Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption

Peter Gottschling
Technische Universität Dresden and SimuNova,
Helmholtzstr. 10
01069 Dresden, Germany
peter.gottschling@tu-dresden.de

Torsten Hoefler
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street
Urbana, IL
htor@illinois.edu

***Abstract*—Sparse linear algebra is a key component of many scientific computations such as computational fluid dynamics, mechanical engineering or the design of new materials to mention only a few. The discretization of complex geometries in unstructured meshes leads to sparse matrices with irregular patterns. Their distribution in turn results in irregular communication patterns within parallel operations.**

In this paper, we show how sparse linear algebra can be implemented effortless on distributed memory architectures. We demonstrate how simple it is to incorporate advanced partitioning, network topology mapping, and data migration techniques into parallel HPC programs by establishing novel abstractions.

For this purpose, we developed a linear algebra library — Parallel Matrix Template Library 4 — based on generic and meta-programming introducing a new paradigm: meta-tuning. The library establishes its own domain-specific language embedded in C++. The simplicity of software development is not paid by lower performance. Moreover, the incorporation of topology mapping demonstrated performance improvements up to 29 %.

I. Motivation

Many scientific simulations, such as computational fluid dynamics, mechanical engineering or the design of new materials use computations on unstructured grids as their core method (§II-A). The operations are expressed as linear algebra (LA) with sparse matrices. These matrices are very often unstructured, that is, the distribution of non-zero values and the data dependencies of typical operations, such as matrix-vector multiplication, are irregular.

Many large-scale scientific HPC applications can highly benefit from specialized data structures and domain-specific algorithms operating on them. On the other hand, strongly specialized implementations are very expensive to expand for new algorithms and new data structures.

The introduction of PETSc [1] in the 90s provided reusable algorithms and data structures for many applications leading to a significant increase of productivity in scientific software development. We aim to raise the productivity further with techniques that did not exist yet at the time PETSc was created.

The goal is that the linear algebra library adapts itself to the scientific application instead of applications designed around libraries. Such adaption can be achieved thanks to the expressiveness and efficiency of the template system of C++ [2]. We developed the Parallel Matrix Template Library 4 (PMTL4 [3], [4]) that allows developers to:

- Program concisely in an intuitive mathematical notation;
- Replace algorithms (like linear solvers) without changing the remainder of the application;
- Substitute data structures (like matrix types) with compatible ones without rewriting the calculations;
- Benefit from specialized codes (like hand-tuned BLAS routines) without stating it explicitly;
- Mix and match orthogonal components leading to an exponential growth of functionality over the size of the library;
- Utilize user-defined and third-party types like quaternions, high-precision numbers, or rationals;
- Debug graphically like with C arrays [5];
- Tune implementations at the function call [6];
- Parallelize applications by simply substituting some type definitions;
- Change the distribution of data without adapting data structures and algorithms;
- Distribute unstructured sparse matrices automatically to parallel processes with balanced load and minimized communication (i.e. a good domain decomposition); and
- Map the parallel processes automatically to the physical network topology (i.e. minimize network congestion or dilation).

Most of the concepts are discussed in [3] and [4]. In this work, we focus on the last two, distributing the unstructured matrices and mapping the resulting communication graph to the network topology. Ideally, these tasks are performed without user assistance leading to convenient libraries that allow developers to program with intuitive abstractions but without sacrificing performance (§II-B–II-F).

Domain-decomposition techniques for structured and unstructured grids have been intensively analyzed and libraries that provide good decompositions are ready for use. In contrast to it, mapping those unstructured grids and their according irregular communication topologies onto static network topologies just recently received the attention it deserves. We integrate LibTopoMap—a library for generic topology mapping—in a fully automatic way into PMTL4.

Today's large-scale supercomputers consist of sparse networks with a relatively low connectivity and hierarchical networks. Large multi- or many-core nodes provide fast intra-

node communication and are connected by multi-dimensional high-performance network topologies. Manual optimization for such architectures is difficult because parameters and topologies change for each system and even from run-to-run based on the current allocation of nodes. Even if the network topology is understood, mapping the unstructured, input-dependent application topology is a daunting task. Thus, an automated and generic technique for mapping application communication topologies to network topologies during runtime is needed.

This fact is acknowledged in several parallel programming frameworks: CHARM++ [7] provides transparent support for topology mapping by process migration and the Message Passing Interface (MPI) [8, §7] allows users to specify the communication relations among processes of a parallel program, enabling the MPI implementation to renumber processes for efficient mapping.

Contributions: In this work, we propose an abstract library interface for parallel sparse matrix computations and effective mapping schemes. We show how the MPI-2.2 graph interface and the topology mapping library can be integrated into parallel applications without increasing code complexity.

