Overview of parFE: A Scalable Finite Element Solver for Bone Modeling

Uche Mennel, Marzio Sala
umennel@student.ethz.ch, marzio@inf.ethz.ch

September, 2006

Institute of Computational Science
ETH Zurich

Contents

1 Introduction

Osteoporotic fractures are a major cause of severe long-term pain and physical disability, and have an enormous impact on the individual, society and health care social systems. Osteoporosis is second only to cardiovascular disease as a leading health care problem. Since global parameters like bone density do not admit to predict the fracture risk, patients have to be treated in a more individual way. Today's approach consists of combining 3D high-resolution CT scans of individual bones with a micro-finite element (μFE) analysis.

With the advent of fast and powerful computers, simulation techniques are becoming popular for investigating the mechanical properties of bones, and predict the strength of a given patient's bones. Ideally, the development of a system with microstructural resolution better than 50 μm would allow in-vivo measurement of patients at different instances in time and at different anatomical sites. Unfortunately, such systems are not yet available, but the resolution at peripheral sites is approaching a level that allows elucidation of individual microstructural bone elements. Faster and more precise peripheral quantitative CT systems (pQCT) are now reaching the market, allowing for in-vivo patient measurements with an isotropic resolution better than 100 μm.

By elaborating the output of a pQCT scan, it is possible to reconstruct a computer image of the trabecular bone under investigation. This computer image eventually leads to a finite element mesh, that can be used to perform a 'virtual experiment', i.e., to simulate a mechanical test in great detail and with high precision.

This report describes the use of a scalable massive parallel μ-FE solver. It is intended to be used on biomechanical applications considering the $\mu-$FE analysis of biological tissue. The physical behavior of the bone tissue is here modeled by the linearized elasticity equations, for whose solution the FE method is a *de facto* standard. The bottleneck of every FE solver is the solution of a linear equation system

$$A\mathbf{u} = \mathbf{f}, \tag{1}$$

where $\boldsymbol{A}$ denotes the *global stiffness matrix.* The solution is obtained by applying the algebraic multigrid (AMG) preconditioned conjugate gradient (PCG) method to the linear system arising from the discretization.

Although the equations their discretization techniques are well-known, much has to be done to solve them efficiently for real-life problems, especially on parallel, distributed memory computers. Several issues arise. First, the application must be able to read large mesh files from disk, compute the desired solution and write it to disk again. Then, a solution technique for (1) must be determined.

To provide an answer to these problems, this report presents the PARFE software library. PARFE is a scalable finite element solver for bone modeling, developed by the Informatics and Biomedical deparments of ETH. A unique feature of PARFE is its flexibility. To meet the demand of most users, two solution techniques are supplied:

- **Global stiffness matrix assembly based PCG**, which assembles the stiffness matrix A, then computes a high-quality preconditioner. We use smoothed aggregation preconditioners, that have been proven to be scalable for this class of problems [**?**]. This approach is very fast, but consumes a lot of memory and might therefore restrict the size of input mesh.

- **Element by element PCG**, which computes the matrix-vector products required by PGC, as well as the preconditioner, without assembling the stiffness matrix. As such, it requires little memory with a modest performance penalty with respect to the globally assembly method. Note that PARFE uses a matrix-free multilevel preconditioner which is significantly faster than the well-known EBE preconditioner proposed about 10 years ago and now common [**?**, **?**].

This report is structured as follows. Section 2 quickly reviews the solution methodologies. Section 3 provides instructions to configure and build the application on various Unix based environments. As examples, the section details how to configure PARFE on a CRAY XT3 and on a Beowulf cluster. Basic usage of application is showed in Section 4, followed by a set of examples. ASCII and binary grid file formats are outlinesin Section 5. Visualization and analysis techniques are reported in Section 6. Few concluding remarks are reported in Section 7.

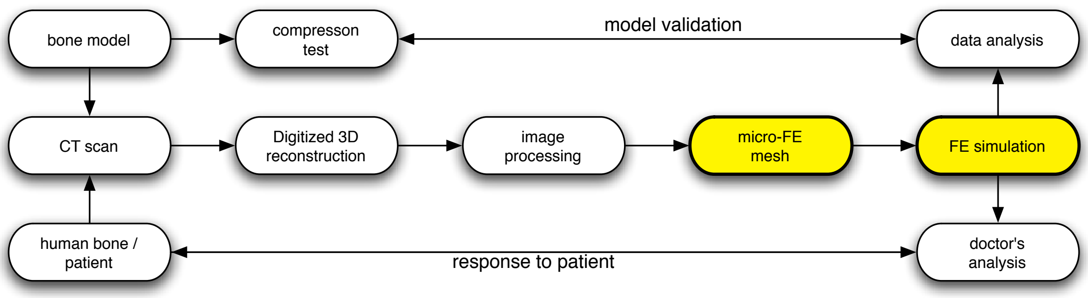

Figure 1: Procedure for the $\mu-$FE analysis. The PARFE code focuses on the the phases in yellow.

