Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS

Peter Gottschling
Institut für Wissenschaftliches Rechnen
Technische Universität Dresden, Germany
Peter.Gottschling@tu-dresden.de

Dag Lindbo
Dept. Numerical Analysis and Computer Science
Kungliga Tekniska Högskolan (KTH), Sweden
dag@kth.se

ABSTRACT

Sparse matrices are indispensable components of most scientific applications. Nevertheless, there is very little general-purpose software support. With the Matrix Template Library 4 (MTL4) we provide a generic library support for dense and compressed sparse matrices. The first challenge in working with compressed matrices is how to set the non-zero entries in an efficient manner. The implementation in MTL4 does not need any pre-allocation or pre-sorting phase, uses a minimal amount of memory and was in all measures as fast or faster than comparable libraries. We demonstrate the performance on well-defined benchmarks.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Arrays; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*software libraries*

Keywords

Sparse matrices, Generic programming, Matrix template library, FEniCS

1. INTRODUCTION

Simulations of physical, chemical, and biological processes play a crucial role in science and economy. The dominant methods to model these processes are *finite element method (FEM)*, *finite volumes (FVM)*, and *finite differences (FDM)*. Although these three methods are mathematically quite different they all involve the solution of large systems of linear equations.

The systems can grow very large in dimension — millions and even billions of unknowns are not rare — but the number of terms per equation is limited. That is the matrices used in these linear systems are *sparse*. Despite the importance of sparse matrices in a large variety of applications the software support in a general-purpose manner is insufficient.

POOSC '09, July 7 2009, Genova, Italy

Instead many simulation packages use their own hand-crafted implementations of sparse matrices and focus on special use-patterns. We are aiming for a general-purpose solution with high performance.

Whereas other algorithms as matrix vector product are implemented similarly in existing software packages, there are extreme differences in the manner how sparse matrices are filled with values or modified. We will show how compressed sparse matrices are set in MTL4 and compare it to other approaches. As yardstick we use a set of well-defined benchmarks.

1.1 Matrix Template Library 4

MTL4 is a generic library for high-performance numeric operations on matrices and vectors [8, 12]. At the moment it provides a dense vector format, compressed sparse matrices, dense matrices and a large spectrum of recursively laid out dense matrices (like Morton-order in the simplest case). Traditional dense matrices can be stored in row-major order like C/C++ arrays or column-major like Fortran arrays. Sparse matrices are available as compressed row storage (CRS) and compressed column storage (CCS). The elements of the vectors can be intrinsic arithmetic or user-defined types as quaternions, intervals, high-precision numeric types and matrices and vectors themselves.

The library contains multiple new techniques as *implicit enable-if* and *meta-tuning* (short for performance tuning based on meta-programming). The latter allows for performance optimization like changing the block size of an unrolled loop or modifying tile sizes in blocked algorithm that otherwise require code rewriting. Thanks to meta-tuning the user can choose such parameters with template arguments in the function call and the compiler will generate corresponding code, e.g., `dot<12>(u, v)` computes a dot product with an unrolled loop using a block size of 12. The technique has proved high efficiency, partly out-performing assembler libraries as GotoBLAS [9]. In contrast to many other linear algebra libraries, we are eager to provide all necessary functionality needed for high-performance FEM/FVM/FDM applications.

1.2 FEniCS

The FEniCS project is free software for solving differential equations. It's primary goals are generality, efficiency and simplicity. FEniCS is organized as a set of tools, some of which are

FFC – the FEniCS Form Compiler [11] The form compiler implements a high-level mathematical language

for formulating differential equations in weak form, i.e. as a set of multi-linear variational forms. An example that will be familiar for anyone who has studied the finite element method (FEM) is the following formulation of Poisson's equation:

```
element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
```

The form compiler generates a representation of this finite element problem formulation in C++ code – the specifications of which are freely available in the so-called Unified Form-assembly Code (UFC).

DOLFIN – generic FEM kernel [10] DOLFIN is organized as a C++ library that provides the key computational components of any FEM solver, such as the assembler, mesh and function abstractions. An interface exists to low-level libraries for matrix and vector operations, as well as to a number of solvers for linear systems. DOLFIN can use MTL4 for the underlying matrix representation via this interface.

Generality is central in DOLFIN – the assembler can assemble any multi-linear variational form over any finite element (defined by other tools in FEniCS), regardless of the dimension of the finite element space or the geometrical dimension of the PDE. This generality places a constraint on the matrix library, that it must be able to insert elements without knowing the sparsity pattern in advance. In this context, MTL4 is particularly interesting.

