On the Generic Parallelisation of Iterative Solvers for the Finite Element Method

P. Bastian and M. Blatt*
Insititut für Parallele und Verteilte Systeme (IPVS),
Universität Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany
Email: {Markus.Blatt|Peter.Bastian}@ipvs.uni-stuttgart.de
*Corresponding author

Abstract: The numerical solution of partial differential equations frequently requires solving large and sparse linear systems. When using the Finite Element Method these systems exhibit a natural block structure that is exploited for efficiency in the "Iterative Solver Template Library" (ISTL). Based on existing sequential preconditioned iterative solvers we present an abstract parallelisation approach which clearly separates the parallelisation aspects from the data structures and solver algorithms by imposing an abstract consistency model onto the building blocks of the iterative solver components. This allows for supporting overlapping and non-overlapping domain decompositions as well as data parallel implementations of standard linear solvers.

Keywords: iterative methods, domain decomposition, C++, generic programming, parallel computing

Reference to this paper should be made as follows: Bastian, P. and Blatt, M. (xxxx) 'On the Generic Parallelisation of Iterative Solvers for the Finite Element Method', *Int. J. Computational Science and Engineering*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Peter Bastian received a Degree in Computer Science at the University of Erlangen-Nürnberg in 1989 and his PhD in Mathematics in 1994 at University of Heidelberg. He is currently a full professor at the Institute of Parallel and Distributed Systems, University of Stuttgart. His research interests include numerical solution of partial differential equations using high performance computing.
Markus Blatt received a Degree in Mathematics at the University of Heidelberg in 2004. He is currently working on his PhD thesis at the Institute of Parallel and Distributed Systems, University of Stuttgart. His research interests include issues related to iterative solvers (especially algebraic multigrid methods) and parallel computing.

1 Introduction

Solving large sparse linear systems is an ubiquitous task in the numerical solution of partial differential equations. Increasing demands of computationally challenging applications both in problem size and algorithm complexity have lead to the development of parallel scalable solver libraries for these tasks. Commonly used parallel iterative solver libraries are hypre, Falgout and Yang [2006], PETSc, Balay et al. [1997], and Trilinos, Heroux et al. [2005]. Of these only the last one provides a C++ interface.

In contrast to these libraries the parallel version of the "Iterative Solver Template Library" (ISTL), Blatt and Bastian [2007], is specifically designed for linear systems stemming from Finite Element discretisations. Depending on how systems of partial differential equations are discretised, e.g. using equation wise ordering or point wise ordering, they may have a different natural hierarchical block structures. As this structure is already known at compile time it is exploited by ISTL using generic programming techniques available in C++. The matrix and vector classes, being template classes, support this structure and it is used in the generic iterative solver kernels for efficiency.

All of the parallel solver libraries mentioned above work with distributed data structures (matrices and vectors) which implicitly know the data distribution and communication patterns. In contrast to this, in our approach the data distribution and communication is not built into the linear algebra data structures. This leads to a clear separation of parallelisation aspects and sequential linear solver components.

The information about the data decomposition and communication interfaces is either provided by a parallel grid implementation or by distributed index sets. With this information an abstract consistency model is imposed on the building blocks (scalar products, preconditioners and parallel operators) of our iterative solvers. As a consequence the framework is able to support overlapping and non-overlapping domain decomposition methods as well as data parallel implementations of standard linear solvers. Furthermore it is easy to switch to other parallel programming paradigms as the consistency model is only loosely coupled with the current paradigm, MPI.

The paper is organised as follows: In Section 2 we describe the proposed domain decomposition together with the parallel discretisation approach. This discretisation approach is a prerequisite for our parallelisation approach. After introducing the sequential version of ISTL in Section 3 we devote Section 4 to the building blocks of our parallel solvers. In Section 5 we outline the abstraction and design of our communication interface, which greatly eases the tasks of developing custom parallel preconditioners and solvers and is easily portable to other parallel programming paradigms. In Section 6 we review related library approaches and point out the differences between the ISTL approach. Finally we show numerical results in Section 7 proving the scalability of the approach and its applicability to real world problems.

2 Domain Decomposition

A crucial part in parallel solvers is the construction of the operators from a given domain decomposition. The usage of these operators is twofold: On the one hand they are applied to vectors, e.g. to compute the current residual in iterative methods, and on the other hand they are used to construct preconditioners. In this section we will show how we set them up to handle both tasks efficiently.

Let us consider a second order selfadjoint partial differential equation in 2D or 3D

$$-\nabla \cdot D(x)\nabla u(x) = f(x), \qquad x \in \Omega \tag{1}$$

$$\alpha(x)u(x) + \beta(x)\frac{\partial u}{\partial n} = g(x), \qquad x \in \partial\Omega \tag{2}$$

where $\Omega \in \mathbb{R}^n$, $n = 2, 3$ is an open domain, $D(x) > 0$ is a diagonal matrix and

$$|\alpha(x)| + |\beta(x)| > 0 \quad \forall x \in \partial\Omega$$

holds.

We decompose the domain Ω into P (the number of processors) non-overlapping subdomains Ω_i with

$$\overline{\Omega} = \bigcup_{i=0}^{P-1} \overline{\Omega}_i, \quad \Omega_i \cap \Omega_j = \emptyset \quad (i \neq j)\,.$$

As we would like to be able to cover both overlapping and non-overlapping domain decompositions we introduce the additional domain $\widetilde{\Omega}_i$, with

$$\Omega_i \subseteq \widetilde{\Omega}_i \subseteq \Omega\,.$$

Furthermore we define the *border* $B_i = \partial\Omega_i$.

Note that for a non-overlapping domain decomposition

$$\widetilde{\Omega}_i = \Omega_i$$

and for the overlapping case

$$\Omega_i \subsetneq \widetilde{\Omega}_i$$

holds.

2.1 Finite Element Spaces

Let V_h be a finite element space generated by a basis $\Phi_h = \{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$. Then it is obvious that any $u \in V_h$ can be represented as $u = \sum_{i=0}^{N-1} x_i\phi_i$. The coefficients form a vector $x = (x_0, \ldots, x_{N-1})^T \in \mathbb{R}^N$.

For any finite dimensional subset $I \subset \mathbb{N}$ we define $\mathbb{R}^I$ as the vectors $x = (x_{k_0}, x_{k_1}^T, \ldots, x_{k_M})$, where $k_i \in I$ and $M = \|I\| - 1$. Assuming that Φ_h is a nodal basis, i.e. every $\phi_i \in \Phi_h$ is associated with some position $z_i \in \overline{\Omega}$, we define for every $\omega \subset \overline{\Omega}$

$$I_w = \{k \in \{0, \ldots, N-1\} | z_k \in \omega\}$$

as the (not necessarily consecutive) index set I_ω corresponding to the domain ω for the nodal basis Φ_h.

Thus I_{Ω_i} denotes all indices of basis functions associated with subdomain Ω_i. Note that $I_{\overline{\Omega}} = \{0, 1, \ldots, N-1\}$ holds.

2.2 Restriction

We define the following restriction operator for an arbitrary subdomain $\omega \in \Omega$:

$$R_\omega : \mathbb{R}^{I_{\overline{\Omega}}} \to \mathbb{R}^{I_\omega}$$

by

$$(R_\omega x)_k = (x)_k \quad \forall k \in I_\omega$$

and the corresponding prolongation operator

$$R_\omega^T : \mathbb{R}^{I_\omega} \to \mathbb{R}^{I_{\overline{\Omega}}}$$

by

$$(R_\omega^T x_\omega)_k = \begin{cases} (x_\omega)_k & k \in I_\omega \\ 0 & k \notin I_\omega \end{cases}\,.$$

Note that R_ω just selects the coefficients x that are associated with the subdomain ω and we have

$$R_\omega R_\omega^T = \mathrm{Id}_\omega$$

where Id_ω denotes the identity on $\mathbb{R}^{I_\omega}$.

