FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Expression Templates and Advanced PDE Software Design on the Hitachi SR8000

Christoph Freundl, Ben Bergen, Frank Hülsemann, and Ulrich Rüde

Lehrstuhlbericht 04-7

Expression Templates and Advanced PDE Software Design on the Hitachi SR8000

Christoph Freundl, Ben Bergen, Frank Hülsemann, and Ulrich Rüde

Lehrstuhl für Systemsimulation, Institut für Informatik, Universität Erlangen-Nürnberg, Cauerstraße 6, D-91058 Erlangen, Germany

Summary. We demonstrate the use of expression templates for the development of a numerical PDE software library for high performance computers. We discuss the library design and show that expression templates on performance tuned data structures achieve both a user-friendly interface and efficient runtime performance. Our performance results on various architectures including the Hitachi SR8000 illustrate that modern programming techniques like expression templates are well suited for large scale parallel computers.

1 Introduction

The aim of the ParExPDE project is to provide a library for the rapid development of numerical PDE solvers on parallel (super-)computers. To this end, the library features a high level and intuitive user interface that hides the complexities of parallel program development without compromising on efficiency. In this paper we demonstrate how to achieve these goals with the help of expression template programming techniques.

The paper is organized as follows. We start with a description of expression template programming. Then we turn to the types of grids that the library covers and the representation of the discretised differential operators. The latter issue has such an influence on the run time performance that it determines the core storage structures used in this project. Efficient PDE solvers require first and foremost efficient algorithms. Therefore we discuss briefly the merits of multigrid algorithms and we emphasize the fact that our grid structures are well suited for this type of method. Next we illustrate the different possibilities where expression templates can be used in different parts of the library. This is followed by an overview of the library's architecture. After going into some details of the implementation we finally present some performance results on different architectures.

1.1 Basic Expression Templates

The main idea of expression templates as presented in [12] is to enclose arithmetic expressions in a tree-like structure by means of C++ template constructs such that the resulting expression object can be passed to other functions. It also enables fast evaluation of vector and matrix expressions by eliminating temporaries and using inline techniques.

We illustrate this by showing how a simple vector assignment is treated by the compiler without going to deep into the details of expression templates. The assignment is a well-known daxpy operation, in the program context it would look like this:

```
Vector x, y, z;
double a;
z = a * x + y;
```

By appropriately overloading the arithmetic operators, the expression `a * x + y` generates an object of the type `Expr< ExprBinOp< Expr< ExprBinOp< ExprLiteral, ExprVector, OpMult > >, ExprVector, OpAdd > >`
as shown in Fig. 1

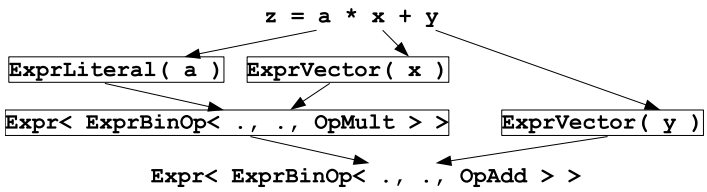

Fig. 1. Generation of the expression object for a daxpy operation

The assignment is performed by the overloaded assignment operator of the `Vector` class.

```
template<class T>
void Vector::operator=( Expr<T>& expr ) {
  for ( int i = 0; i < _size; i++ )
    _values[i] = expr.valueAt( i );
}
```

Each expression class implements the function `valueAt` appropriately, furthermore all `valueAt` functions are declared to be inlined. The resulting assignment in the loop after performing all inlining looks finally like this:

```
_values[i] = a * x._values[i] + y._values[i];
```

An optimizing compiler should have no problems in accelerating the execution of this loop by techniques like loop unrolling or – in case of the Hitachi SR8000 – pseudo-vectorization of the loop (see Sect. 5).

