A User Friendly Toolbox for Parallel PDE-Solvers *

Manfred Liebmann
Institute for Analysis and Computational Mathematics
Johannes Kepler University Linz
manfred.liebmann@uni-graz.at

April 24, 2006

Abstract

The parallelization of numerical algorithms is required to run large scale simulations on clusters of high performance servers. This paper presents a detailed approach parallelizing simulations based on the finite element method. The focus will be on the design of the communicator class, which provides an easy to use interface to all communication functions, that are required for the implementation of parallel linear algebra routines. Benchmarks for a parallel conjugate gradient algorithm on different cluster machines with Infiniband and Gigabit networking validate the parallelization approach.

1 Introduction

The aim of the project is to provide a user friendly C++ toolbox for the parallelization of partial differential equation solvers based on the finite element method. The finite element approach is very flexible in capturing complicated geometries of the simulation domain using simple geometric shapes, the *elements*, but it also provides a natural parallelization approach by evenly distributing the elements on the processing nodes, involved in the parallel computation. To minimize the communication between the processing nodes, a partitioning strategy is necessary that minimizes the number of boundary nodes, that are shared by multiple processors. This can be achieved using partitioning tools like METIS [1], that automatically create optimal partitions based on the mesh connectivity information.
With the partitioning information of the elements, a one-to-one mapping of elements to processors, and the mesh connectivity information it is possible to derive the complete communication setup for parallel algorithms. Specifically, to construct parallel linear algebra routines, it is necessary to communicate information that is located on the shared boundary nodes. To make this communication transparent for the user of the toolbox a communicator object is created, that provides three easy to use communication functions *accumulate*, *distribute*, and *collect*. The constructor of the communicator object needs only the global node numbers, that are located on the current processing node. After construction all the relevant

*The work described in this paper is partially supported by the Austrian Grid Project, funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ 4003/2-VI/4c/2004.

operations in parallel linear algebra can be constructed using the three basic communication functions. Chapter 2 gives details on how to use the communicator object and Chapter 3 gives some insight on the algorithms used to construct the internal data structures of the communicator object. Chapter 4 discusses the extensible binary data format, that is used to store all data, necessary for the finite element computation. Chapter 5 gives a detailed example implementing a parallel conjugate gradient algorithm with preconditioning and including some benchmark results validating the parallelization approach.

2 Communicator Class

The idea of the communicator object is to provide a single object to the user of the toolbox, that can handle all communication, that is required for the parallelization of linear algebra operations.

2.1 Class Definition

The communicator class is defined as a C++ template class. The template arguments are the data type of the node index, typically *int* and the data type of the matrix and vector entries, typically *double*.

```
communicator<int, double> com(con);
```

The code snippet above shows a typical instantiation of the communicator class. The constructor of the *com* object takes a vector of global node numbers *con*, derived from the connectivity information, as input. Vectors of numerical values or node numbers are represented in the toolbox using the C++ Standard Template Library (STL) [7] *vector* template class.

2.2 Accumulated and Distributed Vectors

For parallel computations it is important to distinguish between accumulated and distributed vectors. This distinction concerns only the numerical values of the vector entries and not it's size. Two of the three primary procedures of the communicator class implement the conversion of a distributed vector to an accumulated vector and vice versa.
The procedure *accumulate* takes a distributed vector as input and converts it *in place* to an accumulated vector. The accumulation process requires communication between the processors participating in the parallel computation and is thus an expensive operation.

```
com.accumulate(b);
```

The code snippet implements the accumulation process on the vector *b* using the communicator object *com*.
The procedure *distribute* takes an accumulated vector as input and converts it *in place* to a distributed vector. The distribution process is local and does not require any communication between processors.

```
com.distribute(b);
```

The code snippet implements the distribution process on the vector *b* using the communicator object *com*

2.3 Parallel Scalar Product

The third procedure of the communicator class, *collect*, implements the distribution of numerical data values to all processors, to calculate sums over the data values. The procedure is typically used in the parallel calculation of scalar products, to combine the locally computed partial sums into the final answer.
The parallel scalar product requires some attention on the vector types used in the sequential scalar product routines. For the result to be correct after calling the *collect* procedure, one vector in the scalar product has to be distributed, the other accumulated. If this is not the case one or the other has to be converted using the *accumulate* or *distribute* procedure.

```
scalar_product(x, y, alpha);
com.collect(alpha);
```

The code above implements a parallel scalar product using a distributed vector *x* and an accumulated vector *y* in a sequential scalar product routine. The result of the local scalar product is stored in *alpha* and used as input to the *collect* procedure. On return the output parameter *alpha* contains the full scalar product.

