Parallel solution of systems of linear equations generated by COMSOL 3.2 using the Sun Performance Library

Gersborg, Allan Roulund; Dammann, Bernd; Aage, Niels; Poulsen, Thomas Harpsøe

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

Parallel solution of systems of linear equations generated by COMSOL 3.2 using the Sun Performance Library

Allan Gersborg-Hansen, Department of Mathematics
Technical University of Denmark,
DK-2800 Lyngby, Denmark

Bernd Dammann, Informatics and Mathematical Modelling
Technical University of Denmark,
DK-2800 Lyngby, Denmark

Niels Aage and Thomas Harpsøe Poulsen
M.Sc. students at the Technical University of Denmark,
DK-2800 Lyngby, Denmark

Correspondence to: agh@mek.dtu.dk

March 10, 2006

Mat-Report No. 2006-05
ISSN: 0904-7611

1 Abstract

This note investigates the use of the Sun Performance Library for parallel solution of a system of linear equations generated by COMSOL 3.2. In many engineering disciplines this is a computational bottleneck for large problems which are often met in research practice. Most researches are primarily concerned with developing a proper (COMSOL) model rather than developing efficient linear algebra solvers which motivates this investigation of the efficiency of the coupling COMSOL + SPL. The technicalities of making such a coupling is described in detail along with a measure of the speedup for a testproblem run in 2D and 3D.

Moreover this note quantifies the performance of COMSOL running on a Sparc ULTRA III processor. The study shows that for small problems such as debugging tasks, teaching exercises etc. the Sun computer is not competitive compared with a standard PC.

2 Introduction

This note is written as a brief primer to the Sun computer at Technical University of Denmark (DTU) with focus on using multiple processors in the parallelized Sun Performance Library (SPL). The SUN computer is a high performance computer very different from a standard PC. It has several processors, lots of memory and is capable of running large jobs in 64-bit. The primer is intended for users that need to solve large linear algebra problems possibly generated by MATLAB and/or COMSOL 3.2.

Getting the linking from a JAVA environment like MATLAB to the SPL right is non-trivial and receives the primary focus. Moreover, this note illustrates how a program can be profiled to identify bottlenecks and it proves the principle that multiple processors can be used in the SPL via a call from a FORTRAN or a MATLAB programme running COMSOL.

To measure the performance of the SPL we consider a Poisson test problem and solve it in two and three spatial dimensions. A comparison with the solver used by COMSOL 3.2 shows that a speed up of a factor 2.4 can be achived for a 2D prolem and a factor 3.4 for a 3D problem, if several processors are used in the SPL. If only a single processor is used the SPL is as effective as the solver used in COMSOL 3.2.

The report also addresses the concept of writing an appropriate makefile for compiling a programme written in FORTRAN being a low level language. It turns out to be very critical for the performance that the correct compiler options are used. Links to the SPL and options for using 64-bit are also described.

The files presented in this report can be found and downloaded from the home page at:
http://www2.mat.dtu.dk/people/A.G.Hansen/

3 Compiling a FORTRAN programme

3.1 Script files and the shell environment

To streamline the presentation we use script files to show the necessary commands. A script file is created by making a text file in any editor and changing its filestatus to an executable. Example: `chmod +x scriptfile` changes the file `scriptfile` such that it can be executed. To execute the script file one types its name in the unix prompt. Scriptfiles need not have a file extension.

The command prompt may be flavored to the match the taste of the user. One can use the `bash`, `c`, `k` shell etc. In this note we use the `bash` shell. You change to the bash shell by writing `bash` in the prompt.

Consider the below script file

Listing 1: script_1.txt

```
# Comment line , show content of directory
ls

# Assign value to system variable
export OMP_NUM_THREADS=4

# Check the value of a system variable (useful for debugging)
echo 'number of threads is:' $OMP_NUM_THREADS
```

The system variable `OMP_NUM_THREADS` determines the number of processors to use following the OpenMP standard. In this way it is very simple to control parallelized libraries. Other OpenMP options exist such as `OMP_SCHEDULE=guided`. See the OpenMP literature for more details, e.g., [1, 2].

3.2 Compiling

We wish to compile the FORTRAN programme contained in the files `solver.f` and `call_solver.f` given in Appendix A. It turns out that the Sun Studio 10 compiler has an error that prevents the use of multiple processors, but using the Sun Studio 8 compiler things work fine. Since the Sun computer has a shared memory architecture the parallelization is based on OpenMP.

In order to be able to change the compiler version easily one needs to have
`/appl/htools`
in the file
`~/.grouprc`
if it exists. If the file does not exist, create it and type
`~.  ./.grouprc`
or login again.

Alternatively, one can use the command
`source /appl/htools/bin/grp.profile`
as described below.

Important: To use the bash shell with Sun Studio 8 type the following three commands in the prompt

```
bash
source /appl/htools/bin/grp.profile
init.ss8
```

Once this is done you can check if the FORTRAN compiler is based on Sun Studio 8 by typing
`which f90`

If you get the answer `/opt/FD8/SUNWspro/bin/f90` things are fine and you can proceed with the below script. The script compiles a FORTRAN programme consisting of the files solver.f and call_solver.f.

Listing 2: script_2.txt

```
# This script compiles the FORTRAN programme
# contained in the files solver.f and call_solver.f

# To change to the bash shell running Sun Studio  8 you type
# bash
# source /appl/htools/bin/grp.profile
# init.ss8

# Check that the Sun Studio 8 compiler is used echo 'Using FORTRAN
compiler:'
which f90 echo

# Compile the FORTRAN programme and link to the SPL
echo
echo 'Compiling and linking'
echo
f90 -g -fast solver.f call_solver.f -xlic_lib=sunperf -xarch=v8plusb -autopar

# Indicate that the program is running
```

```
echo
echo 'Testing programme'
echo

# Use one processor
export OMP_NUM_THREADS=1

# run the programme
ptime a.out

# Use four processors
export OMP_NUM_THREADS=4

# run the programme
ptime a.out
```

A brief explanation of the options passed to the f90 compiler, see `man f90` for more details

`-g` The programme is compiled in debugging mode. This is useful if one examines the core file for tracing why the programme crashed or similar.

`-fast` This option increases the compiler optimization to the highest level such that reading of matrices is done in an efficient way. To obtain a fast compilation time and less optimized code the `-dalign` option can be used instead.

`-xlic_lib=sunperf` A link to the SPL is created.

`-autopar` Parallelizes simple segments of code.

`-xarch=v8plusb` Specifies the SUN architecture type.

The command `ptime` measures the process time. It gives the output: `real` being the wall clock (physical) time, `user` is the time that the user spent on the computer and `sys` is some other system time. When using more than one processor one can see that the `user` time increases, type `man ptime` to see more information. However, the `ptime` command does not provide information about if the processors are working on the job.

Output from the programme and a compiled version is available at [3].

A more advanced compiler example using a makefile is treated in section 6.

4 Using the analyzer profiling tool

4.1 Process id

The command `top all` shows all the present jobs running, if you write `top -U$USER` or `ps -ef | grep $USER` you see your jobs. To quit the programme you type `q`, more help is found by typing `man top`. Each job has a process id (pid) which we will use to track the performance of the programme in the next section.