2 The Elasticity Equations and Their Solution

Figure 1 sketches the micro finite element analysis [?]. Starting from either a bone model or a human bone, micro computed tomography (CT) is employed to obtain 3D high-resolution images of the considered bone. For bone models, compression tests are often used to compute the maximal strength of the bone. For the applications considered in this report, the experimental analysis is then used as a validation tool for the finite element solver. However we do not consider the validation phase in this report, and we only focus on the phases in yellow (or light gray): the generation of the $\mu-$FE and the FE simulation itself. Both phases are based on the Lamé equations of elasticity [?], whose weak formulation reads

$$\left\{ \begin{array}{l} \text{Find } \mathbf{u} \in V \text{ such that} \\ \int\limits_{\Omega} [2\mu\, \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v}) + \lambda \ \div \mathbf{u} \ \div \mathbf{v}]\, d\Omega = \int\limits_{\Omega} \mathbf{f}^T \mathbf{v}\, d\Omega + \int\limits_{\Gamma_N} \mathbf{g}_S^T \mathbf{v}\, d\Gamma, \qquad \forall \mathbf{v} \in V, \end{array} \right. \tag{2}$$

where $\Omega \subset \mathbb{R}^3$ is the computational domain resulting from the image processing phase, $V \subset (H^1_{\Gamma_D}(\Omega))^3$ is a Sobolev space where the solution is sought, Γ_D and Γ_N are the Dirichlet and Neumann part of $\partial\Omega$, $\overline{\Gamma_D \cup \Gamma_N} = \overline{\partial\Omega}$, $\Gamma_D \cap \Gamma_N = \emptyset$, and $\mathbf{u}$ describes the nodal displacements, λ and μ are the Lamé constants, $\mathbf{f}$ the volume forces, and $\mathbf{g}$ the boundary tractions. The symmetric strains in (2) are given by

$$\varepsilon(\mathbf{u}) := \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^T).$$

The $\mu-$FE procedure is based on equally shaped micro finite elements, obtained by simply converting all bone voxels to equally sized 8-node brick elements. The finite element discretization of problem (2) on this domain by means of piecewise trilinear polynomials [?] leads to a the linear algebraic system (1), where A is symmetric, and positive definite as long as $\Gamma_D \neq \emptyset$.

μ-FE analyses are computationally demanding[1] and therefore equation (1) is solved with a preconditioned conjugate gradient method. It is well-known that the performances of PCG depends on the spectral properties of the linear system matrix, and in particular on its

[1]For example, a convergent linear analysis of one human vertebral body requires over 130 million elements with a resolution of 30 microns.

condition number $\kappa(A)$. For the unpreconditioned system, one has $\kappa(A) \approx \mathcal{O}(h^{-2})$, where h is the mesh size [**?**]. Therefore, the linear system

$$AB(B^{-1}\mathbf{u}) = \mathbf{f} \tag{3}$$

is solved instead of (2), where the symmetric and positive definite linear operator B is called the *preconditioner.* B is chosen so that $\kappa(A\,B) \ll \kappa(A)$. As such, PCG should converge faster when applied to (3) than to (1). The savings in iterations to converge should not offset for the additional CPU time required for the construction and application of B.

Defining a good preconditioner for a given problem is more art than science. For the considered bone modeling problems, common choices are the element-by-element (EBEP) preconditioner and a diagonal preconditioner (DP). Although cheap to construct and apply, EBEP and DP preconditioners do not effectively reduce the number of PCG iterations to converge. In fact, both preconditioners have only a minimal effect in reducing $\kappa(A)$. Furthermore, their performances degradate drastically with large jumps in the problem coefficients.

To overcome these problems, we here consider scalable *multilevel* (or *multigrid*) preconditioners. Multigrid methods were introduced in the late 70's, and their success and development is testified by the vast literature and the many international conferences organized since then, see [**?**] for a survey.

A multilevel method tries to approximate the original PDE problem of interest on a hierarchy of levels and use 'solutions' from coarse levels to accelerate the convergence on the finest level. When used to define preconditioners for linear systems, multigrid methods are applied using the so-called V-cycle.

The preconditioner for the solution of linear system (1) is applied by calling the procedure MultiLevelSolve($A, \mathbf{x}, \mathbf{b}, 0$) described in Figure 2. For a detailed description of all the operators, as well as the techniques required to build them, we refer to the specialized literature; see for example [**?**, **?**]. We also refer to [**?**, **?**] for results concerning aggregation preconditioners on up to thousands of processors. Here, we limit outselves to few parameters that can be modified to improve the performances of the preconditioner.

The first parameter we consider here is the **damping factor** ω. Aggregation preconditioners can be divided into non-smoothed aggregation (NSA) preconditioners and smoothed aggregation (SA) preconditioners. For elliptic problems as the one considered in this paper, SA performs significalty better [**?**] in terms of iterations to converge; however it requires an additional distributed matrix-matrix product, and produces more fill-in in the coarser-level matrices. NSA, instead, has a faster and less memory-demanding setup phase, but typically requires more iterations to converge. Although we suggest to use SA preconditioners, the optimal choice for a given problem or classes or problems may require a few numerical tests.

The second parameter we consider here is the **aggregate size**. Both NSA and SA methods operates by grouping together grid vertices in the so-called aggregates. The largest the number of vertices included in each aggregate, the faster the preconditioner, and typically the largest the number of iterations to converge. By increasing the number of vertices per aggregate, one can also reduce the memory requirements. parFE uses the ML package of Trilinos to define the multilevel preconditioner. ML offers several strategies to build the aggregates. Among them, we recall here the `Uncoupled` and `MIS` schemes [**?**], whose goal is the definition of aggregates of size 27 (in 3D), and the `METIS` and `ParMETIS` schemes, which can be used for arbitrary aggregate sizes. As general rule, we suggest to use `Uncoupled` for the finest 2 or 3 levels, then switch to the more performance `MIS`. If memory is an issue, then one should use

```
1.  procedure MultiLevelSolve($A_\ell$, $\mathbf{b}_\ell$, $\mathbf{x}_\ell$, $\ell$)
2.      if $\ell == L$ then
3.          $\mathbf{x}_\ell = A_\ell^{-1}\mathbf{b}_\ell$
4.      else
5.          $\mathbf{x}_\ell = S_\ell(\mathbf{b}_\ell, 0)$
6.          $\mathbf{r}_\ell = R_\ell\,(\mathbf{b}_\ell - A_\ell\mathbf{x}_\ell)$
7.          $\mathbf{v}_{\ell+1} = 0$
8.          MultiLevelSolve($A_{\ell+1}, \mathbf{r}_{\ell+1}, \mathbf{v}_{\ell+1}, \ell + 1$)
9.          $\mathbf{x}_\ell = \mathbf{x}_\ell + P_\ell\,\mathbf{v}_{\ell+1}$
10.         $\mathbf{x}_\ell = S_\ell(A_\ell, \mathbf{b}_\ell, \mathbf{x}_\ell)$
11.     endif
12. endprocedure
```

