SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S156–S172

BOXLIB WITH TILING: AN ADAPTIVE MESH REFINEMENT SOFTWARE FRAMEWORK*

WEIQUN ZHANG†, ANN ALMGREN†, MARCUS DAY†, TAN NGUYEN‡, JOHN SHALF§, AND DIDEM UNAT¶

Abstract. In this paper we introduce a block-structured adaptive mesh refinement software framework that incorporates tiling, a well-known loop transformation. Because the multiscale, multiphysics codes built in BoxLib are designed to solve complex systems at high resolution, performance on current and next generation architectures is essential. With the expectation of many more cores per node on next generation architectures, the ability to effectively utilize threads within a node is essential, and the current model for parallelization will not be sufficient. We describe a new version of BoxLib in which the tiling constructs are embedded so that BoxLib-based applications can easily realize expected performance gains without extra effort on the part of the application developer. We also discuss a path forward to enable future versions of BoxLib to take advantage of NUMA-aware optimizations using the TiDA portable library.

Key words. high-performance computing, software framework, tiling

AMS subject classification. 97N80

DOI. 10.1137/15M102616X

1. Introduction.

BoxLib is a mature, publicly available software framework for building massively parallel block-structured adaptive mesh refinement (AMR) applications [1]. It is one of a number of current publicly available AMR frameworks; see [16] for an overview. Numerous application codes are based on these frameworks and are too numerous to list here, but sample research codes in use today that are based on BoxLib include MAESTRO [5, 6, 4, 36, 23] and CASTRO [3, 34, 35] for astrophysical simulations, Nyx [2, 21] for cosmological simulations, LMC [13, 9, 8] for low Mach number combustion simulations, and SMC [17] for compressible combustion simulations, as well as codes for moist atmospheric physics [14, 15] and subsurface flow [24].

BoxLib contains extensive software support for explicit and implicit grid-based operations as well as particle-mesh operation on adaptive hierarchical meshes. Multilevel multigrid solvers are included for cell-based and node-based data. Multiple

*Received by the editors June 16, 2015; accepted for publication (in revised form) April 7, 2016; published electronically October 27, 2016. This work was supported by the SciDAC Program and the Exascale Co-Design Program of the DOE Office of Advanced Scientific Computing Research under U.S. Department of Energy contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

http://www.siam.org/journals/sisc/38-5/M102616.html

†Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (WeiqunZhang@lbl.gov, ASAlmgren@lbl.gov, MSDay@lbl.gov, https://ccse.lbl.gov).

‡Computer Architecture Group, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (tannguyen@lbl.gov, crd.lbl.gov/departments/computer-science/computer-architecture/staff/tan-thanh-nhat-nguyen).

§Computer Science Department, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (JShalf@lbl.gov, crd.lbl.gov/departments/computer-science/).

¶Computer Science and Engineering Department, Koç University, Rumelifeneri Yolu, Sariyer, Istanbul, 34450, Turkey (dunat@ku.edu.tr, http://parcorelab.ku.edu.tr). The research of this author was supported by the Marie Sklodowska Curie Reintegration Grant by the European Commission and by Tubitak grant 215E285.

time-subcyling modes are supported for adaptive mesh simulations. Most current BoxLib application codes evolve multiscale, multiphysics systems in time by solving systems of partial differential equations often accompanied by constraints. Because these codes are designed to solve complex systems at high resolution, performance on current and next generation architectures is always of paramount importance.

BoxLib uses a hybrid MPI/OpenMP approach for parallelization, and the OpenMP parallelism has traditionally been expressed in the individual loops over cells. BoxLib application codes have demonstrated good scaling behavior on up to 100,000 cores on current multicore architectures (see, e.g., [3, 22]). However, with the current trends in system design, the next generation of high-performance computing systems will have node architectures based on many-core processors and nonuniform memory access (NUMA) designs. Due to the irregularity of AMR algorithms, a fine-grained loop-level threading approach is not expected to provide efficient parallelism on a node with hundreds or thousands of threads. In addition, reducing the cost of data movement is expected to be crucial for performance [31]. In response to these architectural challenges, we are adopting a new programming model, *tiling*, in BoxLib to expose additional parallelism and optimize data access behavior.

Tiling is a well-known loop transformation that has been proven to be useful in enhancing data locality and parallelism (e.g., [33, 25]). Tiling reduces the working set size so that the working set of a thread can fit into cache, reducing the number of cache misses. This decreases the memory traffic, resulting in improved performance. Tiling can also expose more parallelism by multidimensionally partitioning data or iteration space. Previously we added tiling to the BoxLib based SMC code [17] manually. However, manual tiling of an application code is both labor intensive and error prone because there is no standard language construct for tiling in any of the general-purpose programming languages. In this paper, we introduce a new version of BoxLib that embeds tiling in the shared framework rather than in each application code. By supporting tiling in the Boxlib framework itself, application developers using BoxLib do not have to recode any of the tiling constructs.

In section 2, we present a brief introduction to tiling. In section 3, we give a brief overview of BoxLib and key abstractions in the pretiling version are presented. We then present the new version of BoxLib with tiling and discuss the threading approach. Several performance results are shown in section 4, demonstrating the improved performance of the tiling approach. In section 5, we discuss our plans for further optimization of BoxLib using the TiDA portable library [29, 30]. We summarize the results of this paper in the final section.

2. Tiling overview and related work. Modern systems contain complex memory hierarchies with multiple levels of caches and multiple NUMA nodes. It is essential that a software framework like BoxLib adapt to the increasing number of cores and possibly increasing number of NUMA domains in order to be able to take advantage of the current and future architectures. Tiling, also known as cache blocking, is a well-known program transformation that has proven to be useful in improving data locality and parallelism. Tiling eliminates some of the memory traffic by reducing the size of the working set so that there are fewer cache misses. In addition, it enables high degrees of parallelism through partitioning iteration space into independent units of execution. At the language level, there is no standard programming support to express tiling information in the program. The programmer has to manually tile individual loops, which can be very labor intensive and error prone for large codes like the BoxLib-based AMR applications. The most common way to