The paper is structured as follows: We will initially introduce sparse matrix formats; then we will discuss existing approaches to fill compressed sparse matrices. Subsequently, we will introduce a new concept for setting compressed matrices and discuss it's time and space complexity. Finally, we will introduce benchmarks and present results for four different matrix insertion concepts.

2. INSERTION ALGORITHMS

The dominant majority of scientific simulation software requires solving large systems of linear equations. Denoting such a system in terms of a matrix A and vectors x and b results in

$$A\,x = b,$$

where A and b are given and x is searched. With some exceptions (e.g. spectral methods [1]), the number of non-zero elements per row and column is limited by a constant, in many cases below 10 and in almost all cases below 100. On the other hand, the number of rows and columns can grow into the millions or even billions. Therefore, storing the entire matrix is much too expensive or even impossible and special matrix formats were developed that only store the non-zero elements.

2.1 Sparse Matrix Formats

The simplest sparse matrix scheme is called *coordinate* matrix. It consists of a triplet of row and column index and the value for each non-zero entry. Alternatively to one array of triplets it can also be stored in three arrays of single values. For instance, the matrix

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix} \tag{1}$$

is represented by the array of row indices $[0, 0, 1, 2, 2]$, column indices $[0, 2, 3, 1, 4]$, and values $[1, 2, 3, 4, 5]$. In this paper we use zero-based indexing as known from C/C++.

If the entries are sorted with respect to their row indices, the respective array becomes repetitive so that a kind of *run-length encoding* can be applied. This is illustrated in Figure 1. Row 0 contains two entries which are stored as element 0 and 1 in the columns and the value array. The position of each row's first entry is stored and the position of the last entry can be easily computed from the position of the next row's first entry. In typical real-world applications this leads to a significant reduction. This format is called *Compressed Row Storage (CRS)*.

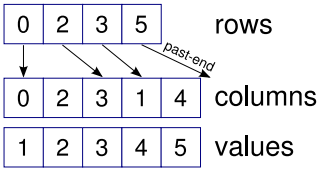

Figure 1: Compressed row storage format

The non-zero entries can also be sorted regarding the column index what is called *Compressed Column Storage (CCS)*. MTL4 supports this format and the implementation is realized in the same template class by interchanging the roles of rows and columns. The insertion is implemented with the same functions for CRS and CCS matrices and provide therefore the same performance behavior.

Some libraries also offer sparse matrices as vector of sparse vectors. Such matrices are very flexible for modifications but impose poor memory locality when used in operations.

There are more sparse matrix formats such as skyline matrices (e.g. [3]) or banded matrices (e.g. [6, 2]) but this is not our scope. For the remainder of the paper, we only consider CRS matrices.

2.2 On-the-fly Insertion

A simple approach for setting up and modifying a compressed sparse matrix is considering every insertion or modification as stand-alone operation. This technique has several advantages:

- All operations are easy to implement;
- The matrices are easy to use; and
- Sparse and dense matrices can be handled in the same manner.

The last argument is important for generic software because it allows programming generic functions that can work on

both sparse and dense matrices. Libraries that use this technique due to the before-mentioned advantages are uBLAS [16] and an older version of MTL [14].

The big disadvantage of insertion on the flight is performance. All entries with larger indices must be shifted in the array for the column indices and for the values. This also requires adding more memory at the end of the array what in turn can demand reallocation with additional copying. The starting positions of all rows larger than the row of the inserted entry must be increased. Figure 2 depicts the insertion of a_{01}, i.e. A[0][1] = 6.

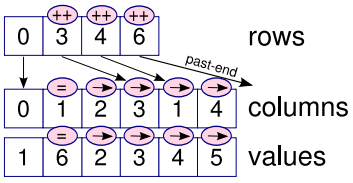

Figure 2: Insertion on the flight

The best case for inserting in this manner is when all entries are inserted in order. In this case the arrays for column indices and the values only need extension at the end. However, increasing in average half of the rows' starting position is already quite expensive. Inserting in reverse order or randomly linearly increases the number of entries that need to be shifted. As the consequence the algorithm is quadratic in the number of non-zero elements.

The dependence on the order of insertion can be circumvented by using an auxiliary vector-of-vector matrix, confer Section 3.1. In addition to the extra memory, the run-time is also impacted by unpredictable memory allocations and by copying the data from the vector-of-vector to the compressed matrix, see Sections 3.2–3.6.

2.3 Two-Phase Matrix Utilization

The before-mentioned performance deficiencies of on-the-fly insertion can be overcome by defining a dedicated insertion phase before using the matrix. Such insertion phase also enables to set up distributed matrices. PETSc (Portable Extensible Toolkit for Scientific computation, [4]) uses this approach in serial and parallel computation. The construction of distributed matrices even includes the creation of communication buffers and how they have to be filled within a matrix vector product. For the library *Janus* — which generically handles data-parallel scientific computations [7] — the two-phase (two-face) concept even inspired the name of the library.