Figure 2: Typical multigrid preconditioner. In the procedure, $\ell = 0$ defines the finest level, the A_ℓ represents the discretization on level ℓ of the problem, P_ℓ and R_ℓ are prolongator and restriction operator from level $\ell + 1$ to ℓ, respectively, and the S_ℓ are approximate solvers (called smoothers).

`METIS` with an aggregate size of about 100 for the finest 2 or 3 levels, then switch to `ParMETIS` with the same aggregate size.

The third parameter is the **coarsest level size**. The stiffness matrix at the coarsest level is solved by a direct solver. parFE uses the Amesos package of Trilinos to interface with direct solvers. Currently, the KLU solver contained in Amesos is used for the solution. KLU is a simple though effective serial solver[2]. The default value used by parFE 1024, should suffice to most users.

The last parameter we consider here is the **smoother operator**, defined by the S_ℓ symbol in Figure 2. Common choices for the smoothers are Chebyshev polynomial smoothers, or processor-based symmetric Gauss-Seidel (SGS) [**?**]. If more powerful smoothers are required, one can increase the degree of the Chebyshev smoother or the number of SGS sweeps. Alternatively, it is possible to adopt processor-based incomplete Cholesky factorizations.

[2]It is not difficult to modify parFE to use other, parallel direct solvers, like SuperLU, SuperLU_DIST or MUMPS. However, in our experience KLU furnishes acceptable performances for the range of coarsest level sizes typically considered in parFE.

3 Configuring and Building PARFE

A configuration script is supplied with the application. It has been created with the GNU autotools. It will (try to) detect compilers and libraries available on your machine and set up things accordingly. Presently it is expected to work on most Linux PCs and clusters and the CRAY XT3. You may tune the configuration by specifying appropriate environment variables and/or command-line options. Note that a FORTRAN90 compiler is required to compile PARFE.

3.1 Required libraries

The implementation is based on the following sofware packages:

- **MPI** [?]. MPI is the *de facto* standard for distributed memory computations, and it is available for almost any parallel machines. The MPICH and LAM/MPI projects offer freely downloadable distributions of MPI.

- **Trilinos**, version 7.0 or higher [?, ?]. The Trilinos framework must be installed with the following packages enabled: AztecOO [?], Epetra [?], EpetraExt, IFPACK [?], ML [?], Amesos [?] and Teuchos. ML should also support the libraries METIS and ParMETIS described below.

- **METIS** and **ParMETIS** [?]. ParMETIS is a graph partitioning library, which can be freely downloaded from the web site of the authors.

- **HDF5** [?]. HDF5 is a portable, high-performance I/O library, developed at the National Center for Supercomputing Applications. HDF5 supports a simple but very flexible file format, which consists of two fundamental object types: datasets, which contains $n-$dimensional arrays of arbitrary data, and groups, which may be used to arrange related datasets in a hierarchical structure, similar to a Unix file system.

  Parallel I/O is performed by HDF5 by the so-called *hyperslabs*. For our applications, a hyperslab is simply a linear distribution of an array of data, whose chunks can be read in a distributed and concurrent manner by the processors involved in the computation. Once a distributed object, like a multivector, has been read from file in a linear distribution, it can be redistributed to match the desired data layout using the Import and Export classes of Epetra.

  If a parallel filesystem (e.g. RAID) is available, the HDF5 library should be installed with the MPI-IO functions enabled [?, ?]. The application can then take advantage of parallel disk I/O.

3.2 Licensing of PARFE

PARFE is released under the Lesser GNU Public License reported below.

Note that some parts of parFE are dependent on a third party code. Each third party code comes with its own copyright and/or licensing requirements. It is responsibility of the user to understand these requirements.

3.3 Obtaining parFE

The parFE sources are stored within an SVN repository. Let `SVNROOT` be a shell variable containing the access method, the name of the computer hosting the SVN repository, and the path of the SVN repository[3]. Then, to check out the latest version of the project and store them in the local directory `parfe`, one should type

```
> svn co $SVNROOT ./parfe
```

where '>' is the shell sign. To synchronize the actual directory with the repository, one should perform an update,

```
> cd parfe
> svn update
```

Finally, local changes can be committed to the repository with a check in:

```
> svn co -m"brief description of changes"
```

It is easy to add new, delete, or restore files, or print the log file related to given files:

```
> svn add new-file.tex
> svn delete bad-file.tex
> svn log a-file.tex
> svn diff file-with-changes.tex
```

SVN keeps track of all the previous versions, so that one can restore old versions as required. We will not cover here all the SVN capabilities, and we refer to [?] for more details. Note that the few commands introduced above should suffice to most parFE users.

[3]For example, one can have `ssh+svn://my.machine.ch/path/to/repository/parfe`.

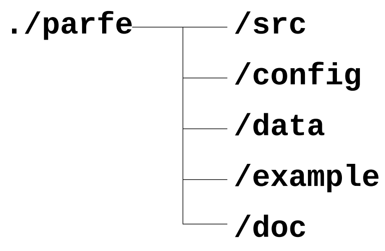

Figure 3: Directory structure of parFE.