3 Algorithmic Details

Communicating the boundary information between different computational nodes, see Figure 1 and Figure 2, requires the setup of various data structures inside the communicator class. One major goal in the design of the communicator class is scalability of the communication setup. With this goal in mind only algorithms of almost linear complexity are used inside the communicator class. The description is partitioned into algorithms required for deriving the data structures for the accumulation process and for the derivation of the distribution process.

3.1 Sorting Algorithms

One interesting point turned out to be the fact, that most algorithmic problems in the communicator setup can be solved using integer sorting algorithms. Although the Standard Template Library (STL) provides a powerful sort function, a binary integer sorting technique is used, based on the radix exchange sort [2], that achieves higher performance for the test examples, than the STL sort function. The binary integer sorting technique also has the advantage, that it works *in place* and does not require any temporary storage, except for some stack space, that is less than 4 KB for all relevant situations. Aside from

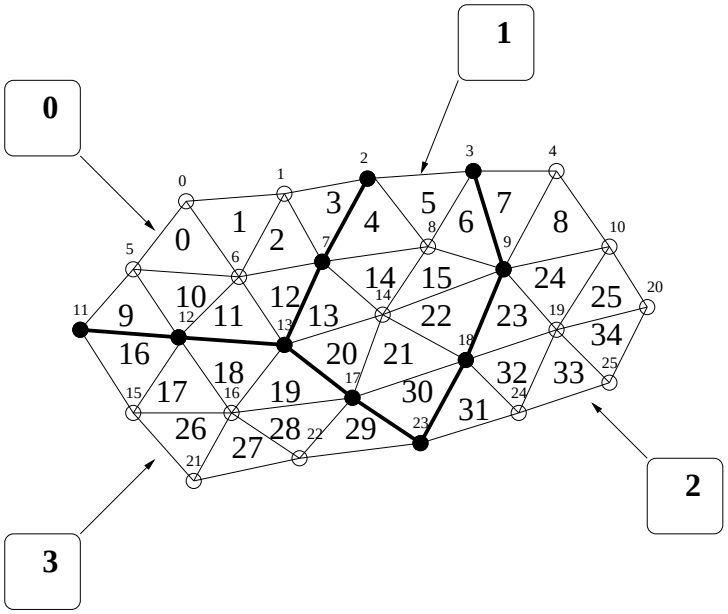

Figure 1: Finite element mesh distributed to four processing nodes with global element and node numbers.

0	0	0	0	1	1	1	2	2	0	0	0	0	1	1	1	3	3	3	3	1	1	1	2	2	2	3	3	3	3	1	2	2	2	2

Figure 2: Partition vector for four processing nodes mapping every finite element to the associated processing node.

0	0	6	5	1	6	0	1	7	6	2	7	1	5	12	11	6	12	5	6	13	12	7	13	6										
1	2	8	7	3	8	2	3	9	8	7	14	13	8	14	7	9	14	8	14	17	13	18	17	14	9	18	14	18	23	17				
2	4	9	3	10	9	4	19	18	9	10	19	9	20	19	10	24	23	18	19	24	18	25	24	19	20	25	19							
3	12	15	11	16	15	12	13	16	12	17	16	13	16	21	15	22	21	16	17	22	16	23	22	17										

Figure 3: Mesh connectivity information for all four processing nodes. Every triple of global node numbers defines a triangle.

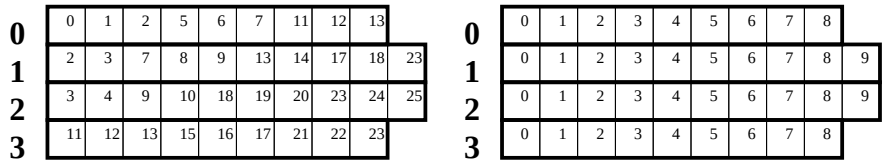

Figure 4: Union of global node numbers over the mesh connectivity information on the left and the corresponding local node numbers on the right.

