Parallelization of an Adaptive Mesh Refinement Method for Low Mach Number Combustion*

Charles A. Rendleman
Vincent E. Beckner
Mike Lijewski

Lawrence Berkeley National Laboratory
Berkeley, CA 94720

January 30, 2001

Abstract

We describe the parallelization of a computer program for the adaptive mesh refinement simulation of variable density, viscous, incompressible fluid flows for low Mach number combustion. The adaptive methodology is based on the use of local grids superimposed on a coarse grid to achieve sufficient resolution in the solution. The key elements of the approach to parallelization are a dynamic load-balancing technique to distribute work to processors and a software methodology for managing data distribution and communications. The methodology is based on a message-passing model that exploits the coarse-grained parallelism inherent in the algorithms. A method is presented for parallelizing weakly sequential loops—loops with sparse dependencies among iterations.

1 Introduction

Advanced, higher-order finite difference methods and local adaptive mesh refinement have proven to be an effective combination of tools for modeling problems in fluid dynamics. However, the dynamic nature of the adaptivity in time dependent simulations makes it considerably more difficult to implement this type of methodology on modern parallel computers, particularly, distributed memory architectures. In this paper we present the parallelization of a computer program using a software framework that facilitates the development of adaptive algorithms for multiple-instruction, multiple-data (MIMD) architectures. The particular form of adaptivity we consider is a block-structured style of refinement, referred to as AMR, that was originally developed by Berger and

*This work was carried out at the Lawrence Berkeley National Laboratory under the auspices of the US Department of Energy Contract No. DE-AC03-76SF00098. Support was provided by the Defense Threat Reduction Agency under subcontract to Lawrence Livermore National Laboratory and by the Applied Mathematics Program in the DOE Office of Science.

Oliger [5]. The methodology uses the approach developed by Berger and Colella [4] for general systems of conservation laws, its extension to three dimensions by Bell et al. [2], and incompressible flows by Almgren et al. [1]. Subsequently, this work was extended to the simulation of low Mach number combustions by Pember et al. [13] and by Day and Bell [10]. This paper discusses the AMR parallel implementation of the algorithm described by Day and Bell [10]

AMR is based on a sequence of nested grids with finer and finer mesh spacing in space, each level being advanced in time with time step intervals determined by the Courant-Friedrich-Level (CFL) condition. The fine grids are recursively embedded in coarser grids until the solution is sufficiently resolved. An error estimation procedure automatically determines the accuracy of the solution and grid management procedures dynamically create rectangular fine grids where required to maintain accuracy or remove rectangular fine grids that are no longer required for accuracy. Special difference equations are used at the interface between coarse and fine grids to insure conservation.

In this paper we describe the application of the framework to the parallelization of an AMR numerical solution of low Mach number reacting flows with complex chemistry. developed by Day and Bell [10] extending work on the modeling of incompressible fluid flows by Almgren et al.[1]. Rendleman et al. [15] have described a general framework for the implementation of parallel AMR algorithms, and demonstrated its application to the parallelization of AMR to hyperbolic conservation laws. That framework was developed based on experience gained from researchers in our group and others [8, 9, 3, 7, 11]. In that approach, data distribution and communication are hidden in C++ class libraries that isolate the application developer from the details of the parallel implementation.

In the next section we will briefly review the basic algorithmic structure of AMR with emphasis on the particular case of incompressible fluid flow. The dynamic character of AMR leads to a dynamic and heterogeneous work load. In section 3 we discuss the basic parallelization strategy and the load-balancing techniques we use for AMR algorithms. Section 4 provides a description of the parallel implementation (described more fully elsewhere [15]), focusing primarily on the additional methods used to parallelize algorithms specific to AMR for incompressible flows. Finally, we illustrate the use of the program with a simulation of a three dimensional premixed combustion flow.