Theorem 2.1. *(Partitioning of $\mathbb{R}^{I_{\overline{\Omega}}}$) There exists a (not necessarily unique) disjoint partitioning*

$$I_{\overline{\Omega}} = \bigcup_{i=0}^{P-1} I_i\,, \quad I_i \cap I_j = \emptyset \quad \forall i \neq j$$

with $I_i \subset I_{\overline{\Omega}_i}$.

Proof. Since $\cup \overline{\Omega}_i = \overline{\Omega}$ it is obvious that

$$\bigcup_{i=0}^{P-1} I_{\overline{\Omega}_i} = I_{\overline{\Omega}}$$

is a possibly overlapping decomposition. This immediately gives a constructive set

$$I_i = I_{\overline{\Omega}_i} \setminus \left(\bigcup_{j=0}^{i-1} I_{\overline{\Omega}_j} \right).$$

□

With $\mathbb{R}^{I_i}$ we associate the canonical restriction $R_i : \mathbb{R}^{I_{\overline{\Omega}}} \to \mathbb{R}^{I_i}$ and the prolongation $R_i^T : \mathbb{R}^{I_i} \to \mathbb{R}^{I_{\overline{\Omega}}}$ in the same way as above.

2.3 Parallel Representations

In a parallel implementation $x \in \mathbb{R}^{I_{\overline{\Omega}}}$ cannot be stored in one process but is represented by individual pieces. As there might be the need to store more entries than the domain $\widetilde{\Omega}_i$ contains we introduce the superset $\widetilde{I}_i \supseteq I_{\widetilde{\Omega}_i}$ denoting all indices for which process i stores values. Each process i, $0 \leq i < P$, stores the piece $x^i \in \mathbb{R}^{\widetilde{I}_i}$ of the global vector x.

The goal is to use purely sequential matrix and vector data structures and operations and still be able to do parallel computations reusing sequential linear algebra kernels. Therefore one has to impose certain constraints onto the local representations of global vectors when entering the kernel methods as well as guarantee certain representations upon exit of the methods. These constraints will be defined here.

Let $\mathcal{P} = \{0, 1, \ldots, P-1\}$ be the set of processes being used.

Definition 2.2 (Valid representation of a vector)**.** *A vector x is stored in a* valid representation *if and only if*

$$(x^i)_k = (R_{\widetilde{I}_i} x)_k \quad \forall k \in I_i\,.$$

for all local pieces x^i, $i \in \mathcal{P}$.

This is the most weak assumption imposed on a parallel vector which should hold at any state of a parallel computation.

Definition 2.3 (Consistent representation of a vector)**.** *A vector x is stored in a* consistent representation *on the decomposition $J_i \subseteq \widetilde{I}_i$, $i \in \mathcal{P}$ and $\cup_{i \in \mathcal{P}} J_i = I_{\overline{\Omega}}$ if and only if for all of its components x^i*

$$(R_{\widetilde{I}_i} x)_k = (x^i)_k \quad \forall k \in J_i\,,$$

on all processes $i \in \mathcal{P}$ holds.

For the case $J_i = \widetilde{I}_i$ this means that all entries in the local vector x^i are the same as the corresponding entries in the global vector x. In this case x is said to be consistent.

Definition 2.4 (Additive representation of a vector)**.** *A vector x is stored in an* additive representation *if and only if*

$$x = \sum_{i=0}^{P-1} R_{\widetilde{I}_i}^T x_i\,.$$

It is obvious that if a vector x is stored in an additive representation, it can easily be transformed into a consistent representation on the decomposition $\widetilde{I}_i$, $i \in \mathcal{P}$, by

$$x_i = R_{\widetilde{I}_i} \sum_{i=0}^{P-1} R_{\widetilde{I}_i}^T x_i\,.$$

A special case of the additive representation is the unique representation of a vector:

Definition 2.5 (Unique representation of a vector)**.** *A vector x is stored in a* unique representation *on the processes of $\mathcal{P}$ if and only if*

$$(R_{\widetilde{I}_i} x)_k = x_k^i \quad \forall k \in I_i \quad \textit{and}$$
$$(R_{\widetilde{I}_i} x)_k = 0 \quad \forall k \in \widetilde{I}_i \setminus I_i$$

holds for all processors $i \in \mathcal{P}$.

Note that each vector x being stored in a valid representation can easily be transformed to a unique representation by a local projection that sets all entries associated to indices of $\widetilde{I}_i \setminus I_i$ to zero.

2.4 Operators

Assume we have an operator

$$A : \mathbb{R}^{I_{\overline{\Omega}}} \to \mathbb{R}^{I_{\overline{\Omega}}}\,.$$

Then the application of the global operator A shall be represented by applying local operators A_i, to be defined later.

As we want to use purely sequential data structures without knowledge of the parallel representation these operators need to be carefully crafted to resemble the global application and consistency constraints have to be imposed on the vector it is applied to and onto the result of the application. A second purpose of the local operators is for the construction of preconditioners, which must not be neglected. Our goal is to define the local operators in a way such that the sequential preconditioners, e.g. SOR, can still compute updates stored in a valid representation.

In order to define the finite element method the domain Ω is partitioned into elements $T(\Omega) = \{t_0, \ldots, t_{M-1}\}$. We assume that the mesh is compatible with the subdomains, i.e.

$$T(\omega) = \{t \in T(\Omega) | t \subseteq \omega\}$$

and

$$\bigcup_{t \in T(\omega)} \bar{t} = \overline{\omega}\,,$$

where $\omega \in \{\Omega_i, \widetilde{\Omega}_i\}$, $0 \le i < P$.

With each element $t \in T(\Omega)$ we associate the restriction $R_t : \mathbb{R}^{I_{\overline{\Omega}}} \to \mathbb{R}^{I_t}$ in the way defined above. The global operator A in the finite element method is constructed locally in an additive way

$$A = \sum_{t \in T(\Omega)} R_t^T A_t R_t \tag{3}$$

where A_t, the so-called *local stiffness matrix*, is associated with the element t.

2.4.1 Non-overlapping case

Although we allow overlapping grids ($\Omega_i \subsetneq \widetilde{\Omega}_i$) here, we want to achieve local operators that only compute values for the unique partition I_i.

Since the subdomains Ω_i define a decomposition of Ω and the mesh is compatible we have

$$\begin{aligned} A &= \sum_{i=0}^{P-1} \sum_{t \in T(\Omega_i)} R_t^T A_t R_t \\ &= \sum_{i=0}^{P-1} \sum_{t \in T(\Omega_i)} R_{\overline{\Omega}_i}^T R_{\overline{\Omega}_i} R_t^T A_t R_t R_{\overline{\Omega}_i}^T R_{\overline{\Omega}_i} \\ &= \sum_{i=0}^{P-1} R_{\overline{\Omega}_i}^T \underbrace{\left(\sum_{t \in T(\Omega_i)} R_{\overline{\Omega}_i} R_t^T A_t R_t R_{\overline{\Omega}_i}^T \right)}_{=: A_{\overline{\Omega}_i}} R_{\overline{\Omega}_i} \qquad (4) \\ &= \sum_{i=0}^{P-1} R_{\overline{\Omega}_i}^T A_{\overline{\Omega}_i} R_{\overline{\Omega}_i} . \end{aligned}$$

The local operator $A_{\overline{\Omega}_i}$ is a mapping $A_{\overline{\Omega}_i} : \mathbb{R}^{I_{\overline{\Omega}_i}} \to \mathbb{R}^{I_{\overline{\Omega}_i}}$. Thus we have obtained an additive decomposition of the operator A where the application of the local operators $A_{\overline{\Omega}_i}$ can be computed in each processor i in parallel without communication as long as the vector is stored in a consistent representation on $\overline{\Omega}_i$, $i \in \mathcal{P}$.

