU	Solver_burg
ISMP	Smoother_press
ISMU	Smoother_burg
NITNS	MaxTimeIterations
NLMAX	NFine
NSLP	SolverMaxIt_press
NSLU	SolverMaxIt_burg
NSMP	PreSteps_press / PostSteps_press
NSMU	PreSteps_burg / PostSteps_burg
OMGINI	OmgIni
RE	EpsEqu
RLXSMP	MGOmega_press
RLXSMU	MGOmega_burg
TIMEIN	TInitPhase
TIMEMX	TEnd
TSTEP	DtStart
UPSAM	UpSam

continued on next page

Table 2: Equivalent options in parameter files of *pp3d* and *parpp3d++*

Remark: The relaxation parameters for the iterative solvers used when dealing with the coarse grid problem (RLXSLU and RLXSLP in FEATFLOW syntax) are hard-coded within the methods CCoarseGrid::SolveExact and CCoarseGrid::SolveConstExact, respectively.

10.2 Changes to coarse grid file

pp3d and *parpp3d++* can handle the same coarse grid files - with one exeption: If any floating point number within a coarse grid file is given in floating point notation, make sure you use only an 'E' for the exponent. Fortran can handle a 'D' as well (which stands for double precision), C++ or more precisely my parser routine can't (as already stated in remark 2 on page 12). So instead of '2.50D0', please use '2.50E0'.

This means, if you create a 3D grid with FEATFLOW's *tr2to3*, you will have to open it with your favourite editor and perform a global search and replace operation on all capital 'D' to turn them into 'E'.

10.3 indat3d.f and parq3d.f

In *pp3d* inflow and boundary conditions are prescribed in the files `indat3d.f` and `parq3d.f`. As explained in chapter 5 and 6 these necessary code adjustments have been merged within the file `Bound.cc` in *parpp3d++*.

References

[1] J. F. Acker. Effiziente Realisierung von hierarchischen Pre- und Postprocessingmethoden in der mathematischen Strömungssimulation. Master's thesis, Universität Dortmund, May 2003.

[2] Ch. Becker and D. Göddeke. Devisor grid. Technical report, Universität Dortmund, `http://www.featflow.de/feast_hp/devisormain.html`, 2002.

[3] Ch. Becker and S. Turek. Featflow – finite element software for the incompressible Navier–Stokes equations. User manual, Universität Dortmund, 1999.

[4] S. H. M. Buijssen. 3d grid generation. Handout FeatFlow Springschool 2002, March 2002.

[5] S. H. M. Buijssen. Numerische Analyse eines parallelen 3-D-Navier-Stokes-Lösers. Master's thesis, Universität Heidelberg, October 2002.

[6] S. H. M. Buijssen, M. Grajewski, S. Turek, and H. Wobker. High performance FEM simulation. Research report, NRW Graduate School of Production Engineering and Logistics, Universität Dortmund, Leonhard-Euler-Str. 5, 44221 Dortmund, September 2004. p.52–55.

[7] S. H. M. Buijssen, H. Wobker, and S. Turek. High performance FEM simulation in CFD and CSM. Research report, NRW Graduate School of Production Engineering and Logistics, Universität Dortmund, Leonhard-Euler-Str. 5, 44221 Dortmund, August 2005. p.46–49.

[8] PG DeViSoR. Endbericht der Projektgruppe DeViSoR. Ergebnisberichte des Instituts für Angewandte Mathematik, Nr. 240t, FB Mathematik, Universität Dortmund, 2003.

[9] International Center for Numerical Methods in Engineering. GiD – the personal pre and post processor. `http://gid.cimne.upc.es`, 2003.

[10] P. Harig, P. Schreiber, and S. Turek. Feat3d – finite element analysis tools in 3 dimensions user manual. release 1.2. Preprints SFB 359, Nr. 94-19, Universität Heidelberg, March 1994.

[11] G. Karypis and V. Kumar. METIS – a software package for partitioning unstructured graphs, partitioning meshes, and computing fill–reducing

orderings of sparse matrices. Technical report, University of Minnesota, Department of Computer Science, 1998. `http://www-users.cs.umn.edu/~karypis/metis/metis/index.html`.

[12] LAM Team. MPI primer / developing with LAM, 1996. www.lam-mpi.org.

[13] F. A. Ortega. *The General Mesh Viewer Version 3.2*. Los Alamos National Laboratory, 2003. `http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html`.

[14] R. Preis and R. Diekmann. The party partitioning library, user guide — version 1.1. Technical report, Department of Mathematics and Computer Science, Universität Paderborn, September 1996. `http://wwwcs.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html`.

[15] Advanced Visual Systems. AVS/Express. `http://www.avs.com`, 2003.

[16] L. Tobiska. Full and weighted upwind finite element methods. In J. W. Schmidt and H. Späth, editors, *Splines in Numerical Analysis*, 1989. Internationales Seminar ISAM 1989 in Weissig.

[17] S. Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*. Springer, Berlin, 1999.

Index