One drawback of this technique is that a sparse matrix cannot be handled in the same fashion as dense matrices so that a more constrained program structure is required. Another disadvantage is that matrices cannot be changed in any form after the setup phase is finished. One possibility for implementing the two-phase technique is collecting all entries first, sort this collection and create the compressed matrix out of the sorted non-zero entries. Such approach allows for relatively good cache performance but requires significant extra memory for storing all entries in an additional list.

2.4 Inserter Concept in MTL4

Previously we stated that on-the-fly insertion has the disadvantage of slow performance while two-phase insertion disables later insertion. Such libraries typically allow for modifying existing non-zeros but not inserting further non-zeros. We will introduce the concept of inserter objects that both enable high performance and multiple insertion phases.

What is an inserter? It is an object that provides operations and data for setting up efficiently another object — i.e. a matrix or vector in case of MTL4. The insertion phase lasts as long as the inserter object exists. This has certain implications:

- When the inserter object is destroyed, the matrix or vector is ready to use.
- As long as the inserter object exists the referred object might be in an undefined state.[1]
- Two inserters must not refer to the same object.

If an object shall be changed later, the programmer can create a new inserter and modify it. Thus, an application can have alternative modification and (read-only) usage phases. However, the user should be aware that certain inserters (like the one described later in this paper) can be expensive to create and destroy.

Apparently, it would be cumbersome to create an inserter every time a dense vector is modified. Dense vectors and matrices can also be changed without inserter in MTL4. Then why MTL4 does provide inserters for dense matrices and vectors?

1. Inserters enable writing generic functions that modify matrices. Such functions create an inserter for the matrix — regardless whether the matrix is sparse or dense — and perform the modifications.
2. Distributed matrices and vectors can be set up in a more general manner. Entries can be inserted on any processor and the inserter ensures that all entries are set correctly at the end of the insertion phase.[2]

In the following section, we will show how to write programs with inserters.

2.4.1 *Using Inserters*

The following program sets up the compressed matrix from Equation (1):

```
int main(int argc, char* argv[])
{
    compressed2D<float> A(3, 5);
    {
        matrix::inserter<compressed2D<float> > ins(A);
        ins[0][0] << 1.0; ins[0][2] << 2.0;
        ins[1][3] << 3.0;
        ins[2][1] << 4.0; ins[2][4] << 5.0;
    }
    std::cout << "A is\n" << A << '\n';
    return 0;
}
```

[1] MTL4 checks that the insertion is finished before accessing data when compiled in debug mode.

[2] Distributed matrices and vectors and the respective operations are not part of the open-source version of MTL4 and can be provided on demand.

We like to draw the reader's attention to two details. Firstly, the inserter class is parameterized by the matrix type. This allows the inserter to treat matrices of different types in a distinct manner. Secondly, the inserter object is defined within a block. The reason for this style (which probably look strange to most programmers at the beginning) is that the final compression of the matrix is realized in the destructor. Thus, by introducing a new scope for the inserter we ensure that the destructor is called by the end of the block and the matrix is usable afterwards.

Alternatively, one can create a new inserter with dynamic allocation and terminate the insertion by **delete**-ing the inserter, for instance:

```
typedef matrix::inserter<compressed2D<float> > ins_type;
ins_type* p_ins= new ins_type(A);
(*p_ins)[0][0] << 1.0;
// : :
delete p_ins;
```

Dealing with a pointer is less elegant and more error-prone but it is helpful if the insertion process is not realized in the same scope. For instance, several FEM packages like FEniCS or AMDiS (**A**daptive **M**ulti-**Di**mensional **S**imulations, [15]) start and terminate the insertion in one member function respectively and each insertion is also performed by a member function call. As a consequence, no common scope exists for all these operations. Instead, the class handling the insertion is extended with a pointer to an inserter, which is allocated/deallocated by the function starting/terminating the insertion process.

The inserter type has a second template argument that specifies the update operation. This argument is by default overwriting, i.e. after performing

```
matrix::inserter<compressed2D<float> > ins(A);
ins[0][0] << 1.0; ins[0][0] << 2.0;
```

A[0][0] is 2 (after ins is destroyed). Matrices in FEM are typically the sum of smaller sub-matrices and the overwriting scheme is not appropriate. To accumulate the contributions to each matrix element the inserter must be parameterized with update_plus, e.g. the code-let

```
matrix::inserter<compressed2D<float>,
                 update_plus<float> > ins(A);
ins[0][0] << 1.0; ins[0][0] << 2.0;
```

results in A[0][0] being 3. Users can add arbitrary update schemes while overwriting and adding will probably cover most applications.