3.4 Structure of parFE

The directory structure of parFE is reported in Figure 3. Directory `config` hosts files required by AutoConf and AutoMake (see next Section) and should be ignored by the user. Directory `src` contains the source files for the general-purpose finite element library distributed with parFE. Directory `example` contains the executable driver, called `parfe.exe`, which will be described in Section 4. Directory `data` contains few example files, while directory `doc` hosts the documentation of the distribution.

parFE uses Doxygen for all technical details that go beyond the scopes of this report. Doxygen is availabe at `http://www.stack.nl/~dimitri/doxygen/`. The online documentation will be enriched with graphics, if the the graphviz (`http://www.graphviz.org/library`) is found on the system. For example, function and method parameters, or class hierarchies are not reported here, and can be found in the Doxygen documentation, which can be generated as follows:

```
> make doxygen-doc
```

File `parfe/doc/html/index.html` is the entry point for the on-line documentation.

3.5 Unix installation

parFE takes advantage of AutoConf and AutoMake to ease the configuration, compilation and installation; as such, the user is not required to hack Makefile's in order to compile parFE. Instead, Makefile's are automatically generated starting from templated included in the distribution by calling `configure`. The shell script `configure` can be customized with several parameters. A complete list of configuration options can be consulted by typing

```
> ./configure --help
```

Common MPI implementations provide compiler wrappers called `mpicc` and `mpiCC` or `mpic++`. It is strongly recommanded to compile the application using these wrappers. To do so, the following command should suffice

```
> ./configure --with-mpi-compilers --with-cxxflags="-DLAM_BUILDING"
```

The above instruction assumes that HDF5 and Trilinos are installed in standard locations. It is also assumed that the `g95` FORTRAN compiled on the system as well. If you use any other FORTRAN 95 compiler, set the environment variable `FC` to this compiler. Depending on the HDF5 installation, it is also necessary to link against the compression z library. The z library can be added by appending `--with-ldflag="-lz"` to the commandline above. For many systems, setting the `LAM_BUILDING` preprocessor variable is not necessary.

3.5.1 Cray XT3 Supercomputer

The Cray XT3 operating system UNICOS/lc is designed to maintain remarkable scalability for large parallel applications. It consists of two primary components — a Catamount micro-kernel for compute PEs and a full-featured SuSE Linux for the service PEs. Compute node applications have to be cross-compiled on the service PEs in order run on the Catamount kernel. Several library paths are not searched by default. They have to be added explicitly to the configure command. Contact your system administrator to find out about the installed software. The following example shows the configure invocation for the Cray XT3 Supercomputer located at CSCS in Manno, TI.

```
> ./configure \
--prefix=$HOME/parfe-build \
--host=x86_64-unknown-linux-gnu \
--enable-static \
--with-ldflags="-L/apps/zlib-1.2.3/lib" \
--with-hdf5=/apps/hdf5-1.6.4 \
--with-trilinos=/users/msala/Trilinos/CRAY_XT3 \
--with-parmetis=/apps/metis/parmetis-3.1 \
--with-cxxflags="-DMPICH_IGNORE_CXX_SEEK"
```

The above script involves the newest Trilinos build located at the home directory of `msala`. The user is free to download the Trilinos[4] tarball from the web site `http://software.sandia.gov`, and install it in his or her directory by using the script `cray_xt3_marzio` contained in the `Trilinos/sampleScripts`.

3.5.2 The Linux Cluster Gonzales at ETHZ

Gonzales is a high-performance cluster of AMD processors, built by Dalco AG for the ETH Zurich. Service nodes run a full-featured SuSE Linux, while the compute nodes run a minimal installation of SuSE Linux. To configure, built and use parFE on gonzales, one can use the following configure invocation.

```
> ./configure \
--prefix=$HOME/parfe-build \
--with-mpi-compilers \
--with-parmetis=/home5/infk/umennel/ParMetis-3.1 \
--with-trilinos=/home5/infk/umennel/Trilinos \
--with-hdf5=/home1/infk/umennel/HDF5 \
--with-ldflags="-lz"
```

The installation process copies header files, libraries and executables in the directory specified with the `--prefix` parameter of `configure`. The default value is `/usr/local`.

This example involves the newest Trilinos, ParMETIS and HDF5 build located at the home directory of `umennel`. After succesfull configuration, the executable is built and installed with the usual sequence of commands

```
> make
> make install
```

[4]Trilinos 7.0 or higher is required by parFE.

which will copy the files in the installation directory specified by `--prefix`.

4 Getting Started

parFE comes as a general-purpose linear FE library, a suite of tools to manage fully parallel grid I/O, and a complete program to solve $\mu-$FE problems, called `pfaim.exe`. We now describe the usage of `pfaim.exe`, whose general syntax is

```
pfaim.exe [OPTIONS] file
```

where `file` is a HDF5 encoded file in the IBT format[5]; see Section 5 for more details. An example input file is included in the `data` subdirectory of the parFE distribution.

To get a list of the command line options, invoke the application with

```
pfaim.exe --help
```

This will print a brief overview of command line options, with some explanations. A more detailed explanation follows in Table 1.

Remark 1 *Important notice concerning Lustre filesystems: If you access the input of output files on a directory that is located on a Lustre filesystem, e.g. on the CRAY XT3, make sure to enable the striping of the files over all I/O nodes. This might substantially increase the I/O performance. Therefore, use the command*

```
lfs setstripe <directory> 1048576 -1 -1
```

for 1MB striping over all available I/O nodes.

