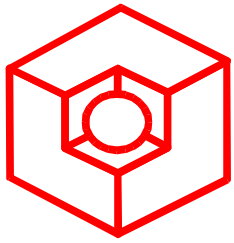
BremHLR
Competence Center of
High Performance Computing Bremen



FoSSI

The Family of Simplified Solver Interfaces for parallel sparse solvers in numerical atmosphere and ocean modeling

Stephan Frickenhaus*
Wolfgang Hiller‡, Meike Best‡

BremHLR Technical Report 01 - 03

* BremHLR, University of Bremen, FB 03 / ZeTeM
‡ Alfred-Wegener-Institute for Polar and Marine Research, Bremerhaven

FoSSI: The Family of Simplified Solver Interfaces for parallel sparse solvers in numerical atmosphere and ocean modeling

Stephan Frickenhaus [a,b,*] Wolfgang Hiller [b] Meike Best [b,1]

[a] *Competence Center of High Performance Computing Bremen BremHLR*

[b] *Alfred-Wegener-Institute for Polar and Marine Research, Bremerhaven, Germany*

Abstract

The portable software system FoSSI is presented that allows for an efficient and scalable parallel solution of large sparse linear equations systems arising in finite element methods. The software offers an efficient and easy, yet flexible interface to several parallel solvers available on the web, such as PETSC, AZTEC, MUMPS, PILUT and HYPRE. FoSSI makes use of the concept of handles for vectors, matrices, preconditioners and solvers, that is frequently used in solver libraries. Hence, FoSSI allows for a flexible treatment of several linear equations systems and associated preconditioners at the same time, even in parallel on separate MPI-communicators. The second special feature in FoSSI ist the task specifier, being a combination of keywords, each configuring a certain phase in the solver setup. This enables the user to control a solver over one unique subroutine. Furthermore, FoSSI has rather similar features for all solvers, making a fast solver intercomparison or exchange an easy task. FoSSI and its followers is a community software, proven in an adaptive 2D-atmosphere model and a 3D-primitive equation ocean model. Performance measurements are given for test matrices from these models, demonstrating a minimal yet scalable overhead of the interface. Speedup experiments and solver optimization are reported, giving a brief overview over some of the solver packages performance. It is shown that FoSSI is easily and efficiently used even on LINUX-clusters. Furthermore, an OpenMP-implementation of parallel iterative solvers based on domain dcomposition methods is discussed. This approach to OpenMP solvers is rather attractive, as the code for domain-local operations of factorization, preconditioning and matrix vector product can be readily taken from a sequential implementation that is also suitable to be used in the MPI-variant.

Key words: Sparse parallel solvers, iterative solvers, finite element, user interface

1 Introduction

Modularized software is the basis for efficient and portable code development and is a prerequisite for rapid software development of cooperating working-groups for example in the field of earth system modeling. The resulting inter-operability of software components nowadays plays an important role also in scientific computing. This is due to the fact that inter-disciplinary research has become a major paradigm. However, infrastructures for efficient cooperation are frequently missing. Furthermore, creation of long-term re-usable and thus portable software requires knowledge about the life-time and capabilities of the used software components. In the field of scientific research another serious problem may arise. During research successful algorithms turn to be no longer applicable with growing problem complexity. As an example one may consider the development of a finite-element-model that in the initial phase uses linear elements and later needs higher-order elements, e.g., for stability reasons. In such a case it may turn out that the initially used solver libraries cannot be used in the final code version. It may then be not feasible to switch to another solver within the remaining project time. The solution to this problem is to use an almost omnipotent solver that integrates almost all solvers present on the web. Such a solver package can be found for example within the SLES module of PETSC (Balay et al., 2002), the Portable Extendible Toolkit for Scientific Computing. However, it still remains an open problem how developers of the model code may test various combinations of solvers and preconditioners offered by the underlying solver library within a reasonable time and without deep knowlegde of the spectrum of applicable methods. Thus there is a strong demand for a simple-to-use programming-interface for the end user that is programmed and maintained by an expert, who is able to rapidly extend the interface on user-demands. FoSSI, the Family of Simplified Solver Interfaces, is such a collection of user-interfaces to different parallel sparse linear solvers grown from user demands. The interfacing is greatly simplified for linear systems with matrix-representations in compressed sparse row (CSR) format. The supported MPI-parallel (Gropp et al., 1994) sparse iterative solvers (Saad, 1996; van der Vorst, 2003) are PETSC, PILUT (Karypis and Kumar, 1997), AZTEC (Tuminaro et al., 1999), HYPRE (Falgout and Yang, 2002), and the parallel direct solver MUMPS (Amestoy et al., 2000). The interfaces are able to keep several matrices and preconditioners within each solver library, allowing to reuse distributed matrix values and factorizations for consecutive solves. Each solver is equipped with a single subroutine for solver configuration and invokation. Different tasks within the interface, e.g., distribution of matrix values, incomplete factorization, solution