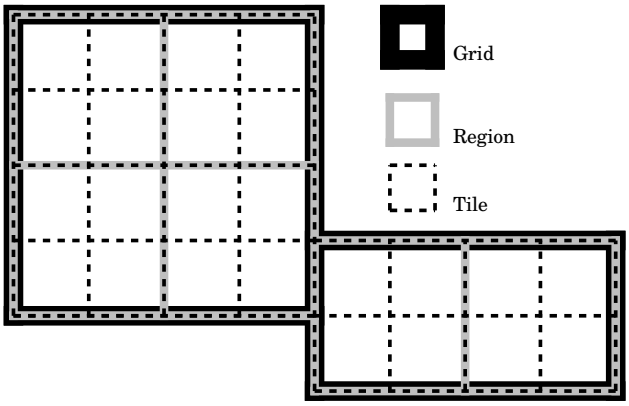

FIG. 1. *Grid, region, and tile. In this example of regional tiling, there are two grids. The grid on the left contains four regions, and the grid on the right contains two regions. Each region is split into four logical tiles in iteration space. The floating point data are allocated contiguously in each region; thus the data within a grid are not in one contiguous block. Note that logical tiles do not affect the data layout in memory.*

implement tiling is to partition a nested loop by introducing *tiling loops* that iterate over tiles and *element loops* that iterate over the data elements within a tile.

There is a long list of literature on iteration space tiling to move the burden from the programmer to the compiler [33, 10, 26, 12, 18, 27]. Compiler methods rely on static loop transformations [28], usually in the form of source-to-source translation. There is only limited support for tiling in commercial compilers because of the code generation complexity. The complexity emerges primarily from the semantics of existing programming languages (e.g., C, C++, or Fortran) and the inability in current programming environments to express essential information required for effective performance analysis and safe code transformations to autogenerate tiled code. It is particularly difficult for AMR codes, where crucial information needed to optimize data locality is available only at runtime. Our approach here is to decouple tiling from the individual loops and instead represent it at the data structure level within the BoxLib framework.

Hierarchically tiled arrays (HTA) [10] offer data structures for describing a hierarchy and topology of tiles where computation and communication are represented by overloaded array operations. Extending array notations hides many details from the programmer but eventually creates performance problems because of the excessive use of temporary arrays or frequent data layout transformations. Adopting HTA in BoxLib would have meant extensive rewriting of BoxLib and BoxLib-based application codes. TiDA [29, 30], on the other hand, is a library that allows parameterized data layout and tiling but does not require restructuring of the original software framework. We are motivated here by TiDA's approach to tiling and have integrated many features of TiDA into the BoxLib framework.

TiDA introduces two concepts: *regional* and *logical* tiling. Figure 1 shows an example of regional tiling. We denote a rectangular domain in index space as a *grid*; each grid is tiled into multiple *regions* with each region's floating point data allocated contiguously in memory. Each region is further split into *logical tiles* in iteration space; the logical tile size can be changed dynamically on a loop-by-loop

basis. It should be emphasized that logical tiles exist only in the iteration space and do not alter the floating point data layout in memory. A tiling approach based on the hierarchy of grid, region, and tile is called regional tiling. Logical tiling is a special case of regional tiling in which each grid contains only one region. Thus the data on a grid in logical tiling are contiguous in memory, whereas in regional tiling they are allocated in multiple contiguous blocks. Regional tiling is intended to address the NUMA nodes and regional coherence domains and is more general than logical tiling; however it requires data structure changes because existing AMR frameworks usually allocate one contiguous block of memory for the data on each grid. In contrast, logical tiling is easily adoptable by the underlying framework, because it requires minimal bookkeeping and incurs little overhead on existing codes.

3. BoxLib. The BoxLib software framework includes both a C++ framework that can call Fortran subroutines and an entirely Fortran framework. Both frameworks have been modified to use tiling, but we will present the illustrative examples in C++ only for simplicity.

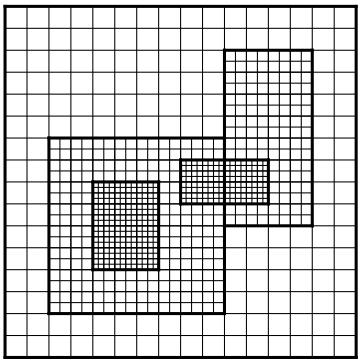

FIG. 2. *Cartoon of AMR grids with two levels of factor* 2 *refinement. The coarsest grid covers the domain with* 16^2 *cells. Bold lines represent grid boundaries. The two intermediate resolution grids are at level* 1 *and the cells are a factor of 2 finer than those at level* 0*. The two finest grids are at level* 2 *and the cells are a factor of 2 finer than the level* 1 *cells. Note that there is no direct parent-child connection. The data for each grid are contained in an object called FArrayBox (see section* 3.1*). In this example, there are* 1*,* 2*, and* 2 *FArrayBox objects on levels* 0*,* 1*, and* 2*, respectively. FArrayBox objects on each level are organized into an object called MultiFab (Multiple FArrayBox).*

The data in BoxLib is defined on a nested hierarchy of logically rectangular grids. Recall we use the term *grid* to refer to a rectangular domain in index space; in the context of block-structured AMR a *level* is composed of a union of (in this case nonintersecting) grids that share the same mesh (or cell) spacing. The ratio of mesh spacings between adjacent levels is typically 2 or 4. Figure 2 shows a cartoon of AMR grids in two dimensions with two levels of refinement. One aspect that distinguishes BoxLib from frameworks such as FLASH [19] is that a fine grid does not necessarily have a unique parent grid. This allows the organization of data into relatively large aggregate grids and as a result amortizes the cost incurred by the irregular nature of adaptive meshes.

The changes to BoxLib described in this paper focus on data locality and parallelism at each level of refinement independently; thus for the rest of the paper we will confine ourselves to the geometry of a single level. However, the fact that a tiling strategy must work in the context of complex multiphysics applications on adaptive

grid hierarchies dictates three necessary features. First, the strategy must work for a union of grids that are not necessarily of equal size and shape and that do not necessarily span the entire rectangular domain. Second, the strategy must be such that the tile size can be modified depending on the nature of the loop, as different parts of the algorithm may have very different computational and communication demands. Finally, the tiling strategy must be sufficiently lightweight to adapt to the frequently changing grid structure at all levels but the coarsest as the simulation evolves.