We will show in §II-B–II-D that using our mapping in the Parallel Matrix Template Library 4 (PMTL4) is as easy as writing Matlab code. The partitioning and the topology mapping is entirely orthogonal to the distributed data layout and hidden from the user. The two transformations can be specified by a single object applicable universally on matrices and vectors. All information for partitioning and mapping can be extracted from a sparse matrix allowing to write the reorganization in a single statement.

A. Related Work

Parallel Linear Algebra: In addition to several libraries for dense parallel LA—e.g., PLAPACK [9]—two libraries outstand for sparse parallel LA: PETSc and Trilinos [10]. PETSc is written in C and can be easily interfaced with applications in C, C++, and Fortran. The drawbacks of the library are the not particularly concise interface and the restriction to one scalar type, that is, programmers using both real and complex arithmetic need two installations of the library. Trilinos is implemented generically—thus enabling arbitrary arithmetic—but the interface is still quite complicated. To our best knowledge, no library uses MPI topology mapping.

Partitioning and Topology Mapping: We use the well known library ParMETIS [11] to compute a domain decomposition that minimizes the communication volume during the computation. ParMETIS computes a (k,1+ϵ)-balanced partitioning which is optimized for partitioning large input domains.

Several researchers investigated techniques to optimize process mappings on parallel computers. Different heuristics have been applied to the problem. Träff [12] uses partitioning strategies, similar to ParMETIS, for mapping to strictly hierarchical networks (e.g., SMPs). Bokhari [13] uses the well-known problem of graph isomorphisms to find mappings by ignoring the influence of non-mapped edges. Lee and Agarwal [14] define a more accurate model including all edges in a simple greedy search strategy followed by a technique similar to simulated annealing. Similar techniques have been discussed by Bollinger and Midkiff [15]. The SCOTCH library offers static mapping based on recursive bisections to map arbitrary graphs [16].

Other, network- or topology-specific approaches exist for a limited set of topologies and systems, such as BlueGene/L [17] and Torus topologies [18].

Hoefler and Snir summarize known algorithms and develop an algorithm based on graph similarity in a survey paper [19] and proof that the problem of finding a mapping that minimizes dilation or congestion is NP-complete.

Generic Programming and DSEL: The principles of generic programming are consequently followed in the non-parallel linear algebra library uBLAS [20]. Parallelism in a generic manner is provided by Boost.MPI [21] and the Parallel Boost Graph Library [22]. The genericity of the latter allowed for expressing parallel sparse linear algebra computations by means of distributed graphs [23]. The elegance and efficiency of domain-specific embedded languages (DSEL) in scientific computing is best demonstrated by the library Life [24]. It uses the source-code transformation ability of the C++ compiler to generate efficient executables from easy-to-read expressions in the finite-element domain.

II. Parallel Sparse Linear Algebra

A. Why are Sparse Matrices Important in HPC

Parallel HPC computers are dominantly used to simulate physical processes described by ordinary or partial differential equations (ODE/PDE). For illustration purpose, we consider the 3D simulation of a droplet sliding down a rippled ramp [25] as shown in Figure 1.

This process can be modeled by a diffuse domain Cahn-Hilliard system:

$$\begin{aligned}
\partial_t(\phi\mathbf{u}) + \nabla\cdot(\phi\mathbf{u}\mathbf{u}^T) &= -\phi\nabla p + \nabla\cdot(\nu\phi\mathbf{D}) + \frac{\tilde{\sigma}}{\epsilon}\phi\mu\nabla c \\
&\quad + \phi\mathbf{F} + BC_u, \\
\nabla\cdot(\phi\mathbf{u}) &= \mathbf{g}\cdot\nabla\phi, \\
\partial_t(\phi c) + \nabla\cdot(\phi c\mathbf{u}) &= \nabla\cdot(\phi M(c)\nabla\mu) + BC_\mu, \\
\phi\mu &= \phi W'(c) - \epsilon^2\nabla\cdot(\phi\nabla c) + BC_c,
\end{aligned}$$

where t is the time, ϕ a phase field function, $\mathbf{u}$ the velocity, p the pressure, $\mathbf{D} = (\nabla\mathbf{u} + \nabla\mathbf{u}^T)$, c the phase field concentration, BC_u, BC_μ, BC_c are internal boundary conditions, g the velocity of Γ, $M(c)$ a mobility, μ the chemical potential, ϵ the interface size of ϕ, and $\mathbf{F}$ the body force.

For numerical treatment, continuous domains are discretized by meshes of simpler geometry like triangles or tetrahedra. The continuous equations are discretized by methods like finite elements (FEM) on specific mesh points. This results in a sparse linear system whose solution approximates the continuous one at the mesh points. The equations above discretized with FEM on the before-mentioned 3D mesh yield

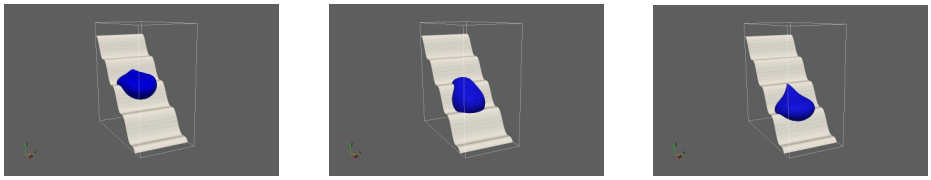


Fig. 1. Numeric simulation example of a droplet sliding down a ramp involving multiple coupled physical processes

a linear system where the sparse matrix has the sparsity pattern shown in Figure 2.