Unfortunately, the result of the application of the local operators $A_{\overline{\Omega}_i}$ is not stored in a valid representation on $\overline{\Omega}_i$ as the entries of the local matrix row might not be equal to the corresponding entries in the global matrix representation. This means that before continuing any computations a communication step is needed to store the vector in a valid representation. In addition it is still not possible to create preconditioners working directly on the operator representation, like SOR, that are able to compute updates that are consistent on $I_{\overline{\Omega}_i}$ without additional communication or storing an additional representation of the operator for the preconditioner.

A remedy to this situation is to store on process i for each local matrix row corresponding to an index in I_i all off-diagonal nonzero entries of the global matrix with the corresponding global value. As this means that our vectors might need to store additional values, due to the additional matrix entries, the index set $I_{\overline{\Omega}_i}$ needs to be augmented in the following way:

Let $G(V, E)$ be the graph of the matrix A, where the set of vertices $V = \{0, 1, \ldots, N-1\}$ represents the N unknowns and the set of edges $E = \{(i,j) | A_{ij} \neq 0\}$ represents the pairs of vertices coupled by a nonzero entry in A. Then we set $\widehat{I}_i = I_{\overline{\Omega}_i} \cup \{ j \in I | \exists i \in I_{\overline{\Omega}_i} : (i,j) \in E \}$ and $\widetilde{I}_i = I_{\widetilde{\overline{\Omega}}_i} \cup \hat{I}_i$.

Now we construct a local operator mapping $A_{\widetilde{I}_i} : \mathbb{R}_{\widetilde{I}_i} \to \mathbb{R}_{\widetilde{I}_i}$ as follows:

$$(A_{\widetilde{I}_i})_{\alpha\beta} = \begin{cases} \left(\sum_{j=0}^{P-1} (R_{\overline{\Omega}_i}^T A_{\overline{\Omega}_i} R_{\overline{\Omega}_i}) \right)_{\alpha\beta} & \text{if } \alpha \in I_i \wedge \beta \in \hat{I}_i \\ \delta_{\alpha,\beta} & \text{if } \alpha \notin I_i \\ 0 & \text{else} \end{cases} , \tag{5}$$

where

$$\delta_{\alpha,\beta} = \begin{cases} 1 & \text{if } \alpha = \beta \\ 0 & \text{else} \end{cases}$$

denotes the Kronecker delta.

Graphically the operator $A_{\widetilde{I}_i}$ has the following structure

$$\widetilde{I}_i \left\{ \hat{I}_i \left\{ I_i \left\{ \begin{array}{|c|c|c|} \hline A_{ii} & * & 0 \\ \hline 0 & I & 0 \\ \hline 0 & 0 & I \\ \hline \end{array} \right. \right. \right.$$

where

$$(A_{ii})_{\alpha\beta} = \left(\sum_{j=0}^{P-1} R_{\overline{\Omega}_i}^T A_{\overline{\Omega}_i} R_{\overline{\Omega}_i} \right)_{\alpha\beta} = (A)_{\alpha\beta}$$

are the entries of the sub matrix (principal sub matrix) A_{ii} of A with respect to the indices I_i.

Note that for the case $\widehat{I}_i \not\subseteq I_{\widetilde{\overline{\Omega}}_i}$, e.g. real non-overlapping grids, computing this local operator requires communication. Using this local operator we are in position to compute an update stored in a valid representation provided that the vector is stored in a consistent representation on the decomposition $\widehat{I}_i$, $i \in \mathcal{P}$.

Using a modified local operator $S_i A_{\widetilde{I}_i}$, where $S_i = R_{\widetilde{I}_i} R_i^T R_i R_{\widetilde{I}_i}^T$ sets all entries x_i, $i \notin I_i$, to 0, results in the additive decomposition:

$$A = \sum_{i=0}^{P-1} R_{\widetilde{I}_i}^T S_i \widetilde{A}_{\widetilde{I}_i} R_{\widetilde{I}_i} .$$

Assuming that the vector to which the operator A is applied is stored in a consistent representation on the decomposition $\widehat{I}_i$, $i \in \mathcal{P}$, the application of the local operators $S_i \widetilde{A}_{\widetilde{I}_i}$, results in a vector being stored in a unique representation. Together with the mentioned constraints this operation itself does not require any communication.

2.4.2 Overlapping case

Next we consider the overlapping case $\Omega_i \subsetneq \widetilde{\Omega}_i$, where we want to construct an operator $A_{\widetilde{\widetilde{\Omega}}_i} : \widetilde{\widetilde{\Omega}}_i \mapsto \widetilde{\widetilde{\Omega}}_i$ for using it in additive overlapping Schwarz methods.

We construct it simply by adding up the local stiffness matrices as described in equation (3) for the finite elements belonging to the local domain $\widetilde{\Omega}_i$ and representing the entries belonging to $\partial\widetilde{\Omega}_i$ as Dirichlet boundary conditions:

$$(A_{\widetilde{\widetilde{\Omega}}_i})_{\alpha\beta} = \begin{cases} \left(\sum_{t\in T(\widetilde{\Omega}_i)} R_t^T A_t R_t\right)_{\alpha\beta} & \text{if } \alpha \in \widetilde{\Omega}_i \\ \delta_{\alpha\beta} & \text{else} \end{cases} .$$

Note that this is the classical definition of the overlapping Schwarz operator.

To apply the operator in parallel we use the additive decomposition

$$A = \sum_{i=0}^{P-1} R_{\widetilde{\widetilde{\Omega}}_i}^T S_i \widetilde{A}_{\widetilde{\widetilde{\Omega}}_i} R_{\widetilde{\widetilde{\Omega}}_i} ,$$

where $S_i = R_{\widetilde{\widetilde{\Omega}}_i} R_i^T R_i R_{\widetilde{\widetilde{\Omega}}_i}^T$ sets all entries x_k^i, $k \notin I_i$, to 0.

Note that applying the local operator $S_i \widetilde{A}_{\widetilde{\widetilde{\Omega}}_i}$ on a vector stored in a consistent representation on the decomposition $\widetilde{\Omega}_i$, $i \in \mathcal{P}$ results in a vector being stored in a unique representation. This is a local operation without any communication.

3 The Sequential Iterative Solvers Template Library

The numerical solution of partial differential equations frequently requires solving large and sparse linear systems. When discretised using the Finite Element method the resulting sparse matrice exhibit a natural block structure.

Assuming one is discretising a system of partial differential equations consisting of the variables $v_1, \dots, v_n$ with a nodal basis attached to the points $z_1, \dots, z_k$ then two representations of the resulting discretisation matrix resulting in a block matrix become apparent.