3.1. BoxLib without tiling. Before we discuss the tiling strategy, we describe the data structures and methods for operating on the data on a level in BoxLib without tiling. In BoxLib, a *Box* is the data structure for a rectangular domain in index space, and a three-dimensional Box can be represented by six integers. Each grid is represented by the Box type. An *FArrayBox* is the data structure that holds the floating point data on a single Box. The data can have multiple components (such as density and velocity) and are stored internally as a one-dimensional array allocated with C++ 's `new` operator and therefore are contiguous in memory. The index space information in an FArrayBox can be used to reshape the one-dimensional array into a multi dimensional array to be used in a Fortran subroutine. We note that while the grids themselves are nonintersecting boxes, the data in an FArrayBox may be defined on a Box larger than the grid if ghost (or halo) cells are needed. A *MultiFab* is the parallel data structure containing multiple FArrayBoxes which holds data on the multiple grids of an AMR level (see Figure 2). The data in a MultiFab are distributed among MPI processes. Thus, on each MPI process the MultiFab contains only the FArrayBox objects owned by this process, and the process operates only on its local data. For operations that require data owned by other MPI processes, ghost cells are filled via MPI communications. To reduce the latency and use the network bandwidth more efficiently, the communication messages are aggregated. For the purpose of communication within each level and between levels, each MPI process has a copy of all the grids.

Many common operations in BoxLib can be performed at the abstract level of MultiFab with the data and implementation details hidden from the application developers. A few examples are given below.

```
// Return the maximum value contained in component 0 of the
// MultiFab mf.
max_val = mf.max(0);

// Fill the ghost cells of each FArrayBox in the MultiFab mf
mf.FillBoundary();
```

To operate on the data in more application-specific ways, BoxLib supplies iterators, called `MFIter`s, over the FArrayBoxes in a MultiFab. The application developer provides a Fortran subroutine that performs the operations; each subroutine operates on one grid's worth of data at a time. For example,

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi) // Loop over grids
{
  // Define vbox to be the grid associated with this iteration.
  // This "valid" region will be used to define the extent
  // of the data
  // that the subroutine will operate on.
  const Box& vbox = mfi.validbox();
```

```
    // Get a reference to the FArrayBox so that we can access
    // both the data and the size of the FArrayBox.
    FArrayBox& fab = mf[mfi];

    // Define the double* pointer to the data of this FArrayBox.
    double* a = fab.dataPtr();

    // Define abox as the Box on which the data in the FArrayBox
    // is defined.
    // Note that "abox" includes ghost cells (if there are any),
    // and is thus
    // larger than or equal to "vbox".
    const Box& abox = fab.box();

    // We can now pass the information to a Fortran routine,
    // which treats the double* as a multi-dimensional array
    // with dimensions specified by the information in "abox".
    // We will also pass the index information in "vbox",
    // which specifies our "work" region.
    // Functions loVect() and hiVect() return the lower and upper
    // indices of a box, respectively.
    f(vbox.loVect(), vbox.hiVect(),
      a, abox.loVect(), abox.hiVect());
  }
```

The Fortran subroutine might look as shown here:

```
  subroutine f(lo, hi, a, alo, ahi)
    integer, intent(in) :: lo(3), hi(3), alo(3), ahi(3)
    double precision, intent(inout) :: a(alo(1):ahi(1),&
                                         alo(2):ahi(2),&
                                         alo(3):ahi(3))
    integer :: i, j, k
    !$OMP PARALLEL DO private(i,j,k)
    do     k = lo(3), hi(3)
      do   j = lo(2), hi(2)
        do i = lo(1), hi(1)
           ! ...
        end do
      end do
    end do
    !$OMP END PARALLEL DO
  end subroutine f
```

Here, for the sake of simplicity, we have omitted the code that facilitates calling Fortran subroutines from C++ . In BoxLib, this is usually handled by C preprocessor macros; one can also use the ISO_C_BINDING module in Fortran 2003. In this example, OpenMP is used on the loop level for work sharing among threads.

3.2. BoxLib with logical tiling. In the new version of BoxLib,[1] we introduce the capability for *logical tiling* as defined earlier. For backward compatibility the default is to have tiles the same size as (or larger than) the grids so that tiling is

[1]BoxLib is under active development and all updates are publicly available through git as soon as they are tested and committed. By "new version," we do not refer to any particular version number.

effectively turned off. In our logical tiling approach, each grid is now logically split into tiles (both grids and tiles are of type Box), and the modified looping construct is completely incorporated into the MFIter iterator which now loops over each tile in each grid rather than just over the grids themselves. With logical tiling, the data layout is unchanged; just the access pattern to the data is changed. An example of using tiling is shown below.

```
bool tiling = true;
// Loop over tiles
for (MFIter mfi(mf,tiling); mfi.isValid(); ++mfi)
{
  // Define the tile of this iteration
  // This tile, rather than the grid that the tile is a part of,
  // will be used to define the extent of the data
  // that the subroutine will operate on.
  const Box& tbox = mfi.tilebox();

  // Get a reference to the FArrayBox so that we can access
  // both the data and the size of the FArrayBox; the FArrayBox
  // is unchanged by using tiling
  FArrayBox& fab = mf[mfi];

  // Define the double* pointer to the data of this FArrayBox.
  // The dataPtr of the FArrayBox is unchanged by using tiling.
  double* a = fab.dataPtr();

  // Define abox as the Box on which the data in the FArrayBox
  // is defined.
  // This is also unchanged by using tiling.
  const Box& abox = fab.box();

  // We can now pass the information to a Fortran routine.
  // We now pass the index information in "tbox" to specify
  // the work region.
  f(tbox.loVect(), tbox.hiVect(),
    a, abox.loVect(), abox.hiVect());
}
```

In this example, the loop controlled by MFIter can be regarded as a nested loop collapsed into one iteration space, where the nested loop is composed of an outer loop of grids and an inner loop of tiles in a grid. Note that the code is almost identical to the one in section 3.1; the Fortran subroutine and the data pointer passed to it are unchanged, but the index space on which the subroutine operates is now defined by the tile rather than the grid. Loop tiling is achieved without any modification in the kernel function f. This is important to application developers using BoxLib since typically the kernels are application-specific. Moreover, any local arrays in the kernel function can now be the size of the tile rather than the potentially much larger grid; thus the working set size is further reduced, resulting in better data access patterns.

The MFIter class has several constructors:

```
MFIter(const MultiFab& mf);
MFIter(const MultiFab& mf, bool do_tiling);
MFIter(const MultiFab& mf, const int tilesize[]);
```

In the first version, each grid has one tile and the tiling is effectively turned off. If the second version is used, the default tile size (that can be specified at runtime) is used. The third version provides more control over the tile size by passing an integer array specifying the tile size in each spatial dimension. This allows applications to use different tile sizes in different parts of the algorithm, which is essential for maximizing performance in a complex multiphysics code. The new `tilebox` member function of MFIter returns a box for the current tile of the iteration. Additional MFIter member functions that are useful for building application codes include

```
Box growntilebox(int nghost) const;
Box nodaltilebox(int direction) const;
```

The function `growntilebox` returns a grown tile box that includes ghost cells at grid boundaries only; therefore the returned boxes originating from the same grid are still nonoverlapping. This is useful when the work region includes ghost cells. The function `nodaltilebox` returns nonoverlapping face-type boxes for tiles, with the face specified by the integer argument. This function is particularly useful for finite-volume methods where face-based fluxes are often used.

3.3. Threading in BoxLib. BoxLib uses OpenMP for threading. As seen in section 3.1, BoxLib traditionally used `OMP DO` in a fine-grained loop-level threading approach. Tiling provides the opportunity for a coarse-grained approach for threading, which enables the removal of OpenMP constructs from the individual Fortran routines. For example, OpenMP parallelism at the tile level can now be achieved by adding OpenMP pragmas around the MFIter loops:

```
#pragma omp parallel
// Loop over tiles
for (MFIter mfi(mf,true); mfi.isValid(); ++mfi)
{
  const Box& tbox = mfi.tilebox();
  FArrayBox& fab = mf[mfi];
  double* a = fab.dataPtr();
  const Box& abox = fab.box();
  f(tbox.loVect(), tbox.hiVect(),
  a, abox.loVect(), abox.hiVect());
}
```

Here `mfi` is a thread private MFIter object because it is created inside an OpenMP parallel region. Because an OpenMP parallel region has already started when function f is called, we remove the OpenMP directives in function f unless nested OpenMP parallelism is desired.

When MFIter detects that it is inside an OpenMP parallel region as in the code above, the iterations over tiles are shared among OpenMP threads. In our current implementation, the work scheduling policy is similar to OpenMP's static scheduling. Let us consider an example. Suppose there are four grids of different sizes at a level, and the tile size is such that the first and fourth grids are each divided into 4 logical tiles, while the second and third are divided into 2 logical tiles each. In the tiling version, the loop body will be run 12 times in total, as opposed to 4 times in the untiled version. Assuming four threads are used, thread 0 will work on 3 tiles from the first grid, thread 1 on 1 tile from the first grid and 2 tiles from the second grid, and so forth.

In addition to the benefits of tiling itself, this coarse-grained tile-level threading approach has several advantages over the fine-grained loop-level approach shown in section 3.1. First, when individual grids are small (such as often occurs after multiple coarsenings near the bottom of a multigrid V-cycle in a multigrid solver), there can still be enough parallelism for the tile-level approach even though there might not have been enough without tiling. Second, the tile-level approach has removed the need to use OpenMP directives in the application-specific Fortran subroutines in most cases, which reduces the chance of application-specific OpenMP coding errors. Finally, it reduces thread overhead and synchronization points because there is only one OpenMP parallel region as opposed to many parallel regions in the fine-grained loop-level approach.

There could be an even coarser-grained grid-level parallelism approach, in which the operations on each grid are performed by only one thread and different threads operate on different grids. This grid-level approach is less desirable than the tile-level approach, because in complex AMR applications the boxes typically have different sizes, resulting in load imbalance among threads, and because there will typically be more threads than grids on many-core architectures unless many small grids are used. Increasing the amount of parallelism by using many small grids in an AMR framework such as BoxLib is not recommended. It increases the size of the AMR metadata, increases the cost of performing box-box intersections needed to fill ghost cells and copy data between AMR levels, and increases the actual computation because of the increased volume of ghost cells. All of the benefits one would achieve with many small grids are achieved by tiling the grids instead.

Besides providing a backward compatible tiling capability and a coarse-grained tile-level threading approach, the new version of BoxLib includes many performance improvements that are made possible by applying the tiling concept. One such example is the operation of filling ghost cells, which involves both local and remote communications. For data available on its own MPI process, simple copy operations are performed, whereas to get data from a remote MPI process, MPI calls must be made. In BoxLib, MPI messages between processes are aggregated by using buffers for MPI send and receive calls to reduce communication overhead and to increase the utility of network bandwidth. Thus the remote communication also involves local data movement. These local data copy operations are usually considered cheap. However, on a many-core architecture with hundreds of threads, they could become a bottleneck if they are not threaded or have low parallel efficiency, as expected from Amdahl's law [7]. In section 4.1, we will present such an example and demonstrate the improvement from using tiling.

The user function called inside the MFIter loop often dynamically allocates its own temporary arrays. In a multithreaded environment, this often results in lock contention and sometimes translation lookaside buffer (TLB) misses or even page faults. One way of avoiding these issues is using Fortran automatic arrays, which are usually put on the stack instead of the heap by default (see [32] for an example of this approach). Another approach we have implemented in BoxLib is creating thread private memory arenas and allocating dynamic arrays from the arenas. One advantage of the latter approach is that it provides the flexibility of allocating the high-bandwidth on-package memory available on the next generation of chips (e.g., Intel Knights Landing).

Finally we note that tiling is not always desired or better. The traditional fine-grained loop-level threading approach coupled with dynamic scheduling is more appropriate for work with unbalanced loads, such as calculating chemical reactions in cells by an implicit solver such as VODE [11].

4. Performance. In this section, we present several results demonstrating the performance benefits of tiling in BoxLib. In the first test we consider an explicit heat equation solver built with the BoxLib C++ framework, and in the second test we use a minimalist version of SMC [17] built with the BoxLib Fortran framework. Unless stated otherwise, we define speedup as the ratio of the runtime to the corresponding time in the 1-thread untiled run. Note that MPI is not used in these tests so that we can accurately measure the on-node performance impact of tiling.