1.2 Structured vs Unstructured Grids

In order to solve a partial differential equation, we first need some type of discretization of the problem. In the current setting, this means that we introduce an unstructured, hexahedral, input grid that provides a discrete representation of the problem domain and optionally apply regular refinement to it until the desired level of resolution is achieved. Such a grid that has had at least one level of regular refinement applied to it will be call a *block–structured grid*. We then apply either the *Finite Difference Method* or the *Finite Element Method*, both of which lead to sparse matrix systems. However, depending on the type of compute grid used to represent the domain, different types of data structures are required to solve these systems on a computer. In the case of the structured grid interiors resulting from regular refinement, we may use stencil based data structures to represent the discretization matrix, as couplings can only occur at regular, known offsets. Unstructured grids sections, like those encountered on the input grid and the shared vertices and edges of the block interfaces, result in generally sparse systems. This implies that, from the point of view of the storage mechanism, the couplings that exist between unknowns cannot be determined beforehand. This makes it necessary to use more complicated data structures to store such systems, and results in added costs during computation. To illustrate this, we will consider the *Jagged Diagonals Storage* scheme for storing a purely unstructured grid, and discuss why it is unable to achieve high performance on modern computer architectures.

The JDS scheme stores the non zero entries of a generally sparse matrix in the following manner: First, the matrix is compressed row-wise by removing all of the zero entries and shifting the remaining terms to the left. The rows are then sorted in descending order by length. Finally,

the entries are stored in a linear array by column. These columns are called the jagged diagonals. This scheme requires four arrays to represent the matrix: The *values* array, already mentioned, stores the jagged diagonals. The *columns* array stores the column index of each entry in the *values* array. The *offsets* array stores the extents of each jagged diagonal, and the *permutations* array stores the permutations that were applied during the row sorting. As an example of how the values are accessed, consider the following section from a matrix-vector multiplication,

```
y[i] += values[offsets[j]+i]*x[columns[permutations[offsets[j]+i]]]
```

The important thing to notice in this algorithm, is that indirect indexing is used to access the entries of the matrix. This indirection can cause several problems for the compiler during optimization. Here, we will only consider the most troublesome problem having to do with prefetch optimizations.

The performance of many numerical algorithms is limited by the speed and bandwidth with which main memory can be accessed. Many, if not most, modern architectures have processors that are able to perform many more operations on data than can be supplied by the memory subsystem. This gap in memory speed access versus processor speed is generally referred to as *memory bandwidth limitation*. In order to overcome this limitation, different architectural and compiler strategies and optimizations have been developed. One such strategy that is particularly relevant for our purposes involves both hardware specialization and compiler optimization. This is the so called *prefetch* optimization. Prefetch allows the compiler to insert special scheduling directives to the processor which affect memory access. This enhanced scheduling cuts down on the *latency* between the time that memory is requested by the CPU and the time that the memory actually begins being streamed by the memory subsystem, and results in a higher percentage of the peak memory bandwidth being available for processing. Our target platform, the Hitachi SR8000, has special hardware modifications that allow it to perform both preload and prefetch operations[1]. These optimizations require that the compiler be able to analyze how data is accessed by a particular algorithm so that it can issue special scheduling instructions to the CPU. This is the root of the problem with the JDS scheme. Because of the additional layer of indirection required to access the matrix entries, the compiler cannot perform this type of optimization, with the result that memory intensive algorithms are unable to achieve a high level of efficiency. For structured grids and stencil based matrix representations, there is no indirection. Therefore, stencil based data structures are more desirable when performance is an issue. To illustrate the difference in performance between structured and unstructured grid representations, consider the following table, which shows the MFLOP/s rates obtained on a single compute node of the Hitachi SR8000 in Munich, Germany.

Table 1. MFLOP/s Rates for Matrix-Vector Multiplication (JDS results courtesy of Dr. Gerhard Wellein, RRZE Erlangen Germany)

# unknowns	729	4913	35937	274625	2146689
JDS	50	260	750	800	1050
Stencils	640	820	3980	5550	7120

Clearly, the stencil based data structures achieve much better performance. The approach presented here draws from experience of the gridlib project [4], [6], [5], in which similar structures for hybrid grids have already achieved high performance results on various platforms [3], [1]. As can be seen from the results in Table 1, the ParExPDE algorithms will have the best performance when several levels of regular refinement have been added to the unstructured input grid.

[1]Here, we make the distinction that preload operations load data from main memory directly into a floating point register, in the same manner as a true vector processor. Prefetch, refers to loading data from main memory into one of the caches.