- The resulting linear system $Ax = b$ can be reordered "unknown-wise" resulting in the system

  $$\begin{pmatrix} A_{[11]} & \cdots & A_{[1n]} \\ \vdots & \ddots & \vdots \\ A_{[n1]} & \cdots & A_{[nn]} \end{pmatrix} \begin{pmatrix} v_{[1]} \\ \vdots \\ v_{[n]} \end{pmatrix} = \begin{pmatrix} b_{[1]} \\ \vdots \\ b_{[n]} \end{pmatrix}$$

  where $v_{[i]}$ denotes the vector of variables associated with the i-th unknown and $A_{[i.j]}$, $i \neq j$, the sparse submatrices of A reflecting the coupling of the i-th with the j-th unknown, see Figure 1(d).

  As an example we mention the Stokes system. Iterative solvers such as the SIMPLE, Patankar [1980], or Uzawa algorithm, Arrow et al. [1968], use this structure.

- There is also the possibility of reordering the system "point-wise" resulting in the system

  $$\begin{pmatrix} A_{(11)} & \cdots & A_{(1k)} \\ \vdots & \ddots & \vdots \\ A_{(k1)} & \cdots & A_{(kk)} \end{pmatrix} \begin{pmatrix} v_{(1)} \\ \vdots \\ v_{(k)} \end{pmatrix} = \begin{pmatrix} b_{(1)} \\ \vdots \\ b_{(k)} \end{pmatrix}$$

  where $v_{(i)}$ denotes the vector of all unknowns associated with the point z_i. Now the matrix blocks $A_{(i,j)}$ are (small) dense submatrices of size $n \times n$, see Figure 1(a). It is straightforward and efficient to treat these small dense blocks as fully coupled and solve them with direct methods within the iterative method, Bastian and Helmig [1999].

When using p-adaptive Discontinuous Galerkin discretisations each finite element $t \in T(\Omega)$ has an associated order p_t, giving rise to $O(p_t^d)$ unknowns associated to this element in the scalar case. Here d represents the space dimension of the grid. Using the natural "element-wise" ordering of the unknowns results in a sparse matrix of dense submatrices whose size varies with the order of the discretisation in each element. This case is depicted in Figure 1(b).

Other structures that can be exploited are the level structure arising from hierarchic meshes, a p-hierarchic structure (e.g. decomposition in linear and quadratic part), geometric structure from decomposition in subdomains or topological structure where unknowns are associated with nodes, edges, faces or elements of a mesh.

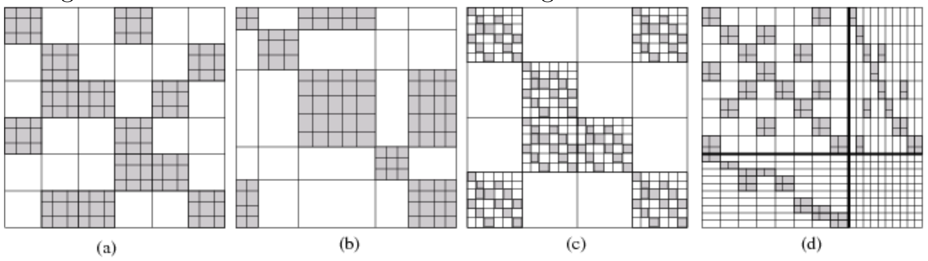
The "Iterative Solver Template Library" (ISTL), Blatt and Bastian [2007], was designed to resemble the described natural block structure of the discretisation matrices as well as the vectors. Each matrix entry is itself either a sparse or dense matrix and each vector entry is again a vector. This natural hierarchic block structure is already known at compile time and using generic programming techniques ISTL exploits this structure to generate algorithms adapted to it.

Currently there are three types of matrices and vectors adhering to the ISTL interface:

- Small dense matrices, `template<class T,int n,int m> class FieldMatrix`, and vectors, `template<class T,int n> class FieldVector`, whose entries are of the numeric type `T`, e.g. `double` or `complex`, and the dimensions `n` and `m` are specified at compile time via template parameters. As the dimensions are already known to the compiler it can apply optimisations such as unrolling loops. Furthermore the dense systems can be solved efficiently using direct methods.

- `template<class T> class BCRSMatrix` denotes a block matrix stored in compressed row storage (CRS), each matrix entry is itself a matrix adhering to the interface. The type of the matrix block is given by the template parameter `T`, e.g. `FieldMatrix` in Figure 1(a) and `BCRSMatrix` in 1(c). The corresponding vector class is `template<class T> class BlockVector`. Each entry is itself a vector of type `T`.

Figure 1: Block structure of matrices arising in the finite element method



- Although currently not implemented, `template<class T> class VariableBCRSMatrix,` will be a CSR matrix of dense block matrices whose dimensions can vary from one entry to another like depicted in Figure 1(b). While it is possible to resemble this situation with a `BCRSMatrix` whose blocks are again of type `BCRSMatrix` a specialised implementation will be much more efficient as the entries can be stored in consecutive memory. `template<class T> class VariableBlockVector<B>` is the associated vector type, where each entry is a vector with a variable number of blocks of type `B`.

Note that the compiler can automatically apply inlining as all the type information is available to it. This allows ISTL to provide a fine grained interface, i.e. provide small functions like access to individual matrix entries, and still be comparable to hand crafted C code.

The block recursion can be exploited in the algorithms. Most preconditioners can be modified to honour this recursive structure for a specific number of block levels k. While treating the off-diagonal matrix blocks as traditional matrix entries a special treatment is applied to the diagonal blocks: If $k > 1$ the diagonal is treated as a matrix itself and the preconditioner is applied recursively on the matrix representing the diagonal value $D = A_{ii}$ with block level $k - 1$. For $k = 1$ the diagonal is treated as a matrix entry resulting in solving the represented linear system or an identity operation depending on the algorithm.

4 Parallel Solver Components

While other parallel solver libraries, like PETSc, Balay et al. [2004], use parallel data structures (matrices and vectors) that (implicitly) know the data distribution and communication patterns we decided to clearly separate the parallelisation aspects from the data structures used. This is done by imposing an abstract consistency model onto our linear algebra.

The solvers only use methods of instances of `LinearOperator`, `ScalarProduct` and `Preconditioner`, described in this section. These are provided in the constructor. Therefore the parallelisation is hidden from the solver algorithms in the parallel implementations of these interfaces.

Based on the description of the domain decomposition and according parallel discretisation we have everything in place to introduce these building blocks of our parallel solvers.

4.1 Scalar Products and Norms

One of the building blocks of Krylov methods is computing scalar products and norms on the underlying vector spaces. The base class `template<class X> ScalarProduct` provides methods `field_type dot(const X& x, const X& y)` and `double norm(const X& x)` to calculate these in the vector space described by the template parameter `X`.

Let x, y be vectors in a valid representation, then a parallel scalar product can easily be computed by calculating

$$x \cdot y = \sum_{i=0}^{P-1} (R_{I_i} x) \cdot (R_{I_i} y) \, .$$

Note that the projection can be done locally and the sum requires one global communication. Therefore we do not impose any additional prerequisites onto x, y.

4.2 Linear Operators

The base class `template<class X, class Y> LinearOperator` represents linear maps. The template parameter `X` is the type of the domain and `Y` is the type of the range of the operator. A linear operator provides the methods `apply(const X& x, Y& y)` and `applyscaledadd(field_type alpha, const X& x, Y& y)` performing the operations $y = A(x)$ and $y = y + \alpha A(x)$, respectively. The subclass `template<class M, class X, class Y> AssembledLinearOperator` represents linear operators that have a matrix representation. Conversion from any matrix into a linear operator is done by the class `template<class M, class X, class Y> MatrixAdapter`.