2 The Adaptive Mesh Refinement Algorithm

AMR solves partial differential equations using a hierarchy of grids of differing resolution. The grid hierarchy is composed of different levels of refinement ranging from coarsest ($l = 0$) to finest ($l = l_{max}$). Each level is represented as the union of non-intersecting logically rectangular grid patches of a given resolution. In this work, we assume the level 0 grid is a grid patch decomposition of a single rectangular parallelepiped, the *problem domain*. In this implementation, the refinement ratio is always even i.e., $\Delta x^{l+1} = \Delta x^l / r$, where r is the refinement ratio, which in the implementation, can be a function of level. The grids are *properly nested*, in the sense that the union of grids at level $l + 1$ is contained in the union of grids at level l for $0 \leq l < l_{max}$, and the level l grids are large enough to guarantee that there is a border at least one level l cell wide surrounding each level $l + 1$ grid. Proper nesting does not require that a fine sub-grid not cross a

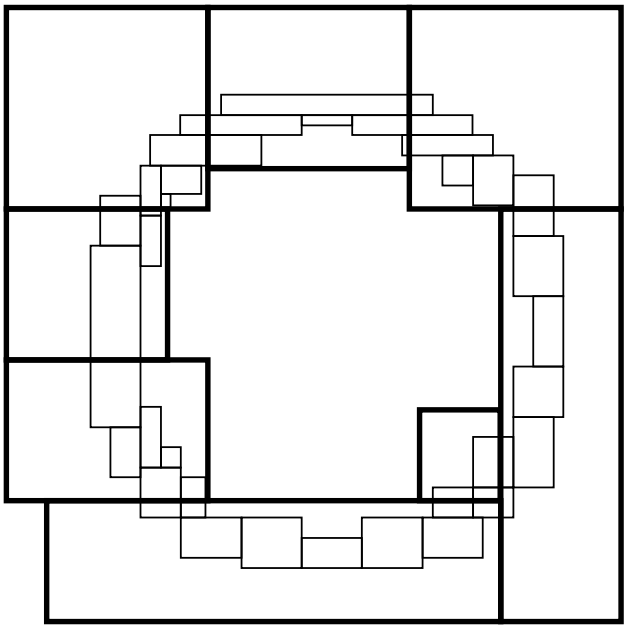

Figure 1: Two levels of refined grids. Grids are properly nested, but may have more that one parent grid. The thick lines represent grids at the coarse level; the thin lines, grids at the fine level.

coarser sub-grid boundary. Grids at all levels are allowed to extend to the physical boundaries so the proper nesting is not strict there. This is illustrated in Figure 1 in two dimensions. (This set of grids was created for a problem with initial conditions specifying a circular discontinuity.)

Both the initial creation of the grid hierarchy and the subsequent regriding operations in which the grids are dynamically changed to reflect changing flow conditions use the same procedures to create new grids. Cells requiring additional refinement are identified and tagged using an error estimation criteria, possibly using Richardson extrapolation [4]. The tagged cells are clustered [6] into rectangular patches, which in general contain cells that were not tagged for refinement; typically 70% of the cells in a new grid have been tagged by the error estimation process. When new grids are created at level $l + 1$, the data on these new grids are copied from the previous grids at level $l + 1$ where possible, otherwise the data is interpolated in space from the underlying level l grids.

The AMR algorithmis a recursive procedure that advances each level l, $0 \leq l \leq l_{max}$, with a time-step appropriate to that level, based on CFL considerations. The adaptive algorithm advances the grids at each level independent of other levels in the hierarchy except for obtaining boundary data and the synchronization between levels. The coarser grids supply boundary data in order to integrate finer grids, by filling *ghost cells* in a band around the fine grid data whose width is determined by the stencil of the finite difference scheme. If this data is available from grids at the same level of refinement the data is provided by a simple copy, otherwise it is obtained by interpolation of coarser grid data in time and space. When the coarse and fine grids have been advanced to the same time and we synchronize, there are three corrections that we need to make. First, we replace the coarse data by the volume weighted average of covering fine grid data. Second, we must correct the coarse cell values by adding the difference between the coarse and fine grid fluxes used to advance grids at their respective levels. Third, we also impose the divergence

constraint to the velocity field over the composite grid system. See Almgren et al. [1] for more details on the synchronizations of data between levels.