* sfrickenhaus@awi-bremerhaven.de
[1] Funded by the German Ministry for Education and Research (DEKLIM-PLASMA)

and cleanup, can be combined within a single task-identifier. The parallelism is completely hidden by specifying an MPI-communicator, allowing to use various solvers at the same runtime, or, alternatively, allowing parallel solution of many different linear equation systems synchronously.

2 FoSSI-Overview

FoSSI emerged from a user interface developed for the PILUT-solver within the FENA project, which is the precursor of the Finite Element Ocean Model FEOM (Danilov et al., 2004). It was found that the usage of such a powerful user interface makes code development very efficient. Nevertheless, PILUT in its MPI-2 variant turned out not to be a good choice on some compute platforms , e.g., the IBM-Regatta. Therefore, and for convenience (user acceptance), the interface principles have been overtaken for other solver libraries, such as PETSC, AZTEC and HYPRE. All three of these offer the user a wide area of accessible data structures and configuration options. Especially the library internal storage of distributed matrices, vectors and preconditioners and their reference by handles can be exploited within the user interfaces by attributing to each parallel linear equations system a FoSSI-handle, which is simply a user selected number out of a limited range. The second principle of the mentioned libraries is the separation of setup of matrices, vectors, preconditioners, the configuration of the solver, the invocation of the solver and the clean up of data structures. This feature has been made transparent to the user within the FoSSI-interfaces by means of the task specifier concept. A task specifier is a sum of elementary task specifiers, each being a configuration or action option for the parallel solver. For example, in the PETSC-interface a parallel matrix structure is generated, filled with symmetrically scaled values by specifying the task `PET_STRUCT+PET_MVALS+PET_SYM_SCAL`. A more detailed preconditioner configuration is given by the task specifier `PET_PCASM+PET_ASMB+PET_ICC+PET_OVL_2`, which sets up an additive Schwarz-preconditioner with overlap 2 in a symmetric implementation (known in PETSC as `PETSC_ASM_BASIC`), and an incomplete Cholesky factorization on the subdomains. This setup is specified together with the solve specifier, e.g., `PET_SOLVE + PET_CG` for a conjugate gradient iterative solver.

For a good solver selection and tuning, the user needs detailed performance (timing) information. At least, the number of applied iterations and the times for factorization and solution should be given back to the user or be printed on output. FoSSI gives some more information back to the user, such as the elapsed time for setup of a parallel matrix, for gathering the solution, for the full call, or even the cumulated time for all calls. The specifier `PET_REPORT` orders performance information in text form before the code returns from the FoSSI-interface routine. The main limitation of the current state of FoSSI is its

required input format of matrices and vectors. FoSSI uses global matrix and vector data. However, the interface needs only values on rows that are actually used by the MPI-task the row is attributed to (according to the user-given partitioning). In this way, the user does not have to take care about distributed data formats. This limitation will be overcome in the implementation of an extra FoSSI-solver, which is independent of other solver packages.

3 FoSSI-Syntax : an example

Here we only present, for brevity, the FORTRAN subroutine syntax and options for the task parameter of the PETSC-interface, as it is the most powerful interface.

petsc_s(`integer*8 Task, integer Handle, integer dim, integer nnonz,`
`integer gnum(1:nloc+1), integer max_iter, integer restart,`
`integer fillin, double precision droptol, double precision soltol,`
`integer PART(1:dim), integer ROWPTR(1:dim+1), integer COLIND(1:nonz),`
`double precision VALS(1:nonz), double precision RHS(1:dim),`
`double precision SOL(1:dim), double precision rinfo(1:10),`
`integer COMM)`

The parameters are explained in the following: `integer(kind=8) Task` : the task specifier, being a sum of predefined parameters declared in the include file petscf.h, documented below.
Handle : the handle number (0..19) to which the input-matrix is attached. All subsequent calls working on the same matrix must have the matching handle.
dim : the dimension of the problem, equals the number of rows in the global matrix.
nnonz is the number of entries in the matrix.
gnum(1) is the number of local rows of the matrix after the first call to a structure setup.
gnum(2..gnum(1)+1) : on output : the global row-indices of processor owned rows. Given back by Task
`PET_GNUM`.
max_iter : maximum number of iterations of iterative solver.
restart : number of iterations of GMRES before restart.
fill : allowed fillin for incomplete factorisation.
tol : threshold for incomplete LU-factorisation.
soltol : desired residuum reduction as convergence criterium.
PART : array of ranks describing the intended row-wise partitioning of the sparse matrix, i.e., `PART(i)=j` means rows i belong to MPI-rank j
Exception: see PET_BLOCKP specifier below.
Sparse matrix format and vectors:

ROWPTR : array of row-pointers, ROWPTR(1)=1;
ROWPTR(i+1)-ROWPTR(i) = number of entries in row i.
COLIND : array of column indices (Fortran indexing, lowest index 1, highest index dim) of matrix entries
VALS : array of matrix entry values in the same order as column indices COLIND
RHS : right hand side vector
SOL : on output : solution vector
Other parameters:
rinfo : on output: array containing information on convergence and timing, shown by the ..._REPORT Task.
COMM : MPI-communicator on which the solver operates for the specified handle.
The task specifiers for the PETSC-interface are:
PET_STRUCT : setup the parallel matrix from ROWPTR, COLIND
PET_MVALS : feed parallel matrix with values in parallel from vals according to partitioning
PET_NOSCALE : no pre-scaling of matrix in PET_MVALS before feeding matrix to PETSC
PET_SYM_SCAL : symmtric scaling of matrix, useful for CG solver
PET_NEWPC : after a previous solve, generate a new preconditioner. Per default the old preconditioner is used in subsequent solves on the same handle
PET_LOADX0 : load initial solution from SOL, e.g., for solution refinement
PET_SOLVE : call default solver restarted GMRES
PET_CG : use CG-solver
PET_GMRES : use default GMRES-solver
PET_BICGSTAB : use BICGSTAB-solver
PET_PCASM : use additive Schwarz for parallel preconditioning , default overlap 1 (default preconditioner, no other parallel preconditioner implemented yet, force overlap 0 with PET_PCBJ instead of PET_PCASM)
PET_OVL_2 : use overlap 2 in ASM
PET_OVL_3 : use overlap 3 in ASM
PET_ASMB : use basic ASM, not restricted ASM (which is default PETSC). ASMB is a symmtric ASM, useful for CG solvers
PET_ICC : use incomplete Cholesky factorization (symmetric matrix values) with fillin level `fill`, default is ILU
PET_ILU : use ILU with fillin specified by a memory factor `tol`
PET_ILUT : use threshold ILU, droptolerance from `tol`
PET_PCBJ : use BlockJacobi preconditioning (purely local on subdomains)
PET_CLEAR : clear interface internal and solver internal memory for the specified handle
PET_QUIET : flag for fewer output
PET_REPORT : flag for performance output
PET_VIEW : flag for PETSC internal configuration output
PET_RCM : use reverse CutHill-McKee renumbering on subdomains with

ILU
PET_ND : use nested disection renumbering on subdomains with ILU
PET_QMD : use quasi minimum degree reordering on subdomains with ILU
PET_BLOCKP : partitioning is blockwise, i.e., continous ranges of rows are attributed to each MPI-task. PART(1)=1, PART(i+1)-PART(i) is the number of rows owned by processor i. The BLOCKP-input matrix and vector take only processor local data. See example code petsc.F90.

PET_BLOCKSORT : variables come in blocks. gnum(2) is number of blocks of equal size; requires same partitioning for each block. Example: Indices 1..10 are the x-components of a 2d-vector, indices 11-20 are the y-component; make shure, `part(i)=part(i+10), i=1..10` and `gnum(2)=2`.
Alternatively, operate with `PET_RCM`, which does also a good renumbering / packing of blocked variables on subdomains, even without the same partitioning. Portability
FoSSI makes use of MPI-2 in some parts of the interfaces. PILUT uses MPI-2 extensively as it ported from a CRAY shmem-code. All other code is MPI-1 compatible. Currently, the interfaces are not compiled for use without MPI, e.g., on a single processor machine. However, single-processor communicators can be used in all solvers. Currently SUN (ClusterTools4, Solaris 2.8, Forte6u2, Forte7), SGI (Irix6.5, SGI-MPI) and IBM (AIX5.1, POE) were tested.

4 FoSSI-Performance

In this section we present and interpret some results of performance measurements of the FoSSI-solvers with different solver configurations. The experiments are seperated into two parts: the 3D-ocean model FEOM (Danilov et al., 2004) tracer matrix problem and a matrix from the 2D-adaptive atmosphere model from the PLASMA (Läuter et al., 2003) project.

