Automatic Parallelization with Expression Templates*

Christoph Pflaum† Robert D. Falgout‡

November 21, 2001

Abstract

A new automatic parallelization technique is presented for numerical algorithms for solving partial differential equations. This technique is based on expression templates and is applied to a finite element discretizations on semi-unstructured grids.

Key words. automatic parallelization, expression templates, numerical solution of PDE's

1 Introduction

Achieving an efficient parallelization of a numerical algorithm is time consuming and difficult. Automatic parallelization techniques can ease this process considerably. For an overview of some of these techniques, see [2].

In this paper, we present a new approach for automatic parallelization, based on expression templates. The key to an efficient and flexible automatic parallelization is the language used to implement a numerical algorithm. Such a language should satisfy two properties. First, the language should be close to the mathematical language for describing the numerical algorithm. Second, the language should provide global information about the algorithm so that automatic parallelization can be done. Such a language can be constructed using *expression templates* [6]. Expression templates provide an efficient implementation of

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

†Universität Würzburg, Institut für Angewandte Mathematik und Statistik, Am Hubland, 97074 Würzburg, Germany.

‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551.

operator overloading in C++. They have been used in the library Blitz++ for numerical linear algebra (see [5]) and later in the library EXPDE for the numerical solutions of partial differential equations (see [3]).

The focus of this paper is the automatic parallelization of algorithms for the numerical solution of PDE's. Here, we restrict ourselves to three-dimensional problems discretized on semi-unstructured grids (see [4]). In Section 2, we first describe the parallelization of two simple iterative methods applied to Poisson's equation on rectangular two-dimensional domains. These parallel algorithms require a small number of communications, even in case of the Gauss-Seidel iteration. They were first presented in [1] and [2], but are recalled here for completeness. In Sections 3 and 4, we extend the approach to general domains in three dimensions. The concept of automatic parallelization with expression templates is described in Section 5, and numerical results are presented in Section 6.

2 Parallelization of Poisson's problem on a square

As a model problem let us first study the parallelization of Poisson's problem on a square. The main idea of the parallelization concept described in this section can be found in [1] and [2].

Let us assume that $f \in \mathcal{C}([0,1]^2)$ is a continuous function on the unit square. Then, the solution of Poisson's problem $u \in \mathcal{C}^2([0,1]^2)$ is the solution of the equation

$$\begin{aligned} -\triangle u &= f \quad \text{on } \Omega =]0,1[^2 \text{ and} && (1)\\ u &= 0 \quad \text{on } \partial\Omega. && (2) \end{aligned}$$

The finite element discretization with bilinear elements of Poisson's equation on the grid $\Omega_h = \{(ih, jh) \mid i, j = 0, 1, \cdots, N = 1/h\}$, where $N = 2^n, n \in \mathbb{N}$, is the solution u_h of the difference equation

$$\begin{aligned} 9U_h(x,y) - \sum_{k,l\in\{-1,0,1\}} U_h(x+kh, y+lh) &= F_h(x,y) \quad \text{for all } (x,y) \in \mathring{\Omega}_h \\ U_h(x,y) &= 0 \qquad \text{for all } (x,y) \in \Omega_h \backslash \mathring{\Omega}_h, \end{aligned}$$

where $\mathring{\Omega}_h = \Omega_h \cap]0,1[^2$ denotes the interior grid points and $F_h(x,y)$ is a suitable scaled local average of the right hand side f. For simplicity, let us denote

$$L_h(U_h)(x,y) = -9U_h(x,y) + \sum_{k,l\in\{-1,0,1\}} U_h(x+kh, y+lh).$$

the discrete Laplace operator.

The Jacobi iteration and the Gauss-Seidel iteration are iterative solvers for calculating the finite element solution U_h. These solvers calculate a sequence of iterative approximations $\tilde{U}_h$, which converge to the solution U_h. The Jacobi iteration can be described as follows:

Jacobi iteration with parameter $0 < \omega < 2$.
Calculate

$$R_h(p) := F_h(p) + L_h(\tilde{U}_h)(p) \quad \text{for all } p \in \mathring{\Omega}_h.$$

and then

$$\tilde{U}_h(p) := \tilde{U}_h,(p) - \omega\, R_h(p) \quad \text{for all } p \in \mathring{\Omega}_h.$$

The Gauss-Seidel iteration we want to study here, is based on a coloring of the grid points $\mathring{\Omega}_h$. A coloring is a set $\mathcal{F}$ of disjoint subsets $F \subset \mathring{\Omega}_h$, defined as follows:

Coloring of structured grid points in 2D.