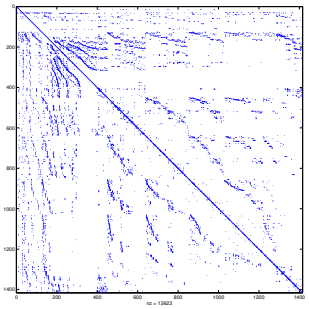


Fig. 2. Sparsity of the discrete problem's matrix

The regularity of the discretization mesh directly affects the pattern of the sparse matrix. If the mesh is an equi-distant grid then the sparse matrix is a regular stencil. Accordingly, irregular meshes yield unstructured matrices. Especially, adaptive mesh refinement (AMR) causes unstructured sparsity. Industrial applications in particular need dominantly simulations with unstructured sparse matrices. Therefore, their efficient handling in large-scale parallel simulations is extremely important.

Solving linear systems iteratively requires local vector operations, vector reductions, and matrix vector products. From the topology prospective, only the latter matters and we focus on its implementation.

B. Abstractions of Distributed Data

The development of parallel HPC software is typically work-intensive. To decrease this effort it is very important to provide highly reusable components. Unfortunately, many approaches to reusability significantly impair performance, e.g. using techniques of object-oriented programming (OOP) at a fine-grained level can lead to slow-downs of one or more orders of magnitude [26].

The static polymorphism of C++ templates avoids the runtime overhead of OOP or function pointers, confer [2]. Moreover, the reusability with templates is even broader then with OOP or function pointers because it is not limited to types derived from a specific base class or to functions with a specific signature but can be used with every type providing the according interface. This approach was formalized by Musser and Stepanov as Generic Programming (GP) [27], [28] based on the algebraic foundations of Formal Concept Analysis [29]. The prototype of generic software is the Standard Template Library in C++ [30].

Generic programming establishes tremendous expressiveness when augmented with Meta-programming [31]. Information on types are available within programs and can be explored to select the best suitable algorithm. Veldhuizen has shown that the template system of C++ is Turing complete [32]. In fact it can be considered a complete functional language. Every expression in terms of integral values and types can be evaluated during compilation.

The Matrix Template Library [4] utilized these paradigms intensively. Incorporating meta-programming in numeric high-performance software enabled a new kind of tuning, called meta-tuning, that allows for generating tuned executables with every standard C++ compiler [6]. For instance, loop unrolling can be parametrized in the function call or by formulas taking the argument type, the computed expression, or the underlying platform into consideration. It has been demonstrated [6] that the generated code has identical performance as its hand-written counter-part.

In the remainder of this section we introduce abstractions provided by the parallel MTL4 using the before-mentioned paradigms.

1) Distribution: We defined a concept Distribution that specifies how an index range can be divided into a given number of sub-sets. So far, we provide three types of Distribution:

- block_distribution;
- cyclic_distribution; and
- block_cyclic_distribution.

These are implemented as classes with functions to calculate:

- The number of local entries;
- The global index of a given local one;
- The location of a global index; ...

With this abstraction we can implement vectors and matrices of different distributions while using the same implementation.

2) Distributed vectors: This is defined by the two orthogonal abstractions of a local vector and a distribution:

```
template <typename LocalVector, typename Dist>
class distributed;
```

The parametrization of the local vector allows for sparse vector and vector types from third-party libraries. The global indices of the vector are mapped to the process according to the vector's distribution object. For instance, an object of type block_distribution always arranges the indices associated with

a process consecutively whereas the size of each block can be set automatically or defined explicitly.

3) Distributed matrices: The design space for distributed matrices is larger since there are already more types for local matrices. Furthermore, we define a second distribution:

```
template <typename LocalMatrix, typename Dist1,
          typename Dist2>
class distributed;
```

Matrices being finite-dimensional linear operators from one vector space into another, a distributed matrix can accordingly map a vector stored in distributed manner (i.e. Dist2) onto a vector stored with a different distribution (i.e. Dist1). To avoid name conflicts the two class templates distributed are defined in the name spaces mtl::vector and mtl::matrix accordingly.

4) Migrations: A migration object provides a generic definition of how distributed vectors and matrices are copied to ones with other distributions, more details follow in §II-D.

C. Parallel Operations

Parallel operations constitute of local numeric operations on blocks and communication. To support user-defined types, e.g., high-precision arithmetic numbers, for elements of distributed matrices and vectors, we use the generic communication library Boost.MPI [21]. This library works with all types that are serializable by Boost.Serialization [33].