1.3 Multigrid and Other Iterative Solvers

As stated in the previous section, the types of discretizations considered here lead to a linear system of equations. For solving large systems it is necessary to turn to iterative methods. This is due to the lower algorithmic complexity of iterative solvers. For example, if we were to use standard *Gaußian Elimination* to solve a system of n unknowns, we would need $\mathcal{O}(n^3)$ operations. Suppose that we would like to solve a system with $1,000,000$ unknowns on a CPU that can perform 2 GFLOP/s. We would then need to perform on the order of 10^{18} operations. On our CPU, at peak theoretical performance, this would take $500,000,000$ seconds which is approximately $1,000$ years. In the special case that the linear system is symmetric positive definite, which is true for the types of discretizations we are considering here, iterative schemes such as *Gauß–Seidel* have better complexity $\mathcal{O}(n^2)$. However, these are also unsatisfactory as they lose convergence as the oscillatory error components are eliminated. Fortunately, there is an asymptotically optimal iterative solver for such systems called *full multigrid* with complexity $\mathcal{O}(n)$. Considering the same constraints as in the previous example, we would now only need 5×10^{-4} seconds to solve the problem. It is not a coincidence that regular refinement of an unstructured input grid was discussed in the previous section, as this leads to a suitable, nested, grid hierarchy for performing geometric multigrid, an important component of full multigrid.

2 Expression Templates for Partial Differential Equations

The idea of using expression templates for the treatment of partial differential equations is not new, the development of the ParExPDE library is based on the already existing EXPDE library ([11], [10]). Although there are differences between the two libraries the common basis is that both provide a comfortable interface for the application programmer. An example is given in Fig. 2 which shows how easy and compact the cg method can be formulated in C++ when using the ParExPDE library. Note that we are working on an arbitrary unstructured hexahedral grid where the variables (`u`, `r`, `g` etc.) are defined. The complexity of the evaluation of the right-hand side of the assignments is hidden to the programmer.

```
for(int i = 1; i <= iteration && delta > eps; ++i) {
  g = laplace(d) | interior_points;
  double tau = delta / pxdScalarProduct(d, g, &interior_points);
  u = u + tau * d | interior_points;
  r = r + tau * g | interior_points;
  double delta_prime = pxdScalarProduct(r, r, &interior_points);
  double beta = delta_prime / delta;
  delta = delta_prime;
  d = beta * d - r | interior_points;
  e = u - u_exakt | interior_points;
  double l2_error =
    sqrt(pxdScalarProduct(e, e, &interior_points) / normi);
}
```

Fig. 2. Implementing the cg method using ParExPDE

In the example listing we can see different applications of expression templates:

- Simple arithmetic operations like addition of two variables or multiplication with a scalar.
- Application of a differential operator to a variable (`laplace(d)`).
- Restriction of the evaluation of an expression to only a part of the whole domain. In this case this is done by using the predefined marker `interior_points`.

Another application of expression templates in the ParExPDE library is the definition of differential operators. For the creation of an operator we must simply provide the function to be integrated (in a finite element context) as shown in Fig. 3. The integration routines which are developed by J. Härdtlein in [2] make also use of expression templates for the formulation of the function. Again they are used for simple arithmetic operations but also for the expression of simple differential operators like `d_dx` for $\frac{d}{dx}$.

```
class pxdLaplaceOperatorFunction {
public:
  static double integrate(_Cuboid_<bgPoint3D>& c,
                          int m, int n, bgPoint3D** cell_point) {
    return c.integrate(d_dx(v(cell_point[m])) *
                       d_dx(v(cell_point[n])) +
                       d_dy(v(cell_point[m])) *
                       d_dy(v(cell_point[n])) +
                       d_dz(v(cell_point[m])) *
                       d_dz(v(cell_point[n])));
  }
};
```

Fig. 3. Definition of the Laplace Operator

3 Software Architecture of ParExPDE

The realization of a library with the desired functionality is clearly not a simple task as many stages of abstraction are involved. A common approach in software development is to structure the library into several *layers*. Each layer has a clearly defined interface and uses only the functionality of lower layers. Ideally, the layers are completely independent of each other if they are strictly based only on the interfaces but not concrete implementations of lower layers. As a result, the implementation of a certain layer could easily be replaced by another implementation without changing the rest of the library. This leads to a clear structure and to a well maintanable library.