$$\begin{aligned}
\mathcal{F} &= \{F_{SW}, F_{SE}, F_{NW}, F_{NE}\},\\
F_{SW} &= \{(2ih, 2jh) \mid i,j = 0,1,\cdots,N/2\},\\
F_{SE} &= \{((2i+1)h, 2jh) \mid i = 0,1,\cdots,N/2-1;\ j = 0,1,\cdots,N/2\},\\
F_{NW} &= \{(2ih, (2j+1)h) \mid j = 0,1,\cdots,N/2-1;\ i = 0,1,\cdots,N/2\},\\
F_{NE} &= \{((2i+1)h, (2j+1)h) \mid i,j = 0,1,\cdots,N/2-1\}.
\end{aligned}$$

Furthermore, let us assume that the colors of $\mathcal{F}$ are ordered by F_1, F_2, F_3, F_4. Then, the Gauss-Seidel iteration can be described as follows:

Gauss-Seidel iteration.
For $i = 1,\cdots,4$ calculate:

$$\tilde{U}_h(p) := \tilde{U}_h(p) - F_h(p) - L_h(\tilde{U}_h)(p) \quad \text{for all } p \in F_i \in \mathcal{F}. \qquad (3)$$

It is important to interpret equation (3) in the following way. $\tilde{U}_h(p)$ on the left hand side is the new value of the approximation $\tilde{U}_h(p)$. $\tilde{U}_h(p)$ on the right hand side is the actual value of $\tilde{U}_h(p)$. These are the old values $\tilde{U}_h(p)$, when no new values exist, and the new values of $\tilde{U}_h(p)$, when they are already calculated.

For the parallelization of the Jacobi and Gauss-Seidel iteration let us subdivide Ω_h in $N_p = 2^{2m}$ processor subregions, where $0 < m < n$. These processor subregions are small squares of length $H = 2^{-m}$:

$$P_{(x,y)} = [x - H/2, x + H/2) \times [y - H/2, y + H/2),$$

where the (x, y) are the middle points of these squares. The set of these middle points is

$$\mathcal{M}_H = \Big\{((i-1/2)H, (j-1/2)H) \;\Big|\; i,j = 1,\cdots,2^m\Big\}.$$

The processor with middle point $c \in \mathcal{M}_H$ is responsible for the local subgrid $\Omega_h \cap P_c$, and will be referred to as processor c in what follows.

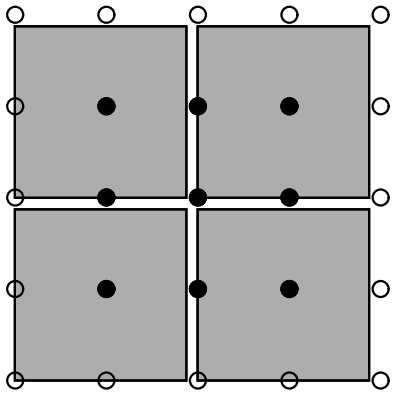

Figure 1: Subdivision of the grid $\Omega_{1/4}$ in 4 processor subregions.

Example:
Figure 1 depicts a subdivision of the grid $\Omega_{1/4}$ in 4 processor subregions. The interior grid points $\mathring{\Omega}_{1/4}$ are marked black. The middle points of the 4 processor subregions are

$$\mathcal{M}_{1/4} = \Big\{(1/4, 1/4), (3/4, 1/4), (1/4, 3/4), (3/4, 3/4)\Big\}.$$

To parallelize the above iterative solvers, we introduce *local boundary subdomains* for every processor. A processor needs the grid points contained in these subdomains for performing an iteration on its local subgrid. We define 4 different kinds of local boundary subdomains:

$$\begin{aligned}
B_{(x,y),W} &= [x - H/2 - h, x - H/2) \times [y - H/2 - h, y + H/2], \\
B_{(x,y),E} &= \{x + H/2\} \times [y - H/2 - h, y + H/2], \\
B_{(x,y),S} &= [x - H/2 - h, x + H/2] \times [y - H/2 - h, y - H/2), \\
B_{(x,y),N} &= [x - H/2 - h, x + H/2] \times \{y + H/2\}.
\end{aligned}$$

Figure 2 depicts these subdomains. One can see that the subdomains $B_{(x,y),W}$ and $B_{(x,y),S}$ are 2-dimensional domains and that $B_{(x,y),E}$ and $B_{(x,y),N}$ are lines. The reason for this will become apparent later in the paper when we extend the method to more general domains.