As final example we like to show how to set up generically a tridiagonal matrix[3] for an arbitrary matrix type:

```
template <typename Matrix>
void poisson1D(Matrix& A)
{
    A= 0;
    matrix::inserter<Matrix> ins(A);
    for (unsigned i= 0; i < num_rows(A); i++) {
        ins[i][i] << 2;
        if (i > 0) ins[i][i-1] << -1;
        if (i < num_rows(A)-1) ins[i][i+1] << -1;
    }
}
```

[3]Arising from discretizing a Poisson equation equidistantly in one dimension.

The first statement zeroes out dense matrices (which is not done by default for performance reasons). For sparse matrices this operation is void. Conversely, the creation of the inserter is void for dense matrices but performs some operations for sparse matrices. The next section will describe in detail how the initial insertion and the modification of compressed matrices is realized algorithmically.

2.4.2 Algorithm

The insertion starts by reserving space for s entries in each row. The space of one row is called slot. By default the slot size s is set to five and users are recommended to choose a size that is appropriate for their application. Advise on the choice of slot size will be given in Section 2.4.3. The slot size is specified in the (optional) second argument of the inserter's constructor:

```
matrix::inserter<Matrix> ins(A, 3);
```

The slots are provided by the constructor of the inserter so that the insertion can begin directly after the inserter's creation.

Each slot is represented by its starting and finishing (past-end) position — using right-open intervals as in the CRS format. Values are inserted directly in the slots as long as space is still available. Figure 3 depicts the inserter's state after inserting the entries from Equation (1).

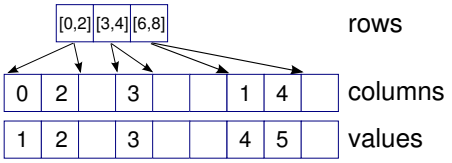

Figure 3: Preparation phase of inserter

The insertion of a new value preserves the column order by shifting all entries with higher column indices. Since this shift is limited to one slot it is rather cheap (compared to shifting additionally all entries with higher row index when changing the matrix on-the-fly). Regarding the row indices, only the slot end of the actual row is incremented. Figure 4 illustrates the impact of introducing a_{01}.

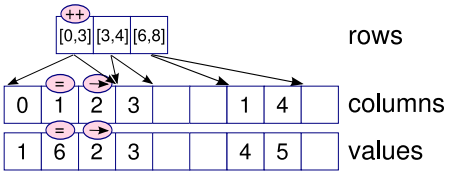

Figure 4: Insertion in available row slot

The slot of row r is filled when the slot end is equal the slot start of row $r+1$. In this case, new entries are put into an extra container as depicted in Figure 5. This container is a std::map and allows for fast determination whether or not a value for the same matrix element was already inserted. Although all operations in the std::map have only logarithmic cost[4], it is significantly more expensive than the direct

[4]With respect to the number of map elements

insertion into slots due to the need of dynamic memory allocation. We will refer to this as *indirect insertion* opposed to *direct insertion* when the entry is stored in the slot.

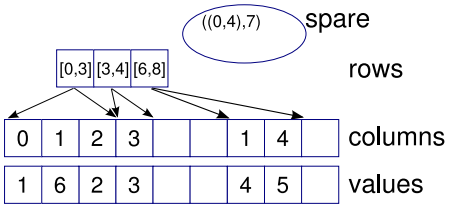

Figure 5: Insertion in container for spare entries

The destructor performs the compression of the matrix. In the most general case, three phases are needed. In the first phase, the total size of each row is computed from the slot size and the entries in the spare container. The arrays for the column indices and for the values are resized at this point if necessary. The second phase moves the entries in the slots to their final location in an appropriate combination of forward and backward copy operations. Finally, the values in the spare container are inserted in the rows and unused elements at the end of the column index and value arrays are cut off.

2.4.3 Time and Space Complexity

One clear advantage of the inserter over certain other approaches is that no memory needs to be pre-allocated nor need entries to be pre-sorted (like compressed graphs in BGL [13]). The inserters work without any kind of (external) preparation phase.

The overall memory consumption of the inserter is extremely small if the slot size is chosen appropriately. In most applications an upper limit for entries per row can be given. For instance, the convection-diffusion equation discretized on a structured 3D mesh might yield a matrix with 27 non-zero entries per row for inner grid points, 18 non-zeros on boundary faces, 12 non-zeros on boundary edges, and 8 non-zeros on the corners. A slot size of 27 enables direct insertion of all entries and uses almost the entire reserved space.