The layered architecture of the ParExPDE library is described next. In the current design we can distinguish six different layers as shown in Fig. 4.

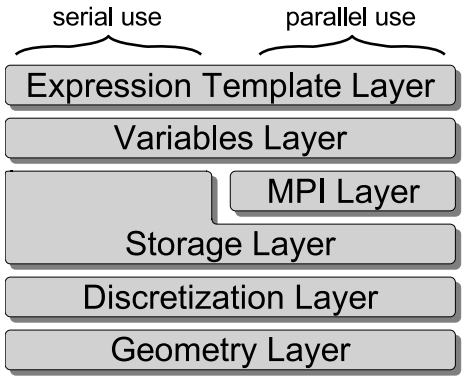

Fig. 4. Software layers of ParExPDE

Geometry Layer The geometry layer holds and handles all information of the underlying hexahedral grid which includes the following:
- Storing all geometric primitives in the grid: in a hexahedral grid there are vertices, edges, (quadrilateral) faces and cuboids
- Storing neighbour information for each geometric primitive
- The ability to retrieve the coordinates for a given point within one of the primitives

Discretization Layer This layers stores for each geometric primitive the number of discretization points. Also, periodicity information is handled here: on a periodic boundary there are for each grid element two geometric elements but they are treated identically.

Storage Layer Using the information from the discretization layer, this layer provides the construction of objects containing memory for each element in the grid. Memory is not only needed for the elements themselves but also each element must be able to access certain data from neighbouring elements.

MPI Layer Placed on top of the storage layer, the MPI layer provides mechanisms for the exchange of data between storage objects. MPI is the de facto standard for distributed memory machines, including supercomputers and high performance clusters. This layer manages the communication buffers and the data exchange between elements on different processes.

Variables Layer Up to this layer we only have storage objects that are not related to each other. Putting a storage object for each geometric primitive together in one place we get the notion of a variable defined on the whole grid. By using the MPI layer — if available and wanted — it is easy to keep the variables synchronized among all involved processors.

Expression Template Layer The user interface is realized by the expression template layer. It enables the user to write program statements that are very close to the mathematical language. An efficient evaluation of expressions is possible.

4 Implementation

The first step in implementing the ParExPDE library for the Hitachi SR8000 consisted in using the EXPDE code as a framework to develop the so-called EXPDE-cuboid version which possessed the functionality of EXPDE except having multigrid operators. This version had already used the Hitachi specific features like PVP and COMPAS but it only worked on a single cuboid and lacked MPI parallelization. The next step was to overcome these deficiencies by enabling the library to work on general hexahedral grids and parallelizing the code. It turned out that the insertion of the new geometric primitives and the inclusion of MPI required major program restructuring.

Expression Templates for Heterogeneous Data Structures

The EXPDE-cuboid code worked on just one cuboid. Therefore, the faces, edges and vertices of the cuboid did not require a treatment as separate entities since there were no adjacent neighbours which would have caused them to become regions with arbitrary structure. With the treatment of general hexahedral grids, this is no longer the case as there might be interior edges or vertices which can possibly have an arbitrary number of neighbours.

When we evaluate an expression template applied to a variable defined on a hexahedral grid, we have to iterate over all storage objects of all types. Furthermore, the application of the expression template to the different element types might require different implementations e.g. during the application of a differential operator.

Therefore, the expression templates provide specializations of themselves for a certain grid element. We demonstrate this for the expression type that represents a variable. The general expression class defines types for the element specializations and provides functions that return specializations for certain elements:

```
class pxdExprVariable {
public:
  typedef pxdExprCuboid CuboidReturnType;
  typedef pxdExprFace FaceReturnType;
  typedef pxdExprEdge EdgeReturnType;
  typedef pxdExprVertex VertexReturnType;
private:
  pxdVariable& variable_;
public:
  pxdExprVariable(pxdVariable& variable) : variable_(variable) { }
  pxdExprCuboid getCuboidExpr(int cuboid_index) const {
    return pxdExprCuboid(variable_, cuboid_index); }
  pxdExprFace getFaceExpr(int face_index) const {
    return pxdExprFace(variable_, face_index); }
  pxdExprEdge getEdgeExpr(int edge_index) const {
    return pxdExprEdge(variable_, edge_index); }
  pxdExprVertex getVertexExpr(int vertex_index) const {
    return pxdExprVertex(variable_, vertex_index); }
  ...and other things
};
```