To parallelize the above iterative solvers we only want to apply updates from a direction $D \in \{W, E, N, S\}$. These updates are defined as follows:

Updates.
Update W: For each $(x-H, y), (x, y) \in \mathcal{M}_H$, an update from direction W is a communication of all values $\tilde{U}_h$ at the points $B_{(x,y),W} \cap \Omega_h$ from processor $(x - H, y)$ to processor (x, y).
Update E,N,S: Define analogously.

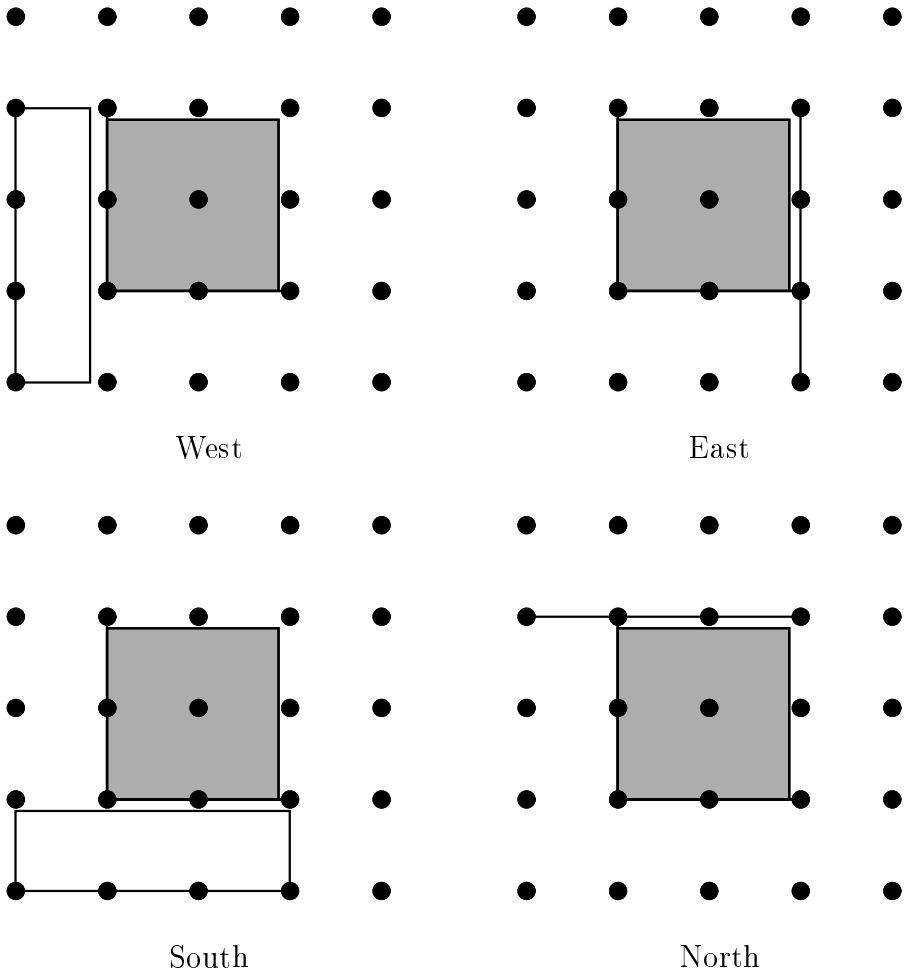

Figure 2: Four local boundary subdomains for a processor.

Since the Jacobi iteration and the Gauss-Seidel iteration involve the evaluation of a nine point stencil, the interior processors need information from 8 neighbor processors. For example the processor with subregion $P_{(x,y)}$ needs informations from the processor with subregion $P_{(x-H,y-H)}$. Since, we do not want to communicate data diagonally from direction SW, subregion $P_{(x,y)}$ must get this data from subregion $P_{(x-H,y-H)}$ through subregion $P_{(x,y-H)}$ via an update W followed by an update S.

In [1], the following efficient parallelization of the Jacobi and Gauss-Seidel iterations is described:

Parallel Jacobi iteration with parameter $0 < \omega < 2$.

1. *Update:* Perform the following updates:
   Update from W.
   Update from S.
   Update from E.
   Update from N.
2. *Smooth:* Calculate in parallel for every processor $c \in \mathcal{M}_H$:

$$R_h(p) := F_h(p) + L_h(\tilde{U}_h)(p) \quad \text{for all } p \in \mathring{\Omega}_h \cap P_c.$$

and

$$\tilde{U}_h(p) := \tilde{U}_h(p) - R_h(p)\omega \quad \text{for all } p \in \mathring{\Omega}_h \cap P_c.$$

Parallel Gauss-Seidel iteration.
Assume that the local boundary values of $\tilde{U}_h$ are updated at the points $(B_{(x,y),N} \cup B_{(x,y),E}) \cap \mathring{\Omega}_h$. Then perform:

1. Parallel smoothing of color F_{NE} for every processor $c \in \mathcal{M}_H$:

$$\tilde{U}_h(p) := \tilde{U}_h(p) - F_h(p) - L_h(\tilde{U}_h)(p) \quad \text{for all } p \in F_{NE} \cap P_c.$$

2. Update from W.
3. Parallel smoothing of color F_{NW} for every processor $c \in \mathcal{M}_H$:

$$\tilde{U}_h(p) := \tilde{U}_h(p) - F_h(p) - L_h(\tilde{U}_h)(p) \quad \text{for all } p \in F_{NW} \cap P_c.$$

4. Update from S.
5. Parallel smoothing of color F_{SW} for every processor $c \in \mathcal{M}_H$:

$$\tilde{U}_h(p) := \tilde{U}_h(p) - F_h(p) - L_h(\tilde{U}_h)(p) \quad \text{for all } p \in F_{SW} \cap P_c.$$

6. Update from E.
7. Parallel smoothing of color F_{SE} for every processor $c \in \mathcal{M}_H$:

$$\tilde{U}_h(p) := \tilde{U}_h(p) - F_h(p) - L_h(\tilde{U}_h)(p) \quad \text{for all } p \in F_{SE} \cap P_c.$$

8. Update from N.

After this iteration, the local boundary values of $\tilde{U}_h$ are updated at the points $(B_{(x,y),N} \cup B_{(x,y),E}) \cap \mathring{\Omega}_h$.

3 Parallelization of Poisson's problem on general domains in 2D using semi-unstructured grids.

In this section, we extend the parallelization approach described in Section 2 to general domains in 2D. To this end, we apply semi-unstructured grids. These grids consist of a large structured grid in the interior of the domain and an unstructured grid, which is only contained in boundary cells. A detailed description of semi-unstructured grids for general domains in 2D and 3D is given in [4]. Here, we describe only the main properties of semi-unstructured grids.

Let us assume that a partial differential equation is given on a domain Ω. For reasons of simplicity, let us assume that $\Omega \subset]0,1[^2$. A semi-unstructured grid generation is based on the structured grid Ω_h, described in Section 2, and leads to the following objects:

- All cells $[ih,(i+1)h] \times [jh,(j+1)h]$, $i,j = 0,\cdots N-1$ are classified in *interior cells*, *boundary cells*, and *exterior cells*. The boundary of Ω cuts the boundary cells. This cut is approximated by triangles for every boundary cell. The union of all these triangles and all interior cells is the *discretization domain* $\Omega_{dis,h}$.

- The semi-unstructured grid is the set of *nodal points*

$$\mathcal{N}_h \;:=\; \mathcal{N}_{h,i} \cup \partial\mathcal{N}_h,$$

  where $\partial\mathcal{N}_h$ are the boundary nodal points and $\mathcal{N}_{h,i} \subset \Omega_h$ are the interior nodal points. The boundary nodal points are constructed in such a way that every boundary nodal point $p \in \mathcal{N}_h$ is contained in the interior of an edge of a boundary cell.

Figure 3 shows a semi-unstructured grid and Figure 4 depicts an example of a boundary cell with two interior nodes and two boundary nodes on the east and west edge of the boundary cell. The boundary cell cut is subdivided in two triangles.

To parallelize algorithms on semi-unstructured grids, we can use the subregions P_c. Let us call a subregion P_c active, if $\bar{P}_c \cap \Omega_{dis,h} \neq \emptyset$. To every active subregion P_c corresponds one processor. To obtain an efficient load balancing one has to move and stretch the box $]0,1[^2$, which surrounds the domain Ω, such that no active subregion P_c contains only a small number of grid points. This can be done automatically before the grid generation process. Furthermore, observe that every subregion P_c adjacent to a nodal point $p \in \mathcal{N}_h$ is an active subregion.

Now, let us assume we want to solve numerically Poisson's problem with Dirichlet and Neumann boundary conditions on the domain Ω. This equation

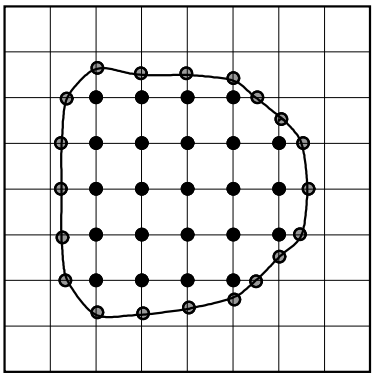

Figure 3: Semi-unstructured grid.

