Generic C++ Implementation of High-Performance BFS-RBF-based Mesh Motion Schemes

Peter Gottschling[‡,*], René Heinzl[†], J. Weinhub[†], Nadejda Kirchner[**], Martin Sauer[**], Arno Klomfass[**], Cornelius Steinhardt[‡] and Jörg Wensch[‡]

[*]*SimuNova UG, Helmholtzstr. 10, 01069 Dresden, Germany*
[†]*TU Wien, Inst. for Microelectronics, Gusshausstrasse 27–29, 1040 Wien, Austria*
[**]*Fraunhofer Inst. for High-Speed Dynamics, EMI, Num. Simulation Dep., Eckerstr. 4, 79104 Freiburg, Germany*
[‡]*TU Dresden, Inst. f. wiss. Rechnen, 01062 Dresden, Germany*

Abstract. Multi-Dimensional fluid- and structural dynamics problems are solved by computational methods based on Arbitrary Lagrange Euler (ALE) formulation of the continuum mechanical conservation equations. The paper presents a new modification of the radial basis function (RBF) based mesh motion scheme, which combines the RBF interpolation with the breadth-first search (BFS) algorithm. In this emerging domain, it is still unknown which algorithmic approach is the most suitable. Therefore, we realized our C++ implementation on a high abstraction level enabling broad customization and easy extension for further algorithmic research without sacrificing performance.

Keywords: Generic programming, Mesh Motion Algorithm, Radial Basis Function Interpolation, Breadth-First Search
PACS: 02.70.-c, 02.70.Wz, 07.05.Bx, 07.05.Tp

INTRODUCTION

Computational methods based on Arbitrary Lagrange Euler (ALE) [1] formulation of conservation laws are usually applied for the solution of multi-dimensional fluid- and structural dynamics problems, e.g., [2, 3, 4]. The goal is to adjust the mesh motion to the movement of the boundary under preservation of the overall mesh topology. These mesh update or mesh motion schemes [1, 5, 6] must provide an acceptable mesh quality even under large deformations. An intensively investigated algorithm for mesh motions is the recently proposed radial basis function (RBF)-based mesh motion scheme [7] from the family of interpolation methods, as it can be applied to arbitrary point clouds and as it is conveniently parallelised. This paper presents the C++ generic implementation of an alternative data reduction technique to the standard RBF method using the breadth-first search (BFS) algorithm, i.e. the BFS-RBF-based mesh motion algorithm introduced in [Kirchner et al.]. Our implementation is based on the C++ programming language that adds power and flexibility to its C ancestor. These added benefits do not always come for free: some C++ language constructs produce run-time overhead. Nevertheless the C++ ability to model an orthogonal software design concept allows for achieving high-performance code. The most important concepts are the following: abstraction; separation of concerns; simplicity; freedom of side effects; transparent code optimization. These concepts can be modeled in C++ orthogonally by using a multi-paradigm approach, e.g., object-oriented, functional, generic, and meta-programming. In this contribution we show how orthogonal design principles can be successfully used to achieve a C++ implementation of BFS-RBF-based mesh motion algorithm, which preserves the languages benefits and obtains optimal computational efficiency at the same time.

RBS-RBF-BASED MESH MOTION ALGORITHM

The BFS-RBF-based mesh motion algorithm introduced in [Kirchner et al.] is divided into the following stages. 1) Construction of the nodal graph of the mesh. Each vertex of the graph represents the mesh node. The graph edges are the links between each pair of connected nodes. 2) Application of the breadth-first search (BFS) algorithm [9] that records the groups of vertices separated from the source vertex by equal shortest paths, i.e. the so-called BFS waves. 3) Traverse of the BFS-waves a) from the boundary surface to the inner of the mesh, if the mesh contains no embedded objects; b) traverse of the BFS waves in two opposite directions starting from the inner and the outer mesh boundaries, simultaneously, see [Kirchner et al.]. In this approach, each pair of neighbor BFS waves contains a reduced set of

CP1281, *ICNAAM, Numerical Analysis and Applied Mathematics, International Conference 2010*
edited by T. E. Simos, G. Psihoyios, and Ch. Tsitouras

constrains and also a reduced set of the mesh nodes, where the velocity must be interpolated. For each reduced point configuration the interpolation matrix is constructed and the linear equation system is solved. The BFS-RBF algorithm represents the front marching technique, where the iteration over the BFS waves allows sequential propagation of the velocity field from the boundary surface to the inner of the mesh.