We now report few examples of usage. The first example we consider here uses assembles the linear system matrix, then uses the ML preconditioner. The problem is stored in file `freetest.mesh.h5`, which is distributed within the parFE distribution:

```
pfaim.exe -p ml ../data/freetest
```

The second example converts the ASCII input file `freetest.mesh` to the HDF5 file `freetest.mesh.h5`:

```
pfaim.exe --toHDF5 ../data/freetest
```

The third example we consider here does not assemble the matrix, and uses the EBE approach to apply the linear system matrix. It also uses the multilevel preconditioner in the PCG solution:

```
pfaim.exe -p matrixfree ../data/freetest
```

where `matrixfree` enables the matrix-free preconditioner. The next two examples explain the how to launch jobs on the CRAY XT3 or on the Gonzales cluster. All applications created to run on compute nodes must be submitted to the batch system.

On the Cray XT3, the application has to be submitted by a parallel job launcher, which requests the necessary CPU resources from the MySQL database driven compute PE allocator (CPA). The following PBS script `freetest.pbs` shows the submission of a job running on 4 CPUs:

[5]Running the application in a parallel environment ususally requires the employment of a job launcher (e.g. `mpirun`). On many computer systems, a parallel application can only be run after submitting it to a batch job queue, like PBS or QSUB.

Table 1: Command line options to run `pfaim.exe`.

`-h, --help`	Print a small usage guide.
`-i, --maxiters=n`	Set the maximum number of solver iterations.
`-t, --tolerance=r`	Set the solver tolerance to determine the stopping criterion.
`-p, --precond=name`	Specify the preconditioner to be used by the solver. Choose one of `ml` (Multilevel), `ic[k]` Incomplete Cholesky with optional k = fill-in), `Jacobi`, and `matrixfree` (matrix-free multilevel preconditioner).
`--memory=quantity`	Specify memory consumption (One of: low/normal/high).
`-a, --ascii`	Read/Write ASCII encoded files, instead of HDF5.
`--norepart`	Don't apply load balancing. Proceed with the initial linear partition.
`--nosolve`	Do not solve the problem. Let the application exit after having assembled the stiffness matrix
`--printmedit`	Print mesh and displacements to be visualized with MEDIT; see Section 6.1.
`--printpmvis`	Print mesh partition to be visualized with PMVIS; see Section 6.2.
`--printmatrix`	Print out the matrix in sparse MATLAB format (HDF5 encoded); see Section 6.3
`--printrhs`	Print out the right hand side vector in MATLAB format (HDF5 encoded); see Section 6.3.
`--printlhs`	Print out the left hand side vector in MATLAB format (HDF5 encoded); see Section 6.3.
`--toASCII`	Convert to the provided problem file to ASCII encoding and exit.
`--toHDF5`	Convert to the provided problem file to HDF5 encoding and exit.
`-v, --verbose`	Will cause the application to produce more detailed console output.

```
#!/bin/sh
#PBS -N freetest
#PBS -l size=4,walltime=00:10:00
#PBS -j oe
#PBS -V
#PBS -q production

yod -small_pages -sz all pfaim.exe -p ml ../data/freetest
```

The job is finally submitted with

```
> qsub freetest.pbs
```

On Gonzales, batch jobs are managed - on different levels - by Platform load sharing facility (LSF) and Quadrics resource management system (RMS). A job can be directly submitted by just starting the job laucher with the specific commandline options. The following example shows the submisson of a job running on 4 CPUs:

```
bsub -J freetest -W 00:10 -o freetest.%J -n 4 prun pfaim.exe -p ml ../data/freetest
```

5 Grid File Formats

We now describe the file formats used by parFE. parFE can handle ASCII and binary grid files. ASCII files are human-readable and simple to generate. ASCII grid files have the `.mesh` ending, and they format is described in Sections 5.1. On the other hand, ASCII files are storage-demanding, slow to read and write because of the required conversion between the ASCII characters and the memory storage of integer and double numbers, and cannot be read and written in parallel. Therefore, ASCII file formats should be used for when experimenting or when running on a single-processor machine.

The binary format, based on the HDF5 library, is especially convenient when when dealing with large files. HDF5 grid files myst have the `.mesh.h5` extension; their format is described in Section 5.2.

5.1 ASCII encoding

The ASCII input file consists of a sequence of whitespace delimited entries. The enumeration below describes each entry. Note that if an entry is described with “scalar”, just a single word or number is requested. The descriptions “vector” or “matrix” just refers to a sequence of words or numbers delimited by whitespace characters. Thus the term “matrix” is actually redundant. It only used here to support the explanation of an entry. The bracketed words refer to a data type or the standard ASCII formatting in this case. A template of an ASCII grid file is reported in Figure 5.

1. scalar string quoted by ’s to indicate the element type. e.g. ’hexahedron’
2. scalar (Integer) Number of elements
3. scalar (Integer) Number of nodes
4. scalar (Integer) Number of Gaussian integration points to compute an element stiffness matrix.
5. scalar (Integer) Number of degrees of freedom per node.
6. scalar (Integer) Dimension of the stress-strain matrix $\boldsymbol{D}$
7. scalar (Integer) Number of spatial dimensions d. Usually $d = 3$.
8. vector (Real) d quantities to describe the spatial size of an element
9. scalar (Integer) Number of material properties.
10. scalar (Integer) Number of different materials.
11. matrix (Compound: 1st column: Integer, other columns: Real) A matrix with a row for each material, where the first row determines the material ID, the other columns contain the material properties. If there is only material type, the first column is omitted.
12. scalar (Real) Solver tolerance.
13. scalar (Integer) Number of maximum solver iterations.

14. vector (Real) Coordinates of the nodes. The coordinates of each node are stored consecutively.

15. vector (Integer) Node connectivity, elements of the mesh. The nodes making up an elements are stored consecutively. The node numbering must comply with the numbering scheme in figure 4.