The prerequisites for both operations are that x is stored in a consistent representation and y is valid. On completion of both functions y has to be stored in a unique representation. Note that the discretisations described in Section 2.3 fulfil these constraints.

4.3 Preconditioners

The `template<class X, class Y> Preconditioner` provides the abstract base class for all preconditioners in ISTL. The

Table 1: Preconditioners

class	implements	rec.
`SeqJac`	Jacobi method	x
`SeqSOR`	successive over relaxation (SOR)	x
`SeqSSOR`	symmetric SSOR	x
`SeqILU`	incomplete LU decomposition (ILU)	
`SeqILUN`	ILU decomposition of order N	
`AMG`	algebraic multigrid method	

Table 2: ISTL Solvers

class	implements
`LoopSolver`	wrapper for looping over preconditioner application
`GradientSolver`	preconditioned gradient method
`CGSolver`	preconditioned conjugate gradient method
`BiCGSTABSolver`	preconditioned bi-conjugate gradient stabilised method

template parameter `X` is the type of the domain and `Y` is the type of the range of the operator it preconditions. The method `void pre(X& x, Y& b)` has to be called once before applying the preconditioner. Here `x` is the left hand side and `b` is the right hand side of the operator equation. The method may, e.g. scale the system, allocate memory or compute an (I)LU decomposition. The method `void post(X& x)` should be called after all computations to clean up allocated resources.

The method `void apply(X& v, const Y& d)` applies one step of the preconditioner to the system $A(v) = d$. Here `d` must contain the current residual in a unique representation and `v` must be 0. Upon exit of the method `v` contains the computed update to the current guess, i.e. $v = M^{-1}d$ where M is the approximate inverse of the operator A characterising the preconditioner. It is stored in a consistent representation.

See Table 1 for a list of available sequential preconditioners. In the list `AMG` can be used in sequential and parallel mode. They have the template parameters `M` representing the type of the matrix they work on, `X` representing the type of the domain and `Y` representing the type of the range of the linear system. The block recursive preconditioners are marked with an “x” in the last column. Their recursion depth is specified via an additional template parameter.

The sequential preconditioners can be used to build parallel ones using the `class BlockPrecondtioner`. It represents a block Jacobi method on the blocks represented by I_i. Instead of solving these blocks exactly, an arbitrary number of iterations of a sequential method is performed and the computed update is communicated to be in a consistent representation. The sequential method is provided in the constructor.

4.4 Solvers

All solvers are subclasses of the abstract base class `template<class X, class Y> InverseOperator`. It represents the inverse of an operator from the domain of type `X` to the range of type `Y`. The actual solving of the system $A(x) = b$ is done in the method `void apply(X& x, Y& b, InverseOperatorResult& r)`. The class `InverseOperatorResult` stores some statistics about the solution process, e.g. iteration count, achieved residual reduction, etc. All solvers only use methods of instances of `LinearOperator`, `ScalarProduct` and `Preconditioner`.

See Table 2 for a list of available solvers. All solvers are template classes with a template parameter `X` specifying the vector implementation.

With the parallel components fulfilling the prerequisites posed in the previous subsection we can now transform the sequential ISTL solvers into parallel solvers by plugging in matching parallel operators, scalar products and preconditioners.

5 Communication Software Components

In this section we will describe some implementational aspects of our communication interface. In parallel representations a random access container x, e.g. a plain C-array, cannot be stored with all entries on each process because of limited memory and efficiency reasons. Therefore it is represented by individual pieces x^p, $p \in \mathcal{P} = \{0, \ldots, P-1\}$, where x^p is the piece stored on process p of the P processes participating in the calculation. Although the global representation of the container is not available on any process, a process p needs to know how the entries of its local piece x^p correspond to the entries of the global container x, which is used in a sequential program. In this section we describe how the mapping of the local pieces to the global view and the communication interfaces are implemented and set up based on these mappings.

5.1 Index Sets

From an abstract point of view a random access container $x : I \to K$ provides a mapping from an index set $I \subset \mathbb{N}_0$ onto a set of objects K. Note that we do not require I to be consecutive. The piece x^p of the container x stored on process p is a mapping $x^p : I_p \to K$, where $I_p \subset I$. Due to efficiency the entries of x^p should be stored consecutively in memory. This means that for the local computation the data must be addressable by a consecutive index starting from 0.

When using adaptive discretisation methods there might be the need to reorder the indices after adding and/or deleting and adding some of the discretisation points afterwards. Therefore this index does not need to be persistent and can easily be changed. We will call this index *local index*.

For the communication phases of our algorithms these locally stored entries must also be addressable by a global identifier to be able to store the received values at and retrieve the values to be sent from the correct local index in the consecutive memory chunk. To ease the addition and removal of discretisation points this global identifier has to be persistent but does not need to be consecutive. We will call this global identifier *global index*.

5.1.1 IndexSet

Let $I \subset \mathbb{N}_0$ be an arbitrary, not necessarily consecutive, index set identifying all discretisation points of the computation. Furthermore let

$$(I_p)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p = I$$

be an overlapping decomposition of the global index set I into the sets of indices I_p corresponding to the global indices of the values stored locally in the chunk of process p.

Then the

```
template<typename TG, typename TL>
class IndexSet;
```

realises the one to one mapping

$$\gamma_p \,:\, I_p \longrightarrow I_p^{\text{loc}} := [0, n_p)$$

of the globally unique index onto the local index.

The template parameter `TG` is the type of the global index and `TL` is the type of the local index.

To be able to attach further information to the index the only prerequisite for the type of the local index is that it is convertible to `std::size_t` as it is used to address array elements.

The pairs of global and local indices are ordered by ascending global index. Thus it is possible to access the pairs via `operator[](TG& global)` in $log(n)$ time, where n is the number of pairs in the set. In an efficient code it is advisable to access the index pairs using iterators.

Due to the ordering, the index set can only be changed, i.e. indices added or deleted, in a special resize phase. By calling the functions `beginResize()` and `endResize()` the programmer indicates that the resize phase starts and ends, respectively. During the call of `endResize()` the deleted indices will be removed and the added indices will be merged with the existing ones.

5.1.2 ParallelLocalIndex

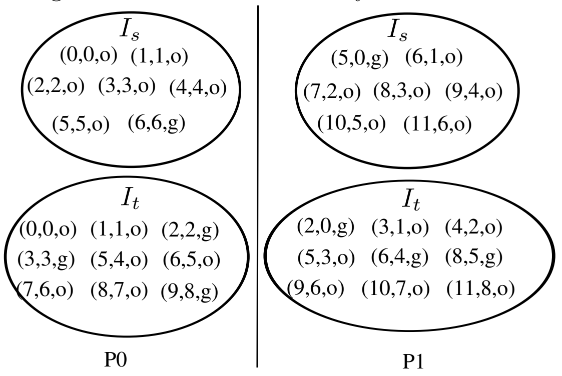
When dealing with overlapping index sets in distributed computing there often is the need to distinguish different partitions of the index set, e.g. I_i and $\tilde{I}_i \setminus I_i$ as introduced in Section 2.

This is accomplished by using the class

```
template<typename TA>
class ParallelLocalIndex;
```

where the template parameter `TA` is the type of the attributes used, e.g. `Flags` defined by

Figure 2: Index sets for array redistribution



```
enum Flags {owner, ghost};
```

where `owner` marks the indices $k \in I_i$ owned by process i and `ghost` the indices $k \notin I_i$ which are owned by other processes.