IMPLEMENTATION OF BFS-RBF BASED MESH MOTION ALGORITHM

Conjugate gradient solver with incomplete LU factorization

In the first test of the BFS-RBF algorithm described in [Kirchner et al.] the linear system was solved using the conjugate gradient solver with incomplete low up factorization provided by the Matrix Template Library 4 (MTL4) [Gottschling and Lumsdaine]. MTL4 is a generic library for linear algebra operations on matrices and vectors. It supports BLAS-like operations with an intuitive interface but its main focus is sparse matrix and vector arithmetic for simulation software. The library contains advanced meta-programming mechanisms to transform the programs during compilation into an optimal executables. These techniques enable good performance without the need for exhaustive code replication. In addition to the performance demands, MTL4 hides all this code complexity from the user who writes his/her applications in a natural mathematical notation. **Conjugate Gradient:** A prototypical example for the intuitive syntax of the library is the entirely abstract implementation of the conjugate gradient method (CG) [11]. The program source shown below demonstrates the purely mathematical programming style on the user side.

```
template < typename LinearOperator, typename HilbertSpaceX,
           typename HilbertSpaceB, typename Preconditioner, typename Iteration >
int cg(const LinearOperator& A, HilbertSpaceX& x, const HilbertSpaceB& b,
       const Preconditioner& M, Iteration& iter)
{
    typedef typename mtl::Collection<Vector>::value_type Scalar;
    Scalar rho(0), rho_1(0), alpha(0), beta(0);
    HilbertSpaceX p(size(x)), q(size(x)), r(b − A*x), z(size(x));
    while (! iter.finished(r)) {
        z = solve(M, r); rho = dot(r, z);
        if (iter.first())
            p = z;
        else {
            beta = rho / rho_1; p = z + beta * p;
        }
        q = A * p; alpha = rho / dot(p, q);
        x += alpha * p; r −= alpha * q;
        rho_1 = rho; ++iter;
    }
    return iter;
}
```

The implementation is identical with algorithmic descriptions in text books. On the other hand, many source code optimizations happen transparently during compilation. For instance, naïve implementations of vector operations use and return temporary vectors. That causes significant overhead due to repeated memory traversal and dynamic allocations. The statement $p = z + beta * p$; requires in the naïve manner three loops and two temporary vectors while our implementation performs the statement in one single loop without temporaries using expression templates. The genericity of the function allows for arbitrary matrix and vector types (with the according interface respectively). The method can be called as well with dense, as with tri-diagonal or skyline matrices. Thus, the data formats can benefit from domain knowledge without affecting the semantics of the CG implementation. Certainly, the matrix can be stored in a symmetric layout since the algorithm requires symmetry in any event. The generic implementation does not restrict how single operations—like matrix vector product—are performed as long as they provide the corresponding semantics. This generality even includes parallel computation on distributed data. The CG method, as shown before, works without modifying any single line on distributed data as long as all operations are realized accordingly on the distributed objects. As preconditioner we used incomplete LU factorization (ILU).