4.1 3D FEOM tracer matrix

The tracer matrix of the 3D FEOM model configured for the north atlantic is taken as a starting point. As the FEOM-mesh is a stacked mesh of layers of tetrahedrals the structure of the matrix appears semi-regular due to the layer ordering (see fig. 1). By inspection of the sparsity structure it is seen that the surface layer (top part of the matrix) has more nodes than the layers below. The very deep regions of the north atlantic are represented by the bottom part of the matrix, consisting of much fewer elements. The upper curve in the left drawing plots entries for the neighbourhood with the layer below, the middle curve represents the 2D-horizontal neighbourhood relationships,

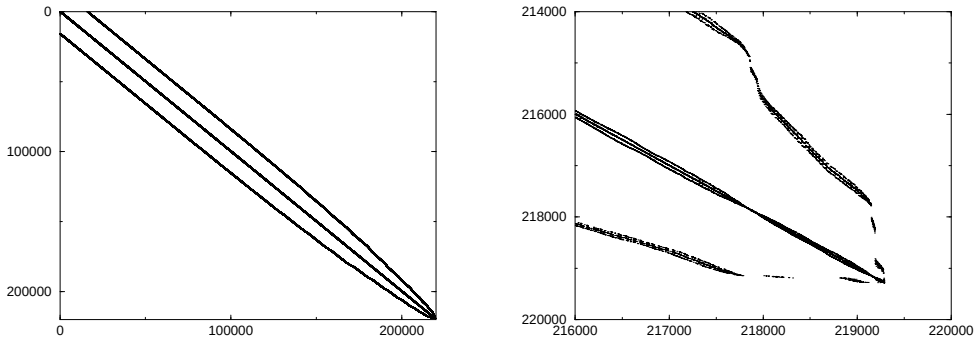


Fig. 1. Sparsity structure of the FEOM 3D-tracer matrix. Left: Full matrix; Right: Zoom into the lower-right part of the full matrix, displaying the finestructure (see text).

PETSC Solvers on 2 CPUs SunFIRE 6800

Preconditioner	factorization	solution	iterations
Diag. Scal.	0.24	8.2	73
BJ ILU(0)	0.5	3.9	13
ASM(1) ILU(0)	1.0	2.8	8
ASM(1) ILU(1)	2.2	3.2	6
ASM(1) ILU(1) RCM	2.0	4.0	8
ASM(1) ILU(2)	4.8	4.3	5
ASM(2) ILU(1)	2.4	3.6	6
ASM(2) ILU(2)	5.4	4.1	4

Table 1
Performance of the PETSC-solver configured over the FoSSI-interface for the 3D ocean model FEOM. The overlap of ASM (additive Schwarz) is given in brackets; ILU(i) means incomplete cholesky with the level of fillin i. All solutions have been obtained with the restarted GMRES solver on 2 CPUs of a SUNFire 6800 1.05 GHz machine with SUN-MPI. Times are given in seconds.

while the lower curve is for the neighbourhood with the next layer below. The curves finestructure is partially resolved in the right drawing within fig. 1. The matrix has 219286 rows and 3116206 entries, representing the advection of a scalar quantity on 219286 nodes of the mesh. It is seen, that the level of fillin reduces the interations needed, while the solution times grow. This is due to the more expensive communication and the higher numerical load for the more entries in the factorized matrix. The same is true for higher overlap in the additive Schwarz method. The best performance is found for the block-Jacobi ILU(0) case. However, for several tracers, the same preconditioner may be used, such that the overhead of computing a more efficient preconditioner

PETSC speedup on SUNFire 6800

#CPUS	factorization	solution	iterations	speedup(fact+solve)
1	1.8	4.9	8	1
2	1.1	2.7	8	1.8
4	0.7	2.5	12	2.1
8	0.5	2.5	18	2.2
16	0.3	3.4	24	1.8

Table 2
Performance speedup of the PETSC-solver configured over the FoSSI-interface for the 3D ocean model FEOM. The solver is restrticted ASM(1) ILU(0) GMRES(15). Speedup of factorization and solution is given in the last column. Times for factorization and solution are given in seconds.

PETSC speedup

	SunFire 6800		Regatta p655		XEON-cluster	
#CPUS	fact	sol	fact	sol	fact	sol
1	1.8	4.9	0.77	0.81	0.68	1.02
2	1.1	2.7	0.57	0.48	0.48	0.64
4	0.7	2.6	0.46	0.53	0.41	1.17
8	0.5	2.5	0.22	0.86	0.28	1.28

Table 3
Speedup measurement for three modern compute platforms. The solver is the same as in table 2. Times are given in seconds.

can be compensated by a reduction in the solution times. In table 2 we present the speedup measurements for the additive Schwarz method and ILU(0). It is seen in table 2, that the overall speedup is rather limited, as the number of iterations grows with the number of processors. This is a well known feature of domain-decomposition methods that do not involve coarse grid corrections (Chan and Mathew, 1994). At the same time, the factorization speeds up up to 16 CPUs.