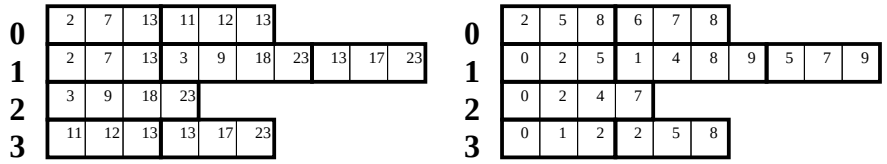

Figure 5: Global intersection data structure using global node numbers on the left and the corresponding local node numbers on the right.

sorting an integer sequence in ascending order, also in place Hilbert-order sorting is naturally supported by the binary sorting technique, which can be used for a cache optimized matrix-multiplication scheme [3].
The next two sections discuss the algorithms in the constructor of the communicator class.

3.2 Accumulation

The single input to the constructor is a vector that represents the mapping of the local to the global node numbers, see Figure 4. This vector is typically calculated as the union of the mesh connectivity information on the current processor, see Figure 3. The ordering of the global node numbers is arbitrary, but the mapping from local to global numbers must be one-to-one. This is the only restriction on the input vector.
The setup for the accumulation process starts with sorting the global node numbers in ascending order. The permutation introduced in the local to global mapping is recorded in a temporary vector of local node numbers. This sorting is required to have a unique definition of node ordering on every processor. After the sorting process of the global nodes, every

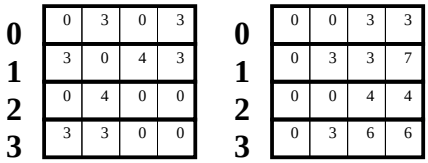

Figure 6: Count vectors on the left and displacement vectors on the right for the global intersection data structure using four processing nodes.

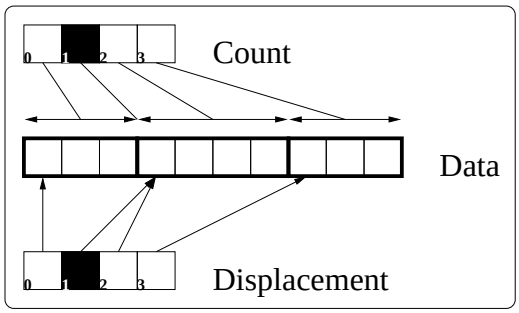

Figure 7: Complete global intersection data structure for processing node 1. On top the count vector, in the middle the data vector and at the bottom the displacement vector

processor sends out his ordered sequence of global node numbers to every other processor participating in the parallel computation. After the completion of this communication step, every processor has collected all sets of global node numbers. To calculate the shared global nodes, every processor intersects his ordered set of global nodes, with the sets received from the other processors. If there are n processors, then every processor calculates n intersections. Globally n^2 different intersections are calculated, thus the computational load is evenly distributed. After the intersection process is complete the intersecting global nodes, these are the nodes, that have to be exchanged between a pair of processors, are mapped back to their local node numbers and stored in a flexible data structure, see Figure 7, using the three vectors *_rcom*, the data vector, *_rcnt*, the count vector and *_rdsp*, the displacement vector. The complete data structures for the global intersection are depicted in Figure 5 and Figure 6.

```
vector<T> _rcom;
vector<T> _rcnt;
vector<T> _rdsp;
```

- *_rcom* stores all the local node numbers that are used in the communication with n processors.
- *_rcnt* stores the number of local nodes to communicate with processor number $i \in 1, \ldots, n$.
- *_rdsp* stores the displacements for accessing the local nodes for processor number $i \in 1, \ldots, n$.

The flexible data structure is directly used for packing the local node values into the send buffer, for the data exchange using an all-to-all communication call [6], see Figure 8, and for unpacking the receive buffer and accumulation back to the local node values.

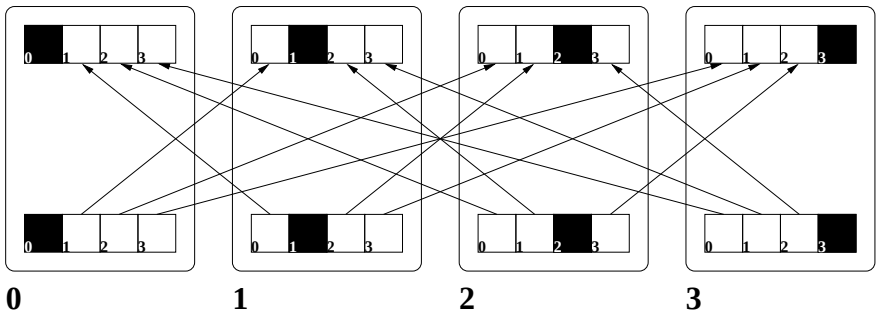

Figure 8: All-to-all communication pattern for the accumulation process using four processing nodes.

```
vector<S> _rvec;
vector<S> _svec;
```

- *_rvec* is the communication receive buffer for the node values to be received.
- *_svec* is the communication send buffer for the node values to be sent out.