4.1. Heat equation solver. In our tests of tiling in the heat equation solver, the code took 1000 time steps on a single level using a forward Euler method with second-order centered differences for the spatial derivatives.[2] The domain contained a single grid with 128^3 cells, and periodic boundaries were assumed in all three dimensions. The computation of second derivatives in the heat equation kernel is performed in two steps with three separate loops for the flux components and another loop for the divergence of the flux. Although these loops in this memory-bound situation can be fused to improve cache locality, this implementation is a typical pattern for application developers without specific computer architecture knowledge. This test was intentionally simple so that the impact of tiling can be clearly identified.

The first test was a series of nonthreaded runs carried out on a single core of Edison at the National Energy Research Scientific Computing Center (NERSC), using either the default Gnu or Intel compiler. In these runs we varied the tile size and found that even with no parallelism, tiling can significantly improve the performance due to the smaller working set size (Table 1). In this serial test, a factor of 1.8 speedup (Intel) and a factor of 3.4 speedup (Gnu), due entirely to logical tiling, were achieved.

TABLE 1

Impact of tiling on the performance of the heat equation kernel on the Edison machine at NERSC using the Gnu and Intel compilers. We show the runtime and speedup of the computational kernel for various tile sizes. The tile size refers to the number of cells in a logical tile in the x-, y-, and z-directions, respectively.

Tile size	Time, Gnu (s)	Speedup, Gnu	Time, Intel (s)	Speedup, Intel
$128 \times 4 \times 4$	8.5	3.4	8.7	1.8
$128 \times 8 \times 8$	9.0	3.2	9.6	1.6
$128 \times 16 \times 16$	9.6	3.0	10.5	1.5
$128 \times 32 \times 32$	23.7	1.2	10.4	1.5
$128 \times 64 \times 64$	24.4	1.2	10.9	1.4
No tiling	28.6	–	15.5	–

We next explored the parallel performance of the tiled heat equation solver. We performed a strong scaling study on Edison, which has 12-core Intel Ivy Bridge processors, and used OpenMP for threading. Figure 3 compares the tiled runs with untiled runs where a tile size of $128 \times 4 \times 4$ cells was used for all of the tiled runs. Two different threading approaches were used; the tiled runs used the coarse-grained tile-level threading approach as described in section 3.2, whereas the untiled runs used the fine-grained loop-level threading approach described in section 3.3. The speedup shown in Figure 3 is for the computational kernel only. In this figure the tile-level threading clearly demonstrates much better scaling behavior than the loop-level approach. The 12-thread tiled run was more than 5.3 times faster than the 12-thread untiled run, and it had a 16.5x speedup over the single thread untiled run. For the tiled runs, a

[2] The source code of the test is available at `Tutorials/Tiling_Heat_C` in BoxLib.

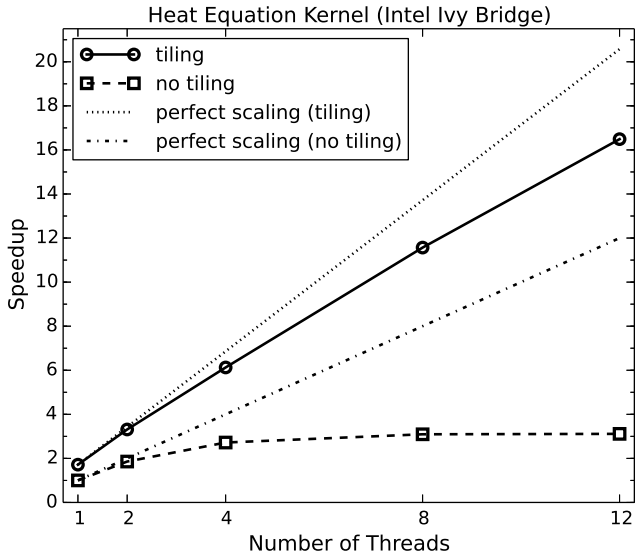

FIG. 3. *Heat equation kernel speedup on* 12*-core Intel Ivy Bridge. We compare the results of tiled and untiled runs. A* 16.5x *speedup was obtained in the* 12*-thread tiled run.*

9.6x speedup over the single thread tiled run was obtained with 12 threads. We note that perfect strong scaling within a socket is difficult to achieve because Edison has 30 MB L3 cache shared among 12 cores on a processor.

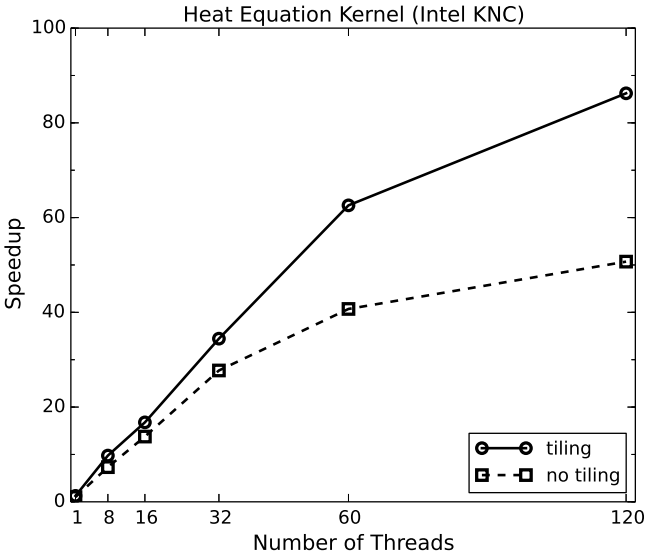

FIG. 4. *Heat equation kernel speedup on* 60*-core Intel KNC. We compare the results of tiled and untiled runs. A speedup of* 86x *was obtained for the* 120*-thread tiled run.*

A similar strong scaling study of the heat equation was carried out on a single 60-core Intel Knights Corner (KNC) processor on NERSC's Babbage computer. We again observed the performance benefit of the tiling approach with coarse-grained

tile-level parallelism over the nontiling approach with fine-grained loop-level parallelism. Figure 4 shows the speedup of the computational kernel for untiled and tiled runs. The 120-thread tiled run had a speedup of 69x and 86x with respect to the 1-thread tiled and nontiled runs, respectively. The 60-core Intel KNC processors on Babbage are capable of spawning 4 hardware threads per core. The results using 180 and 240 threads are not shown in the figure because no speedup of the computational kernel was obtained beyond 120 threads. In fact the 240-thread run was about 20% slower than the 120-thread run. We note that no effort was spent on optimizing the kernel for KNC.

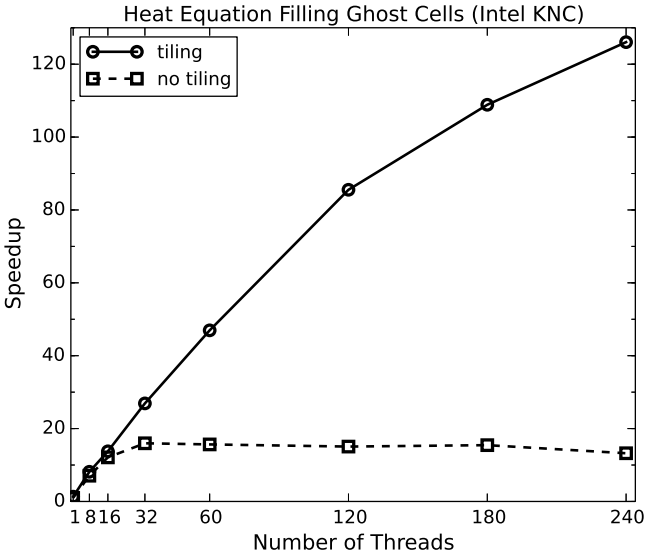

FIG. 5. *Speedup in filling ghost cells in the heat equation solver on* 60*-core Intel KNC. We compare the results of tiled and untiled runs. A speedup of* 126x *was obtained for the* 240*-thread tiled run.*

In the strong scaling study described above, we also measured the time to fill ghost cells at the periodic boundaries. Note again that MPI was not used and the data movement was completely local. The scaling results for just filling the ghost cells are shown in Figure 5. The 240-thread tiled run had a speedup of 96x and 126x in filling ghost cells with respect to the 1-thread tiled and nontiled runs, respectively. Figure 6 shows the fraction of time spent filling ghost cells defined as fill time / (fill time + kernel time). It is a striking reminder of Amdahl's law [7] that the 120-thread nontiled run spent more than 40% of the time filling ghost cells.

4.2. SMC. The SMC code solves the multicomponent, reacting, compressible Navier–Stokes equations using a mixture model for species diffusion [17]. The spatial derivatives are computed with eighth-order stencils that span nine cells in each coordinate direction. Here we use a minimalist version of SMC for performance testing.[3] In this minimalist version, Sutherland's viscosity law is adopted, a constant Schmidt number is used for mass diffusivity, and a constant Prandtl number is used to compute thermal diffusivity. The runs considered here used a nine-species

[3]The source code of this minimalist version of SMC is available at `MiniApps/SMC` in BoxLib.

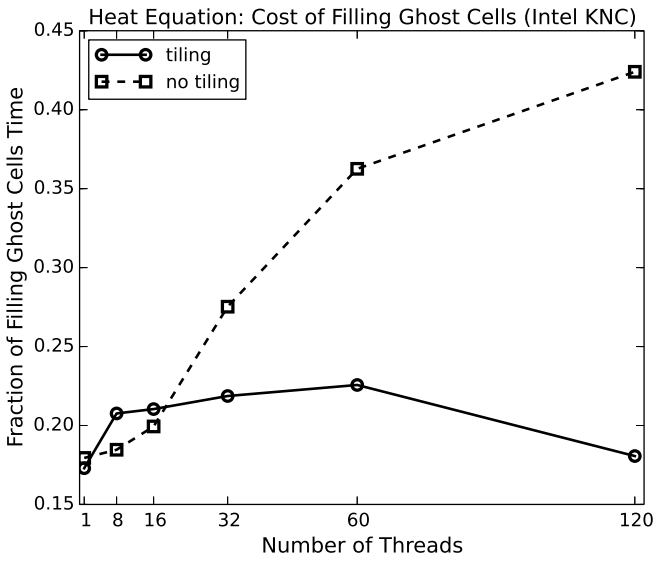

FIG. 6. *Heat equation: cost of filling ghost cells time on* 60*-core Intel KNC. We show the ratio of the time spent filling ghost cells to the sum of filling ghost cells and kernel times. Note that a significant fraction of the time was spent filling ghost cells for the untiled runs.*