As the programmer often knows in advance which indices might also be present on other processes there is the possibility to mark an index as public.

As an example let us look at an array distributed between two processors. In Figure 3 one can see the array a as it appears in a sequential program. Below there are two different distributions of that array. The local views s_0 and s_1 are the parts process 0 and 1 store in the case that a is divided into two blocks. The local views t_0 and t_1 are the parts of a that process 0 and 1 store in the case that a is divided into 4 blocks and process 0 stores the first and third block and process 1 the second and fourth block. The decompositions have an overlap of one and the indices have the attributes `owner` and `ghost` visualised by white and shaded cells. The index sets I_s and I_t corresponding to the decompositions s_i and t_i, $i \in \{0, 1\}$, are shown in Figure 2 as sets of triples (g, l, a). Here g is the global index, l is the local index and a is the attribute (either o for `owner` or g for `ghost`).

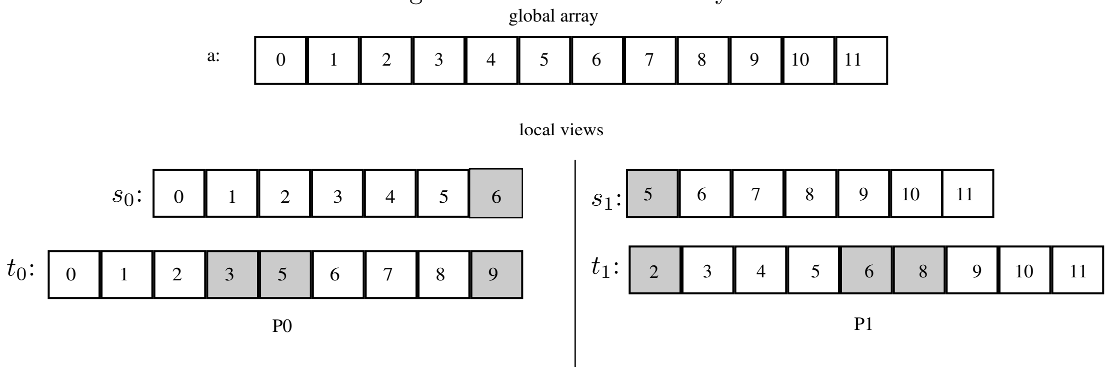
The following code snippet demonstrates how to set up the index set I_s on process 0:

```
// shortcut for index set type
typedef
   IndexSet<int,ParallelLocalIndex<Flags> >
   PIndexSet;
PIndexSet sis;
sis.beginResize();
for(int i=0; i<6; i++)
  sis.add(i, ParallelLocalIndex(i, owner));
sis.add(6, ParallelLocalIndex(6, ghost);
sis.endResize();
```

5.1.3 Remote Indices

To set up communication between the processes every process needs to know which indices are also known to other processes and which attributes are attached to them on the remote side. As there might be scenarios where data is exchanged between different index sets, e.g. if the data

Figure 3: Redistributed array



is agglomerated on lesser processes or redistributed, the communication is allowed to occur between different decompositions of the same index set.

Let $I \subset \mathbb{N}$ be the global index set and

$$(I_p^s)_{p\in\mathcal{P}}, \qquad \bigcup_{p\in\mathcal{P}} I_p^s = I$$

and

$$(I_p^t)_{p\in\mathcal{P}}, \qquad \bigcup_{p\in\mathcal{P}} I_p^t = I$$

be two overlapping decompositions of the same index set I. Then an instance of class `RemoteIndices` on process $p \in \mathcal{P}$ stores the sets of triples

$$\begin{aligned} r_s^{p\to q} = \{(g,(l,a),b)\,|\,g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^s(g),\\ a = \alpha_p^s(l), b = \alpha_q^t(\gamma_p^t(g))\} \end{aligned} \tag{6}$$

and

$$\begin{aligned} r_t^{p\to q} = \{(g,(l,a),b)\,|\,g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^t(g),\\ a = \alpha_p^t(l), b = \alpha_q^s(\gamma_p^s(g))\}\,, \end{aligned} \tag{7}$$

for all $q \in \mathcal{P}$. Here α_p^s and α_p^t denote the mapping of local indices on process p onto attributes for the index set I_p^s and I_p^t as realised by `ParallelLocalIndex`. Note that the sets $r_s^{p\to q}$ and $r_t^{p\to q}$ will only be nonempty if the processors p and q manage overlapping indexsetsets.

For our example in Figure 3 and Figure 2 the interface between I_s and I_t on process 0 is:

$$\begin{aligned}
r_s^{0\to 0} &= \{(0,(0,o),o),(1,(1,o),o),(3,(3,o),g),(5,(5,o),g),\\
&\quad (6,(6,g),o)\}\\
r_t^{0\to 0} &= \{(0,(0,o),o),(1,(1,o),o),(3,(3,g),o),(5,(4,g),o),\\
&\quad (6,(5,o),g)\}\\
r_s^{0\to 1} &= \{(2(2,o),g),(3,(3,o),o),(4,(4,o),o),(5,(5,o),o),\\
&\quad (6,(6,g),g)\}\\
r_t^{0\to 1} &= \{(5,(4.g),g),(6,(5,o),o),(7,(6,o),o),(8,(7,o),o),\\
&\quad (9,(8,g),o)\}
\end{aligned}$$

This information can either be calculated automatically by communicating all indices in a ring or set up by hand if the user has this information available. Assuming that `sis` is the index set I_s and `tis` the index set I_t setup as in the previous subsection and `comm` is an MPI communicator then the simple call

```
RemoteIndexSet<PIndexSet> riRedist(sis, tis,
     comm);
riRedist.rebuild<true>();
```

on all processes automatically calculates this information and stores it in `riRedist`. For a parallel calculation on the local views s_0 and s_1 calling

```
RemoteIndexSet<PIndexSet> riS(sis,sis,
    comm);
riS.rebuild<true>();
```

on all processes builds the necessary information in `riS`.

5.1.4 Communication Interface

With the information provided by class `RemoteIndices` the user can set up arbitrary communication interfaces. These interfaces are realised in `template<typename T> class Interface`, where template parameter `T` is the custom type of the `IndexSet` representing the index sets with attached attributes. Using these attributes attached to the indices by `ParallelLocalIndex` the user can select subsets of the indices for exchanging data, e.g. send data from indices marked as `owner` to indices marked as `ghost`.

Basically the interface on process p manages two sets for each process q it shares common indices with:

$$i_s^{p\to q} = \{l|(g,(l.a),b) \in r_s^{p\to q}|a \in A_s \wedge b \in A_t\}$$

and

$$i_t^{p\to q} = \{l|(g,(l,a),b) \in r_t^{p\to q}|a \in A_t \wedge b \in A_s\}\,,$$

where A_s and A_t are the attributes marking the indices where the source and target of the communication will be, respectively.

In our example these sets on process 0 will be stored for communication if $A_s = \{o\}$ and $A_t = \{o, g\}$:

$$\begin{aligned} i_s^{0\to 0} &= \{0,1,3,5\} \\ i_t^{0\to 0} &= \{0,1,3,4\} \\ i_s^{0\to 1} &= \{2,3,4,5\} \\ i_t^{0\to 1} &= \{5,6,7,8\}\,. \end{aligned}$$

The following code snippet whould build the interface above in `infRedist` as well as the interface `infS` to communicate between indices marked as `owner` and `ghost` on the local array views s_0 and s_1:

```
EnumItem<Flags,ghost> ghostFlags;
EnumItem<Flags,owner> ownerFlags;
Combine<EnumItem<Flags,ghost>,
        EnumItem<Flags,owner> > goFlags;
Interface<PIndexSet> infRedist;
Interface<PIndexSet> infS;
infRedist.build(riRedist, ownerFlags,
    goFlags);
infS.build(riS, ownerFlags, ghostFlags);
```

5.1.5 Communicator

Using the classes from the previous sections all information about the communication is available and we are set to communicate arbitrary data types. The only prerequisite for the data type is that its values are addressable via `operator[](size_t index)`, which should be safe to assume.