To avoid *bad standing* with the databar support make sure to kill the jobs that are not doing any good. This can be done in different ways. In `top` you can kill a job by pressing `k` and enter the pid. Sometimes the above way of killing a job is not sufficient. If the job has a window you can use `xkill` to kill the window. If everything fails you can use the process id (pid) and write
`kill -9` *pid*

5 Profiling

There are two ways of getting a profile for a FORTRAN programme. The easiest is to use the `collect` command

`collect a.out`

then the programme runs and data is collected in the directory `test.#.er/`. To analyze the data you type

`analyzer` or `analyzer test.#.er`

In a script this becomes (the results are shown in Figure 1)

Listing 3: profile_script.txt

```
# This script runs the programme twice with profiling

# To change to the bash shell running Sun Studio  8 you type
# bash
# source /appl/htools/bin/grp.profile
# init.ss8

# Use one processor
export OMP_NUM_THREADS=1

# Run the programme
collect -o run1.er a.out

# Use four processors
export OMP_NUM_THREADS=4

# Run the programme
collect -o run2.er a.out

# Run the analyzer with the first set of profiling data
analyzer run1.er
```

Alternatively, when profiling a programme that is not an ELF executable – i.e. MATLAB – one wants to attach to a certain process and collect the data. Reconsider the FORTRAN programme from Appendix A, if you change the `go_no_go` file to 1 there will be a pause once the programme has started. The procedure then becomes:

Run the programme

`a.out`

open a new terminal and get the pid. Then you type

`dbx`

`attach` pid

if you type `list` you recognize that it is the right programme that we are monitoring. Now, enable the collector

`collector enable`

`cont`

Return to the window of the program and type `go`. Finally, return to the window of the profiler an type

`quit`

`analyzer`

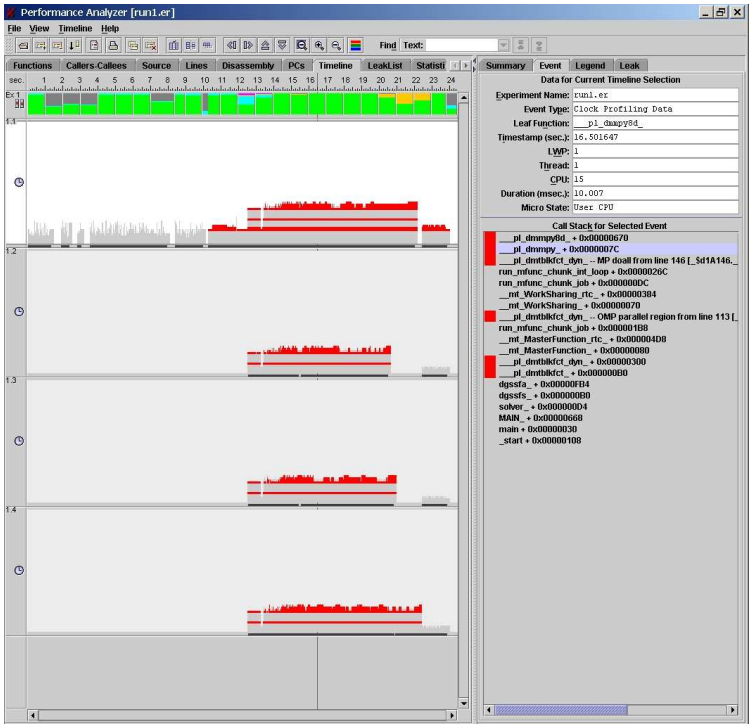

Figure 1: Screen shot from the timeline in the analyzer when solving a large matrix problem using a FORTRAN programme (no JAVA). 'Light grey' peaks are functions calls not in the SUN Performance Library(SPL) and 'dark (red)' peaks are calls from the SPL. Each row in the picture shows the work done on a processor.

5.1 Analyzer software

In the analyzer one can identify the bottleneck in the code. First, open an experiment that contains the data that has been profiled earlier, see Figure 1. The programme is user friendly and we only give a few remarks:

- In the menu `View, Data representation, Timeline` one can choose how many functions that are shown in the big timeline picture. Set the stack size to 50.
- The `Timeline` is where we see the performance of many processors. Push the color chooser. Set all functions to white, then choose a different color. In the text box write `___pl` (3 _ and pl), choose "contains" and the set button. Returning to the big picture we now see where the SPL is called. If the stack size is too small one may not see the SPL.

Having run the analyzer, it is clear that multiple processors are used and they are working on the job.

6 A makefile format for a larger programme

When dealing with large FORTRAN programmes, it becomes apparent that the compilation and linking is a non trivial task. The one-line command is non-optimal because libraries are linked to all programme modules and with a very limited compilation overview. An alternative is to

use the makefile format, which is a script language created for the exact purpose of making the compilation/linking process easy to overview and easy for outsiders to interpret.

The basic idea is to compile each module with distinct commands, e.g. if the NAG library is used in a solver module, only this module is linked to NAG, while the remainder is left as is. The use of this approach provides a transparent compilation scheme, which can relatively easily be adopted by other users.

The layout of the makefile was presented by Ruud van der Pas at an application tuning seminar at DTU in May, 2005. The below code is more practical since the programme consists of several files that are taken care of individually, see line 135 to line 145 for the individual compilation. Also note that the dependencies are easily distinguished for each module. The last part of the compilation is to make the actual linking, and this must still be done in a global manner, see line 125 to line 129.

Getting an environment variable from the makefile to the FORTRAN programme is also delicate. This is because each executable line in the makefile opens a new shell, and thus information exported here is not kept when moving on to the next line. Therefore a trick must be used, in which the variables are first given in a regular manner, line 7 to line 21, and afterwards saved as one variable in line 60. This variable can then be called just before the programme/profiling is executed, see line 94 and line 99 as examples.