In table 3 a comparison of the same solver is given for two other compute platforms: the IBM-Regatta p655 (1.1 GHz Power4), and a Intel-32bit-Linux cluster (2.4 GHz dual-XEON IBM x335), coupled over switched gigabit ethernet under LAM-MPI. It is seen, that the Regatta is a factor 2 to 3 faster than the SUNFire system, while the Linux cluster is a factor 2 faster. This clearly demonstrates the potential performance of parallel solvers on moderate cost platforms, such as Linux-clusters.

In table 4 we present performance results of other solvers called by the respec-

#CPUs	Fact.	Solve	Iter	Fact.	Solve	Iter
HYPRE-AMG, NOSCAL				HYPRE-PILUT($10^{-3}, 15$)		
1	0.26	19.5	134	4.3	4.6	36
2	0.17	12.5	152	2.8	2.7	36
4	0.19	5.0	94	4.6	1.4	33
7	0.17	2.5	60	4.8	1.3	26
HYPRE-Euclid BJ ILU(0)				HYPRE-PILUT($10^{-4}, 25$)		
1	1.7	1.5	13	8.1	7.7	44
2	0.8	1.3	21	5.6	4.4	43
4	0.4	1.5	42	6.9	2.3	35
7	0.2	1.2	52	8.6	2.1	26
MUMPS 4.2 β				AZTEC BJ ILU(1))		
1	85.9	8.3	-	12.3	1.7	8
2	52.3	12.0	-	6.6	2.8	20
4	29.7	8.6	-	3.1	3.0	42
7	22.7	7.7	-	1.6	2.4	53

Table 4
Performance of some other solvers on the 3D FEOM tracer matrix. All Times given in seconds.

tive FoSSI-interfaces on the p655.

The observed rather limited scalability of the FoSSI-solvers applied to the FEOM matrix is partially due to the inefficient numbering and partitioning within the FEOM code. Future development within FEOM will reorder processor-local data in a much more fill-reducing order such that incomplete factorization preconditioners will become more efficient with a high impact on the scalability.

4.2 2D adaptive PLASMA model

In adaptive models the setup time for solvers and preconditioners must be short, as this computation must be repeated at each adaptive step. The test matrices in this subsection is taken from equations arising in the solution of the shallow water equations written in potential form discretized in finite elements on the sphere (Läuter et al., 2003; Rakowsky et al., 2002). The elements are triangular with second order basis functions. This makes a matrix

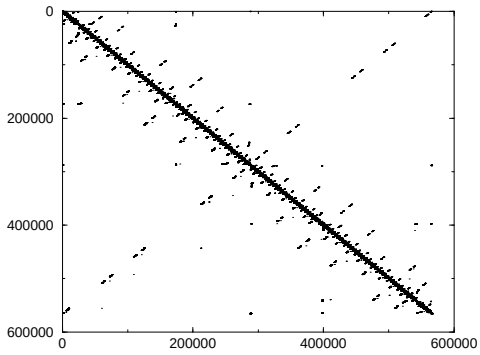


Fig. 2. Sparsity structure of the PLASMA 2D-Poisson matrix.

PETSC CG-solvers on 2 CPUs IBM-Regatta p655, 2D poisson problem

Solver	fact	sol	iter	Solver	fact	sol	iter
ASM(2) ICC(2)	2.3	45	381	ASM(2) ICC(3)	7.26	55.7	252
ASM(1) ICC(1)	1.6	62.4	634	ASM(1) ICC(2)	2.2	43.7	374
BJ ICC(2)	1.4	43.6	414	BJ ICC(1)	0.85	57.0	645
BJ ILU(2)	1.9	72.9	548	BJ ILU(2) RCM	2.4	78.5	559
BJ ILU(2) ND	4.9	113	583	BJ ILU(2) QMD	6.5	113	650

Table 5
CG solver from PETSC with different preconditioners for the 2D adaptive poisson problem. Times in seconds.

of 565910 rows with 6507944 non-zero eintries (structure plotted in figure 2). The problem is symmetric and can be solved with a CG-solver, provided, a symmetrical preconditioner is chosen. The matrix is ordered and partitioned with a reverse space filling curve approach, which is generated on-the-fly during the mesh adaption. In table 5 timing values are listed for a variety of solver configurations, each running on 2 CPUs of an IBM-Regatta p655.