3 Parallelization of AMR

We have adopted a coarse-grained, message-passing model, using MPI [17, 12], in our approach parallelization. This approach is generally associated with distributed memory MIMD architectures, but it can be used on shared memory architectures as well. We make this choice because it enhances portability on distributed memory architectures, and because we feel message-passing programs are more robust. In message-passing parallel programs, the only communication between processors is through the exchange of messages. Direct access to another processor's memory is not provided. In this approach, it is critical to choose carefully which processor has which piece of data. As is apparent from Figure 1, grids vary considerably in size and shape. In AMR the number of grids also changes and is seldom an integer multiple of the number of processors. It is therefore inefficient to assign the grids sequentially to the processors, since the result is unlikely to be load balanced. We will use a load-balancing strategy based on the approach developed by Crutchfield [8, 15]. Because most of the data and computational effort is required by the finest level of grids, we need only be concerned with load-balancing the grids on the finest level. In general, the effort required by the coarser grids will be a minor perturbation.

We accept the set of grids provided by the regriding algorithm and seek to find a well-balanced assignment of grids to processors. It turns out to be possible to find well-balanced assignments if we can make the following assumptions.

1. The computational cost of a grid can be estimated using some type of work estimate.

2. The total computational cost of the algorithm is well approximated as the sum of the costs of time-stepping the grids on the finest level. Other costs such as communications, time-stepping coarser grids, regriding, refluxing, etc., are treated as ignorable perturbations.

3. The grids can be approximated as having a broad random distribution, i.e., that the standard deviation of the distribution is not small compared to the average.

4. The average number of grids per processor is at least three.

We have developed an algorithm [8, 15] based on an application of the well-known knapsack dynamic programming algorithm, a description of which may be found in Sedgewick's book on algorithms [16]. This approach finds the distribution of grid blocks among processors that results in the smallest total computation time.

While the cost of communication is generally important in message-passing parallel programs—and effort is devoted to its reduction—such costs are ignored in this load balance scheme. No effort is made to reduce communication costs by placing adjacent grids on the same processor, or on adjacent processors. Since the ratio of communication cost to calculation cost for modern multiprocessors is not overly large, it is reasonable to ignore communication costs in the load balance only if significant computation is done relative to communications. In general, this assumption holds for the application described in this paper because of the amount of work in the chemical reactions calculation.

4 Implementation

The methodology described in this paper has been embodied in a software system that allows for a broad range of physics applications. It is implemented in a hybrid C++/FORTRAN programming environment where memory management and control flow are expressed in the C++ portions of the program and the numerically intensive portions of the computation are handled in FORTRAN. The software is written using a layered approach, with a foundation library, BoxLib, that is responsible for the basic algorithm domain abstractions at the bottom, and a framework library, AMRLib, that marshals the components of the AMR algorithm, at the top. Support libraries built on BoxLib are used as necessary to implement application components such as interpolation of data between levels, the coarse-fine interface synchronization routines, and linear solvers used in the projection.

The fundamental parallel abstraction is the MultiFab, which encapsulates the FORTRAN compatible data defined on unions of Boxs; a MultiFab can used as if it were an array of FORTRAN compatible grids. The grids that make up the MultiFab are distributed among the processors, with the implementation assigning grids to processors using the distribution given by the load balance scheme described in section 3. Non-MultiFab operations and data-structures are replicated across all of the processors. This non-parallel work is usually measured to be small. Because each processor possesses the global data layout, the processor can post data send and receive request without a prior query for data size and location.

MultiFab operations are performed in one of three ways depending on the implicit communications pattern. In the simplest case, there is no interprocessor communication; the calculation is parallelized trivially with an *owner computes* rule with each processor operating independently on its local data. This is the case with chemistry state evaluations. Different parallel constructs are necessary when data communication involves more than one MultiFab, an example of which is the *fill patch* operation, which interpolates from coarse cell data on to overlying fine grid patches. Such constructs cannot be implemented by simply nesting loops because outer loop bodies for subgrids that are off-processor will not be executed. They must be implemented by our second method using two stages: data is exchanged between processors and then the local targets are updated.