The main goal of our communicators is that we are not only able to send one data item per index, but also different numbers of data elements for each index (provided they have the same type). This is supported in a generic way by the traits class `template<class V> struct CommPolicy` describing the the array-like data type `V`. The `typedef IndexedType` is the atomic type to be communicated and `typedef IndexedTypeFlag` is either `SizeOne` if there is only one data item per index or `VariableSize` if the number is variable.

The default implementation works for all array-like containers, which provide only one data item per index. For all other containers the user has to provide its own custom specialisation. For the vector classes of ISTL (up to two block levels) those specialisations are already implemented.

Either `template<class V> class DatatypeCommunicator` or `template<class V> class BufferedCommunicator` perform the actual communication. From the information provided in `RemoteIndices` a custom `MPI_Datatype` for the `class V` is created by `DatatypeComunicator` using the source and target attribute sets. This directly tells MPI what values of the containers to send and at what indices to store the received data. Only plain send and receive operations are supported. The interface information described in the previous subsection is managed directly by the `MPI_Datatype`. This means that it is tightly coupled to the layout of the container and it is not possible to communicate values of containers represented by a different class even if they have the same layout in terms of index access.

The `template<class T> class BufferedCommunicator`, where `T` is the type of the `RemoteIndices` provides far more flexibility. As the information about the communication interface is managed separately by class `Interface` it is possible to communicate data of arbitrary containers without reconstructing the interface.

Before the communication can start one has to call the `build` method with the data source and target containers as well as the communication interface as arguments. Assuming `s` and `t` as arrays s_i and t_i, respectively, then

```
BufferedCommunicator<
  RemoteIndices<PindexSet> > comm;
BufferedCommunicator<
  RemoteIndices<PindexSet> > commRedist;
comm.build(s, s, infS);
commRedist.build(s, t, infRedist);
```

demonstrates how to set up the communicator `commRedist` for the array redistribution and `comm` for a parallel calculation on the local views s_i. The `build` function calculates the size of the messages to send to other processes and allocates buffers for the send and receive actions. The representatives `s` and `t` are needed to get the number of data values at each index in the case of variable numbers of data items per index. Note that, due to the generic programming techniques used, the compiler knows if the number of data points is constant for each index and will apply a specialised algorithm for calculating the message size without querying neither `s` nor `t`. Clean up of allocated resources is done either by calling the method `free()` or automatically in the destructor.

The actual communication takes place if one of the methods `forward` and `backward` is called. In our case in `commRedist` the `forward` method sends data from the local views s_i to the local views t_i according to the interface information and the `backward` method in the opposite direction.

The following code snippet first redistributes the local views s_i of the global array to the local views t_i and performs some calculation on this representation. Afterwards the result is communicated backwards.

```
comm.forward<CopyData>(s,t);
 // calculate on the redistributed array
doCalculations(&t);
comm.backward<AddData>(s,t);
```

Note that both methods have a different template parameter, either `CopyData` or `AddData`. These are policies for gathering and scattering the data items. The former just copies the data from and to the location while the latter copies from the location but adds the received data items to the target entries. Assuming our data is stored in simple C-arrays `AddData` could be implemented like this

```
struct AddData{
  typedef typename double IndexedType;

  double gather(const T* v, int i){
    return v[i];
  }
  void scatter(T* v, double item, int i){
    v[i]+=item;
  }
};
```

Note that arbitrary manipulations can be applied to the communicated data in both methods.

For containers with multiple data items associated with one index `gather` and `scatter` must have an additional integer argument specifying the subindex.

5.2 Collective Communication

While communicating entries of array-like structures is a prominent task in parallel iterative solver codes one must not neglect collective communication operations, like gathering and scattering data from and to all processes, respectively, or waiting for other processes. An abstraction for these operations is crucial for decoupling the communication from the parallel programming paradigm used.

Therefore we designed `template<class T> class CollectiveCommunication` which provides information of the underlying parallel programming paradigm as well as the collective communication operations as known from MPI. See Table 3 for a list of all functions.

Currently there is a default implementation for sequential programs as well as a specialisation working with MPI. This approach allows for running parallel programs sequentially without any parallel overhead simply by choosing the sequential specialisation at compile time. Note that the interface is far more convenient to use than the C++ interface of MPI. The latter is a simple wrapper around the C implementation without taking advantage of the power of generic programming.

Note that we do not attempt to replicate the complete set of MPI functions using generic programming as done in Boost.MPI, Gregor and Troyer [2006].

5.3 Use Case

Using the ISTL framework it becomes rather easy to develop custom parallel preconditioners. As an example we will implement the so-called parallel hybrid smoother, as described in Yang [2004]. The main idea is to apply the sequential smoother on each subdomain corresponding to the indices of the unique partition. With the discretisation matrices in the non-overlapping case as described in Section 2.4 this can be done rather easy. Note that all matrix rows corresponding to indices $k \notin I_i$ on process i are set up as Dirichlet boundary conditions. Furthermore the prerequisite of the preconditioner is that the vector the preconditioner is applied to is stored in a unique representation. This means that entries of the vector corresponding to indices $k \in I_i$ are not influenced by those corresponding to indices outside of I_i.

Let the index sets be set up and all indices corresponding to I_i on process i be marked as `owner` and all other indices as `ghost`. Furthermore let `comm` be a communicator set up just like the one in Subsection 5.1.5 for the communication between `owner` to `ghost` indices. Then the following code snippet presents the `apply` method of our preconditioner.

```
void apply(X& v, const X& d){
  // apply sequential preconditioner
  seqPrec.apply(v,d);
  // make v consistent
  comm.forward<CopyData>(v,v);
}
```

Here `seqPrec` can be an arbitrary sequential preconditioner described in Table 1. Note that in the `pre` method one could calculate a relaxation factor and apply it before the communication in `apply` to get a smoother that is more scalable.

6 Related Work

Naturally, there are many software packages out there for solving large sparse linear systems. While many of them provide full grown sparse linear algebra, few of them are specifically designed for linear systems created by finite element discretisations of systems of partial differential equations. Only approaches with a scope similar to ISTL are discussed here. For a more general overview see Dongarra [2006] and Eijkhout [1997].

Epetra of Trilinos Epetra provides the linear algebra system used in the Trilinos project of Sandia National Labs. Within the Trilinos project parallel solver algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific applications are developed.

Epetra supports block matrices of dense matrices, where each entry can have a different size. Each matrix entry is a dense matrix object of which the outer sparse matrix manages a pointer to. For the case of constant block sizes Epetra block matrices provide the possibility to copy the block values to contiguous memory. Then they provide hand coded loop unrolling in the matrix-vector-multiplication in the case of of blocks with less than 5 entries per block row. Still the operation to optimise the storage needs additional memory which might be a problem for large systems. Even if all blocks have the same size the information about the block sizes is not available at compile time which prevents the compiler from optimising code like it can be done in ISTL.