It is seen that additive Schwarz with overlap 1 ASM(1) ICC(1) gives a solution time nearly as short as the BlockJacobi version with ICC(2), which needs 40 iterations more. To demonstrate the effect of domain-local reordering, which is possible in PETSC only for ILU-preconditioners, a BlockJacobi ILU(2) has been repeated with RCM, QMD and ND reordering. It is seen, that all of these local reorderings have a negative effect on the performance of the solver, indicating that the ILU(2) is most efficient with the original reverse SFC numbering.

In table 6 speedup measurements are presented for the most promising solver configuration BlockJacobi ICC(2) from table 5. It is seen, that on the IBM-Regatta, the PETSC solver speeds up exceptionally well for the given type of

Speedup of PETSC-solver CG BJ ICC(2) on IBM-Regatta p655 / p690

#CPUs	mvals	factorize	solve	iterations
1	1.5	2.9	66.5	326
2	0.9	1.4	43.7	414
4	0.47	0.7	27.7	468
8	0.27	0.35	19.1	514
16*	0.35	0.42	10.8	485
32*	0.30	0.22	5.7	513

Table 6
Timing measurements (in seconds) and iteration counts for the PETSC CG Block-Jacobi ICC(2) solver on IBM-Regatta p655 1.1 GHz(1..8 CPUS) and p690 (16..32 CPUs). mvals represents the matrix value distribution. The matrix is from a Poisson-problem.

equation system. The observed increase in iteration count is acceptable. On the IBM p655, which has 8 CPUs at 1.1 GHz on one system board, the distribution of matrix values (mvals) speed up in an optimal way. On the IBM p690, operating with 1.3 GHz Power4 CPUs and an SP2-switch, coupling 8-CPU nodes, the speed up is continued, although with a break. For each machine the purely local factorization (requiring no communication) speeds up well, but a moderate decrease in performance is observed between the p655 and p690 system. This may be attributed to the problems on the p690 with respect to reproducability of program runtimes. It is noteworthy, that the same binary executable, compiled on the p655, has been taken on both machines. The FoSSI-interface has a constant overhead for setting up the parallel matrix structure. This takes roughly 0.2 seconds on both of the mentioned IBM systems, independent of the number of used CPUs. A more efficient preconditioner ASM(1) ICC(2) on 8 CPUs (p655) needs 0.6 s for factorization and 20.1 s for solution in 482 interations (data not shown in the tables). This, in comparison to the 8 CPU results in table 5, shows, that a decrease in iteration count due to a better preconditioner is, in this case, compensated by the cost of the increased communication (ASM(1) vs. BJ).

A more complicated equations system emerges from the coupling of the pressure (geopotential), described by second order basis functions, to the potentials of the 2D velocity, namely divergence and vorticity, each represented by linear basis functions. The structurally symmetric matrix has 410148 rows and 7655998 entries (sparsity pattern not shown). The matrix values are not symmetric. The reverse SFC numbering samples all the variables along a path through the elements. In this way, the second order values of the geopotential are sorted into the full matrix structure, leading to a fill-in reducing matrix ordering. Table 7 shows the results of the PETSC BICGSTAB solver on the

PETSC-solver BICGSTAB ASM(1) ILU(2)				
#CPUs	mvals	factorize	solve	iterations
2	0.9	5.2	42.1	103
4	0.47	2.7	22.0	97
8	0.25	1.3	14.0	97
PETSC-solver BICGSTAB BJ ILU(2)				
2	0.9	4.3	96.1	247
4	0.46	2.1	73.5	351
8	0.25	1.1	59.4	454

Table 7
Timing measurements (in seconds) and iteration counts for the PETSC BICGSTAB solver on IBM-Regatta p655. The matrix is from the coupled equations system of geopotential, divergence and vorticity. The upper part is for additive Schwarz with overlap 1, the lower part for Block-Jacobi. ILU(2) is taken as the subdomain-local preconditioner.

IBM p655 for up to 8 CPUs. In contrast to the Poisson-problem, the coupled system can be solved with improved efficiency by employing the additive Schwarz method rather than the Block-Jacobi method. This can be attributed in part to the fact that with the restricted ASM the iteration count is not increasing with the number of subdomains (number of CPUs). The restricted ASM can be used in this case, as the matrix is not symmetric, and thus, the preconditioner need not be symmetric, either. In general, BICGSTAB needs roughly twice as much operations as CG, making it not attractive for symmetric matrices. For the Poisson-problem measurements have shown, that the benefit of restricted ASM cannot compensate for the additional computation time that BICGSTAB needs, compared to CG (data not shown).