16. vector (Integer) Material IDs of each element. This vector only exists if the number of materials is greater than one.

17. scalar (Integer) Number of nodes that have at least one degree of restrained, i.e. set to zero.

18. matrix (Integer) Each row of this matrix contains the node number in the first column. The other columns contain either a zero value if the corresponding degree of freedom is restrained or a one otherwise.

19. scalar (Integer) Number of nodes of the load vector bearing a non-zero load.

20. matrix (Compound: 1st column: Integer, other columns: Real) A matrix with a row for each loaded node, where the first column contains the node number, the other columns contain the components of the corresponding load.

21. scalar (Integer) Number of nodes that have at least one degree of freedom fixed to a non-zero value.

22. matrix (Compound: 1st column: Integer, 2nd column Integer, 3rd column Integer) A matrix, of which each row contains the node number in the 1st column, the number of the degree of freedom of that node in the 2nd column, and the fixed value in the last column.

23. scalar (Integer) Number of nodes that constitute the boundary at the bottom of the mesh.

24. vector (Integer) Node numbers of the bottom nodes of the mesh.

25. scalar (Integer) Number of nodes that constitute the boundary at the top of the mesh.

26. vector (Integer) Node numbers of the top nodes of the mesh.

27. scalar (Integer) Number of elements that constitute the boundary at the bottom of the mesh.

28. vector (Integer) Node numbers of the bottom elements of the mesh.

29. scalar (Integer) Number of elements that constitute the boundary at the top of the mesh.

30. vector (Integer) Node numbers of the top elements of the mesh.

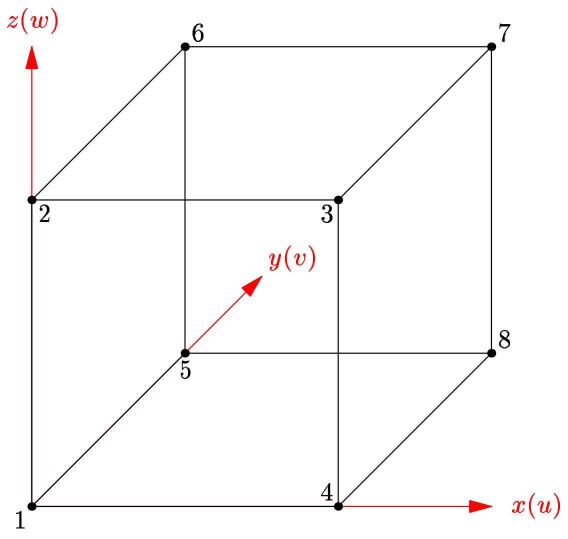

Figure 4: Numbering scheme of a 8-node hexahedral element.

```
# Begin model:
'hexahedron' 125 216   8   3   8   6   3
 0.034000 0.034000 0.034000
  2   1
5000.0 0.3
0.00001000 10000
# Nodes
    0.000   0.000   0.000
    <here all the other nodes>

# Elements
    1     37    38     2     7    43    44     8
    <here all the other elements, first vertex is 1>
# Boundary conditions
# Restrained displacements: node number, dofs (1=free, 0=restrained)
36
    181 0 0 0
    182 1 1 0
    <here other restrained displacements>
# Loads: values
0
# Fixed displacements: node number, sense, value
36
    1 3 1.70000e-03
    2 3 1.70000e-03
    <here other fixed displacements>
# Node and element sets
36
    1   2   3   4   5   6   7   8
    <here all the other sets>
```

Figure 5: Template of an ASCII grid file, extracted from the `parfe/data/freetest.mesh` file included in the distribution.

5.2 HDF5 encoding

In analogy, to the ASCII, also a HDF5 version of the input file has been designed. Each dataset belongs to a group. The term “scalar” describes a dataset that contains just one data point. The term “simple” refers to a dataset containing a regular N-dimensional array of data points. The data type between the brackets are explained in the HDF5 documentation [?]. The hierarchy of the datasets of this format with their names and contents is listed below:
A template of a HDF5 is found in the data directory. It can be viewed with

```
> h5dump data/freetest.mesh.h5
```

Group: **“Parameters”**

- scalar **“element type”** (H5T_STRING {
  STRSIZE 80;
  STRPAD H5T_STR_NULLTERM;
  CSET H5T_CSET_ASCII; CTYPE H5T_C_S1;
  })
  A string quoted by ’s to indicate the element type. e.g. ’hexahedron’
- scalar **“#elements”** (H5T_STD_I32LE) Number of elements.
- scalar **“#nodes”** Number of nodes.
- scalar **“#integration points”** (H5T_STD_I32LE) Number of Gaussian integration points to compute an element stiffness matrix.
- scalar **“#dofs per node”** (H5T_STD_I32LE) Number of degrees of freedom per node.
- scalar **“size of stress-strain matrix”** (H5T_STD_I32LE) Dimension of the stress-strain matrix $\boldsymbol{D}$
- scalar **“#dimensions”** (H5T_STD_I32LE) Number of spatial dimensions d. Usually $d = 3$.
- scalar **“aa”** (H5T_IEEE_F64LE) quantity to describe the spatial size of an element .
- scalar **“bb”** (H5T_IEEE_F64LE) quantity to describe the spatial size of an element.
- scalar **“cc”** (H5T_IEEE_F64LE) quantity to describe the spatial size of an element.
- scalar **“#material properties”** (H5T_STD_I32LE) Number of material properties.
- scalar **“#material types”** (H5T_STD_I32LE) Number of different materials.
- simple **“materials”** (H5T_COMPOUND {
  H5T_STD_I32LE ”ids”;
  H5T_ARRAY { [Number of material properties] H5T_IEEE_F64LE } ”properties”;
  })
  A matrix with a row for each material, where the first row determines the material ID, the other columns contain the material properties. If there is only material type, the first column is omitted.