Implementation of BFS-RBF interpolation scheme

In the following, the implementation of the BFS-RBF-based interpolation scheme is explained. We applied the concept-bounded (static) polymorphism to achieve a better performance and maximal software reuse. The objects CELL, FACE and NODE were designed in a manner similar to [12]. All vector properties of the mentioned objects like position, velocity, list of the grid entities adjacent to the selected object like cell etc. were implemented using the MTL4 [13] dense vectors of the fixed size N (e.g. N=3 in 3D space). All objects representing the grid entities were stored in the intrusive unordered_set container [Krzikalla and Gaztañaga]. The boundary node velocities (i.e. all constraints requiring the interpolation) implemented as std::hash_map<std::size_t, position_type> and the user-defined spatial coordinates of the BFS source vertex position represent the input parameters of the BFS-RBF algorithm, which starts with the construction of the (undirected) nodal graph of the mesh using the Boost Graph Library (BGL) [9]:

```
adjacency_list<vecS, vecS, undirectedS, property< vertex_index2_t, typename NODE::counter_t> > g;
```

The user-defined indexation of each grid node is related to the internal numbering system of the graph. Three public methods of the class template VELOCITY_FIELD_INIT(record_BFS_waves, sort_waves and perform_interpolation) are responsible for data manipulation. The following source code shows how to record the BFS waves within the function record_BFS_waves:

```
1  typedef typename graph_traits<Graph>::vertex_descriptor VERTEX;
2  typedef typename graph_traits<Graph>::vertices_size_type SIZE;
3  SIZE dist[num_vertices(g)];
4
5  fill_n(dist,num_vertices(g), 0);
6  VERTEX source_vertex= vertex(internal_idx_source, g);
7  breadth_first_search(g, source_vertex, visitor(make_bfs_visitor(record_distances(dist, on_tree_edge()))));
8  num_waves= dist[0];
9  for (uint i= 1; i < num_vertices(g); ++i)
10     if (dist[i]>num_waves)
11         num_waves=dist[i];
12 waves.resize(num_waves+1);
13 for (uint i= 0; i < num_vertices(g); ++i)
14     waves[ dist[i] ].push_back( index[vertex(i,g)] );
```

As soon as the distances from the user-defined source vertex (line 6) to all mesh nodes are recorded (lines 7–8) and the total number of the BFS waves is known (lines 9–11), the sequences of the node indices are inserted to the corresponding wave (lines 12–14). After spliting the node indices of each BFS wave into two groups (to constraints and approximation points, respectively), the iterative BFS-RBF interpolation begins. At each iteration step, we call the dynamic linked library SCATTERED_DATA_INTERPOLATION with the following implementation:

```
void SCATTERED_DATA_INTERPOLATION( const size_t& modus,
      const vector<Node::counter_t>& constraints, const vector<Node::counter_t>& X,
      nodes_type& nodes, const size_t& num_inner_nodes )
{ ...
   intrusive_ptr< RBF_INTERPOLATION<Node> > rbf_interpolation =
      new RBF_INTERPOLATION<Node>( constraints, X, nodes, modus, num_inner_nodes
);
   rbf_interpolation → solve_system_of_equations<0>(); // x
   rbf_interpolation → solve_system_of_equations<1>(); // y
}
```

In the code above, modus characterizes the way of the scaling factor calculation. The vector constraints and X contain indices of the data sites and the points, where the RBF approximation must be found, respectively. The properties of all grid nodes are stored in the container nodes. The object rbf_interpolation performs the standard RBF interpolation over the reduced point configuration extracted from the mesh by BFS. To calculate the interpolant for each spatial dimension, the method solve_system_of_equations of the class template RBF_INTERPOLATION is applied:

```
template<const size_t Dimension>
void solve_system_of_equations()
{
```

```
    typedef typename NODE::value_t value_t;
    fill_rhs<dense_vector,NODE, Dimension>(b, data ); ...
    cg(A, lambda, b, Precond, iter); ...
    typename NODE::coords_t s(0);
    for (size_t i= 0; i < num_x; ++i) {
        for (size_t j=0; j < num_data; ++j)
                b[j]= RBF::CompactSupport::CP_C2<value_t >(Dist[i][j], radius_of_support);
        s = x.at(i)→ get_velocity();
        s[Dimension] = dot(lambda,b);
        x.at(i)→ set_velocity(s); };
};
```

The linear equation system above determines the approximation coefficients of the standard RBF scheme. After that, vector b is updated with Wendland's C2 compactly supported RBF [15]. The velocity field of the inner mesh becomes initialized and the time development of the mesh can easily be evaluated.