Listing 4: Makefile.txt

```
# Makefile to be run on the Gbar/Sun Studio compiler vers. 11

############################################################
################## ENVIRONMENT VARIABLES ###################
############################################################
## GENBRAL VARIABLES
SOLVER_NAME='sparse'     ## 'NAG', 'standard' and 'sparse' (SPL)
BANDED=.true.        ## .true or .false.
BAND2SPARSE=.true.   ## .true or .false.
GAUSS_POINTS=2       ## from 1 to 9
## MODAL ANALYSIS
NEIG=4           ## number of eigenfreq.
STURM_SEQUENCE='off'     ## either 'on' or 'off'
## MIXED ANALYSIS
MIXED_MODEL='fluid' ## 'fluid' or 'static'
MASS_CHECK=.true.    ## .true or .false.
PRESSURE=.true.      ## .true or .false.
PARABOLIC=.true.     ## .true or .false.
## Inputfile to analyze
FILE_NAME='big3_modal'
FILE_NAME='big4'
## Options for parallel execution
SUNW_MP_THR_IDLE=SPIN
OMP_NUM_THREADS=4

############################################################
################## COMPILER VARIABLES ######################
############################################################
# Define program names and objects:
PGM = fem
OBJ = fedata.o link1.o plane42.o processor.o numeth.o fea.o main.o

# General compiler commands # -xautopar -reduction -stackvar
OMP       = -openmp -xloopinfo      #-XlistMP    # OpenMP
#ISA       =-xarch=v8plusb               #32-bit architecture

ISA       =-xarch=v9b                   #64-bit architecture
LIB64     = sparcv9

DEBUG     =-g
OPT_BASE = -dalign              # Basic routines
OPT   =-fast                 # Most important for performance (-xO5 e.g)
NAG   =-M/appl/gnag/fnsol04db/nag_mod_dir
FFLAGS    = $(OPT) $(ISA)
LDFLAGS   = $(OPT) $(ISA)  $(OMP)
COMPILE   = f90
LINK      = f90

# Linking to:
LIBPATH =  -xparallel -xlic_lib=sunperf -lmtmalloc -L/appl/htools/pgplot-5.2.2/lib/ -L/usr/openwin/lib/$(LIB64) \
    -L/usr/sfw/lib -L/appl/gnag/fnsol04db
LIBS = -lpgplot -lX -lpng -lnagf190

############################################################
############ The following should be left as is #############
############################################################
```

```
envvar = SOLVER_NAME=$(SOLVER_NAME) \
    BANDED=$(BANDED) \
    BAND2SPARSE=$(BAND2SPARSE)  \
    GAUSS_POINTS=$(GAUSS_POINTS)\
    NEIG=$(NEIG)\
    STURM_SEQUENCE=$(STURM_SEQUENCE)\
    MIXED_MODEL=$(MIXED_MODEL)\
    MASS_CHECK=$(MASS_CHECK) \
    PRESSURE=$(PRESSURE)\
    PARABOLIC=$(PARABOLIC)\
    FILE_NAME=$(FILE_NAME)\
    SUNW_MP_THR_IDLE=$(SUNW_MP_THR_IDLE) \
    OMP_NUM_THREADS=$(OMP_NUM_THREADS)

default: help                #help menu

.SUFFIXES: .f90

build:
    make $(PGM)

check:
    @f90 -V
    @which f90
    @collect -V
    @analyzer -V

# Clean-up rules:
clean:
    @rm -f $(OBJ) core *.inc *.vo *.mod *.out *.lst *.hf

veryclean: clean
    @rm -f $(PGM)
    @er_rm -f *.er

run_$(PGM):
    @$(envvar) ./$(PGM)

collect_$(PGM):$(PGM)
    @er_rm -f prof_$(PGM).er
    @mkdir /tmp/$(USER)
    @$(envvar) collect -d /tmp/$(USER) -o prof_$(PGM).er ./$(PGM)
    @er_mv /tmp/$(USER)/prof_$(PGM).er .
    @rm -R /tmp/$(USER)

profile_$(PGM):
    analyzer prof_$(PGM).er &

help:
    @echo "############################################################"
    @echo "Help on Makefile usage"
    @echo ""
    @echo "Use make -n <key> to see Makefile command without running program"
    @echo ""
    @echo "Program targets are:"
    @echo "   fem - serial version"
    @echo ""
    @echo "Commands used in this Makefile"
    @echo "make <target>        : compiles and link program <target>"
    @echo "make collect_<target>: Runs program with profiler"
    @echo "make profile_<target>: Runs analyzer"
    @echo "make check           : Checks versions"
    @echo "make clean           : Basic clean up"
    @echo "make veryclean       : Removes all data created by <target>"
    @echo ""
    @echo "############################################################"

# Rule for building the program:
$(PGM): $(OBJ)
    $(LINK) -o $(PGM) $(LDFLAGS) $(NAG) $(OBJ) $(LIBPATH) $(LIBS)
    chmod go+rx $(PGM)

# Dependencies:
main.o:         main.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $*.f90
fea.o:          fea.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $(OMP) $(NAG) $*.f90
processor.o:    processor.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $*.f90
fedata.o:       fedata.f90
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $*.f90
link1.o:        link1.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $*.f90
plane42.o:      plane42.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $(OMP) $*.f90
numeth.o:       numeth.f90 fedata.o
    $(COMPILE) -c $(FFLAGS) $(DEBUG) $*.f90
```

A few comments about general makefile usage:

- @ means that the command should not be printed to screen.

- `make` is the makefile format identifier - e.g. the standard command for compiling is `make`. Note that `make` will call the `build` command if one is present.
- New makefile commands can be added such that the makefile can be used to compile, run, clean up, profile, etc. the FORTRAN programme. All extra commands are called on the generic form `make <key>`, where `<key>` is any of the highlighted and left oriented words from lines 72-124.

To read an environment variable into a FORTRAN variable the below generic command can be used

```
call getenv('SOLVER_NAME', ENV_VALUE)
read(unit=ENV_VALUE,*) SOLVER_NAME
```

7 Making a call from MATLAB running COMSOL

7.1 Compiling a mex file

"Calling (C and) FORTRAN programs from MATLAB" is described in the MATLAB manual (to find the maunual go to the MATLAB help menu). One has to write a gateway function in FORTRAN that creates the link between MATLAB and FORTRAN. The gateway function is compiled from MATLAB into a mex file.

To make the linking to the SPL correctly we need to set the mex compiler options, see the MATLAB help by typing *help mex* in the MATLAB prompt. This is done in the file `./matlab/R#/mexopts.sh` where # is the release number. If the file does not exist you can create it by typing *mex -setup*.

In this file we have used the below settings for FORTRAN in the `sol2` section

Listing 5: part of mexopts.sh

```
#           f90 -V
#           Sun Fortran 95 7.1 Patch 112762-09 2004/01/26
            FC='/opt/FD8/SUNWspro/bin/f90'
            FFLAGS='-KPIC -fast -xarch=v8plusb -autopar -mt'
            FLIBS="$MLIBS -lfui -lfsu -lm -lc -Bstatic -lsunperf_mt -Bstatic -lmtsk -xparallel"
```

To compile the mex file we start MATLAB after initializing Sun Studio 8:
First, type the below commands

```
bash
source /appl/htools/bin/grp.profile
init.ss8
```

Now, run the below script

7.1.1 Script file: `submit_test`

```
# To change to the bash shell running Solaris 8 you type
#
# bash
# source /appl/htools/bin/grp.profile
# init.ss8
#
# All commands use the sentinel #$
```

```
#
#$ -N run1
#$ -S /bin/bash
#$ -j y
#$ -o run1.out
#$ -e run1.err
#$ -cwd
###$ -M yourmail@dtu.dk
###$ -m be
### Request  hours run time
#$ -l  cre
#$ -pe HPC 8

# Increase the stack limit to avoid problems running FEMLAB with MATLAB
ulimit -s 32768
#ulimit -s 65536
#ulimit -s 131072

# Show libs loaded in MATLAB
export LAPACK_VERBOSITY=1

# Change the BLAS library that MATLAB uses
# see: http://www.mathworks.com/access/helpdesk/help/techdoc/rn/r14sp1_math_new.html

# We don't load the SPL here, instead we make a link to SPL in the FORTRAN programme.
# If one wants load a library in MATLAB it is done this way
#export BLAS_VERSION=libsunperf.so.4

# Number of processors to use
#export OMP_NUM_THREADS=4

# Cleanup directory
#rm -rf p*.txt

# Startup FEMLAB with MATLAB and run a MATLAB script (useful for submitting batch jobs at HPC)
export OMP_NUM_THREADS=1
run_comsol32_matlab14.txt matlab path -ml -nodesktop -ml -nosplash -mlr test_2d

export OMP_NUM_THREADS=2
run_comsol32_matlab14.txt matlab path -ml -nodesktop -ml -nosplash -mlr test_2d

export OMP_NUM_THREADS=4
run_comsol32_matlab14.txt matlab path -ml -nodesktop -ml -nosplash -mlr test_2d

export OMP_NUM_THREADS=8
run_comsol32_matlab14.txt matlab path -ml -nodesktop -ml -nosplash -mlr test_2d
```

Comments to the script:

- The first lines are related to batch jobs at HPC, see section 7.2.4.
- The `ulimit` command increases the stack size (memory) which is relevant when running MATLAB with COMSOL. The command `limit` displays the resources available to the user, see `man limit`.
- We start COMSOL using MATLAB 14 in an independent script which is present in the working directory, see below.