H_2/O_2 reaction mechanism [20] and a third-order Runge–Kutta method was used for time integration. We have previously demonstrated excellent scaling behavior of manually tiled SMC [17]. The version of SMC described here is based on the new version of BoxLib described in this paper.

A series of SMC runs were carried out on a single 60-core processor on Babbage. As with the heat equation solver, the domain contained a single grid of 128^3 cells and periodic boundaries were assumed in all three dimensions. The tile size was again chosen to be $128 \times 4 \times 4$ in the x-, y-, and z-directions, respectively. For comparison, we have performed both tiled and untiled runs. As above, the tiled runs used the coarse-grained tile-level threading approach, whereas the untiled runs used the fine-grained loop-level threading approach. The scaling results are shown in Figure 7. A speedup of 86x was obtained for the 180-thread tiled run over the single thread tiled run, similar to the result obtained with the manual tiling. Moreover, the best tiled run was 2.4 times faster than the best untiled run, and a 92x speedup was obtained for the 180-thread tiled run over the single thread untiled run. The results demonstrate clearly that excellent performance can be obtained using BoxLib with tiling on many-core architectures with more than 100 threads.

5. Future work. Future work for the BoxLib framework includes integrating the TiDA library currently under development into the BoxLib framework in order to enable regional tiling as well as logical tiling. Regional tiling addresses potential NUMA issues that are likely to become even more prevalent on next-generation architectures. Such architectures enable a thread placed on a NUMA domain to access remote memory of another domain at a cost of higher latency and lower bandwidth, often referred to as NUMA effects. One can address NUMA effects either by placing data in the memory local to where most computations will be carried out or by moving computations to the NUMA domain that holds the data. However, neither memory

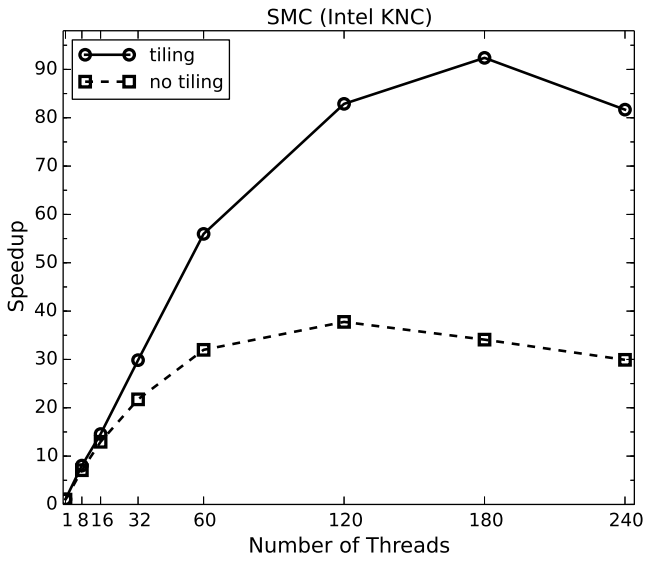

FIG. 7. *SMC speedup on* 60*-core Intel KNC. We show the speedup of the whole run excluding the one-time initialization. The* 180*-thread tiled run had a* 92x *speedup over the single thread untiled run.*

management nor thread management is a trivial task for the application developer. The approach currently adopted in BoxLib is to place one MPI process per NUMA node and use a threaded model such as OpenMP within a NUMA node. However, such a model results in higher overhead for interprocess synchronization (e.g., barrier and reduction), due to more MPI processes, than a model in which a single MPI process spans multiple NUMA nodes. In addition, MPI often requires additional memory to buffer messages at both the application and system levels. In future architectures we expect to see a significant reduction in memory capacity per core and a substantial increase in the number of NUMA domains per compute node [31]. Regional tiling allows one to respect NUMA effects while operating with one MPI process per compute node.

The TiDA library [29, 30] will provide developers with NUMA-aware tiling and layout abstractions at the application level and hides details of how regions and tiles are constructed, mapped, and executed on various hardware platforms. The layout abstraction manages the memory allocation for regions and keeps track of their location on physical partitions of the memory. In Figure 8 we show a performance comparison between prototype logical and regional tiling implementations of SMC. In this example, we conducted a strong scaling study on up to 8 NUMA domains (i.e., 32 cores) of a compute node on Trestles, a system at San Diego Supercomputer Center. It can be seen that when only one NUMA node (4 cores) is used, the performance of both implementations matches. However, with more NUMA domains regional tiling overtakes logical tiling, and the performance gap expands as the number of NUMA domains increases. The reason is that regional tiling reduces NUMA effects by allocating each region on a different NUMA node. In particular, with logical tiling a tile may access remote memory if a neighbor resides in a different NUMA domain. Latency costs will become significant when there are many remote memory accesses on small amounts of data. With regional tiling, tiles at the boundary exchange ghost cells,

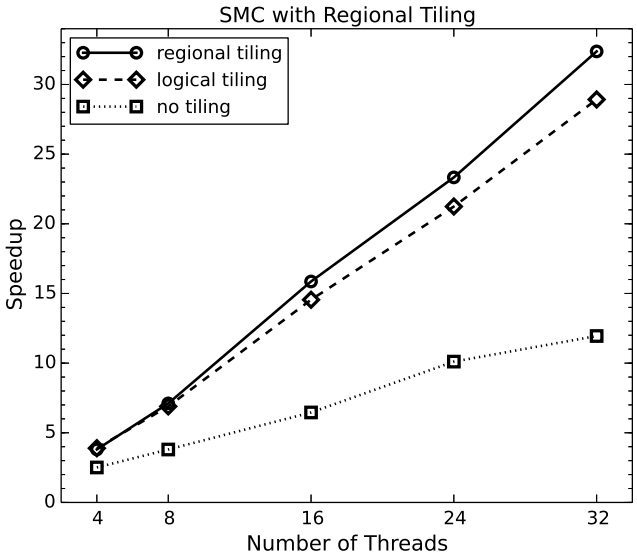

FIG. 8. *SMC with regional tiling on Trestles.*

realizing higher bandwidth and amortizing the latency overhead. In this prototype version of the SMC code, regional tiling achieves 32x the performance over one thread on 32 cores.

6. Summary. In order to address the performance challenges that will accompany next generation node architectures based on many-core processors with NUMA domains, we have implemented logical tiling in BoxLib, a block-structured adaptive mesh refinement software framework. In this paper we have described how the tiling constructs are embedded in the iterator constructs in BoxLib so that BoxLib-based applications can easily realize the expected serial and parallel performance gains without extra effort on the part of the application developer. We also discussed a path forward to enable future versions of BoxLib to take advantage of NUMA-aware optimizations using the TiDA portable library.

Acknowledgments. We would like to acknowledge and thank Brian Friesen for running some of the performance tests and John Bell for many useful discussions concerning tiling.

REFERENCES

[1] *BoxLib*, Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory, https://ccse.lbl.gov/BoxLib/.

[2] A. S. ALMGREN, J. B. BELL, M. J. LIJEWSKI, Z. LUKIC, AND E. VAN ANDEL, *Nyx: A massively parallel AMR code for computational cosmology*, Astrophys. J., 765 (2013), p. 39.

[3] A. S. ALMGREN, V. E. BECKNER, J. B. BELL, M. S. DAY, L. H. HOWELL, C. C. JOGGERST, M. J. LIJEWSKI, A. NONAKA, M. SINGER, AND M. ZINGALE, *CASTRO: A New Compressible Astrophysical Solver.* I. *Hydrodynamics and Self-gravity*, Astrophys. J., 715 (2010), pp. 1221–1238.