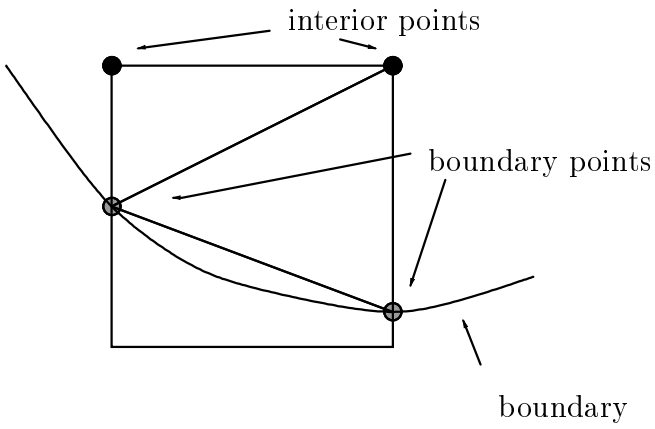

Figure 4: Boundary cell with two interior and two boundary nodes.

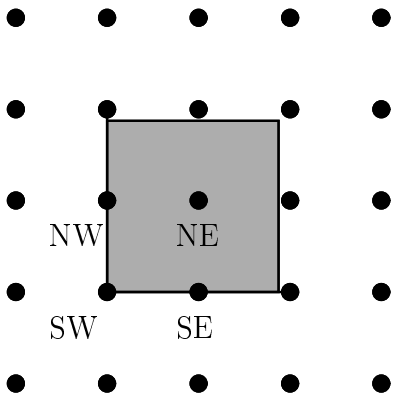

Figure 5: Colors of grid points for processor subgrid $\Omega_h \cap P_c$.

can be described as follows:

$$\begin{aligned} -\triangle u &= f \quad \text{on } \Omega, \\ u &= 0 \quad \text{on } \Gamma_D \subset \partial\Omega \\ \frac{\partial u}{\partial \vec{n}} &= 0 \quad \text{on } \Gamma_N \subset \partial\Omega. \end{aligned}$$

Here, the disjoint sets Γ_D and Γ_N must satisfy the property $\Gamma_D \cup \Gamma_N = \partial\Omega$. Similar to the discretization in Section 2, the finite element discretization of the above equation with linear elements leads to an equation of the form

$$L_h(U_h)(x,y) = F_h(x,y) \quad \text{for all } (x,y) \in \mathcal{N}_h \cap (\Omega \cup \Gamma_N),$$

where the operator L_h has a similar local structure as the operator L_h in Section 2. For the definition of a Gauss-Seidel iteration, we color the interior nodal points $\mathcal{N}_{h,i}$ as in Section 2. But, we additionally have to color the boundary nodal points $\partial\mathcal{N}_h$. This has to be done in such a way that a Gauss-Seidel iteration can be performed with a low number of communications.

To this end, observe that a point on a processor subdomain $p \in \Omega_h \cap P_c$ is either contained in the interior of P_c (denoted $\mathring{P}_c$) or contained in the W-edge or S-edge of P_c. Furthermore, the coloring in Section 2 has the following property:

- The points $p \in F_{NE} \cap P_c$ are contained in the interior of P_c.
- The points $p \in F_{NW} \cap P_c$ are not contained in the S-edge of P_c.
- The points $p \in F_{SE} \cap P_c$ are not contained in the W-edge of P_c.

The coloring of the boundary nodal points $\partial\mathcal{N}_h$ has to be done in such a way that this property is satisfied. This can be achieved in the following way:

Coloring of semi-unstructured nodal grid points in 2D.
The interior nodal points $\mathcal{N}_{h,i}$ are colored as in Section 2. This leads to the sets $F_{SW,i}$, $F_{SE,i}$, $F_{NW,i}$, and $F_{NE,i}$ such that

$$\mathcal{N}_{h,i} = F_{SW,i} \cup F_{SE,i} \cup F_{NW,i} \cup F_{NE,i}.$$

Furthermore, define the sets $F_{NE,b}$, $F_{SE,b}$, $F_{NW,b}$ as follows. $p \in \partial\mathcal{N}_h$ is contained in

- $F_{NE,b}$, if there is a $c \in \mathcal{M}_H$ such that $p \in \mathring{P}_c$,
- $F_{NW,b}$, if there is a $c \in \mathcal{M}_H$ such that p is contained in the W-edge of $\mathcal{M}_H$,
- $F_{SE,b}$, if there is a $c \in \mathcal{M}_H$ such that p is contained in the S-edge of $\mathcal{M}_H$.