7.1.2 Script file: `run_comsol32_matlab14.txt`

This is a *copy* of the COMSOL file which is run to start COMSOL. Only the first few lines have been changed such that MATLAB14 is run. The reason is that MATLAB14 is not using the SPL, hence compiler options blocking the use of SPL is not present with MATLAB14. Other options

such as the stacksize for JAVA can also be edited this way. This we can do because we created the file and therefore have permission to write. The addition of the command *MAINCLS* prevents COMSOL from opening of a GUI when COMSOL is started with MATLAB.

```
#!/bin/sh

MLROOT=/appl/gmatlab/matlab14
FLROOT=/appl/gmatlab/comsol32
LIBGLPATH=

MAXHEAP=256m
MAXHEAPCLIENT=512m
MAXHEAPSERVER=256m
MAXHEAPSERVER64=1024m
JAVAOPTS=on
STACKSIZE=2m
export STACKSIZE

JVMTYPE_CLIENT=client
JVMTYPE_SERVER=server

SETLOCALE=on

#------------------------------------------------------------------------------
#   DO NOT EDIT CODE BELOW!
#------------------------------------------------------------------------------

...

    MAXHEAP=$MAXHEAPCLIENT
    setcommon $1 ${DM_DEF_GUI} ${DM_SUP_GUI}
    MAINCLS=com.femlab.gui.Femlab
    shift
    APPLARGS="client $*"
    ;;
  matlab)
    # MAXHEAP affects JVMARGS that is used when starting client from Matlab
    # heap size settings used by Matlab is MAXHEAPSERVER (search for java.opts)
    MAXHEAP=$MAXHEAPCLIENT
    if [ "${TMPARCH}" = "glnxa64" ]; then
      setcommon $1 32 -
      MAINCLS=com.femlab.script.BatchRunner
    else
      setcommon $1 32 -
      MAINCLS=com.femlab.script.BatchRunner
    fi
    shift
    runmatlab $*
    exit
    ;;
  compile)
    setflcp
    shift

....
```

Now we are able to compile the mex file by typing the MATLAB command
mex -v solver.f solverg.f

Finally, we can run a the test script that uses SPL for solving the linear algebra problem by the command
test_2d

The code of the test script is given in Appendix B.2.

7.2 Running big jobs in the G-bar

7.2.1 Remote access

The best thing to use is the `ThinLinc` software that the G-bar supports, since it enables you to run OpenGL applications such as stand alone COMSOL with the Graphical User Interface (GUI). Go to the home page of G-bar support (www.gbar.dtu.dk),
http://www.gbar.dtu.dk/old/guide?cat=1&subcat=14

Read the user's guide and download the programme from the software menu. Use your student ID and password. Some alternatives to `ThinLinc` are stated below.

7.2.2 Xwin32

1. Download Xwin32 from the www. Try to ask the databar support if they have bought a site licence such that you can register. It is possible in the m-bar
2. Use the wizard and log on to the G-bar using the StarNetSSH.
   Host: `bohr.gbar.dtu.dk`

7.2.3 A nice job without a shell

The below method is relevant to users/students that are not allowed to submit jobs using the grid engine on DTU. In order to have code running for a long time, say 14 hours, one needs to behave nicely.

To start MATLAB in a UNIX shell without any pop–up graphics and immediately run the MATLAB file (a script) `test.m` you write

```
matlab -nodesktop -nosplash -r test
```

Suppose that `test.m` does a big calculation but with no graphics. It also saves the result in a data file termed `output.mat`, then it terminates MATLAB using the `exit` command. The unix command `nohup` starts a job but does not require a UNIX shell. Hence we can start a big MATLAB job this way

```
nice nohup matlab -nodesktop -nosplash -r test
```

and then close the window and log out. The `nice` command ensures that the big job does not use all the CPU–power and is required to avoid *bad standing*. The next day we can pick up the results in the `output.mat` file using the MATLAB command `uiload` from the MATLAB prompt, select the file `output.mat`.

7.2.4 A batch job

For users associated with HPC [5] all jobs have to be run as batch jobs. A batch job is just a script where comment lines with #$ are options to the Grid Engine batch system, see section 7.1.1. A few remarks are appropriate

`#$ -N` Assigns a name to the job.

`#$ -S` Is the shell used.

`#$ -o` Name of output file.

`#$ -e` Name of error file.

`#$ -cwd` Current working directory.

`#$ -M` Your email address.

`#$ -m be` You will receive an email once the job has begun ("b") and once it has terminated("e").

`#$ -pe HPC 4` Gives you 4 processors at HPC.

To place a job in the batch queue the `qsub` command is used to see the status of your programme use `qstat`. See `man qsub` and [5] for more information.

8 Results

Some time studies are presented below. We consider a scalar problem based on the code given in Appendix B and solve the resulting linear algebra problem using the routine “dgssfs” from the SPL.

It is not documented which numerical routine is implemented behind the “dgssfs” interface although some references are suggested in [6]. However, the youngest of these references are from 1993 and new improvements in the solution techniques of sparse linear algebra problems such as UMFPACK [8] and TAUCS [7] do not seem to be included in the SPL.

The tests were run on a Sun Fire 6800 with a total of 24 CPUs of the type 750 MHz/8 MB cache Sun UltraSparc III with a total of 48 GB memory. The tests were not run using the Grid Engine, but as a standard job. This may explain that scaling using more than four processors is not observed. For comparison some of the tests have also been carried out on a PC is equipped with an Intel Pentium 4 CPU, 2.4 GHz, 1 GB RAM and windows 2000 and COMSOL 3.1 installed.

8.1 2D Poisson problem

The below table shows the wall clock time (in seconds) it takes to obtain a solution (factorization and back substitution) for a Poisson problem for a varying number of Sun processors.

	p=1	p=2	p=4	p=8		PC
DOFs 65535^2						
nnz-% : 1.4e-02						
Assembly	4.4	4.3	4.2	4.4		1.84
SPL(symrcm)	3.4 (57%)	2.9 (60%, **1.2**)	2.6 (62%, **1.3**)	2.6 (63%, **1.3**)		
SPL(mmd)	3.5 (56%)	2.7 (61%, **1.3**)	2.3 (65%, **1.5**)	2.3 (66%, **1.5**)		
COMSOL	6.4 (34%)	6.2 (34%)	6.2 (34%)	6.5 (34%)		
DOFs 262144^2						
nnz-% : 3.4e-03						
Assembly	2.2e1	2.0e1	2.0e1	1.9e1		9.08
SPL(symrcm)	2.5e1 (47%)	1.8e1 (53%, **1.4**)	1.5e1 (57%, **1.7**)	1.6e1(55%, **1.6**)		
SPL(mmd)	3.2e1 (41%)	1.9e1 (51%, **1.7**)	1.5e1 (57%, **2.1**)	1.6e1 (55%, **2.0**)		
COMSOL	3.6e1 (27%)	3.5e1 (28%)	3.7e1 (27%)	3.7e1 (28%)		
DOFs 1048576^2						
nnz-% : 8.6e-04						
Assembly	1.1e2	1.0e2	1.0e2	1.0e2		5.61e1
SPL(symrcm)	2.4e2 (31%)	1.5e2 (41%, **1.6**)	1.0e2 (50%, **2.4**)	2.8e2 (26%, **0.9**)		
SPL(mmd)	3.3e2 (25%)	1.8e2 (36%, **1.8**)	1.1e2 (48%, **3.0**)	2.4e2 (29%, **1.4**)		
COMSOL	2.5e2 (21%)	2.4e2 (24%)	2.4e2 (21%)	2.4e2 (22%)		

DOFs denotes the number of unknowns, the first number in parenthesis is the ratio assembly time to total solution time and the second number given in boldface is the speedup. Moreover, the COMSOL solution is the version cf. Appendix B.2 where the problem is first assembled then solved – since this is faster than the one-line solution command.