Providing the number of non-zeros in rows for matrices associated with unstructured meshes is more difficult. However, information like the minimal angle in cells and the discretion scheme (e.g. element type in FEM) enable the estimation of an upper limit.

In many cases, unstructured meshes arise from adaptive or non-adaptive refinement processes. The initial mesh is usually rather small and the efficiency of its treatment is not relevant for the application's overall performance. Matrices built from refined meshes can use the sparsity information on the next coarser mesh. As a rule of thumb, using the average number of non-zeros plus 20 to 30 % as slot size on the fine mesh should provide on one side enough space to insert most entries directly and on the other side not reserving too much additional space. For this analysis, we assume that sets of sparse matrices originate from the same class of problems (e.g. same FEM/FVM/FDM disretization on different meshes) and that the number of non-zeros per row is limited by some problem-inherent constant k.

Although adding entries to the spare container is considerably more expensive than direct insertion, we estimate that less than 10 % entries in the std::map should not decrease the overall performance significantly.

A sharp estimation of the slot size — i.e. the number of rows times slot size is just a bit smaller than the number of non-zeros — enables the creation of compressed matrices that fill 90 % of the memory. To our best knowledge, there is no other general-purpose technique allowing this. This feature is very important for extremely memory-intensive application (e.g. from quantum physics [5]).

Resuming, the space complexity for inserting n non-zero entries in a compressed matrix with m rows is $\max\{n, m\,s\}$. Unless s is chosen much too large ($s \gg n/m$) the memory need is linear. For well-chosen slot sizes ($s \approx n/m$) the inserter needs only marginally more space than the compressed matrix itself.

Time complexity also depends on how often the spare container is used and to what extend entries can be inserted directly in the slot. Finding the position for inserting new entries is performed by linear search and above a certain threshold by binary search. The threshold is by default set to 10 and can be modified by a compiler flag. Thus, the search effort is asymptotically logarithmic in s. Shifting all entries with larger column index requires in average moving half of the entries previously inserted in the row in question. Therefore, the shifting effort for filling one slot is quadratic in the slot size. Since the slot size s is a constant the overall effort for direct insertion is still linear.

Inserting an entry in the spare container takes logarithmic time to find the position and constant time to create a new entry in the container. Although constant, it demands dynamic memory allocation, which is on most platforms orders of magnitudes more expensive than the search or shift operations.

In total the insertion time for n entries is

$$t_i(n) = \underbrace{\alpha\frac{\hat{s}}{4}n}_{\text{direct}} + \underbrace{\beta n \log n}_{\text{indirect}}, \tag{2}$$

where α and β are constant factors enclosing the run-time factors and the respective fractions of direct and indirect insertion and $\hat{s}$ the average used slot size. Later insertion/-modification phases are characterized by the same formulae whereby $\hat{s}$ and β increase with increasing n. The reassembly from Section 3 has similar behavior. The difference is that there are no shift operations but the search effort is larger in average because the slots are filled from the beginning

$$t_r(n) = \alpha\frac{\hat{s}}{2}n \tag{3}$$

This explains that reassembly is sometimes more expensive, especially when slot sizes are long.

2.5 Generic Matrix Insertion in FEniCS

The FEM solver in the FEniCS project, DOLFIN, does not contain a custom matrix or vector implementation. Instead, one anticipates that new matrix libraries will become available and provide greater efficiency. Currently DOLFIN has four fully functional back-ends: PETSc, uBLAS, Trilinos and MTL4. A generic matrix and vector interface is defined, and a bit of wrapper code is put in place for each back-end. It should be noted that in DOLFIN, the particular linear algebra back-end being used is determined at run-time. This

is in contrast to MTL4, which provides genericity through template instantiation by the compiler.

For some back-ends, such as uBLAS, a full sparsity pattern must be computed before the assembler can insert matrix elements. A particular benefit with MTL4 is that this is not required. However, MTL4 benefits from knowing the number of non-zeros to anticipate in each row. There is currently no satisfactory way to provide this estimate in DOLFIN. The user code may provide it if precise control is required.

DOLFIN makes use of dynamically allocated inserters, as described above, since various contributions have to be computed before the assembler finalizes the matrix. The same approach is used in AMDiS.

3. PERFOMANCE

3.1 Benchmark Method

Let A be a N by N matrix and let each row contain s nonzero values. Our benchmark is designed to evaluate how efficiently different libraries handle the task of inserting rows into a CRS matrix. This is a common task, e.g. when treating differential equation either with finite element or finite difference methods.