1) Initializing Distributed Data: For this purpose we use the same abstraction as for non-distributed data: inserter. The concept is described in detail in the MTL4 tutorial [3] and we only discuss its genericity in the parallel context here. Thus, the following code snippet is valid for both distributed and non-distributed matrices:

```
{
    matrix::inserter<type_of_A, updater> ins(A);
    for ( ... )
        ins[i][j] << x;
}
```

In case that A is a distributed matrix, the inserter is specialized to the distributed implementation. The indices are assumed to be global. As in the non-distributed case, the behavior of A is undefined as long as the inserter exists. The inserter's destructor sets up A including sending data to the owning processes and creating buffers for future communications.

The inserter concept allows programmers to write initialization functions that work both on distributed and non-distributed matrices. For instance, in a finite element code one can just insert the element matrices if the elements are distributed in the parallel case (what they usually are). Redundant insertions of entries are incorrect with incremental updates and with overwriting update it is correct but rather inefficient.

2) Calculations: Parallel calculations on distributed data are syntactically equal with their non-parallel counterparts:

```
y= A * x;
```

represents in our library a parallel matrix vector product when distributed matrix/vectors are used. The dispatching among operations is entirely performed at compile time.

Internally, the operation above is implemented by the function template:

```
template <typename Matrix, typename VectorIn,
          typename VectorOut>
void inline dist_mat_cvec_mult(const Matrix& A,
                const VectorIn& v, VectorOut& w)
{
    dist_mat_cvec_mult_handle h=
        dist_mat_cvec_mult_start(A, v, w);
    local(w)= local(A) * local(v);
    dist_mat_cvec_mult_wait(A, v, w, h);
}
```

The calculation is split into three parts: the start of the communication, the operations on local blocks, and the completion with the received remote data. The encircling communicative parts will be realizable with eight schemes resulting from the respective choices between:

- Blocking or non-blocking;
- Collective or point-to-point;
- Derived-type-based or packed.

Not all combinations are implemented yet. Our benchmarks use non-blocking point-to-point communication overlapped with computations.

D. Migration

To minimize communication by better locality, we perform a graph partitioning on the matrix's sparsity graph. The result is a *partition*, i.e. a mapping from index to processor

$$\mathrm{p}\colon \mathcal{I} \to \mathcal{P}.$$

Subsequently, the processes can be optionally permuted according to the topologies

$$\pi\colon \mathcal{P} \to \mathcal{P}.$$

Both functions can be combined to a topology-aware partitioning

$$\hat{\mathrm{p}} = \pi \circ \mathrm{p}\colon \mathcal{I} \to \mathcal{P}.$$

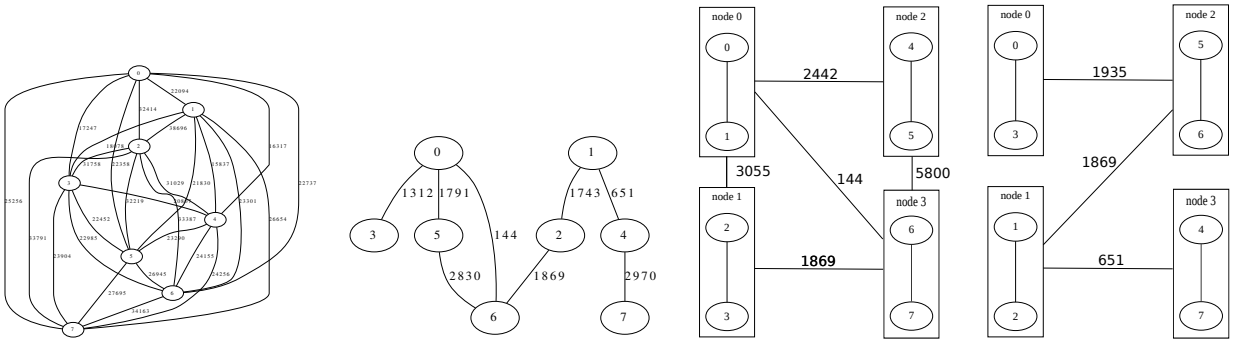
Sorting the indices by assigned processors yields a new block_distribution. The migration from old to new distribution is represented by a block_migration object:

```
block_migration migr= parmetis_migration(A);
matrix_type B(A, migr);
```

Passing the migration object to a matrix or vector constructor provides a convenient way for migration (not the only one). The entire partitioning, topology mapping, and migration can be realized with a single statement like:

```
matrix_type B(A, parmetis_migration(A));
```

However, since multiple objects usually need to be migrated consistently, it is better to keep the migration object as in the first listing. This object can also be used for migrating back results to the original distribution (as needed for post-processing):



(a) Naïve distribution. (b) Optimized distribution. (c) Naïve mapping. (d) Optimized mapping.