Now, define

$$\begin{aligned} \mathcal{F} &= \{F_{SW}, F_{SE}, F_{NW}, F_{NE}\}, \\ F_{SW} &= F_{SW,i}, \\ F_{SE} &= F_{SE,i} \cup F_{SE,b}, \\ F_{NW} &= F_{NW,i} \cup F_{NW,b}, \\ F_{NE} &= F_{NE,i} \cup F_{NE,b}. \end{aligned}$$

Updates can easily defined as follows:

Updates.
Update W: For each $(x-H, y), (x, y) \in \mathcal{M}_H$, an update from direction W is a communication of all values $\tilde{U}_h$ at the points $B_{(x,y),W} \cap \mathcal{N}_h$ from processor $(x - H, y)$ to processor (x, y).
Update E,N,S: Define analogously.

Using the above coloring of semi-unstructured grid points and the above definition of updates, one can apply the parallel Gauss-Seidel iteration and the parallel Jacobi iteration defined in Section 2.

4 Parallelization on general domains in 3D using semi-unstructured grids.

In this section, we generalize the concepts described in the last two sections to the 3D case. For the most part, this is straightforward. For example the generalization of the processor subregion is

$$P_{(x,y,z)} = [x - H/2, x + H/2) \times [y - H/2, y + H/2) \times [z - H/2, z + H/2)$$

for every $(x, y, z) \in \mathcal{P}_H$, where $H = 2^{-m}$ and

$$\mathcal{P}_H = \Big\{((i-1/2)H, (j-1/2)H, (k-1/2)H) \;\Big|\; i, j, k = 1, \cdots, 2^m\Big\}.$$

The coloring of the nodal points leads to eight colors

$$\mathcal{F} = \{F_{DSW}, F_{DSE}, F_{DNW}, F_{DNE}, F_{TSW}, F_{TSE}, F_{TNW}, F_{TNE}\}.$$

One difference between semi-unstructured grids in 2D and 3D is that semi-unstructured grids in 3D may contain additional nodal points $\mathcal{N}_{h,b}$ in the interior of a few boundary cells. These grid points must be colored by F_{TNE}.

The generalization of the update leads to 6 update directions *W,E,S,N,D,T*. Since the number of update directions is not the number of colors as in 2D, the generalization of the parallel Gauss-Seidel and Jacobi iterations to the 3D case is not straightforward. The iterations are as follows:

Parallel Jacobi iteration in 3D

1. *Update:* Perform the following updates:
   Update from *W*.
   Update from *S*.
   Update from *E*.
   Update from *N*.
   Update from *T*.
   Update from *D*.
2. *Smooth.*

Parallel Gauss-Seidel iteration in 3D

Assume that the local boundary values of $\tilde{U}_h$ are updated at the points $(B_{(x,y,z),E} \cup B_{(x,y,z),N} \cup B_{(x,y,z),T}) \cap \mathring{\Omega}_h$. Then perform:

1. Parallel smoothing of color F_{TNE}.
2. Update from *W*.
3. Parallel smoothing of color F_{TNW}.
4. Update from *S*.
5. Parallel smoothing of color F_{TSW}.
6. Update from *D*.
7. Parallel smoothing of color F_{DSW}.
8. Update from *N*.
9. Parallel smoothing of color F_{DNW}.
10. Update from *E*.
11. Parallel smoothing of color F_{DNE}.
12. Update from *S*.
13. Parallel smoothing of color F_{DSE}.
14. Update from *T*.
15. Parallel smoothing of color F_{TSE}.

16. Update from N.
After this iteration, the local boundary values of $\tilde{U}_h$ are updated at the points $(B_{(x,y,z),E} \cup B_{(x,y,z),N} \cup B_{(x,y,z),T}) \cap \mathring{\Omega}_h$.

5 Automatic Parallelization with Expression Templates

An automatic parallelization of a code can only be achieved if the code is implemented in a suitable language. Such a language can be provided by expression templates in C++. Using expression templates, one can implement operators like `+,-, ...` in such a way that expressions like

```
u = a+b+c;
```

are evaluated in an efficient way for vectors `a,b,c`. The main idea of this concept is to implement the operator `+` such that `a+b` does not return the resulting vector, but a template object which is able to evaluate `a+b` efficiently for every component of the vector. This idea can originally be found in [6]. To explain the expression template concept in more detail, let us assume that the following is a class representing a vector:

```
enum update_type { fully, partially, no_update } // needed for
                                                 // parallelization
class vector {
  private:
    int length;           // length of the vector
    double a[];

    int number;           // needed for parallelization
    update_type upd_type; // needed for parallelization
  public:
    vector(int l);
    double component(int i) { return a[i]; }
    ...
};
```