Looking at the times that are processor independent, i.e. the assembly time and on the COMSOL solution time, we see that the first digit is constant but the second depends on the current system load. This observation motivates an interpretation of the results based only on the first digit to account for varying load on the computer due to the multiple user environment.

First, we notice that the PC is at least twice as fast as the Sun computer for small problems. However cache problems seem to be present for the PC since the assembly time decreases from being 2.4 to 1.8 times faster than the Sun computer as the problem size grows. For the Sun computer the assembly time increases linearly which can probably be attributed to the shared memory architecture. It is disappointing to see such a big difference in performance due to the choice of hardware/platform because program development, program debugging, teaching exercises etc. are typically done on small problems.

We see that a small scaling effect is gained if more processors are used up to a factor of 2.4. This study also shows that the time it takes to assemble the problem in COMSOL is comparable to the solution time of the linear algebra problem. Hence the COMSOL assembly procedure can not be classified as 'high performance'. This is relevant to keep in mind and is likely to be a consequence of the very general PDE toolbox that is available through COMSOL 3.2.

For the current test problem we tested the matrix reordering schemes MMD (Multiple Minimum Degree) implemented in the SPL and symrcm (Symmetric reverse Cuthill-McKee permutation) as a MATLAB preprocessor used before the SPL call. For this test problem it pays off to use the symrcm however it is not implemented in the SPL. Instead it is possible to supply the SPL sparse solver with a user defined permutation vector. Other reordering schemes from MATLAB were also

tested but did not perform as well as the symrcm.

One may expect the SPL to be even more efficient if one has to consider several right hand sides. The SPL can handle this - like all numerical libraries - whereas COMSOL 3.2 does not allow a separate factorization step in the solution procedure of a linear problem.

Looking at profiles (not shown) for the above test showed that a user lock is present prohibiting optimal performance. This could be due to JAVA being present at runtime which is necessary for programs like MATLAB and COMSOL. This work does not investigate this issue further, but it is relevant to clarify if JAVA is the reason for the user lock.

8.2 3D Poisson problem

The below table shows similar timings as the problem presented above, but with the PDE solved in three spatial dimensions. Although it was possible to solve a 2D problem two magnitudes larger memory problems already appeared for a small 3D problem. The latter is the reason for only considering one small problem of size DOFs$= 33759^2$.

Here we see a speed up when using SPL of a factor 3.4 which is encouraging. For this test problem it appears that the three different reordering schemes 'nat', 'gnd' and 'mmd' does not significantly alter the solution time.

	p=1	p=2	p=4	p=8
DOFs 33759^2				
nnz-% : 7.5e-02				
Assembly	1.0e1	1.1e1	1.0e1	1.0e1
SPL(nat)	1.1e2 (9%)	5.6e1 (16%, **2.0**)	3.3e1 (24%, **3.3**)	1.8e2 (5%, **0.6**)
SPL(gnd)	1.1e2 (9%)	5.5e1 (17%, **2.0**)	3.2e1 (25%, **3.4**)	1.9e2 (5%, **0.6**)
SPL(mmd)	1.1e2 (9%)	5.6e1 (16%, **2.0**)	3.2e1 (25%, **3.4**)	1.8e2 (5%, **0.6**)
COMSOL	1.0e2 (7%)	1.0e2 (7%)	1.1e2 (7%)	1.1e2 (7%)

9 Conclusions

This study shows that it is possible to link MATLAB, COMSOL and the Sun Performance Library (SPL) together using multiple processors on the Sun High Performance Computing platform at the Technical University of Denmark. It turns out that solution times for the test problem considered can be reduced with a factor of 2.4 for a 2D problem and a factor of 3.4 for a 3D problem.

Moreover, based on the time it takes to assemble the system matrix, we conclude that COMSOL runs slowly on the considered Sun platform compared to a standard PC. Debugging and solution of smaller problems (teaching exercises etc.) should therefore not be carried out on the Sun platform. For large problems the use of the Sun computer may be the only way to achieve a solution.

COMSOL 3.2 does offer a very attractive PDE research environment which reduces the time it takes to setup and solve a PDE. However, high computational performance is needed in particular for 3D problems which can be approached by using the Sun Performance Library. Use of the SPL is not currently available in the COMSOL package although the library is available on the platform.

Acknowledgements

The authors thank Jukka Komminaho, Department of Information Technology, Uppsala University for initial help with the implementation on a Sun computer. For some of the activities at Uppsala University see http://www.uppmax.uu.se/

In addition we thank the members of the TOPOPT group for hints and discussions to the material presented.

References

[1] See www.openmp.org and www.compunity.org

[2] Informatics and modelling, Technical University of Denmark: *Tuning of Parallel Applications*, Parallel Computing & OpenMP, January 2006.

[3] See http://www2.mat.dtu.dk/people/A.G.Hansen/

[4] Xwin32 is distributed by http://www.starnet.com/

[5] Sun High Performance Computing (HPC) Systems at the Technical University of Denmark, see http://www.hpc.dtu.dk/ . How to setup an accout is described here http://www.hpc.dtu.dk/GridEngine/

[6] Sun Performance Library User's Guide, Sun Studio 10, Part No. 819-0498-10 Janaury 2005, Revision A. Sun Microsystems, Inc. www.sun.com

[7] Homepage of the TAUCS project. http://www.tau.ac.il/ stoledo/taucs/

[8] Homepage of Tim Davis, University of Florida. http://www.cise.ufl.edu/research/sparse/ See also the recent "Summary of available software for sparse direct methods" at http://www.cise.ufl.edu/research/sparse/codes/