All vector and sparse matrix data structures within Epetra are distributed objects which know their data distribution. While this means that one does not need to impose a special additive representation onto the local matrix entries, like described in Section 2.4, Epetra has to initiate communication for each global matrix-vector-operation to always keep the vectors stored in consistent representation. Unfortunately two additional vectors need to be allocated

Table 3: Collective Communication Functions

Function	Description
`int rank()`	Get the rank of the process
`int size()`	Get the number of processes
`template<typename T> T sum (T& in)`	Compute global sum
`template<typename T> T prod (T& in)`	Compute global product
`template<typename T> T min (T& in)`	Compute global minimum
`template<typename T> T max (T& in)`	Compute global maximum
`void barrier()`	Wait for all processes.
`template<typename T> int broadcast (T* inout, int len, int root)`	Broadcast an array from root to all other processes
`template<typename T> int gather (T* in, T* out, int len, int root)`	Gather arrays at a root process
`template<typename BinaryFunction, typename Type> int allreduce(Type* in, Type* out, int len)`	Combine values from all processes on all processes. Combine function is given with `BinaryFunction`

temporarily for this and this approach results in more communication steps then needed during iteratively solving a linear system as the solver programmer is not given the power to initiate the communication when he thinks it is necessary.

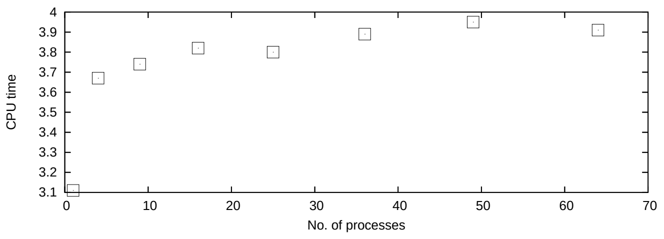
PETSc The "Portable, Extensible Toolkit for Scientific Computation" (PETSc), Balay et al. [2004] and Balay et al. [1997], also supports parallel sparse block matrices with square dense blocks of fixed size. The parallel sparse matrices are partitioned by matrix rows. The partitioning is non-overlapping. Internally two matrices are stored, one square matrix with just the columns corresponding to the local rows and another matrix with all off-diagonal values that correspond to rows not owned by the process. This allows PETSc to overlap the computation of the matrix-vector-product with the communication needed to get values not owned by the process. Still as with Epetra this means that the amount of communication needed is not optimal for the scenario described here.

PETSc supports basic index sets which provide a mapping from local to global indices. These can be used to scatter or gather floating point vectors. In contrast to ISTL more complex communication schemes or communication of other numeric types or data structures cannot be realised.

Hypre The parallel sparse matrix formats of the "high performance preconditioners" (hypre), Falgout and Yang [2006], do not support any kind of block structure although some of its grid interfaces and the finite element interface do so. The matrices are distributed row-wise and each process can only manage a consecutive set of global matrix rows.

Unfortunately all parallelisation aspects are hidden from the user and application developer. This black box approach makes it very hard to supplement or customise the library.

Figure 4: Time for one iteration of `BlockPreconditioner` ($4 \cdot 10^6$ unknowns per process)



7 Numerical Results

The hybrid smoother described in the previous section without outer relaxation factor is not really suitable as preconditioner because of its h-dependency. Still it can used to measure the overhead the parallel components of ISTL pose. Figure 4 shows the time per iteration for different numbers of processors. The problem size is scaled with the number of processes and each process performs the computation of $4 \cdot 10^6$ unknowns. The Laplace equation discretised on a 2D tensor product mesh is solved with the conjugate gradient method preconditioned with the `BlockPreconditioner` with one sequential SSOR sweep as the inner relaxation. The calculation was done on Helics at IWR in Heidelberg, a Linux cluster of 256 AMD 1.4 Ghz dual processor nodes connected by 2 Gb/s Myrinet. It shows that the parallel overhead of ISTL is negligible.

The next model application is a simulation of water infiltration into a macroporous soil. The problem is modelled by the Richards equation, which is solved with an incomplete Newton method within a time stepping BDF scheme. For a complete description of the scheme see Vogel et al. [2006].

The resulting linear system of the first Newton step in the first time step, discretised with cell centred Finite Volumes on a 3D tensor product mesh, is solved using the agglomeration AMG method available in ISTL until a relative defect reduction of 10^{-5} is achieved. It

Table 4: Water infiltration into macroporous soil

P	grid size	setup time	solve time	no. of its.	time / iteration	total time
1	60	2.21	0.44	2	0.22	1.96
8	120	3.76	0.82	3	0.27	4.58
27	180	5.90	1.29	4	0.32	7.19

uses simple piecewise constant prolongation and restriction. `BlockPreconditioner` with inner ILU0 relaxation is used as a smoother. AMG serves as a preconditioner to the bi-conjugate gradient stabilised method.

In Table 4 for all numbers of processes (P) the number of cells in each grid dimension, the setup time, the solution time, the number of iterations, the time per iteration and the total time to solution are shown. The calculation was done on the Linux cluster Mozart at IPVS in Stuttgart, a 64 node dual processor Intel Xeon (3.066 GHz, 1 MByte Level 3 Cache) cluster with Infiniband 4x networking. This shows that with our approach it is possible to create reasonable scalable solvers for real world problems.

8 Conclusion

We have shown that it is possible to parallelise existing sequential linear solvers by imposing an abstract consistency model upon the building blocks of preconditioned Krylov methods. The operators constructed are suitable for being used directly in preconditioners. With the approach presented here it is possible to reuse the existing sequential algorithm for the parallel solvers.

Using the power of generic programming in C++ it is possible to base the parallel communication on the abstraction of distributed index sets. This allows for decoupling the communication methods from the actual parallel programming paradigm used. Thus it is easy to switch to other paradigms if necessary.

As currently the only abstraction of the communication library is MPI, a next step is to support SMP systems using OpenMP or POSIX threads.

The whole DUNE project, of which ISTL is only one part, is licensed under the LGPL and can be obtained from http://www.dune-project.org.

REFERENCES

K. Arrow, L. Hurwicz, and H. Uzawa. *Studies in Nonlinear Programming.* Stanford University Press, Stanford, Ca, 1968.

S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

P. Bastian and R. Helmig. Efficient fully-coupled solution techniques for two-phase flow in porous media. Parallel multigrid solution and large scale computations. *Adv. Water Res.*, 23:199–216, 1999.

M. Blatt and P. Bastian. The Iterative Solver Template Library. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 666–675. Springer, 2007.

J. Dongarra. List of freely available software for linear algebra on the web, 2006. `http://netlib.org/utk/people/JackDongarra/la-sw.html`.

V. Eijkhout. Overview of iterative linear system solver packages, 1997. ftp://math.ucla.edu/pub/eijhout/papers/packages.ps.

R. D. Falgout and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 267–294. Springer-Verlag, 2006.

D. Gregor and M. Troyer. Boost.MPI. `http://www.boost.org/`, 2006.

M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31, September 2005.

S. Patankar. *Numerical Heat Transfer and Fluid Flow.* MCGraw-Hill, 1980.

H.-J. Vogel, I. Cousin, O. Ippisch, and P. Bastian. The dominant role of structure for solute transport in soil: experimental evidence and modelling of structure and transport in a field experiment. *Hydrology and Earth System Sciences*, 10:495–506, July 2006.

U. M. Yang. On the use of relaxation parameters in hybrid smoothers. *Numerical Linear Algebra With Applications*, 11(2–3):155–172, 2004.