The specialized class for cuboids is presented here, the specializations for the other element types are similar.

```
class pxdExprCuboid {
private:
  pxdVariable& variable_;
  double* memory_;
public:
  pxdExprCuboid(pxdVariable& variable, int cuboid_index) : variable_(variable) {
    memory_ = variable_.getCuboidArray(cuboid_index)->getRawMemory(); }
  double valueAt(int index) const {
    return memory_[index]; }
};
```

Finally, for the application of differential operators, we present the specializations of the expressions for cuboids and edges to illustrate that their evaluation requires different implementations. For the evaluation inside a cuboid we can exploit its regular structure i.e. we know that inside the cuboid we only have 27-point stencils, at least for piecewise linear discretizations.

```
template <class A, class DiffOp>
class pxdExprDiffOpCuboid {
public:
  typedef typename A::Type Type;
private:
  A a_;
  DiffOp op_;
  bgComputeCuboid* compute_cuboid_;
  const bgCuboidOffsets* offsets_;
public:
  pxdExprDiffOpCuboid(const A& a, const DiffOp& op, bgComputeCuboid* compute_cuboid)
    : a_(a), op_(op), compute_cuboid_(compute_cuboid) {
    offsets_ = compute_cuboid_->getOffsets(); }
  Type valueAt(int index) const {
    // apply 27-point stencil
    return op_.getStencilAt(index).apply
      (a_.valueAt(index + offsets_->offset_left_front_bottom),
       a_.valueAt(index + offsets_->offset_front_bottom),
       ...the values for the other neighbours...
       a_.valueAt(index + offsets_->offset_right_back_top)); }
};
```

In contrast to cuboids we do not know the precise number of entries of the stencils on an edge at compile time. This leads to a quite different evaluation of an differential operator there.

```
template <class A, class DiffOp>
class pxdExprDiffOpEdge {
public:
  typedef typename A::Type Type;
private:
  A a_;
  DiffOp op_;
  bgComputeEdge* edge_;
public:
```

```
  pxdExprDiffOpEdge(const A& a, const DiffOp& op, bgComputeEdge* edge)
    : a_(a), op_(op), edge_(edge) { }
  Type valueAt(int index) const {
    const bgEdgeOffsets* offsets = edge_->getOffsets();
    // create array for the values of the point and its neighbours
    Type* edge_value = new Type[op_.stencilAt(index).getSize()];
    int vertical_offset =
      edge_->getTotalNumberNeighbours() + 1;
    // values on the edge
    edge_value[0] = a_.valueAt(index-1);
    edge_value[vertical_offset] = a_.valueAt(index);
    edge_value[2*vertical_offset] = a_.valueAt(index+1);

    // values on all neighbouring cuboids
    for (int n = 0; n < edge_->getNumberCuboidNeighbours(); n++) {
      edge_value[2*n+2] =
        a_.valueAt(offsets->offset_neighbour_cuboid[n]+index-1);
      edge_value[vertical_offset+2*n+2] =
        a_.valueAt(offsets->offset_neighbour_cuboid[n]+index);
      edge_value[2*vertical_offset+2*n+2] =
        a_.valueAt(offsets->offset_neighbour_cuboid[n]+index+1); }

    // values on all neighbouring faces
    for (int n = 0; n < edge_->getNumberFaceNeighbours(); n++) {
      similar as above }

    // apply the unstructured stencil to the value array
    typename A::Type result = op_.stencilAt(index).apply(edge_value);
    delete[] edge_value;
    return result;
  }
};
```

5 Performance Analysis

At first sight it seems that the use of expression templates should cause an overhead which prevents the resulting code to be executed efficiently. But to the opposite, we can demonstrate good performance with our expression template codes.