Fig. 3. Domain distribution and topology Mapping of an Example Matrix (F1) running with eight processes on four dual-core compute nodes.

```
vector_type x2(x, reverse(migr));
```

Figure 3 illustrates the impact of partitioning and topology mapping on a sample matrix (F1 from Table I). First, the matrix was read from file and stored block-wise with the rows in the same order as in the file, see Figure 3(a). Although the details are difficult to read in this representation it is obvious that the sub-problems are almost completely connected. In a matrix vector product 715,655 entries must be sent, that is all vector entries are sent twice in average. The longest message already contains 38,696 entries.

Repartitioning the matrix—Figure 3(b)—i.e. permuting the rows for better locality reduces the connectivity of the sub-domains, the overall message size to 13,310 entries and the longest message to 2,970 entries. Let us assume that the computations are performed on four dual-core nodes. If the sub-domains are assigned in order to the cores as depicted in Figure 3(c), the longest message would contain 5,800 entries and the entire communication is between nodes without any intra-node communication, i.e. the total message size is still 13,310. Mapping the sub-problems according to their communication topology—as in Figure 3(d)—reduces the communication load to 4,455 entries.

E. Complete Solution

The following example contains a complete application that sets up a linear system, computes a partitioning including topology mapping, migrates the data, solves the linear system and migrates the data back:

```
#include <iostream>
#include <boost/numeric/mtl/mtl.hpp>
#include <boost/numeric/itl/itl.hpp>

int main(int argc, char* argv[])
{
    using namespace mtl;
    typedef mtl::vector::distributed<dense_vector<double> >
        vector_type;
    typedef matrix::distributed<compressed2D<double> >
        matrix_type;

    par::environment env(argc, argv);

    // Set up linear system
    matrix_type A(io::matrix_market("heat_transfer.mtx"));
    vector_type b(num_rows(A), 1.0);

    // Migrate data
    par::block_migration migration= parmetis_migration(A);
    vector_type b2(b, migration), x2(resource(b2));
    matrix_type A2(A, migration);

    // Solve A2 * x2 = b2
    itl::pc::ilu_0<matrix_type, float> P(A2);
    itl::cyclic_iteration<double> iter(b2, 500, 1.e-8, 0.0, 100);
    bicgstab_2(A2, x2, b2, P, iter);

    // Migrate back and post-process
    vector_type x(x2, reverse(migration));
    par::sout << "solution is " << x << std::endl;
    return 0;
}
```

The preconditioner ILU_0 (incomplete LU factorization without fill-in) is specialized for distributed matrices and performs a block-wise ILU on each local matrix. By default the values of the precoditioners have the same type as those of the preconditioned matrix. In our example, we explicitly used single-precision as this allows for faster calculation without significant loss of precision or convergence.

Although such comparisons are problematic, realizing the same task with PETSc takes according to experts several 100 lines of code in order to achieve good performance. Performing the preconditioner with a lower precision is not possible at all. Figure 4 shows for a 2D Poisson equation that our conjugate gradient (CG) solver performed slightly better than PETSc for realistically large examples. In this weak scaling the tests (from top down) contain about 1, $1/2$, and $1/4$ millions unknowns per core, more details and other benchmarks in [3].

F. Domain-specific Embedded Language

The manner how to define a DSEL in C++ is driven by the fact that the language provides about 30 user-definable operators but allows at the same time for the introduction of an infinite number of types. Thus, in order to provide a broad functionality with a compact notation, the user-definable operators must be appropriately overloaded for multiple types.

For instance, the MTL4 contains a sub-matrix function:

```
sub_matrix(A, i, j, k, l)
```

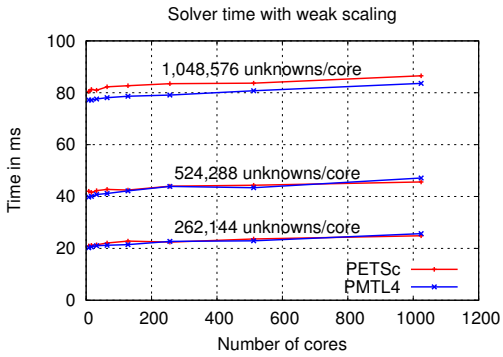


Fig. 4. Runtime of one iteration of a CG solver in PETSc and PMTL4

Unless programmers work frequently with this function, they will need to look up the documentation whether "j" is the last row or the first column of the sub-matrix. One might even wonder whether the matrix is the first or the last argument of the function. In contrast to it, by introducing a right-open index range the expression:

```
A[irange(i, j)][irange(k, l)]
```

is understandable intuitively. Although we are not able to provide the same notation as in Matlab, we can still establish a similar interface by defining operators type-dependently.