Then, an efficient implementation of `u = a+b` can be obtained by the following constructions

```
class SumVector {
  private:
    vector* va;
    vector* vb;
```

```
  public:
    SumVector(vector* Va, vector* Vb) va(Va),  vb(Vb) {};
    double component(int i) {
       return va->component(i) + vb->component(i);
    }
};

SumVector
operator+(vector& va, vector& vb) {
  return SumVector(&va,&vb);
};

void vector::operator= (const SumVector& ea) {
   int i;
   for(i=0;i<length;++i) {
     a[i] = ea.component[i];
   }
};
```

The disadvantage of the above implementation of the operator `+` is that it cannot be used for long sums as `a+b+c` or `a+b+c+d`. Therefore, expression templates are needed. For the implementation of expression templates one first has to define a class, which represents all possible expressions:

```
// wrapper class for all expressions
template<class A>
class DExpr {
  private:
    A wa;
  public:
    DExpr(const A& a) : wa(a) {}
    double component(int i) { return wa.component(i); }
};
```

Instead of the class `SumVector`, we now define the more general class

```
// expression for the sum of two expressions
template<class A, class B>
class DExprSum {
  A va;
  B vb;
public:
  DExprSum(const A& a, const B& b) : va(a), vb(b) {}
  double component(int i) {
```

```
    return va.component(i) + vb.component(i);
  }
};
```

Now, one has to implement several types of operators +. One of them is

```
// operator for the sum of two expressions
template<class A, class B>
DExpr<DExprSum<DExpr<A>, DExpr<B> > >
operator+(const DExpr<A>& a,const  DExpr<B>& b) {
  typedef DExprSum<DExpr<A>, DExpr<B> > ExprT;
  return DExpr<ExprT>(ExprT(a,b));
}
```

The expression template concept can also be extended to solvers for finite element discretizations. Of course, in this case, one needs a more complicated data structure for storing the vectors on a discretization grid. Let us call such vectors `Variable`. Then, the Jacobi iteration and the Gauss-Seidel iteration for Poisson's equation can be implemented as follows:

```
// Algorithm:
// Jacobi iteration for Poisson's equation
void Smoother(Variable& u, Variable& f, Variable& r) {
  r = f + Laplace_FE(u);
  u = u - r ω;
}
```

```
// Algorithm:
// Gauss-Seidel iteration for Poisson's equation
void Smoother(Variable& u, Variable& f, Variable& r) {
  u = u - Laplace_FE(u) - f;
}
```

Here `Laplace_FE(u)` means the discrete Laplace operator L_h for a finite element discretization.

By the above implementation, the ordering of the Gauss-Seidel relaxation is hidden in the library. In [3], it is explained how to implement boundary conditions, systems of equations, and other concepts which are needed in the PDE context. Here, we want to study how an automatic parallelization of general expressions can be obtained. To this end, let us assume that different vectors `u,a,b,c,...` and a right hand side expression `Expr(...)` are given. Furthermore let us write those vectors in brackets

```
Expr(a,b,c,...)
```

which are inserted in a stencil operator, such as the discrete Laplace operator `Laplace_FE(u)`. Then, we can distinguish two types of expressions, which have to be evaluated:

- *Jacobi type expression:*
  u = Expr(a,b,c,...);
- *Gauss-Seidel type expression:*
  u = Expr(u,a,b,c,...);

The difference between these two types of expressions is that the right hand side of the Jacobi type expression does not contain an operator with the left hand side as an argument. Using expression templates one can find out if an expression is of Jacobi type or Gauss-Seidel type. To this end, one first has to number all vectors (see member `int number;` in class `vector`). Then, one has to extend the expression template concept such that the above template class `class DExpr` contains a member function

```
bool DExpr::is_number_operator_argument(int number);
```

This member function has to be implemented such that it tests whether a certain vector is the argument of a stencil operator. In the above example this can be obtained as follows for the operator `+`

```
bool DExpr::is_number_operator_argument(int number) {
  return a.is_number_operator_argument(number);
}
bool DExprSum::is_number_operator_argument(int number) {
  return va.is_number_operator_argument(number) ||
         vb.is_number_operator_argument(number);
}
```

Of course, additional constructions are needed for a suitable class representing the Laplace operator and other operators.

By applying `is_number_operator_argument` to the left hand side vector, one can find out whether the whole expression is of Jacobi or Gauss-Seidel type.