The third, more complicated case, arises in parallelizing loops in the multi-level projection method. The difficulties arise from several causes. First, the original projection method was implemented using libraries that differ somewhat from the BoxLib/AMRLib libraries, though they share a number of features. For example, the projection uses fill patch operations with different treatments of physical and interior boundary conditions. The major difficulty, however, is a result of the special requirements of the adaptive projection itself. The formulation of the projection algorithm requires that loops be applied in a specific order with possible coupling from loop body to loop body. For example, stencils are evaluated by looping over faces, edges and then corners of grids in the associated MultiFab, where results of earlier iterations of the loops may affect the results of subsequent iterates. In addition, the boundary patch filling operation copies in stages, with data from initial stages contributing to data at later stages. The same output patch in an operation may be repeatedly updated in an order specific fashion, and a source patch may require to be updated before being used by an output patch. These order dependencies in operator evaluation give rise to what may be called *weakly sequential loops*.

Weakly sequential loops can be characterized using the language of graph theory. Loop bodies correspond to nodes in the graph and dependencies in the iterations of the loop body correspond

```
WeakSeqCopy(MultiFab& mf_to, const MultiFab& mf_from)
{
  task_list tl;
  for(int i = 0; i < mf_to.length(); ++i)
  {
    for(int j = 0; j < mf_from.length(); ++j)
    {
      tl.add_task(new task_copy(tl, mf_to, i, mf_from, j));
    }
  }
  tl.execute();
}
```

Figure 2: A weakly sequential implementation of a fill boundary operation

to directed edges in the graph. The nodes in the graph can have more than one prior node, and can themselves be the prior nodes of more than one subsequent node. Analogously, weakly sequential loops are similar to the model used in the standard Unix utility, *make*, that manages compilation dependencies. Here a node, usually representing a file, is said to depend on other nodes (files), and can itself be a dependency of other files. The method we use to evaluate weakly dependent loops is again analogous to way the *make* program operates: the *make* program traverses the dependency graph defined in the `Makefile` in such a way that a file is processed by a rule if its dependencies are up to date.

We will use the example in Figure 2 to illustrate the use of weakly sequential loops. The `add_task` method of the `task_list`, is responsible for inserting a task into the task loop and evaluating dependencies of the current task on prior tasks in the loop. The `task_copy` is a helper-class that implements copying of data from sub-grid j of the `MultiFab` mf_from to sub-grid i of mf_to. The `task_list` is passed to the constructor of the `task_fill` operation so that dependencies can be detected among loop iterations. This style of coding has the effect of flattening multiply nested parallel loops into a single serial loop that is processed as indicated below.

The `execute` member of `task_list` used in Figure 2 causes each task to be executed in `task_list` using the algorithm in Figure 3. The loop attempts to maximize concurrency by using asynchronous message passing calls. Potentially, many messages requests can be outstanding, though in practice MPI implementations restrict the number of outstanding posted messages. For that reason, the loop is "throttled" by limiting the number of active members of the task loop. When a task is marked as finished, it is cleared from the dependency list of all remaining tasks in the loop.

Naive implementation of the construction of the dependency graph results in an operation count of $O(N^2)$, where N is the number of loop elements, usually proportional to the number of grids at a level. This could be significant for the case when there are thousands of elements in the loop, a situation that is not uncommon. However, careful implementation reduces computation cost to $O(N/P)^2$, which exhibits lower growth because the number of processors, P, used in a calculation is an increasing function of the number of grids. The careful implementation removes tasks from

```
Procedure task_list::execute()
    while task list not empty
        Pop head of task list into task T.
        if T has no outstanding dependencies
            if T has not been started, Start T, post message passing requests
            if T is message complete, Execute T and mark as finished;
        else
            Push T at end of task list
        endif
    end while
End Procedure task_list::execute
```

Figure 3: Algorithm for `task_list::execute`

the task loop as they are added if the task does not use data local to that processor, or if it uses data only local to that processor and the task does not depend on prior tasks in the task list.

The principle disadvantage of the task list approach is that it encourages an unnatural coding style: a helper class must be implemented for each loop in the program. For the projection, which consists of approximately 13,000 lines of C++ (and 16,000 lines of FORTRAN), less than a dozen helper classes are needed.