The bracket operator for matrices is currently overloaded five-fold:

```
A[int][int]       ⇒ matrix element;
A[irange][irange] ⇒ sub-matrix;
A[int][irange]    ⇒ row vector;
A[irange][int]    ⇒ column vector;
A[iset][iset]     ⇒ sub-matrix;
```

iset is a set of indices and the sub-matrix resulting from the last expression may refer to arbitrary rows and columns of the referred matrix.

Using this notation, the implementation of an LU factorization is compact and very readable:

```
for (std::size_t k= 0; k < num_rows(LU)−1; k++) {
    if(abs(LU[k][k]) <= eps) throw matrix_singular();
    irange r(k+1, imax); // Interval [k+1, n−1]
    LU[r][k]/= LU[k][k];
    LU[r][r]−= LU[r][k] * LU[k][r];
}
```

A notation particularly introduced for performance tuning is:

```
expr1 || expr2;
```

The combined expression declares two operations that are usually calculated sequentially to be performed in parallel. In parallel does not mean here that two threads are started but the evaluation is fused. For example, the following two calculations are calculated consecutively in an iterative solver:

```
r−= alpha * q;
rho= dot(r, r);
```

Merging this two operations into one single loop can provide perceivable acceleration due to better locality. We established a domain-specific notation for such fusion to be realized by meta-programming:

```
(lazy(r) −= alpha * q) || (lazy(rho) = lazy_unary_dot(r));
```

Whether two operations can be merged depends on the argument types, e.g. column-major matrices are not suitable. The meta-programming methods in MTL4 ascertain that only fusible expressions are merged. A detailed description with benchmark results will be published in the near future.

III. Topology Mapping

We use and adapt the open-source mapping library, LibTopoMap [19], that incorporates all published algorithms and provides topology mapping for arbitrary multicore architectures and topologies. LibTopoMap offers mappings based on greedy, recursive bisection, graph similarity, and simulated annealing techniques. It also integrates support for SCOTCH [16] static mapping.

We compare LibTopoMap's greedy and reverse Cuthill McKee (RCM) mapping strategies. Further, we apply threshold accepting (TA) to improve the solution further. However, TA can take prohibitively long at larger scales, cf. [19]. We minimize the maximum congestion per link in order to achieve the best runtime.

A. Example Matrices

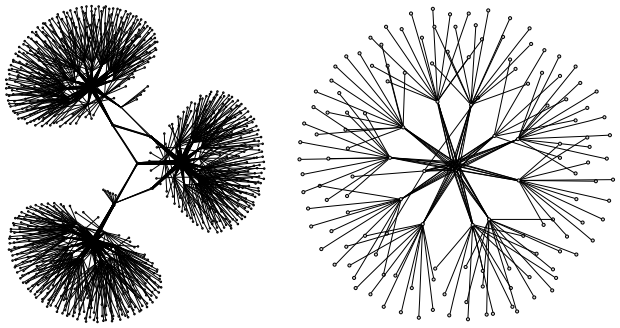
In order to allow everybody reproducing our results we switch here from the introduction example to publicly accessible matrices. We chose three real-world matrices from the University of Florida Sparse Matrix Collection [34]: F1, Ga41As41H72, and ldoor. All matrices represent unstructured meshes. F1 and ldoor are symmetric stiffness matrices—approximating elasticities in structural mechanics—of real-world objects: F1 models an Audi engine crankshaft and ldoor a van's door. Ga41As41H72 belongs to a symmetric eigenvalue problem in density functional theory. Table I lists sizes and numbers of non-zero (nnz) entries for each matrix.

Matrix Name	Rows/Columns	NNZ (sparsity)
F1	343,791	26,837,113 (0.227‰)
ldoor	952,203	42,493,817 (0.047‰)
Ga41As41H72	268,096	18,488,476 (0.257‰)

TABLE I
Matrix dimensions and number of non-zero (nnz) entries.

B. Network Topologies

We use two different test systems, Deimos and Odin. Deimos consists of 729 quad-core nodes connected with three 288 port InfiniBand switches as shown in Figure 5(a). The network has a rather low bisection bandwidth due to its unique topology. The batch system on Deimos allocates cores and not necessarily full nodes, thus, large jobs are usually scattered across different nodes on the entire machine.



(a) Deimos with 729 nodes. (b) Odin with 128 nodes.

Fig. 5. Visualization of network topologies used in the experiments.

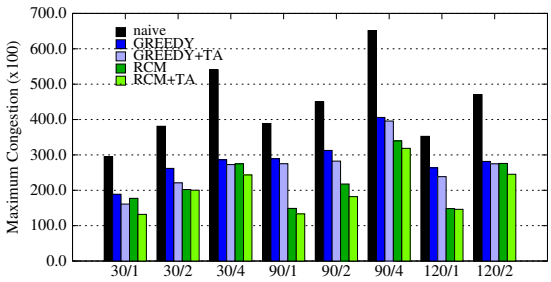


Fig. 6. Odin F1 Congestion.