[4] A. S. ALMGREN, J. B. BELL, A. NONAKA, AND M. ZINGALE, *Low Mach number modeling of Type* Ia *supernovae.* III. *Reactions*, Astrophys. J., 684 (2008), pp. 449–470.

[5] A. S. Almgren, J. B. Bell, C. A. Rendleman, and M. Zingale, *Low Mach number modeling of Type* Ia *supernovae.* I. *Hydrodynamics*, Astrophys. J., 637 (2006), pp. 922–936.

[6] A. S. Almgren, J. B. Bell, C. A. Rendleman, and M. Zingale, *Low Mach number modeling of Type* Ia *supernovae.* II. *Energy Evolution*, Astrophys. J., 649 (2006), pp. 927–938.

[7] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, in Proceedings of the Spring Joint Computer Conference, ACM, 1967, pp. 483–485.

[8] A. J. Aspden, M. S. Day, and J. B. Bell, *Turbulence-chemistry interaction in lean premixed hydrogen combustion*, Proc. Comb. Inst., 35 (2014), pp. 1321–1329.

[9] J. B. Bell, M. S. Day, and M. J. Lijewski, *Simulation of nitrogen emissions in a premixed hydrogen flame stabilized on a low swirl burner*, Proc. Comb. Inst., 34 (2013), pp. 1173–1182.

[10] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C̃. von Praun, *Programming for parallelism and locality with hierarchically tiled arrays*, in Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, ACM, 2006, pp. 48–57.

[11] P. Brown, G. Byrne, and A. Hindmarsh, *Vode: A variable-coefficient ODE solver*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1038–1051.

[12] S. Carr and K. Kennedy, *Compiler blockability of numerical algorithms*, in Proceedings of the ACM/IEEE Conference on Supercomputing, Los Alamitos, CA, 1992, pp. 114–124.

[13] M. S. Day and J. B. Bell, *Numerical simulation of laminar reacting flows with complex chemistry*, Combust. Theory Model., 4 (2000), pp. 535–556.

[14] M. Duarte, A. S. Almgren, K. Balakrishnan, J. B. Bell, and D. M. Romps, *A numerical study of methods for moist atmospheric flows: Compressible equations*, Monthly Weather Rev., 142 (2014), pp. 4269–4283.

[15] M. Duarte, A. S. Almgren, and J. B. Bell, *A low mach number model for moist atmospheric flows*, J. Atmos. Sci., 72 (2015), pp. 1605–1620.

[16] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, D. Graves P. Colella, M. Lijewski, F. Loffler, B. O'Shea, E. Schnetter, B. Van Straalen, and K. Weide, *A survey of high level frameworks in block-structured adaptive mesh refinement packages*, J. Parallel Distributed Comput., 74 (2014), pp. 3217–3227.

[17] M. Emmett, W. Zhang, and J. B. Bell, *High-order algorithms for compressible reacting flow with complex chemistry*, Combust. Theory Model., 18 (2014), pp. 361–387.

[18] G. Goumas et al., *Automatic parallel code generation for tiled nested loops*, in Proceedings of the ACM Symposium on Applied Computing, 2004.

[19] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, *FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes*, Astrophys. J. Suppl., 131 (2000), pp. 273–334.

[20] J. Li, Z. Zhao, A. Kazakov, and F. L. Dryer, *An updated comprehensive kinetic model of hydrogen combustion*, Internat. J. of Chemical Kinetics, 36 (2004), pp. 566–575.

[21] Z. Lukic, C. Stark, P. Nugent, M. White, A. Meiksin, and A. Almgren, *The Lyman-alpha forest in optically-thin hydrodynamical simulations*, Monthly Notices R. Astro. Soc. 446 (2015), pp. 3697–3724.

[22] C. M. Malone, A. Nonaka, S. E. Woosley, A. S. Almgren, J. B. Bell, S. Dong, and M. Zingale, *The deflagration stage of Chandrasekhar mass models for type* Ia *supernovae.* I. *Early evolution*, Astrophys. J., 782 (2014), p. 11.

[23] A. Nonaka, A. S. Almgren, J. B. Bell, M. J. Lijewski, C. M. Malone, and M. Zingale, `MAESTRO`*: An adaptive low mach number hydrodynamics algorithm for stellar flows*, Astrophys. J., 188 (2010), pp. 358–383.

[24] G. S. H. Pau, J. B Bell, A. S Almgren, K. M Fagnan, and M. J. Lijewski, *An adaptive mesh refinement algorithm for compressible two-phase flow in porous media*, Comput. Geosci., 16 (2012), pp. 577–592.

[25] G. Rivera and C.-W. Tseng, *Tiling optimizations for* 3*D scientific computations*, in Proceedings of the ACM/IEEE Conference on Supercomputing, Washington, DC, 2000.

[26] G. Rivera and C. W. Tseng, *Tiling optimizations for* 3*D scientific computations*, in Proceedings of the ACM/IEEE Conference on Supercomputing, Washington, DC, 2000.

[27] Y. Song and Z. Li, *New tiling techniques to improve cache temporal locality*, SIGPLAN Not., 34 (1999), pp. 215–228.

[28] D. Unat, X. Cai, and S. B. Baden, *MINT: Realizing CUDA performance in* 3*D stencil methods with annotated C*, in Proceedings of the International Conference on Supercomputing, New York, ACM, 2011, pp. 214–224.

[29] D. Unat, C. Chan, W. Zhang, J. Bell, and J. Shalf, *Tiling as a durable abstraction for parallelism and data locality*, presented at Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, Nov. 18, 2013.

[30] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, *TiDA: High-level programming abstractions for data locality management*, in High Performance Computing, Lecture Notes in Comput. Sci. 9697, J. M. Kunkel, P. Balaji, and J. Dongarra, eds., Springer, Berlin, 2016, pp. 116–135.

[31] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey, et al., eds., *Programming Abstractions for Data Locality*, Technical report 04, Lugano, Switzerland, 2014, https://sites.google.com/a/lbl.gov/padal-workshop/white-paper.

[32] A. Valles and W. Zhang, *Optimizing for reacting Navier-Stokes equations*, in High Performance Parallelism Pearls Volume One: Multicore and Many-Core Programming Approaches, J. Reinders and J. Jeffers, eds., Morgan Kaufmann, Burlington, MA, 2014.

[33] M. Wolfe, *More iteration space tiling*, in Proceedings of the ACM/IEEE Conference on Supercomputing, New York, ACM, 1989, pp. 655–664.

[34] W. Zhang, L. Howell, A. Almgren, A. Burrows, and J. Bell, *Castro: A new compressible astrophysical solver.* II. *Gray radiation hydrodynamics*, Astrophys. J., 196 (2011), p. 20.

[35] W. Zhang, L. Howell, A. Almgren, A. Burrows, J. Dolence, and J. Bell, *Castro: A new compressible astrophysical solver.* III. *Multigroup radiation hydrodynamics*, Astrophys. J., 204 (2013), p. 7.

[36] M. Zingale, A. S. Almgren, J. B. Bell, A. Nonaka, and S. E. Woosley, *Low mach number modeling of Type* Ia *supernovae.* IV. *White dwarf convection*, Astrophys. J., 704 (2009), pp. 196–210.