A FORTRAN Code

A.1 File: `call_solver.f`

Listing 6: call_solver.f

```
      program call_solver

c
c     This program is an example driver that calls the sparse solver.
c     It factors and solves a symmetric system, by calling the
c     one-call interface.
c
c     see http://docs.sun.com/source/816-2463/plug_matrices.html

      implicit none

      integer           neqns, ier, msglvl, outunt, ldrhs, nrhs, withgo
      character         mtxtyp*2, pivot*1, ordmthd*3
      double precision  handle(150)
      integer,dimension(:),allocatable :: colstr, rowind
      double precision, dimension(:), allocatable ::  values
      double precision, dimension(:,:), allocatable::  rhs, xexpct
      integer           i, j, M, N, L, small_prob

c
c  Sparse matrix structure and value arrays.  From George and Liu,
```

```
c  page 3.
c    Ax = b, (solve for x) where:
c
c       4.0    1.0    2.0    0.5    2.0         2.0         7.0
c       1.0    0.5    0.0    0.0    0.0         2.0         3.0
c  A = 2.0    0.0    3.0    0.0    0.0    x = 1.0    b = 7.0
c       0.5    0.0    0.0    0.625  0.0        -8.0        -4.0
c       2.0    0.0    0.0    0.0   16.0        -0.5        -4.0
c
c


c
c      Determine if a pause is wanted
c
       open(8,file="go_no_go")
       read(8,*) withgo
       close(8)

       if (withgo .eq. 1) then
         pause
       end if

c
c      Determine if we want to solve the 5x5 problem shown above or
c      a larger problem
c
       small_prob = 0

       if (small_prob .eq. 1) then
         allocate(colstr(6))
         allocate(rowind(9))
         allocate(values(9))
         allocate(rhs(5,2))
         allocate(xexpct(5,2))

         colstr = (/ 1, 6, 7, 8, 9, 10 /)
         rowind = (/ 1, 2, 3, 4, 5, 2, 3, 4, 5 /)
         values =(/ 4.0d0, 1.0d0, 2.0d0, 0.5d0, 2.0d0, 0.5d0, 3.0d0,
     &              0.625d0, 16.0d0 /)
C        data rhs     / 7.0d0, 3.0d0, 7.0d0, -4.0d0, -4.0d0 /
         rhs(:,1)=     (/ 7.0d0, 3.0d0, 7.0d0, -4.0d0, -4.0d0 /)
         rhs(:,2)=     (/ 7.0d0, 3.0d0, 7.0d0, -4.0d0, -4.0d0 /)

         xexpct(:,1) =(/ 2.0d0, 2.0d0, 1.0d0, -8.0d0, -0.5d0 /)
         xexpct(:,2) =(/ 2.0d0, 2.0d0, 1.0d0, -8.0d0, -0.5d0 /)
       else
c
c        Read a larger matrix in the above format
c
         open(8,file="problem_data")

         read(8,*) M
         allocate(colstr(M+1))
         do i=1,M+1
           read(8,*) colstr(i)
         end do
C        print*,'.. colstr done'
         read(8,*) L
         allocate(rowind(L))
         allocate(values(L))
         do i=1,L
           read(8,*) rowind(i),values(i)
         end do
C        print*,'.. values done'
         read(8,*) M,N
         allocate(rhs(M,N))
         allocate(xexpct(M,N))
         do i=1,M
           do j=1,N
           read(8,*) rhs(i,j)
           end do
         end do
         do i=1,M
           do j=1,N
           read(8,*) xexpct(i,j)
           end do
         end do
C        print*,'.. rhs done'
       end if

       ordmthd = 'mmd'
c
c  call single call interface
c

       M = size(rhs,1)
       N = size(rhs,2)


       print*,"call solver ....."


       call solver( colstr, rowind, values, M, N,
     &              rhs, ordmthd, ier  )

       write(6,*) 'i', 'rhs(i)', 'expected rhs(i)', 'error'
       do j = 1, N
         do i = 1, min(M,50)
           write(6,*) i, rhs(i,j), xexpct(i,j), (rhs(i,j)-xexpct(i,j))
```

```
        enddo
      enddo

      print*,'error flag: ier = ', ier

      deallocate(rhs)
      deallocate(rowind)
      deallocate(values)
      deallocate(colstr)


      end
```

A.2 File: `solver.f`

Listing 7: solver.f

```
      subroutine solver( colstr, rowind, values,
     &                   M, N, rhs, ordmthd, ier)

      implicit none

      integer*4          ier, ier2, outunt
      character          ordmthd*3
      double precision   handle(150)
      integer*4          colstr(*), rowind(*)
      real*8             values(*)
      real*8             rhs(M,N)
      integer*4          i, j, M, N


      print*,'number of threads:'
      call system('echo $OMP_NUM_THREADS')

      call dgssfs ( 'ss', 'n', M, colstr, rowind,
     &              values, N  , rhs,    M , ordmthd,
     &              outunt, 0, handle, ier             )
      print*,'Error flag: ier = ', ier
      print*,'number of threads after solution:'
      call system('echo $OMP_NUM_THREADS')
c
c  deallocate sparse solver storage
c
      call dgssda ( handle, ier2 )

      ier = max(ier,ier2)

      return

      end
```

A.3 File: go_no_go

This file contains a 0 or a 1 controlling if a pause is used in the programme.