The benchmark cases presented below have been constructed to illuminate several aspects, which have impact on the insertion efficiency, that arise in different applications:

- The order in which rows are visited.
- The order of column indices within each row.
- The fill rate, or number of non-zeros per row and matrix size.
- Inserting new values in the matrix without changing the sparsity structure, i.e. reassembly.

That rows are visited out-of-order, and in a non-uniform fashion, is typical in applications with unstructured meshes. It is well known that this can have a devastating impact on the computational efficiency of e.g. a finite element matrix assembler. One typically tries different reordering strategies to circumvent this problem, or specialized insertion algorithms.

However, in the context of generic insertion, we are interested in how the libraries hide the insertion complexity from the application code. Therefore we do not benchmark the peak insertion rate attainable with each library for each of the cases below. Instead we make one insertion routine for each library and run all the benchmark cases through it, in the simplest possible setting:

```
insert_row(Matrix& A, int row_idx,
           int* cols_idx, double* a, int s){
    for(int j=0; j<s; j++)
        A(row_idx , cols_idx[j]) += a[j];
}
```

The different libraries used are:

MTL4 We use the insertion mechanism described in previous sections for the compressed row-major format. The slot size of the inserter is set to s.

uBLAS [16] We use both compressed row-major and generalized vector-of-vector (GVOV) formats, since insertion efficiency into uBLAS CRS matrices is strongly dependent on row order. Note that the GVOV matrix is converted to CRS after the insertion process, and, for sake of comparison, that the time required for this operation is included.

PETSc [4] A serial (i.e. non-MPI, uni-processor) build of PETSc was used, and matrices in the `SeqAIJ` matrix format initialized with capacity for s non-zeros per row. PETSc provides facilities for tailoring the insertion algorithm to each situation. We do not use these, something our conclusions will take into account.

We measure insertion efficiency in units of non-zero matrix elements inserted per second (#nz/s). All results are averaged over several repetitions. The platform used for these test was a commodity workstation: Inter E6600 CPU (Core2 Duo, 2.4 GHz), running 32-bit Linux.

3.2 Benchmark A: Ascending row order

Here we insert rows in sequential order. Within each row the column indices are random, but sorted in advance. This represents a best case scenario for insertion algorithms.

	Assembly	Reassembly (#nz/s)
$s = 5,\ N = 10^4$		
MTL4	4.66E+07	5.80E+07
uBLAS	5.89E+06	4.29E+07
uBLAS (gvov)	2.03E+06	4.29E+07
PETSc	2.20E+07	3.75E+07
$s = 50,\ N = 10^5$		
MTL4	2.96E+07	2.50E+07
uBLAS	6.50E+06	1.75E+07
uBLAS (gvov)	2.83E+06	1.89E+07
PETSc	3.23E+07	4.09E+07

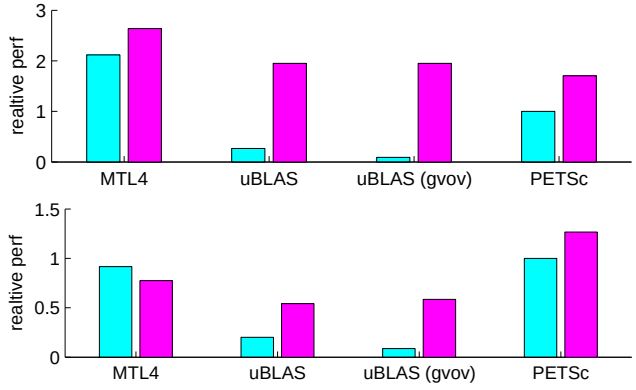

Figure 6: Assembly and reassembly rate for benchmark A, normalized to PETSc assembly rate. Top: $s = 10,\ N = 10^4$. Bottom: $s = 50,\ N = 10^5$

3.3 Benchmark B: Reverse row order

Here we reverse the row order. Columns are still in sorted order within each row.