Odin consists of 128 quad-core machines connected with InfiniBand. The network is a two-stage folded Clos network with full bisection width. The physical topology is shown in Figure 5(b). However, the *effective bisection bandwidth* is only 80% due to static routing [35]. The batch system guarantees the allocation of complete nodes. One important fact is that the injection bandwidth in Odin is limited by a slow PCI-X link to ≈ 500 MiB/s which is only half of the link bandwidth. The resulting higher bandwidth in the network limits the effects of congestion significantly.

C. Mapping Results

We first discuss the strategies' theoretical improvements of maximum congestion (from here on simply "congestion") and average dilation (from here on simply "dilation") on different process and network topologies. Congestion represents the maximum number of data items that have to traverse a single link in the communication network and can thus be seen as a measure of performance. Dilation measures the average distance that a message has to travel in the network and is thus an indicator of power consumption. The optimization goal can be chosen on a case-by-case basis. We saw very similar optimization potential and results for the different matrices, however, heavily depend on the underlying topology. Thus, we discuss the matrix F1 as a representative example for all matrices.

Figure 6 compares the congestion of the different mapping strategies. The leftmost bar indicates the naive (identity) mapping (cf. Figure 3(c)) followed by greedy mapping without and with TA, and the RCM mappings without and with TA, respectively. Each cluster of bars indicates a specific allocation strategy, e.g., "30/2" means the job run on 30 nodes with 2 ppn, that is 60 MPI processes. We see that while our greedy strategy reduces the congestion significantly, RCM performs much better. TA results in small improvement of all mappings.

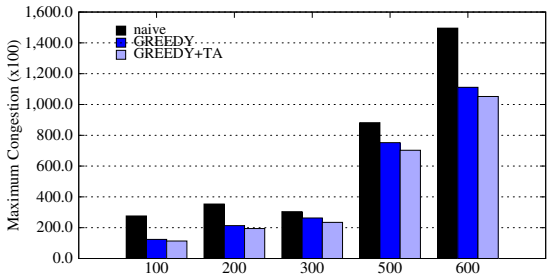


Fig. 7. Deimos F1 Congestion.

We could not perform RCM mapping on Deimos because the allocation policy does not guarantee the required "symmetric allocation", where each physical node has exactly the same number of processes (cf. [19]). Thus, Figure 7 shows only greedy results that still perform well.

D. Benchmarks

Now we present benchmark results for sparse unstructured matrix-vector multiplications (100 times) with a *naive* (round-robin) distribution (cf. Figure 3(a)), simple *graph partitioning* (cf. Figures 3(b) and 3(c)), and *topology mapping* of the partitioned graph (cf. Figure 3(d)).

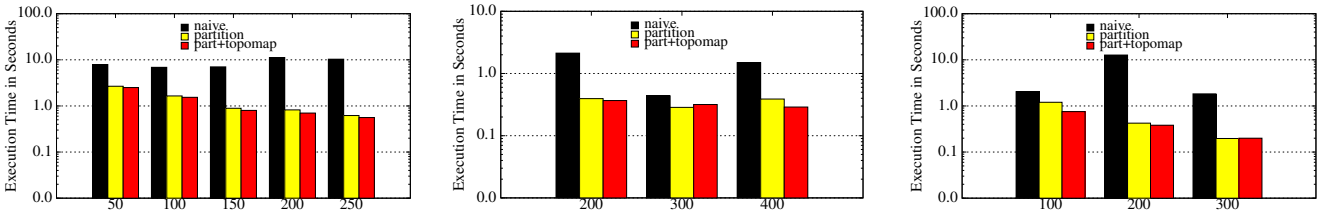
We found that the measured performance benefit of topology mapping on Odin was less than 5% even though the simulation (cf. Figure 6) suggests a 2x reduction in congestion. The link bandwidth on Odin's InfiniBand is nearly twice as high as the injection bandwidth of each node and a two-fold improvement congestion is not measurable. We show an example measurement on Odin in Figure 8(a).

Figures 8(b) and 8(c) show the benchmark results on Deimos which, unlike Odin, has full injection bandwidth. We see improvements of up to 75% from partitioning the input and an additional 26% after applying greedy topology mapping. More matrices are examined in [19].

IV. Discussion and Conclusions

We show how to utilize generic programming techniques to design and implement an efficient and productive linear algebra library using advanced concepts such as graph partitioning and topology mapping for unstructured matrices. Our generic library design using operator overload provides essentially a domain-specific language for linear algebra computations.

We demonstrate that an intuitive easy-to-use interface does not conflict with the goal of achieving highest performance on parallel computers. In the contrary, we show that the full potential of partitioning and advanced topology mapping can be provided "under the hood". Our library follows the guidelines for good MPI library design [36] and completely hides all communication and data-distribution functions from the user. Thus, it enables highest performance portability across a wide variety of architectures and arbitrary network topologies.