5 Numerical Example

The example we choose to illustrate the use of the program represents a simplified model of spray fueling in an internal combustion engine. A spray of premixed fuel droplets enters a turbulent combustion chamber, where it is heated, evaporated and burned. Here, the fuel-rich spray is assumed to be heated to 1000 K by other processes, and the calculation evolves the combustion of fuel in two simultaneous combustion modes: a fast premixed burn of the heated fuel, and a slower diffusion flame at the interface between the air and remaining fuel. In this model, a 1 cm radius sphere of 1000 K hydrogen-air mixture (equivalence ratio = 4) is used to represent the fuel. A domain of 10x10x10 cm, co-centered with the initial fuel sphere, is initialized with room temperature isotropically turbulent air. All the boundaries of the domain are outflowing. In the initial stages, the fuel sphere expands from the temperature rise due to the fast premixed burn. The expansion, and resulting interaction with the background turbulence, generates surface instabilities at the interface between the fuel and air, increasing the interface area, reactant mixing, and overall consumption rate of the fuel. 9 chemistry species (H_2, H, O, O_2, OH, H_2O, HO_2, H_2O_2, and N_2) with 27 reactions among them are used to model the combustion process.

The simulation was performed on the IBM Power3 SMP system at the U.S. Army Engineer Research and Development Center, Major Shared Resource Center in Vicksburg, MS. The AMR parameters used specified 2 levels of refinement with a refinement ratio of 2. The concentration of species H_2O_2 is used to mark the zones in the model that require refinement. The coarse grid consisted of 32 zones in each coordinate direction. With these refinement ratios the finest level has

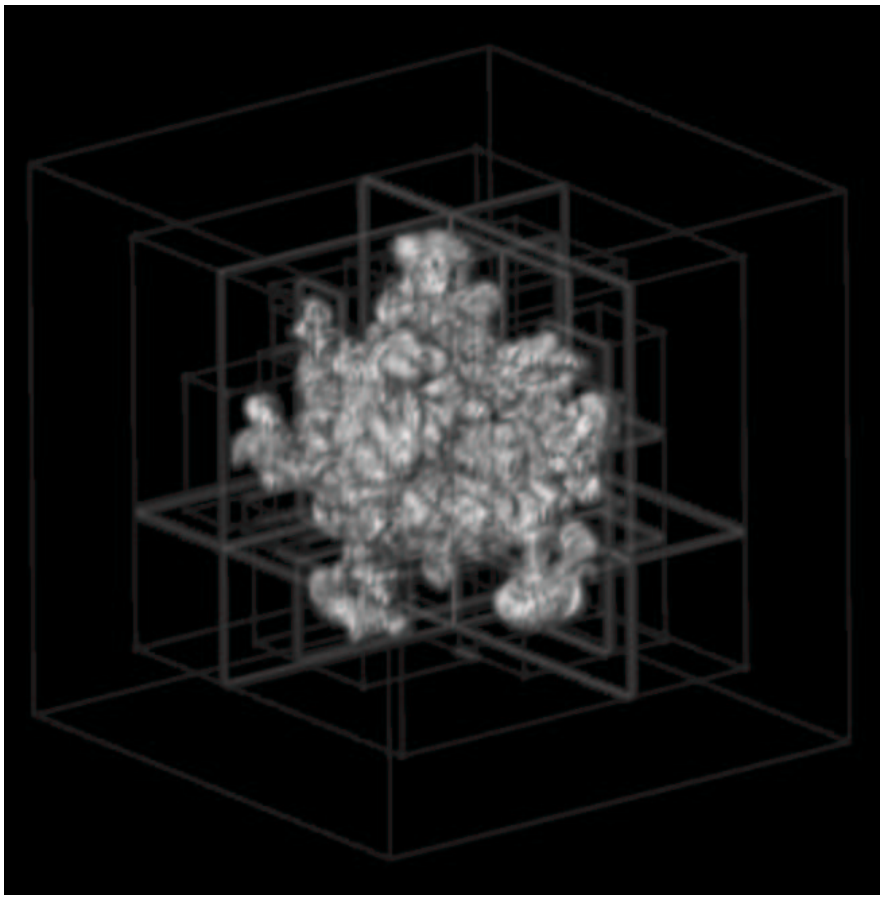

Figure 4: High temperature at fuel-air interface rendered at time step 380. Solid lines indicate refined zones at the finest two resolutions. The coarse grids is not shown.