	Assembly	Reassembly (#nz/s)
$s = 5,\ N = 10^4$		
MTL4	4.68E+07	6.02E+07
uBLAS	1.74E+04	4.22E+07
uBLAS (gvov)	1.97E+06	4.21E+07
PETSc	2.18E+07	3.71E+07
$s = 50,\ N = 10^5$		
MTL4	2.71E+07	2.34E+07
uBLAS (gvov)	2.84E+06	1.75E+07
PETSc	3.04E+07	3.76E+07

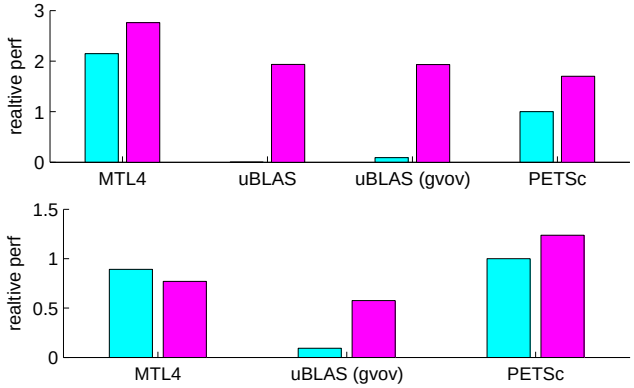

Figure 7: Assembly and reassembly rate for benchmark B, normalized to PETSc assembly rate. Top: $s = 10,\ N = 10^4$, bottom: $s = 50,\ N = 10^5$

3.4 Benchmark C: Randomly permuted row order

Here the row order is randomly permuted, with columns in sorted order within each row.

	Assembly	Reassembly (#nz/s)
$s = 5,\ N = 10^4$		
MTL4	4.14E+07	5.24E+07
uBLAS	3.13E+04	3.88E+07
uBLAS (gvov)	1.89E+06	3.85E+07
PETSc	1.99E+07	3.31E+07
$s = 50,\ N = 10^5$		
MTL4	2.56E+07	2.17E+07
uBLAS (gvov)	2.73E+06	1.62E+07
PETSc	2.56E+07	3.00E+07

3.5 Benchmark D: Random columns within row

As the final row insertion case, we take rows in permuted order again, but let the columns be unsorted within each row.

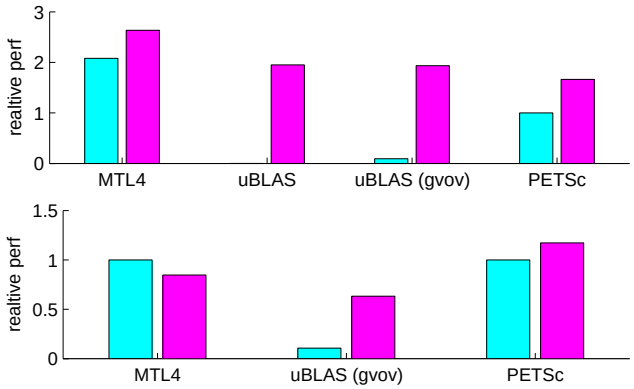

Figure 8: Assembly and reassembly rate for benchmark C, normalized to PETSc assembly rate. Top: $s = 10,\ N = 10^4$. Bottom: $s = 50,\ N = 10^5$

	Assembly	Reassembly (#nz/s)
$s = 5,\ N = 10^4$		
MTL4	1.05E+07	3.32E+07
uBLAS	3.12E+04	2.77E+07
uBLAS (gvov)	1.92E+06	2.78E+07
PETSc	1.55E+07	2.25E+07
$s = 50,\ N = 10^5$		
MTL4	6.09E+06	1.61E+07
uBLAS (gvov)	2.34E+06	1.19E+07
PETSc	7.69E+06	1.44E+07
$s = 50,\ N = 10^6$		
MTL4	5.95E+06	1.54E+07
uBLAS (gvov)	2.29E+06	1.14E+07
PETSc	7.49E+06	1.37E+07

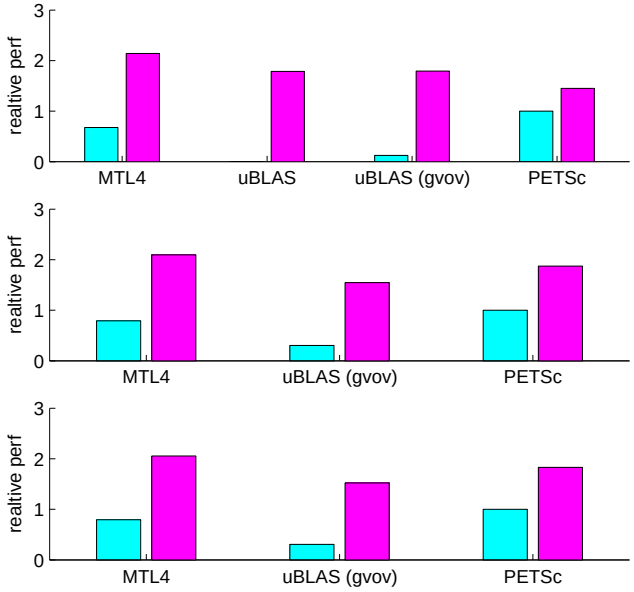

Figure 9: Assembly and reassembly rate for benchmark D, normalized to PETSc assembly rate. Top: $s = 10,\ N = 10^4$. Middle: $s = 50,\ N = 10^5$. Bottom: $s = 50,\ N = 10^6$