3.3 Distribution

The setup of the distribution process requires the number of processors, that share a single global node. Only the actually present global nodes on a processor are relevant for the calculation. The calculation can be done using a modified union function on the vector of all shared local nodes, i.e. *_rcom*, that also records the multiplicity of the elements in the union. Since the locally present nodes are not included in the *_rcom* vector, the calculated multiplicity is off by one. This can be easily fixed, and the reciprocal of the node multiplicity is stored in the vector *_rsca* with the associated local node number stored in *_rpos*.

```
vector<T> _rpos;
vector<S> _rsca;
```

- *_rpos* stores the local node numbers, that have to be scaled in the distribution process.
- *_rsca* stores the reciprocal multiplicities, the scaling factors for the distribution process.

3.4 Collection

The collection process requires the *_rcol* vector as receive buffer in the *collect* procedure. The *collect* procedure is implemented using one-to-all communication for data distribution and a simple local accumulation of the data values, to calculate the sums. No additional setup is required.

```
vector<S> _rcol;
```

- *_rcol* is the communication receive buffer for the *collect* procedure.

3.5 Formal Description

This section gives a formal description of the algorithms above. Most of the algorithmic complexity can be captured using formal set operations, like unions and intersections of node sets. To implement set operations efficiently it is required to have sorted node vectors, typically in ascending order. With this prerequisite set union and intersection can be implemented with linear complexity, accessing every node only once and in sequential order. Furthermore the algorithms can be executed in place, not requiring any additional memory.

3.5.1 Accumulation

Let

$$G^p = (g_i^p)_{i=1}^{N^p} \tag{1}$$

be the sequence of global node numbers located on the processor $p \in \{1, \ldots, P\}$. P is the total number of processors in the parallel computation and N^p is the number of nodes located on processor p. Define a permutation

$$\Sigma^p = \begin{pmatrix} 1 & 2 & \ldots & N^p \\ \sigma^p(1) & \sigma^p(2) & \ldots & \sigma^p(N^p) \end{pmatrix} \tag{2}$$

such that the sequence

$$\hat{G}^p = (g^p_{\sigma^p(1)}, \ldots, g^p_{\sigma^p(N^p)}) \tag{3}$$

is in ascending order. Define the concatenation of all node sequences

$$\hat{G} = (\hat{G}^1, \ldots, \hat{G}^P) \tag{4}$$

For every processor $p \in \{1, \ldots, P\}$ define the P subsequences

$$\tilde{G}^{p,q} = \hat{G}^p \cap \hat{G}^q = (g^p_{\sigma^p(\chi^{p,q}(1))}, \ldots, g^p_{\sigma^p(\chi^{p,q}(M^{p,q}))}) \tag{5}$$

with $q \in \{1, \ldots, P\}$. Here the intersection symbol means set intersection, without changing the order in the sequence of the nodes. The function $\chi^{p,q} : \{1, \ldots, M^{p,q}\} \rightarrow \{1, \ldots, N^p\}$ defines a map from the local node number in the intersecting sequence $\tilde{G}^{p,q}$ to the local node number in the original sequence $\hat{G}^p$ The sequence of local intersecting nodes on the processor p is defined as

$$L^{p,q} = (\sigma^p \circ \chi^{p,q}(1), \ldots, \sigma^p \circ \chi^{p,q}(M^{p,q})) \tag{6}$$

These sequences define the local node numbers, that have to be communicated in the accumulation process. Processor p sends the values of the local nodes $L^{p,q}$ to all other processors $q \neq p$. The received node values are finally accumulated to the local values using the same node sets $L^{p,q}$, due to the symmetry of the communication process.

3.5.2 Distribution

Let

$$\alpha_s(r) := \begin{cases} 1, & r = s \\ 0, & r \neq s \end{cases} \tag{7}$$

then the multiplicity of the local node $s \in \{1, \ldots, N^p\}$ on processor $p \in \{1, \ldots, P\}$ is

$$C_s^p = \sum_{q=1}^{P} \sum_{r=1}^{M^{p,q}} \alpha_s(L_r^{p,q}) \tag{8}$$

In the distribution process the value of the local node s on processor p is divided by C_s^p.