- scalar **“tolerance”** (H5T_IEEE_F64LE) Solver tolerance.
- scalar **“iteration limit”** (H5T_STD_I32LE) Number of maximum solver iterations.

Group: **“Mesh”**

- simple **“Coordinates”** (H5T_IEEE_F64LE) Coordinates of the nodes. The coordinates of each node are stored consecutively.
- simple **“Elements”** (H5T_STD_I32LE) Node connectivity, elements of the mesh. The nodes making up an elements are stored consecutively. The node numbering must comply with the numbering scheme in figure 4.
- simple **“Material IDs”** (H5T_STD_I32LE) Material IDs of each element. This vector only exists if the number of materials is greater than one.

Group: **“Boundary Conditions”**

- scalar **“Restrained nodes size”** (H5T_STD_I32LE) Number of nodes that have at least one degree of freedom restrained, i.e. set to zero.
- simple **“Restrained nodes”** (H5T_COMPOUND {
  H5T_STD_I32LE ”Node number”;
  H5T_ARRAY { [Number of DOFs per node] H5T_STD_I32LE } ”Nodal freedom”;
  })
  Each row of this matrix contains the node number in the first column. The other columns contain either a zero value if the corresponding degree of freedom is restrained or a one otherwise.
- scalar **“Loaded nodes size”** (H5T_STD_I32LE) Number of nodes of the load vector bearing a non-zero load.
- simple **“Loaded nodes”** (H5T_COMPOUND {
  H5T_STD_I32LE ”Node number”;
  H5T_ARRAY { [Number of DOFs per node] H5T_STD_I32LE } ”Loads”;
  })
  A matrix with a row for each loaded node, where the first column contains the node number, the other columns contain the components of the corresponding load.
- scalar **“Fixed nodes size”** (H5T_STD_I32LE) Number of nodes that have at least one degree of freedom fixed to a non-zero value.
- simple **“Fixed nodes”** (H5T_COMPOUND {
  H5T_STD_I32LE ”Node number”;
  H5T_STD_I32LE ”Sense”;
  H5T_IEEE_F64LE ”Value”;
  })
  A matrix, of which each row contains the node number in the 1st column, the number of the degree of freedom of that node in the 2nd column, and the fixed value in the last column.

Note that the solution is written into the same file just by adding another group.
Group: **"Solution"**

- simple **"Nodal displacements"** (H5T_COMPOUND {
  H5T_STD_I32LE "Node";
  H5T_ARRAY { [Number of DOFs per node] H5T_IEEE_F64LE } "Ux Uy Uz";
  })
  A matrix with the node number in the first column and the resulting displacements in the other columns.

Figure 6: Template of an HDF5 grid file, extracted from the `parfe/data/freetest.mesh` file included in the distribution.

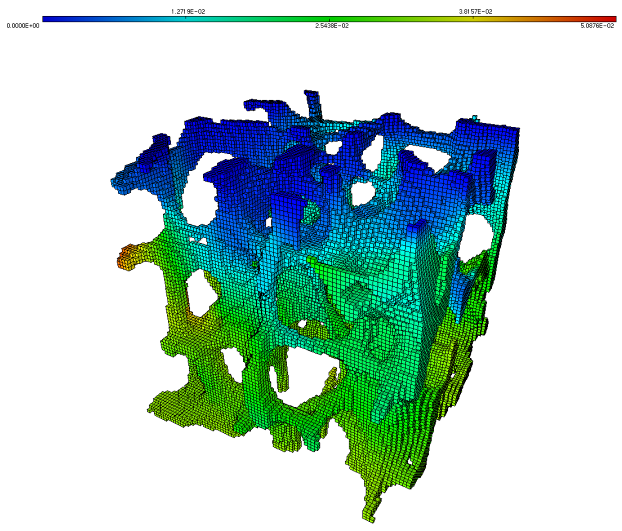

Figure 7: Visualization of displacements in stressed bone tissue with MEDIT.

6 Visualization and Analysis

parFE has been developed to take advantage of several visualization tools. These tools are serial, and they are based on ASCII file formats. As such, they should be used for debugging purposes, or to quickly visualize results corresponding to small- and medium-size problems.

6.1 Visualization of mesh and solution

MEDIT is a program for scientific visualization. It is designed to visualize results of computations involving 2D or 3D meshes. It is available at `http://www.ann.jussieu.fr/~frey/logiciels/medit.html`. By adding the commandline parameter `--printMDEIT`, parFE will generate a mesh file and, if computed, also the solution file containing the displacements. It is required by MEDIT, that the mesh file ends with '.mesh'. Therefore, the name of the generated mesh file output is `<filename>.medit.mesh`, where `<filename>` denotes the name of the file containing the parFEinput problem (without the endings). The corresponding solution file has the same name execept for the ending. It is '.medit.bb'. However, it is possible to view the mesh without any solution file given. To invoke MEDIT to visualize the mesh, enter

```
> medit <mesh file>
```

If there exist a solution file, it will automatically read and included in the visualization. An example of visualizing the displacements of a cubic bone tissue model is depicted in Figure 7.