To obtain an optimal parallelization, it is important to reduce the amount of data which has to be sent from one processor to another. Therefore, it is important to know which data have to be updated before evaluating an expression. To do this, we store in every vector information about whether the vector data is fully updated, partially updated, or not updated (see member `update_type int upd_type;` in class `vector`). In 2D this means ($\mathcal{N}_h$ is the set of nodal points defined in Section 3.):

fully updated: $(B_{c,N} \cup B_{c,E} \cup B_{c,S} \cup B_{c,W}) \cap \mathcal{N}_h$ is updated.

partially updated: $(B_{c,N} \cup B_{c,E}) \cap \mathcal{N}_h$ is updated.

not updated: No values are updated.

Using this information, one can parallelize expressions efficiently. According to the parallelization described in Section 2, Table 1 shows how variables are updated before the evaluation of an expression, and how the update type of the left hand side changes after evaluation of an expression. Furthermore, Table 2 shows how many updates are necessary to change the update type of a vector. Similar results can be obtained for the 3D case.

expression type	necessary update type for vectors in operators	update type of the left hand side after evaluation
Jacobi	fully updated	not updated
Gauss-Seidel	partially updated	partially updated

Table 1: Necessary update type before evaluation of an expression and status of update after evaluation.

			direction of updates	number of updates
not	$\rightarrow$	fully	W,S,E,N	4
not	$\rightarrow$	partially	E,N	2
partially	$\rightarrow$	fully	W,S	2

Table 2: Necessary updates for changing the update type of a vector.

6 Numerical Results

We have applied automatic parallelization with expression templates to different kinds of partial differential equations like Stokes equation, elasticity, and some non-linear partial differential equations. But for illustrating the efficiency of the technique, we restrict ourselves to a multigrid algorithm with two types of smoothers: Gauss-Seidel and Jacobi (see Section 5). For the implementation of a multigrid algorithm, one needs suitable expression template constructions,

explained in [3]. The following code can be parallelized automatically by expression templates:

```
// Algorithm: Multigrid Algorithm
void MG(Variable& u, Variable& f, Variable& r, int level) {
  if(level>1) {
    // pre-smoothing
    Smoother(u,f,r);

    // restriction
    r = Laplace_FE(u) - f;
    f = Restriction_FE(r);
    u.Level_down();
    u = 0.0;

    // coarse-grid correction
    MG(u,f,r,iter_GS,level-1);
    u = u + Prolongation_FE(u);
    f.Level_up();

    // post-smoothing
    Smoother(u,f,r);
  }
  else {
    // smoothing on coarsest grid
    Smoother(u,f,r);
  }
}
```

Counting the number of communications in this algorithm leads to nearly the same number of communications for a multigrid algorithm with either Gauss-Seidel or Jacobi smoothing. This is shown in Table 3. For example, for a multigrid algorithm in 2D with Gauss-Seidel as the smoother, the number of communications is counted as follows: 2 communications for a partial update before Gauss-Seidel smoothing; 4 communications for the Gauss-Seidel smoothing itself; and 2 additional updates needed after Gauss-Seidel smoothing. The above multigrid algorithms consists of 2 Gauss-Seidel smoothing steps on each level. This leads to $(2 + 4 + 2) * 2 = 16$ updates.

Table 4 depicts the computational time of the above multigrid algorithm on ASCI Blue-Pacific. The domain Ω is the union of a ball and two cubes. One can see that the computational time for the multigrid algorithm is roughly the same, whether Jacobi smoothing or Gauss-Seidel smoothing is used.

MG with smoother	2D	3D
Jacobi	16	24
Gauss-Seidel	16	28

Table 3: Number of communications per level in a multigrid algorithm for different kinds of smoothers.

number of processors	8		131			597
number of grid points	5302	36,782	259,609	1,973,996	15,356,509	15,356,509
MG with Jacobi	0.073	0.5	0.27	1.86	9.3	2.35
MG with Gauss-Seidel	0.082	0.51	0.41	1.83	8.6	2.8

Table 4: Computational time (in seconds) of one multigrid algorithm with different kinds of smoothers.

References

[1] U. Block, A.Frommer, and G. Mayer. Block colouring schemes for SOR method on local memory parallel computers. *Parallel Computing*, 14:61–75, 1990.

[2] E.W.Evans, S.P. Johnson, P.F. Leggett, and M. Cross. Automatic and effective multi-dimensional parallelization of structured mesh based codes. *Parallel Computing*, 26:677–703, 2000.

[3] C. Pflaum. Expression templates for partial differential equations. *to appear in Computing and Visualization in Science*, 2001.

[4] C. Pflaum. Semi-unstructured grids. *Computing*, 67(2):141–166, 2001.

[5] T. Veldhuizen. Blitz ++. http://oonumerics.org/blitz/index.html.

[6] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.