3.6 Benchmark E: Fully random insertion

Here we measure completely unstructured element insertion:

```
random_insert(Matrix& A, int N, int nzr,
              double value){
    for(int k=0; k<N*nzr; k++){
        A(rand()%N, rand()%N) += value;
    }
}
```

	Assembly (#nz/s)
$s = 10,\ N = 10^4$	
MTL4	4.82E+06
uBLAS	1.67E+04
uBLAS (gvov)	1.80E+06
PETSc	1.59E+04
$s = 50,\ N = 10^4$	
MTL4	2.87E+06
uBLAS	3.34E+03
uBLAS (gvov)	1.72E+06
PETSc	1.34E+04

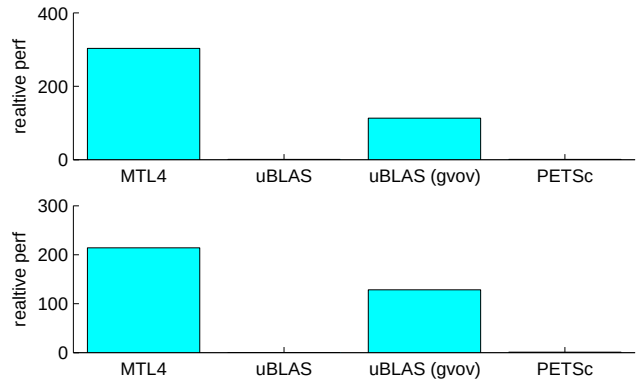

Figure 10: Assembly and reassembly rate for benchmark E, normalized to PETSc assembly rate. Top: $s = 10,\ N = 10^4$. Bottom: $s = 50,\ N = 10^4$

4. CONCLUSIONS AND FUTURE WORK

We presented a new approach for setting and modifying compressed sparse matrices. The technique does not need any preparation phase, uses minimal memory and provides high performance. Especially for randomly inserted entries the advantage of the method is apparent.

An insertion mechanism especially tailored for large blocks is under development. By sorting the entries within the block, the linearly growing impact of the slot filling is substituted by logarithmic complexity of the sorting method. MTL4 and PETSc also provide distributed insertion, which will be compared in future publications. Likewise, the performance behavior on other platforms like multi-core will be examined.

We will extend MTL4 towards other sparse formats, e.g. blocked CSR (BCSR), and the inserter concept will applies naturally to these formats as well. Thus, all applications that set up matrices generically with an inserter can use the new formats without code modifications.

5. REFERENCES

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002.

[2] Göddeke et al. Using GPUs to improve multigrid solver performance on a cluster. *IJCSE*.

[3] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Press, Philadelphia, 1994.

[4] S. Balay et al. (PETSc team). PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[5] F. Alet et al. (the ALPS coll.). The ALPS project: open source software for strongly correlated systems. *J.PHYS.SOC.JPN.*, 74:30, 2005.

[6] G. C Fox. Matrix operations on the homogeneous machine. Technical report, California Institute of Technology, Pasadena, CA, July 1982.

[7] J. Gerlach, P. Gottschling, and U. Der. A generic C++ framework for parallel mesh based scientific applications. In *Proc. HIPS, San Francisco*, 2001.

[8] P. Gottschling, D.S. Wise, and M.D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proc. ICS '07*, pages 116–125, New York, NY, USA, 2007. ACM Press.

[9] P. Gottschling, D.S. Wise, and A. Joshi. Generic support of algorithmic and structural recursion for scientific computing. In *POOSC'09 at ECOOP08*, Cyprus, Greece, 2008.

[10] J. Hoffman, J. Jansson, A. Logg, G. N. Wells, et al. DOLFIN. `http//www.fenics.org/dolfin/`, 2006.

[11] R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006.

[12] Andrew Lumsdaine, Jeremy Siek, Lie-Quan Lee, and Peter Gottschling. The Matrix Template Library home page. `http://www.osl.iu.edu/research/mtl`, 2006.

[13] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, Boston, MA, USA, 2002.

[14] J. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, number 1505 in LNCS, pages 59–70, 1998.

[15] Simon Vey and Axel Voigt. AMDiS: adaptive multidimensional simulations. *Computational Visualization and Science*, 10:57–67, 2007.

[16] Jörg Walter and Mathias Koch. uBLAS – Boost Basic Linear Algebra. `www.boost.org/doc/libs/release/libs/numeric/ublas/doc/index.htm`, 2002.