5 An OpenMP implementation performance outlook

For some model codes, based on OpenMP-parallelization (see Chandra et al. (2000)), a solver based on the same parallelization technique is needed, as the coupling of a multi-threaded (OpenMP) code to an MPI-solver, waiting in the background, may become rather costful: although there is possible to communicate data between threads and MPI-tasks in a safe way (Rakowsky et al., 2002), an optimal scheduling of the threads and/or MPI-tasks may require each of them to run on a separate processor. In particular, a "spinning" MPI-Task is awakening faster, but does not allow for a fast context switch to a corresponding OpenMP-thread running on the same processor. The consequence is, that on some compute platforms, for a scalable version of code the

number of required processors is the number of MPI-tasks in the solver plus the number of threads in the rest of the code. Such a processor setup obviously is not easily load-balanced, as the processors running the multi-threaded code have to wait for the solution of the linear equations system, thus wasting compute time.

Generally, an OpenMP (multi-threaded) parallel iterative solver can be programmed on the basis of a sequential (single CPU) version of iterative solvers by distributing parallel work within OpenMP parallel do-loops. Here, the problem of memory affinity arises: for larger SMP computers memory is distributed over several CPU-boards of the system, being remotely accessible at an increased latency time, typically an extra 200-300 ns. Thus, requests for data from a remote board generate overhead in terms of idle CPU time. On some systems this can be overcome by a so called "first touch memory allocation policy", meaning, that physical memory addresses are generated at the first time memory is accessed (e.g., initiallized), rather than when it is allocated. This allows the operating system to place pages of memory near the accessing CPU, e.g., on the same system board. As memory access on cache-based systems is organized in the cache hierachy by reference to pages, the memory of a data array may be stored physically on more than one system board. Obviously, to reach performance, the programmer must organize memory accesses in his code in such a way that memory affinity is preserved throughout the runtime of the program code. This means, that parallel memory access patterns must be repeated, i.e., access to an array of data must be parallelized in the same way throughout the code, such that each thread deals most of the time with processor near memory.

Iterative sparse solvers are based on three time intensive operations: preparation of a preconditioner (in most cases an incomplete factorizatrion), the matrix-vector product and the precoditioning step (application of a preconditioner to approximately solve a subproblem within the iterative algorithm). The matrix vector product can be easily OpenMP-parallelized through a parallel outer (row-) loop. If the matrix is ordered with near diagonal structure, it makes sense to store vectors and matrix values in a processor affine memory, such that only a minor fraction of matrix/vector values is accessed from remote memory. In contrast, the more robust preconditioners based on incomplete factorizations of higher level, have an immanent sequential part in their preparation as well as in their application. One standard way to overcome this problem is to treat the problem with a domain decomposition technique (see Chan and Mathew (1994)), separating work into independent subdomains, each treated by its own processor. Within OpenMP, regarding the memory affinity, this is naturaly done by introducing a thread-index into shared data structures. Technically spoken, a static global data array is always shared between the threads, and its pages physical locations may be gouverned by memory affinity. To preserve memory affinity in a transparent way, each sub-

#CPUs	MPI-update	OMP-update	MPI-mvu	OMP-mvu	mv omp-for
2	0.06	0.06	23.9	20.5	20.6
4	0.09	0.05	12.47	10.4	10.4
8	0.11	0.03	6.26	6.1	5.7
16	0.13	0.06	3.51	3.5	3.3
30	0.15	0.2	2.32	3.1	4.1

Table 8
Commmunication times in seconds for 1000 vector updates / matrix-vector products on IBM-p690 (1.3 GHz); the matrix is from the 2D-Poisson problem, same as in table 5. mvu means domain decomposed matrix-vector product with vector update. mv omp-for is the row-loop-parallel OpenMP version of the global sparse matrix-vector product.

domain may be given a leading index to the global arrays used for communication, e.g., a buffer vector `Buff[i]` becomes `Buff[t][i]`, where `t` is the logical thread number (obtained from `omp_get_thread_num()`). Within this approach, it is straight forward to code the basis of OpenMP-parallel solvers from MPI-parallel solvers. The MPI-send and receive operations can simply be replaced by memory copy operations between buffers. Furthermore, the code for factorization and preconditioner application can be taken over from an MPI-implementation without changes. In the next version of FoSSI, an efficient and robust parallel solver will be provided to the community, that is independent of the parallelisation paradigm, namely, the user may chose the MPI- or the OpenMP-version with the same set of features and rather similar performace characteristics.