6.2 Visualization of partitioning

PMVIS (Partitioned Mesh Visualizer) is a program to view partitioned, unstructured meshes, consisting of triangular, quadrilateral, tetrahedral, or hexahedral elements. It is available at `http://www-users.cs.umn.edu/~oztekin/pmvis/`. It requires 3 input files: a coordinates file, a connectivity file and a partition file. By adding the commandline parameter

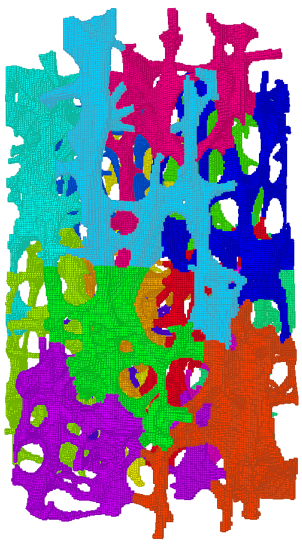

Figure 8: Visualization of 16 mesh partitones with PMVIS.

`--printPMVIS`, parFE will generate the following 3 files: `<filename>.xyz`, `<filename>.c` and `<filename>.p`, where `<filename>` denotes the problem file given as input for parFE. The endings of these files represent the 3 different kinds of PMVIS input files mentioned above (in the same order). To invoke PMVIS to visualize the partitioned mesh, enter

```
> pmvis -n <coordintes file> -c <connectivity file> \
        -p <partition file> -g hex -o 1
```

Figure 8 shows an example of visualizing a 16-way partition generated with PMVIS.

Remark 2 *By adding the commandline option* `--norepart` *to* parFE*, it is possible to visualize the initial (linearly distributed) partitions.*

.

6.3 Matrix and vector analyis

MATLAB is a numerical computing environment and programming language created by The MathWorks. parFE can write vectors and matrices in a MATLAB-compatible format. The matrices or vectors are stored using the MATLAB sparse format. Besides ASCII encoding, also HDF5 is encoding is supported. The HDF5 data organization is explained below.
The output of matrix and vector data is activated with the (independent) commandline options `--printmatrix`, `--printlhs`, and `--printrhs`. They correspond to the routines printing the assembled global stiffness matrix A, the left-hand-side (LHS) vector $\mathbf{u}$ and the right-hand-side (RHS) vector $\mathbf{f}$ of the linear system

$$A\mathbf{u} = \mathbf{f}. \tag{4}$$

By adding these commandline parameters, parFE will generate the following files: `<filename>.matrix.h5` for the matrix, `<filename>.lhs.h5` for the LHS vector and `<filename>.rhs.h5` for the RHS

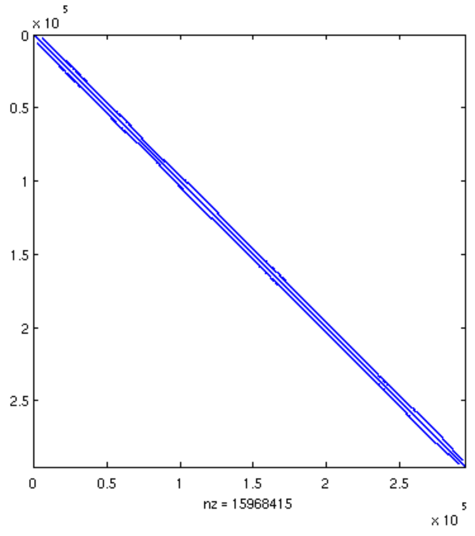

Figure 9: Visualization of the non-zero pattern of the global stiffnes matrix. Generated with MATLAB's 'spy'.

vector, where `<filename>` corresponds to the initial problem file given as input for parFE. Vector data written to a file just consists of a contiguous array of vector values. Matrix data is stored in the MATLAB sparse format. Three columns are written to the file: The first two columns contain the row and column indices while the third contains the corresponding values. In ASCII encoding, all data entities are separated by whitespace characters. If using HDF5 encoding, the format of the dataset comprises a simple dataspace called "Vector" of type `H5T_IEEE_F64LE` for vectors. For matrices, the format is a simple dataspace called "Matrix" of type:

```
H5T_COMPOUND {
  H5T_STD_I32LE "Row indices";
  H5T_STD_I32LE "Column indices";
  H5T_IEEE_F64LE "Values";
}
```

If using HDF5 I/O, enter the MATLAB command 'hdf5read' to read the matrix:

```
>> data = hdf5read(<filename>, "Matrix");
```

and to read a vector, enter

```
>> data = hdf5read(<filename>, "Vector");
```

If using just ASCII I/O, enter the command 'load'. Note that an ASCII file must not end with '.mat'.

```
>> data = load(<filename>);
```

An example of visualizing the matrix non-zero pattern is given in Figure 9.

Remark 3 *When dealing with large files, I/O performance is significantly higher with HDF5 encoded files. If really ASCII files are desired, they can be forced by adding the commandline option* `--ascii`*. In this case the file endings '.h5' will not be added to the output files.*

7 Concluding Remarks

We have presented the parFE library, which allows scalable solutions of linear elasticity problems arising from the micro finite element discretizations of bone modeling problems. The most notable capabilities of parFE are fully parallel binary I/O and the usage of scalable multilevel preconditioners based on smoothed aggregation, for both assembled and unassembled matrices. These capabilities make parFE far more efficient than other implementations, based on serial element-by-element preconditioning.

parFE can be easily extended to support other visualization format and incorporate nonlinear material properties. It is easy to add pre- and post-processing operations like grid smoothing, or analysis of the strain distribution. It can also be extended to perform time-dependent analysis, since it already contains all the ingredients to compute the mass matrix term.

Acknowledgments

We want to thank Prof. Peter Arbenz for his suggestions and guidance, and Harry van Lenthe and Prof. Ralph Müller for a first working version of the code, the test problems, and the fruitful discussions. parFE is based on several public-domain software projects: the Epetra, AztecOO, Amesos and ML projects of Trilinos, METIS and ParMETIS, and HDF5. The developers of these packages are acknowledged for their work.