Listing 8: go_no_go

```
0
```

B MATLAB Code

B.1 Gateway file: `solverg.f`

Listing 9: solverg.f

```
!
!=======================================
! Compile a FORTRAN program with a gateway using:
! mex -v -g solver.f solverg.f
!
! then run the test script: test_femlabmatlab.m
!
```

```
!=======================================
!
! solverg.f - Gateway function for solver.f
!
! This is an example of the FORTRAN code required for interfacing
! a .MEX file to MATLAB.
!
! This subroutine is the main gateway to MATLAB.  When a MEX function
!  is executed MATLAB calls the MEXFUNCTION subroutine in the corresponding
!  MEX file.
!
!
      SUBROUTINE MEXFUNCTION(NLHS, PLHS, NRHS, PRHS)
!
!********** DEFINE POINTERS FOR MEXFUNCTION ****************
!     (pointer) Replace integer by integer*8 on 64-bit platforms
!
      INTEGER*4 PLHS(*), PRHS(*)
      INTEGER*4 NLHS, NRHS
!
!********** DEFINE POINTERS FOR SUBROUTINES CALLED BY MEXFUNCTION ****************
!
      INTEGER*4 MXCREATEDOUBLEMATRIX, MXGETPR
      INTEGER*4 MXCREATENUMERICMATRIX
      INTEGER*4 MXGETM, MXGETN
!
!********** CREATE POINTERS TO INPUT/OUTPUT ARGUMENTS, format: <variable>P ****************
!
      INTEGER*4 colstrP, rowindP, valuesP,rhsP, ordmthdP, IierP
      INTEGER*4 write_matrix_as_file
!
!********** CREATE DUMMY VARIABLES USED FOR FORTRAN COMPUTATIONS, format: R<variable>P  ****
!
      INTEGER*4                                :: RierP
      INTEGER*4, DIMENSION(:),ALLOCATABLE
     &             :: RcolstrP, RrowindP

      CHARACTER*3                              :: RordmthdP
      REAL*8, DIMENSION(:), ALLOCATABLE    :: RvaluesP
      REAL*8, DIMENSION(:,:), ALLOCATABLE :: RrhsP
!
!********** AUXILARY VARIABLES ****************
!
      INTEGER*4 M, N, L
      INTEGER*4 p, ni, nj, jlast
!
!********** INPUT - RHS ****************
!
       colstrP   = MXGETPR(PRHS(1))
       rowindP   = MXGETPR(PRHS(2))
       valuesP   = MXGETPR(PRHS(3))
       rhsP      = MXGETPR(PRHS(6))
       ordmthdP = MXGETPR(PRHS(7))

       M     = MXGETM(PRHS(6))
       N     = MXGETN(PRHS(6))
!
!********** CREATE OUTPUT POINTERS ****************
!

!      Status flag
       PLHS(1) = MXCREATENUMERICMATRIX(1,1,
     &mxClassIDFromClassName('int32'),0)
!      Solution
       PLHS(2) = MXCREATEDOUBLEMATRIX(M,N,0)

!
!********** GET THE DIMENSIONS ****************
!

      if (MXGETM(PRHS(1)) .eq. M+1) then
C
C       The input has propably been numbered correctly
C
         print*,'****************'
         print*,'  ORDINARY NUMBERING'
         print*,'****************'
         ALLOCATE(RcolstrP(M+1))
         CALL mxCopyPtrToInteger4(colstrP, RcolstrP, M+1)

         L = RcolstrP(M+1)-1

         ALLOCATE(RrowindP(L))
         ALLOCATE(RvaluesP(L))
         ALLOCATE(RrhsP(M,N))

         CALL mxCopyPtrToInteger4(rowindP, RrowindP, L)
         CALL MXCOPYPTRTOREAL8(valuesP, RvaluesP, L)
         CALL MXCOPYPTRTOREAL8(rhsP    , RrhsP    , M*N)
         CALL mxCopyPtrToCharacter(ordmthdP,RordmthdP,3)

      elseif ((MXGETM(PRHS(1)) .eq. MXGETM(PRHS(2))) .and.
     &(MXGETM(PRHS(2)) .eq. MXGETM(PRHS(3)))) then
C
C       The input propably originates from MATLABs [I,J,val]=find(K)
C
         print*,'****************'
         print*,'  MATLAB-FIND NUMBERING'
         print*,'****************'
```

```
        L = MXGETM(PRHS(1))
        ALLOCATE(RcolstrP(L))
        ALLOCATE(RrowindP(L))
        ALLOCATE(RvaluesP(L))
        ALLOCATE(RrhsP(M,N))

        CALL mxCopyPtrToInteger4(colstrP, RcolstrP, L)
        CALL mxCopyPtrToInteger4(rowindP, RrowindP, L)
        CALL MXCOPYPTRTOREAL8(valuesP, RvaluesP, L)
        CALL MXCOPYPTRTOREAL8(rhsP   , RrhsP   , M*N)
        CALL mxCopyPtrToCharacter(ordmthdP,RordmthdP,3)

        ni = 0
        nj = 0
        jlast = 0
        do p = 1,MXGETM(PRHS(1))
          if (RcolstrP(p)<=RrowindP(p)) then
C
C use symmetry
C
            nj = nj +1;
              if (RcolstrP(p) .ne. jlast) then
C
C               new column
C
                ni = ni + 1;
              RcolstrP(ni) = nj;
              end if
              jlast = RcolstrP(p);
              RrowindP(nj) = RrowindP(p);
              RvaluesP(nj) = RvaluesP(p);
          end if
        end do
        ni = ni +1;
        RcolstrP(ni) = RcolstrP(ni-1) + 1;
        L = nj
        print*,'       sorting done'

      else
C
C       The numbering is wrong
C
        print*,'****************'
        print*,' ERROR'
        print*,'****************'

        ierP       = MXGETPR(PLHS(1))
        RierP = -100
        CALL MXCOPYInteger4TOPTR(RierP, ierP, 1)
        return
      end if

C
C      write matrix in a file that can be read by the FORTRAN programme
C
       write_matrix_as_file = 0
       if (write_matrix_as_file .eq. 1) then
         open(8,file="problem_data")
         write(8,*) M
         do i=1,M+1
           write(8,*) RcolstrP(i)
         end do

         write(8,*) L
         do i=1,L
           write(8,*) RrowindP(i),  RvaluesP(i)
         end do
         write(8,*) M,N
         do i=1,M
           do j=1,N
             write(8,*) RrhsP(i,j)
           end do
         end do
       end if

!
! MAKE CALL AND COPY LOCAL LHS ARRAY TO MATRIX OUTPUT
!
       RierP = -1
       CALL solver(RcolstrP(1:M+1), RrowindP(1:L), RvaluesP(1:L),
     &            M, N, RrhsP, RordmthP, RierP )

C
C      write the solution obtained to a file if wanted
C
       if (write_matrix_as_file .eq. 1) then
         do i=1,M
           do j=1,N
             write(8,*) RrhsP(i,j)
           end do
         end do
         close(8)
       end if
!
! ********** OUTPUT - LHS ****************
!
      ierP       = MXGETPR(PLHS(1))
      rhsP       = MXGETPR(PLHS(2))
```

```
      CALL MXCOPYInteger4TOPTR(RierP, ierP, 1)
      CALL MXCOPYREAL8TOPTR(RrhsP, rhsP, M*N)

      DEALLOCATE(RcolstrP)
      DEALLOCATE(RrowindP)
      DEALLOCATE(RvaluesP)
      DEALLOCATE(RrhsP)

      RETURN
      END
```

B.2 MATLAB script: `test_2d.m`

Remark: The option of using symmetry in the COMSOL assembly procedure was also tested. This affects the memory use, but does not change the solution time significantly.