In order to get good performance on the Hitachi SR8000, the code should be prepared to be optimized using the pseudo-vectorization feature (PVP). In particular, this means that we have to design the data layout carefully. In the storage layer, we have chosen to arrange the data in single large arrays wherever this is possible. PVP should work well for operations on large arrays of known size. This applies not only to the SR8000 architecture but also to other architectures. On cache based machines these data structures are well suited for layout and access modification techniques presented in [9], [7], [8] that take the memory hierarchy into account.

Due to the peculiarities of Hitachi's ANSI-compliant C++ compiler sCC, the application of differential operators could not yet be optimized. Unfortunately, sCC is the only C++ compiler on this machine that makes use of the machine's special features. Therefore, it has not been possible to obtain satifactory performance for programs involving differential operators. However, we can show that the evaluation of expression templates consisting only of simple arithmetic operations shows a reasonably good performance. On other architectures we can demonstrate that we get good performance for both regular arithmetic expressions and expressions with differential operators (see Table 2).

Performance results that we have obtained with a more procedural-style implementation of the top layer show that the suboptimal treatment of complex expression templates by the Hitachi compiler is responsible for the unsatisfactory performance.

Therefore, the new top layer has been implemented in a mix of C++ and FORTRAN77 and it still uses C++ features like inheritance and templates but in a more conservative way than the original expression template layer.

The components of the geometric multigrid method are well known in the literature. We use a V(2,2) cycle with a Gauß-Seidel smoother on a seven point finite difference discretisation of a

Table 2. MFLOPs obtained on a single processor for the evaluation of expression templates. The "Simple Expression" is of the type "constant + scalar * variable + scalar * variable * variable", the other expression consists just of the application of the Laplace operator to a variable. The evaluation has been performed on a single cuboid containing 128^3 unknowns

	Hitachi SR8000	Pentium 4 2.4 GHz
Simple Expression	513	407
Expr. with Differential Operator	25	918

Poisson problem with Dirichlet boundary conditions. The problem domains for the scale up experiment were $\Omega_1 = [0,2] \times [0,2] \times [0,2]$, partitioned into 8 hexahedra, $\Omega_2 = [0,4] \times [0,2] \times [0,2]$, partitioned into 16 hexahedra, $\Omega_3 = [0,4] \times [0,4] \times [0,2]$, partitioned into 32 hexahedra. The mesh-size was always $h = \frac{1}{256}$. The performance results in Table 3 underline that the ParExPDE layers result in runtime efficient programs. When comparing these performance values with the ones in Table 1, one has to take several differences into account. The most significant difference is that Table 3 states the performance for the whole multigrid program including program start, reading of configuration files and memory allocation in addition to the geometric multigrid part, whereas Table 1 concentrates on one computationally intensive routine. Usually, the computational cost of a geometric multigrid implementation is dominated by the smoothing operation. We utilise a Gauß-Seidel smoother, which is similar to a matrix-vector product in terms of memory access pattern and which, in fact, achieves a similar, if slightly lower, MFLOP performance. However, other components of the parallel multigrid implementation, in particular the data exchanges, do not and cannot run at the same speed, hence it cannot be expected to see a whole program performance on par with the matrix-vector product. Furthermore, the scale up experiments include inter-process communication via MPI for both, inter-node and intra-node data exchange, whereas the matrix-vector routine was run on a single node using Hitachi's shared memory programming model COMPAS. Despite all the differences, the computation of the matrix vector product, or, more precisely, the application of the discrete operator to a problem variable using a stencil based operator representation, does achieve a similar performance as in Table 1.

Table 3. Scale up results for parallel geometric multigrid in 3D: *Dof* are the degrees of freedom, *Time* is the wall clock solution time for the linear system, the value *MFLOPs per node* is computed by multiplying the average performance over all MPI processes by eight, the number of processes per node. The average performance of all MPI processes is taken over the whole program run, including start up, reading of configuration files, initialisation in addition to the numerically intensive part

CPU	Dof $\times 10^6$	Time in (s)	V-cycles	MFLOPs per Node
8	133.4	78	9	2209.6
16	267.1	81	10	2337.6
32	534.7	95	11	2167.2

6 Conclusions

In this paper, we have shown that expression templates are an appropriate programming paradigm for the development of parallel numerical software on supercomputers. With expression templates, ParExPDE provides the application programmer with an intuitive interface to the underlying parallel grid software library. This software infrastructure enables the rapid development of PDE solvers and numerical simulation programs.