4 Extensible Binary Data Format

In this section the Extensible Binary Data Format (EBDF) will be introduced. Finite element computations require a variety of different data structures. For example element connectivity information, element to processor mappings for different numbers of processors, coordinate data for the geometric nodes, info for Dirichlet nodes and values, an array of all element matrices, and data for various initial and solution vectors.
Often input data is stored in text files, but parsing textual information is an expensive operation. This performance penalty is especially relevant for large data files. Binary data files on the other hand provide high performance access to stored data. But it is necessary to carefully design the file format to enable painless data creation and transformation from other formats using software packages like *Mathematica* and *Matlab*. In the parallel toolbox all binary data files are stored in little-endian format.

4.1 Folder Concept

For an extensible data format it is necessary to have a container, that stores all data chunks. Traditionally a single file is used for this purpose, but this requires specialized reading and writing routines. To avoid this complexity in the parallel toolbox the container is a single folder or directory.

4.2 Block Data

With the folder concept every piece of data of the finite element simulation can be stored in a separate file. Now to have a unified concept for the storage of the pieces of data, only arrays of a single data type are allowed in the data files. This is very convenient for reading and writing the data, because the whole binary data block on the disk can be directly mapped to an array or vector in memory. For this purpose the parallel toolbox provides the routines *binary_read* and *binary_write*. Specialized routines for reading binary input files in parallel are also provided for some data structures. As a further requirement all vectors are C-style 0-based arrays. Index counting in the parallel toolbox always starts at 0!

4.3 File Structure

Information listed below gives a detailed description of files, that are found in a typical finite element simulations.

4.3.1 Header.bin

```
vector<int> hdr(8);
```

The header file stores common values for the finite element data structures.

```
int magic = hdr[0];
int version = hdr[1];
int gnumelem = hdr[2];
int elemsize = hdr[3];
int gnumnode = hdr[4];
int nodesize = hdr[5];
int gnumdir = hdr[6];
int dirsize = hdr[7];
```

The size of other data structures can be determined using the values in the header file. A detailed description of the vales is listed below.

- *magic* is a magic number for file identification.
- *version* is the version of the header file.
- *gnumelem* is the global number of elements.
- *elemsize* is the number of geometric nodes per element for a uniform mesh.
- *gnumnode* is the global number of geometric nodes.
- *nodesize* is the dimension of the associated coordinate tuple of a node.
- *gnumdir* is the global number of Dirichlet nodes.
- *dirsize* is the dimension of the associated Dirichlet values.

4.3.2 Connection.bin

```
vector<int> con(gnumelem * elemsize);
```

The mesh connectivity information consists of *elemsize*-tuples of global node numbers for all elements.

4.3.3 Partition$p$.bin

```
vector<int> par(gnumelem);
```

The partition information represents the mapping of an element to a processor. The appended number p in the filename denotes the number of processors, for which the partition was created. This information is typically generated by the mesh partitioning tool METIS.

4.3.4 Element.bin

```
vector<double> ele(gnumelem * elemsize * elemsize);
```

The element matrices are stored as a sequence of small matrices of dimension *elemsize*.

4.3.5 Type.bin

```
vector<int> typ(gnumelem);
```

Material information for an element.

4.3.6 VectorB.bin

```
vector<double> b(gnumnode);
```

Right hand side for the finite element calculation.

4.3.7 VectorX.bin

```
vector<double> x(gnumnode);
```

Initial value for the finite element calculation.

4.3.8 Coordinate.bin

```
vector<double> coo(gnumnode * nodesize);
```

Coordinate tuples for the global nodes.

4.3.9 Dirichlet.bin

```
vector<int> dir(gnumdir);
```

Global node numbers for Dirichlet nodes.

4.3.10 Values.bin

```
vector<double> val(gnumdir * dirsize);
```

An array of multidimensional Dirichlet values.

5 Parallel Conjugate Gradient Algorithm

To validate the usability of the communication class, a parallel conjugate gradient (CG) algorithm with diagonal preconditioning [5] is used as a test case. As a performance optimization the algorithm iterates on packed vectors, calculating the solution simultaneously for multiple right hand sides.