To demonstrate the efficiency of the approach discussed above, performance measurements are presented in table 8, allowing to compare the overhead of communication of the MPI- and the OpenMP version of code. It is seen that the vector update (also known as vector gather) in the domain decompostion method is more efficient in OpenMP than in MPI (columns 2 vs. 3). This is due to the overhead of the MPI-implementation over the direct memory-to-memory-copy in the OpenMP-code, although the MPI communication is forced to work over shared memory. In the matrix-vector product, the MPI version of code shows more overhead than expected from the pure update timings. Obviously, the matrix vector product of the domain-decomposed code performs similarly fast as the parallel loop code. On a p690, four 8-CPU subsystems are coupled by a communication bus, thus the communication overhead is expected to increase when more than 8 CPUs are used. It is noteworthy that we did not analyse how processes and threads were distributed over the 32 CPUs. To achieve a reliable memory affinity it is necessary to bind processes/threads to CPUs.

6 Conclusion

We presented a convenient way to access various parallel solvers for finite element modeling codes of geophysical flow, even for the unexperienced user. The Family of Simplified Solver Interfaces provides a flexible and highly performing parallel tool. Further development in the field of robust iterative solvers parallelized in OpenMP and MPI are in progress and necessary, in particular, for real domain decomposed model codes, i.e., where global matrices and vectors are not available. The emerging solvers will also take into consideration system specific optimization options such as processor binding for memory affinity. In contrast to the existing solvers with FoSSI-interfaces, these new solvers will allow for a matrix and vector distribution in such a way that the user may program the code around the solver in a completely domain decomposed way with only processor local data structures. This is achieved by handing over the communication routine for vector updates to the user. It is planned to make these solvers accessible to the scientific community in source form as a portable maintained stand-alone software package.

Acknowledgement

The development of FoSSI in the context of PLASMA is funded by the German Ministry for Education and Research within the german climate research programme DEKLIM. The authors are very grateful for the contribution of test problems by the FEOM-developers Genady Kivman and Sergej Danilov as well as by the PLASMA co-developer Natalja Rakowsky.

References

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods in Appl. Mech. Eng. 184, 501–520, See http://www.enseeiht.fr/apo/MUMPS/.

Balay, S., Gropp, W., Curfman-McInnes, L., Smith, B., 2002. Petsc users manual. Technical Report ANL-95/11 - Revision 2.1.3. See http://www-fp.mcs.anl.gov/petsc/.

Chan, T. F., Mathew, T. P., 1994. Domain decomposition algorithms. Acta Numerica, 61–143.

Chandra, R., Dagum, L., Kohr, D., 2000. Parallel Programming in OpenMP. Morgan Kaufmann Publishers.

Danilov, S., Kivman, G., Schröter, J., 2004. A finite element ocean model: principles and evaluation. Ocean Modelling 6, 125–150.

Falgout, R. D., Yang, U. M., 2002. hypre: a library of high performance preconditioners. in computational science - iccs 2002 part iii.

Gropp, W., Lusk, E., Skjellum, A., 1994. Using MPI. MIT Press, Cambrige.

Karypis, G., Kumar, V., 1997. Parallel threshold-based ilu factorization. In: Proceedings of 9th Supercomputing Conference, San Diego. ACM SIGARCH, pp. 1–24, http://www.supercomp.org/sc97/proceedings/.

Läuter, M., Handorf, D., Dethloff, K., Frickenhaus, S., Rakowsky, N., Hiller, W., 2003. An adaptive Lagrange–Galerkin shallow–water model on the sphere. In: Heinze, T., Lanser, D., Layton, A. T. (Eds.), Proceedings of the Workshop "Current Development in Shallow Water Models on the Sphere", 10–14 March 2003. Munich University of Technology, Munich, Germany.

Rakowsky, N., Frickenhaus, S., Hiller, W., Läuter, M., Handorf, D., Dethloff, K., 2002. A self-adaptive finite element model of the atmosphere. In: Zwieflhofer, W., Kreitz, N. (Eds.), Proceedings of the Tenth ECMWF Workshop on the Use of High Performance Computing in Meteorology: Realizing TeraComputing. World Scientific.

Saad, Y., 1996. Iterative Methods for Sparse Linear Systems. PWS Publishing Company.

Tuminaro, R. S., Heroux, M., Hutchinson, S. A., Shadid, J. N., 1999. Official aztec users guide: Version 2.1. See http://www.cs.sandia.gov/CRF/pspapers/Aztec_ug_2.1.ps.

van der Vorst, H., 2003. Iterative Methods for Large Linear Systems. Cambridge University Press.