RESULTS AND CONCLUSIONS

The computational efficiency of the BFS-RBF algorithm was compared with the standard RBF scheme studied in [7], see [Kirchner et al.]. There we indicated that the BFS-RBF algorithm in our implementation is a factor of 30 faster. Comparison of the elapsed time for the single run of BFS-RBF algorithm with other mesh motion schemes like [6, 5] are currently difficult due to the lack of information in these. For instance [6] presents no measured absolute computation times. Despite the already achieved good performance we still see potential for acceleration (e.g. implicit storage schemes). Ultimately, little is known about the application of distinct solvers and their influence on the algorithm stability and performance. Optimization of the overall BFS-RBF computational performance as well as the comparative study with other mesh motion schemes will be addressed in the next study. For these new studies, the generic software design based on MTL4 and BGL is paramount to realize new algorithms with minimal implementation changes or extentions.

REFERENCES

1. J. Donea, A. Huerta, J.-P. Pontot, and R. Ferran, "Arbitrary Lagrangian Eulerian Methods," in *Encyclopedia of Computational mechanics*, John Wiley & Sons, 2004.
2. M. Stiemer, J. Unger, B. Svendsen, and H. Blum, *Comput. Methods Appl. Mech* pp. 1535–1547 (2009).
3. H. Bungartz, and M. Schäfer, *Fluid-structure interaction: modelling, simulation, optimisation*, Springer, 2006.
4. M. Kucharik, R. Garimella, S. Schofield, and M. Shashkov, *Journal of Computational Physics* (2009).
5. R. Löhner, and C. Yang, *Communications in numerical methods in engineering* pp. 599–608 (1996), vol.12.
6. T. Rendall, and C. Allen, *Journal of Computational Physics* **228**, 6231–6249 (2009).
7. A. de Boer, M. van der Schoot, and H. Bijl, *Computers and structures* **85**, 784–795 (2007).

Kirchner et al.. N. Kirchner, M. Sauer, A. Klomfass, P. Gottschling, and R. Heinzl, "A new mesh deformation algorithm for the ALE technique," in *Proc. of 1st Int. Conf. Multiphysics Simulation*

9. J. Siek, L. Lee, and A. Lumsdaine, *The boost graph library*, Addison-Wesley, 2002.

Gottschling and Lumsdaine. P. Gottschling, and A. Lumsdaine, The matrix template library 4, `http://www.osl.iu.edu/research/mtl/mtl4/`.

11. M. Križek, P. Neittaanmäki, R. Glowinski, and S. Korotov p. 93 (2004), series: Scientific Computation, XV, 382.
12. N. Kirchner, O. Herzog, S. Knell, V. Holzwarth, U. Ziegenhagel, M. Sauer, and A. Klomfass, "C++ code design for multi-purpose explicit finite volume methods: requirement and solutions," in *Proc. 8th POOSC*, Genova, Italy, 2009.
13. P. Gottschling, and D. Lindbo, "Generic compressed sparse matrix insertion: algorithms and implementations in MTL4 and FEniCS," in *Proc. 8th POOSC*, ACM, New York, NY, USA, 2009, pp. 1–8, ISBN 978-1-60558-547-5.

Krzikalla and Gaztañaga. O. Krzikalla, and I. Gaztañaga, Boost intrusive, `http://www.boost.org/doc/libs/1_38_0/doc/html/intrusive.html`.

15. R. Schaback, "Kernel Based Meshless Methods," 2007, pp. 6231–6249, `http://num.math.uni-goettingen.de/schaback/teaching/07SS/kernel.pdf`.