5.1 Numerical Results

The finite element matrix for the example is derived from a 3D bunny heart simulation [4]. The overall size of the problem is 862515 nodes and 5082272 elements. The right hand sides for the calculation are taken from the heart simulation. As a benchmark setup 512 CG iterations with a packed vector representing four right hand sides are used.
The MegaFLOP rates are calculated only counting the operations for the matrix vector product in the CG loop. One multiplication and one addition is counted for every non-zero matrix element. Scalar products and the diagonal preconditioner are not included in the instruction count.
Benchmark runs were carried out on the Opteron clusters *kepler*, *pregl*, and *archimedes*. The *kepler* cluster uses an Infiniband interconnect and the two other Gigabit Ethernet networks. See Table 1 for the cluster node configuration.

Cluster	CPU	RAM	Network
kepler	2x Opteron 248 2,2GHz	4GB DDR	1x Infiniband Interconnect
pregl	2x Opteron 248 2,2GHz	4GB DDR	1x Gigabit Ethernet
archimedes	2x Opteron 250 2,4GHz	4GB DDR	2x Gigabit Ethernet

Table 1: Cluster node configuration

NP	kepler	pregl	archimedes
1	222.42	225.224	308.17
2	219.64	193.56	253.62
4	202.32	190.45	241.59
8	198.93	79.29	224.50
16	194.85	35.90	163.65
32	159.43	18.96	140.55

Table 2: MFLOP rates for cluster computers

The benchmark data was collected using the Sun N1 Grid Engine (SGE) [9] batch system on the cluster computers, see Table 2. The SGE handles the resource allocation on the cluster for different users. Although the computational resources are efficiently assigned, the benchmark indicates that the allocation of the network resources is not optimal. This results in a serious performance degradation observed on the *pregl* cluster.

For 32 processors the *kepler* cluster with Infiniband interconnect is 8.5 times faster, than the Gigabit Ethernet machine *pregl*. In this case Gigabit Ethernet provides only for up to four processors reasonable performance scaling, for more processors there is a complete stagnation.
The Infiniband machine only loses about 12.5% from the peak performance of one processor up to sixteen processors. With 32 processors the relative performance is still about 72% of the peak performance of a single processor. In comparison 32 processors on the *pregl* cluster deliver only about 8.5% of the single processor peak performance.
That the performance degradation is not solely caused by the Gigabit Ethernet network shows the benchmark on the *archimedes* cluster. For eight processors the machine loses about 27% from the single processor peak performance. Even with 32 processors it still delivers 45% of the single processor peak performance.

5.2 Communicator Setup

The setup time for the communicator, that is the time for the construction of the communicator object, is measured for $1 - 32$ processors. The timings are in milliseconds.

NP	kepler	pregl	archimedes
1	55.9	70.3	54.7
2	47.6	78.1	66.4
4	35.5	87.9	86.9
8	29.9	98.6	91.3
16	27.4	102.8	95.2
32	34.5	676.6	622.9

Table 3: Timing for the construction of the communicator object in milliseconds.

The setup timings show a clear advantage for the Infiniband interconnect on the *kepler* cluster. The timings for the Gigabit Ethernet clusters are consistent considering the 10% performance advantage of the *archimedes* cluster over the *pregl* cluster.

6 Future Work

The parallel toolkit introduces a simple and effective framework for parallelization of basic linear algebra routines. A conjugate gradient algorithm with simple diagonal preconditioning has been implemented as a simple performance benchmark. To make the toolkit more useful more advanced preconditioners like algebraic multigrid will be investigated in the future.

References

[1] K. Schloegel, G. Karypis, V. Kumar, *Parallel static and dynamic multi-constraint graph partitioning* Concurrency and Computation: Practice and Experience. Volume 14, Issue 3, pages 219-240, 2002.

[2] P. Hildebrandt, H. Isbitz, *Radix exchange-an internal sorting method for digital computers*, JACM 6, pp. 156-163 (1959).

[3] G. Haase, M. Liebmann, G. Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, Technical Report IMA03-05, University of Graz, Institute for Mathematics and Scientific Computing, 2005.

[4] G. Plank, M. Liebmann, R. Weber dos Santos, E.J. Vigmond, G. Haase, *Algebraic Multigrid Preconditioner for the Cardiac Bidomain Model*, IEEE Transactions on Biomedical Engineering (to appear), 2006.

[5] C.C. Douglas, G. Haase, U. Langer, *A Tutorial on Elliptic PDE Solvers and Their Parallelization*, SIAM, Philadelphia (2003).

[6] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, Cambridge (1999).

[7] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley (2000)

[8] Parallel toolbox home page and source code. *http://paralleltoolbox.sourceforge.net/*

[9] Sun N1 Grid Engine. *http://www.sun.com/software/gridware/*