an effective resolution of 128 zones in each coordinate direction. For this relatively small model, only 8 processors were used, in accord with our load-balance guidelines of assigning 3 or more grids per processor (section 3). Figure 4 show a rendering of the temperature at time step 380. At this point in the simulation, there are 34 grids at the finest level, covering approximately 18% of the domain. Approximately 30% of the computational time is consumed in evaluating the chemical reaction processes. The remainder of the time is roughly allocated to 5% for scalar advection, 20% for the velocity projection, and 30% for diffusion of the chemical species. The remainder of the time is charged to overhead associated with the adaptive algorithm and with the technique, not described here, used to load balance the chemistry [14].

6 Conclusions

We have described the techniques used to parallelize an AMR variable density, viscous, incompressible flow solver targeted to low-Mach number reacting flows. One of the applications of this program, together with its companion program for hyperbolic systems of conservation laws described in Rendleman et al. [15], is to provide end-to-end simulation capabilities for explosions in

buried chamber systems. The hyperbolic code is used to examine the prompt effects of the explosion, while the low-Mach number code is used to monitor longer time scale processes associated with burning of the chamber system's contents. The methods used are rectangular sub-grid decomposition of data among parallel processors and a use of SPMD style programming constructs. Load balance is achieved using an efficient and effective dynamic-programming algorithm. We also described our software methodology including a novel technique for identifying and evaluating weakly sequential loops. We demonstrated the use of the program for an example of pre-mixed fuel air combustion in a room temperature isotropically turbulent body of air.

Acknowledgements

The authors wish to thank J. B. Bell and W. Y. Crutchfield for valuable input. We also wish to thank M. S. Day for providing the numerical example used in section 5.

References

[1] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *J. Comput. Phys.*, 142:1–46, May 1998.

[2] J. Bell, M. Berger, J. Saltzman, and M. Welcome. A three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Statist. Comput.*, 15(1):127–138, 1994.

[3] M. Berger and J. Saltzman. AMR on the CM-2. *Applied Numerical Mathematics*, 14:239–253, 1994.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[5] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, March 1984.

[6] M. J. Berger and J. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1278–1286, 1991.

[7] P. Colella and W. Y. Crutchfield. A parallel adaptive mesh refinement algorithm on the C-90. In *Proceedings of the Energy Research Power Users Symposium*, July 11–12, 1994. `http://www.nersc.gov/aboutnersc/ERSUG/meeting_info/ERPUS/colella.ps`.

[8] W. Y. Crutchfield. Load balancing irregular algorithms. Technical Report UCRL-JC-107679, Lawrence Livermore National Laboratory, July 1991.

[9] W. Y. Crutchfield. Parallel adaptive mesh refinement: An example of parallel data encapsulation. Technical Report UCRL-JC-107680, Lawrence Livermore National Laboratory, July 1991.

[10] M. S. Day and J. B. Bell. Numerical simulation of laminar reacting flows with complex chemistry. *Combust. Theory Modelling*, 4:535–556, 2000.

[11] J. A. Greenough, W. Y. Crutchfield, and C. A. Rendleman. Numerical simulation of a wave guide mixing layer on a Cray C-90. In *Proceedings of the Twenty-sixth AIAA Fluid Dynamics Conference*. AIAA-95-2174, June 1995.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. The MIT Press, Cambridge, Mass, 1994.

[13] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. An adaptive projection method for unsteady low-Mach number combustion. *Comb. Sci. Tech.*, 140:123–168, 1998.

[14] Charles A. Rendleman, Vincent E. Beckner, and Marc S. Day Mike Lijewski. Parallel performance of an adaptive mesh refinement method for low mach number combustion. in preparation, January 2001.

[15] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Y. Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3(3):147–157, 2000.

[16] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[17] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation. The Mit Press, Cambridge, Mass, 1996.