Listing 10: test_2d.m

```
clear all
clc

%*********************************************
% Input parameters
%*********************************************
% Vector of mesh parameters
no_mesh        = [10];
%no_mesh        = [9 8 ];
spl_symfactor = {'nat' 'gnd' 'mmd' ...
  'matlab_colamd' 'matlab_colperm' 'matlab_symamd' 'matlab_symrcm'};

%spl_symfactor = {'nat'};


sdim = [2];

% matlab parameters are set et spparms

% Vector that determines if a direct solution is performed (0=no/1=yes)
call_matlab         = 1*ones(1,length(no_mesh));
call_spl            = 1*ones(1,length(no_mesh));
call_femlab         = 0*ones(1,length(no_mesh));
call_femlab_femlin = 1*ones(1,length(no_mesh));
% Vector that determines if a non-uniform conductivity field is used (0=no/1=yes)
nonuniform          = ones(1,length(no_mesh));
pause_on_off = 0;

% Create an output file
[s,w]= system('echo $OMP_NUM_THREADS');
try
  wstr = num2str(eval(w));
catch
  wstr = '1';
end
% Generate time format and filename
yy_mm_dd = datestr(now,25);yymmdd = [ yy_mm_dd(1:2) yy_mm_dd(4:5) yy_mm_dd(7:8)];
hh_mm_ss = datestr(now,13);hhmmss = [hh_mm_ss(1:2)  hh_mm_ss(4:5)  hh_mm_ss(7:8)];
filename = ['p_' wstr '_' yymmdd  '_' hhmmss '.txt'];
diary(filename)

for i=1:length(sdim)
if sdim(i)==3
  no_mesh = no_mesh -3;
end
clear times

for j=1:length(no_mesh)
  clear KC LC NULL UD
  clear rowind colstr values rhs
  clear K L M N

  flclear fem

  % Define geometry
  r2 = rect2(0,1,0,1);
  fem.geom = r2;

  % Define mesh
  nx = 2^no_mesh(j); ny = 2^no_mesh(j);
  fem.mesh = meshmap(fem,'edgelem',{[1 3] [nx] [2 4] [ny]});
  fem.xmesh = meshextend(fem);

  if sdim(i)==3
    fem = meshextrude(fem,'distance',1,'elextlayers',{nx});
  end
  % Store mesh information
  times{j}.mesh = fem.mesh;
  times{j}.nxny = [nx ny];
  if sdim(i)==2
```

```
    times{j}.ndv = nx*ny;
  else
    times{j}.ndv = nx*ny*nx;
  end
  % Variational problem
  if sdim(i)==2
    fem.sdim  = {'x','y'};
  else
    fem.sdim  = {'x','y','z'};
  end
  fem.dim   = {'T' 'b'}; % T: Temperature field, a: Design field
  fem.shape = {shlag(1,'T') shdisc(length(fem.sdim),0,'b')};
  fem.form  = 'weak';

  % Define problem data
  fem.const = {'f',1,'G',2};

  % Variational problem in Omega
  if sdim(i)==2
    fem.equ.weak{1}(1,:) = {'T_test*f -(Tx_test*b^3*Tx + Ty_test*b^3*Ty)'};
  else
    fem.equ.weak{1}(1,:) = {'T_test*f -(Tx_test*b^3*Tx + Ty_test*b^3*Ty + Tz_test*b^3*Tz)'};
  end

  % Neumann BCs on Gamma_N
  fem.bnd.weak = {{'0'},{'0'},{'T_test*0'}};
  % Dirichlet BCs on Gamma_D
  if sdim(i)==2
    fem.bnd.ind={[1],[3],[2 4]};
  else
    fem.bnd.ind={[1],[6],[2:5]};
  end
  fem.bnd.constr = {{'T-0'},{'T-G'},{'0'}};

  % Prepare FEMLAB model
  fem.xmesh = meshextend(fem);

  % Display input data
  disp(['ndv =' num2str(times{j}.ndv) ', no_proc=' wstr ])

  % Initialize solution
  if 1==nonuniform
    % non-uniform conductivity
    if sdim(i)==2
      fem.sol = asseminit(fem,'init',{'T' '0' 'b' '0.001 + (x*y/2)^3'});
    else
      fem.sol = asseminit(fem,'init',{'T' '0' 'b' '0.001 + (x*y*z/4)^3'});
    end
  else
    % uniform conductivity
    fem.sol = asseminit(fem,'init',{'T' '0' 'b' '1'});
  end

  % Assemble matrix problem
  try
    system('echo FEMLAB ASSEMBLY');
  end
  times{j}.start = clock;
  [KC,LC,NULL,UD] = femlin(fem,'U',fem.sol.u,'Solcomp','T','report','off');
  times{j}.assembly = etime(clock,times{j}.start);
  times{j}.sys_matrix_size = size(KC);
  times{j}.sys_matrix_nnz  = nnz(KC);
  times{j}.sys_matrix_nnz = times{j}.sys_matrix_nnz/ ...
          (times{j}.sys_matrix_size(1)*times{j}.sys_matrix_size(2))*100;
  times{j}.sys_matrix_size = [num2str(times{j}.sys_matrix_size(1)) ...
           ' x ' num2str(times{j}.sys_matrix_size(2))];

  if pause_on_off
    disp('.. press enter')
    pause
  end

  % MATLAB Backslash solution of linear problem
  if call_matlab(j)
    try
      system('echo .. calling MATLAB solver');
    end
    times{j}.start = clock;
    solvec = NULL*(KC\LC) + UD;

    times{j}.BS_solution = etime(clock,times{j}.start);
    disp(['..... ', sprintf('%1.1e ',times{j}.BS_solution) 's'])
  end

  % SUN PL solution of linear problem
  if call_spl(j)
   for jj=1:length(spl_symfactor)
%    try
%      system('echo .. SPL preproccsing in MATLAB');
%    end
    times{j}.start = clock;
    if length(spl_symfactor{jj})>7
       P=feval(spl_symfactor{jj}(8:end),KC);

      [rowind, colstr, values] = find(KC(P,P));
      colstr = int32(colstr);
      rowind = int32(rowind);
      rhs(:,1) = LC(P);
```

```
      ordmthd = 'nat';
    else

      [rowind, colstr, values] = find(KC);
      colstr = int32(colstr);
      rowind = int32(rowind);
      rhs(:,1) = LC;
      ordmthd = spl_symfactor{jj};
    end

    [M,N]=size(rhs);
%   try
%     system(['echo .. calling spl solver, ordmthd = ' spl_symfactor{jj}]);
%    end
    try
      [ier,s_spl]=solver( colstr, rowind, values, M, N, rhs, ordmthd );

    if length(spl_symfactor{jj})>7
        ss = zeros(M,1);
        for iii=1:M
           ss(P(iii))=s_spl(iii);
        end
        sol_spl = NULL*(ss) + UD;
    else
        sol_spl = NULL*(s_spl) + UD;
    end
    catch
      disp('.. Warning: Call to SPL solver failed')
    end
    splt(jj) = etime(clock,times{j}.start);
    disp(['.... spl, ordering = ' spl_symfactor{jj} ', time = ' sprintf('%1.1e',splt(jj)) 's'])
   end
    [val,index]=min(splt);
    times{j}.SPL_solution = splt(index);
    times{j}.SPL_ordering = spl_symfactor{index};
  end
  %FEMLAB solution of linear problem
  if call_femlab(j)
    try
      system('echo .. calling FEMLAB');
    end
    times{j}.start = clock;
    fem.sol = femlin(fem,'U',fem.sol.u,'Solcomp','T','report','off');
    times{j}.fl_solution = etime(clock,times{j}.start);
    disp(['..... ', sprintf('%1.1e',times{j}.fl_solution) 's'])
  end
  %FEMLAB solution of linear problem using FEMLIN
  if call_femlab_femlin(j)
    try
      system('echo .. calling FEMLAB using femlin');
    end
    times{j}.start = clock;
    [K,L,M,N] = assemble(fem,'U',fem.sol.u,'Solcomp','T','report','off');
    times{j}.fl_lin_solution_a = etime(clock,times{j}.start);
    times{j}.start = clock;
    fem.sol = femlin('in',{'K' K 'L' L 'M' M 'N' N},'report','off');
    times{j}.fl_lin_solution = etime(clock,times{j}.start);
    disp(['..... ', sprintf('%1.1e',times{j}.fl_lin_solution) 's'])
  end

  %Release file, such that it can be read while the programme is running
  diary off
  diary(filename)

end

% print a summary
%clc
for j=1:length(no_mesh)
  disp('*****************************************')
  %display mesh information
  if sdim(i)==2
    disp(' 2D problem')
  else
    disp(' 3D problem')
  end
  %times{j}.mesh
  if  nonuniform(j)
    disp(['ndv = ' num2str(times{j}.ndv) ' (non-uniform design)'])
  else
    disp(['ndv = ' num2str(times{j}.ndv) ' (uniform design)'])
  end
  disp(['Size of system matrix: ' times{j}.sys_matrix_size])
  disp(['nnz % : ' sprintf('%1.1e',times{j}.sys_matrix_nnz)])
  disp(['Number of processors used: ' wstr])
  fla = times{j}.assembly;
  disp(['FEMLAB assembly: ' sprintf('%1.1e',fla) 's'])
  disp(['         Time [s]  Total [s]  Assembly/Total -%'])
  if call_matlab(j)
    bss = times{j}.BS_solution;
    disp(['MATLAB: ' sprintf('%1.1e %1.1e    %3.0f',[bss fla+bss fla*100/(fla+bss)])])
  end
  if call_spl(j)
    spls = times{j}.SPL_solution;
    disp(['SPL(' times{j}.SPL_ordering ')   : ' sprintf('%1.1e %1.1e    %3.0f',[spls fla+spls fla*100/(fla+spls)])])
  end
  if call_femlab(j)
    fls = times{j}.fl_solution;
    disp(['FEMLAB: ' sprintf('%1.1e %1.1e    %3.0f',[fls fls NaN])])
```

```
  end
  if call_femlab_femlin(j)
    fla  = times{j}.fl_lin_solution_a;
    flsl = times{j}.fl_lin_solution;
    disp(['FEMLAB: ' sprintf('%1.1e %1.1e     %3.0f',[flsl flsl+fla fla*100/(fla+flsl)])])
  end

end

end
diary off

% If run as a batch job: Include the exit command such that MATLAB is shut down
%
 exit
%
```