Depending on the quality of the compiler, this can be achieved without having to compromise on runtime performance. Unfortunately, the SR8000 is one of the examples where currently only relatively simple expression templates can be executed with full efficiency. With more complex expression templates the optimizer still fails to exploit the full inherent performance potential. The benchmarks of ParExPDE presented above show clearly that, when using a different compiler, the performance degradation does not occur and is due to the limited functionality and optimization capabilities of the current C++ implementation on the Hitachi.

This claim has been cross-checked by presenting results on the Hitachi with a more conservative implementation of a multigrid code (derived from the associated gridlib project) using the same ParExPDE data structures and communication routines. This program variant avoids using more complex expression templates. The performance results show that the basic software architecture has excellent performance, speedup and scalability characteristics. Different from unstructured grid algorithms on the Hitachi, it achieves a fully satisfactory sustained performance and thus justifies the software design of the ParExPDE and gridlib projects. Both projects are based on the assessment that the Hitachi is much better suited for computations on structured grids than on unstructured grids or general sparse matrix implementations.

Summarizing, the ParExPDE project has resulted in a highly efficient software infrastructure for parallel grid based computations. It provides both a very high level expression template interface and a somewhat lower level C++ library interface. The performance degradation that is still observed with complex expression templates on the Hitachi must be attributed to the state of the currently available compiler. This limitation will disappear with increasing compiler maturity so that the use of expression template programming will become even more attractive and feasible.

Acknowledgements

The authors gratefully acknowlegde the support of the Bavarian high performance computing initiative KONWIHR. We are also indebted to Professor C. Pflaum without whom this project would not have been possible.

References

1. B. Bergen and F. Hülsemann. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical Linear Algebra with Applications*, 11:279–291, 2004.
2. J. Härdtlein, A. Linke, and C. Pflaum. Fast expression templates. To appear.
3. F. Hülsemann, B. Bergen, and U. Rüde. Hierarchical hybrid grids as basis for parallel numerical solution of PDE. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 840–843, Berlin, 2003. Springer.
4. F. Hülsemann, P. Kipfer, U. Rüde, and G. Greiner. gridlib: Flexible and efficient grid management for simulation and visualization. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 652–661, Berlin, 2002. Springer.
5. F. Hülsemann, S. Meinlschmidt, B. Bergen, G. Greiner, and U. Rüde. *gridlib* – a parallel, object-oriented framework for hierarchical-hybrid grid structures in technical simulation and scientific visualization. In *High Performance Computing in Science and Engineering, Munich 2004. Transactions of the Second Joint HLRB and KONWIHR Result and Reviewing Workshop*, pages 37–50. Springer-Verlag Berlin Heidelberg New York, 2004.
6. P. Kipfer, F. Hülsemann, S. Meinlschmidt, B. Bergen, G. Greiner, and U. Rüde. gridlib: A parallel, object-oriented framework for hierarchical hybrid grid structures in technical simulation and scientific visualizations. In S. Wagner, W. Hanke, A. Bode, and F. Durst, editors, *High Performance Computing in Science and Engineering 2000-2002*, pages 489–501, Berlin, 2003. Springer.

7. M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science (LNCS)*, pages 307–316, Espoo, Finland, 2002. Springer.
8. M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*, pages 213–232. Springer, 2003.
9. M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proc. of the 2002 Int. Conf. on Computational Science (ICCS 2002), Part III*, volume 2331 of *Lecture Notes in Computer Science (LNCS)*, pages 642–651, Amsterdam, The Netherlands, 2002. Springer.
10. C. Pflaum. *EXPDE — Expression Templates for Partial Differential Equations*. `http://www10.informatik.uni-erlangen.de/~pflaum/expde/public_html/`.
11. C. Pflaum. Expression templates for partial differential equations. *Computing and Visualization in Science*, 4(1):1–8, November 2001.
12. T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.