(a) Odin ldoor. (b) Deimos Ga41As41H72. (c) Deimos F1.

Fig. 8. Benchmark Results for different numbers of cores. "naive" is a simple round-robin distribution (cf. Figure 3(a)), "partition" is using the migration technique for graph partitioning (cf. Figures 3(b) and 3(c)), and "part+topomap" uses the topology-aware partitioning (cf. Figure 3(d).

Our automated and transparent partitioning and topology mapping results show a significant speedup due to partitioning and load-balancing (up to 75%) as well as due to topology mapping (up to 26%). We showed that LibTopoMap can handle communicators up to 1,024 processes with low overheads in a realistic application run.

Facing larger systems with deeper hierarchies and more heterogeneity, good topology mapping will be paramount. The algorithms will certainly evolve to cope with the increasing complexity of future architectures.

Acknowledgments We thank Sebastian Aland for the CFD example and Thomas Witkowski and Andreas Naumann for the benchmarks with PMTL4 and PETSc.

References

[1] W. D. Gropp and B. Smith. PETSc. Technical report, Argonne National Laboratory, 1994.

[2] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 3 edition, February 2000.

[3] Peter Gottschling et al. PMTL4 web page. http://www.simunova.com/node/142 (accessed Dec. 2011).

[4] Peter Gottschling, David S. Wise, and Adwait Joshi. Generic support of algorithmic and structural recursion for scientific computing. *IJPEDS*, 24(6):479–503, Dec. 2009.

[5] Peter Gottschling et al. Debugging MTL4. https://simunova.zih.tu-dresden.de/mtl4/docs/debugger_support.html (accessed Dec. 2011).

[6] Peter Gottschling and Cornelius Steinhardt. Meta-tuning in MTL4. In *ICNAAM 2010: International Conference of Numerical Analysis and Applied Mathematics*, volume 1281, pages 778–782, September 2010.

[7] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *OOPSLA'93*, pages 91–108. ACM Press, Sep. 1993.

[8] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2*, June 23rd 2009.

[9] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, and Robert van de Geijn. PLAPACK: Parallel linear algebra package. In *SIAM Parallel Processing Conference*, 1997.

[10] M.A. Heroux et al. An overview of the Trilinos project. *TOMS*, 31(3):397–423, 2005.

[11] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[12] Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing '02*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[13] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, 30(3):207–214, 1981.

[14] S.-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Comput.*, 36(4):433–442, 1987.

[15] S. Wayne Bollinger and Scott F. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans. Comput.*, 40(3):325–333, 1991.

[16] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN Europe*, volume 1067 of *LNCS*, pages 493–498. Springer, 1996.

[17] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology mapping for blue gene/l supercomputer. In *SC'06*, page 116, New York, NY, USA, 2006. ACM.

[18] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *ICS '09*, NY, USA, 2009. ACM.

[19] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*, pages 75–85. ACM, Jun. 2011.

[20] Jörg Walter and Mathias Koch. *Boost Basic Linear Algebra*. Boost, 2002. www.boost.org/libs/numeric.

[21] Douglas Gregor and Matthias Troyer. Boost.MPI, November 2006.

[22] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, 2005.

[23] Alex Breuer, Peter Gottschling, Douglas Gregor, and Andrew Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Performance Optimization for High-Level Languages and Libraries (POHLL)*, April 2006.

[24] Christophe Prud'homme. A DSEL in C++ for AD, projection, integration and variational formulations. *Sci. Program.*, 14:81–110, April 2006.

[25] S. Aland, J. Lowengrub, and A. Voigt. Two-phase flow in complex geometries: A diffuse domain approach. *CMES*, 57(1):77–106, 2010.

[26] P. Gottschling, T. Witkowski, and A. Voigt. Integrating object-oriented and generic programming paradigms: Experiences with AMDiS and MTL4. In *POOSC'08, Cyprus*, 2008.

[27] David R. Musser and Alexander A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Proceedings*, volume 358, pages 13–25, Berlin, 1989. Springer.

[28] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1 edition, June 2009.

[29] Rudolf Wille. Conceptual graphs and formal concept analysis. In *Conceptual Structures: Fulfilling Peirce's Dream*, volume 1257 of *LNCS*, pages 290–303. Springer Berlin / Heidelberg, 1997.

[30] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[31] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[32] T.L. Veldhuizen. C++ templates are Turing complete. citeseer.ist.psu.edu/581150.html, 2003.

[33] Robert Ramey. *Boost Serialization Library*. Boost, 2004. http://www.boost.org/libs/release/libs/serialization.

[34] Timothy A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[35] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Cluster Computing'08*. IEEE Computer Society, Oct. 2008.

[36] T. Hoefler and M. Snir. Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. In *Proceedings of EuroMPI 2011*, volume 6960, pages 345–355. Springer, Sep. 2011.