A Cache-Aware Approach to Adaptive Mesh Refinement in Parallel Stencil-based Solvers

Gaurav Saxena, Peter K. Jimack and Mark A. Walkley
School of Computing
University of Leeds
Leeds LS2 9JT
Email: {scgs, P.K.Jimack, M.A.Walkley}@leeds.ac.uk

***Abstract*—In prior-research the authors have demonstrated that, for stencil-based numerical solvers for Partial Differential Equations (PDEs), the parallel performance can be significantly improved by selecting sub-domains that are not cubic in shape (Saxena et. al., HPCS 2016, pp. 875-885). This is achieved through accounting for cache utilization in both the message passing and the computational kernel, where it is demonstrated that the optimal domain decompositions not only depend on the communication and load balance but also on the cache-misses, amongst other factors. In this work we demonstrate that those conclusions may also be extended to more advanced numerical discretizations, based upon Adaptive Mesh Refinement (AMR). In particular, we show that when basing our AMR strategy on the local refinement of patches of the mesh, the optimal patch shape is not typically cubic. We provide specific examples, with accompanying explanation, to show that communication minimizing strategies are not necessarily the best choice when applying AMR in parallel. All numerical tests undertaken in this work are based upon the open source BoxLib library.**

***Index Terms*—Partial Differential Equations ; Adaptive Mesh Refinement ; Finite Difference ; Domain Decomposition ; Cache-misses**

I. Introduction

Solving Partial Differential Equations (PDEs) forms a central part of Scientific Computing. Since the analytical solution is infeasible in most cases, they are discretized over a domain and solved using direct or iterative methods on computer systems. The discretized domain is most simply represented using a single uniform grid. However, for increased accuracy in particular regions, instead of refining the entire grid, those particular regions may be further refined according to some criteria. Adaptive Mesh Refinement (AMR) [1], [2], [3] is a technique where the compute resources are directed towards obtaining an increased precision in the solution in a particular region of interest. The region of interest is dependent on the particular application and can, for example, be a space-region where the solution transitions rapidly. Thus, instead of approximating the solution on a globally refined grid, the solution can be obtained with less overall compute work by refining in a certain local region. BoxLib [4] is a software library which may be used for developing parallel block structured Adaptive Mesh Refinement (AMR) applications in two and three dimensions. BoxLib has been written with a combination of C++ and Fortran90. In addition, a pure Fortran90 version also exists. For simplicity and clarity the current work will only use and refer to the pure Fortran90 version of BoxLib. However, alternative parallel AMR libraries do exist, p4est [5] and Paramesh [6], for example.

II. Motivation, Focus and Contribution

Our work in [7] showed that *only minimizing communication is not sufficient* to obtain optimal domain partitions for stencil codes on uniform 3-D grids. Thus, contrary to the generally accepted practice of using cubic sub-domains which only minimize the volume of communication, using non-cubic partitions which optimize cache-misses [7] can yield better performance. To this effect we developed a high level cache-aware model by utilizing only the cache-line length that analyzed and minimized the sub-domain level cache-misses to optimize domain partitions in Stencil-based codes. To model the cache-misses, we used a 7-pt stencil in 3-D, an unweighted Jacobi solver and assumed sub-domains with a 1-element deep ghost/halo zone. The 7-pt stencil used a weighted contribution from the six immediate data neighbours to update the solution at a point. Though the model was derived using a Finite Difference discretization and a 7-pt stencil, it applies in general to a 19-pt/27-pt stencil and Finite Element/Finite Volume discretizations with appropriate qualitative and quantitative differences. Our experiments demonstrated the efficacy of our model by simulating the Laplace equation (second order Elliptic PDE) on a unit cube with Dirichlet boundaries.

For the purpose of identifying and demarcating the sources of cache-misses, each sub-domain was visualized to be made up of three distinct parts : (i) The *Independent Compute* (IC) kernel which does not require data from other processes for its update. (ii) The *Dependent Planes* (DP) i.e. the next-to-boundary layers that need data from other processes for updating the mesh points. (iii) The *ghost/halo* zone which acts as a buffer to store the incoming data from other processes. The cache-misses then arise from updating the IC and updating/packing/unpacking the DP. We concluded that "close to 2-D" partitions for 3-D Stencil codes offered better performance than the communication minimizing partitions returned by the default `MPI_Dims_create()` Cartesian topology of MPI. A Cartesian Topology in MPI [8] is a *virtual* *geometrical* arrangement of processes. We advocated that a *balance* be maintained between minimizing cache-misses and the communication volume for obtaining optimal partitions *instead of* minimizing *only* the communication volume.

Our aim in the current work is to evaluate the performance and extendibility of our model when solving problems using AMR software to produce locally refined meshes. We contribute in the following ways :

- Implementation of a *new layout* simulating the MPI Cartesian Process Topology [8] in BoxLib and its performance evaluation (Section V and Section VII-A).
- Demonstrate that the hypothesis formulated in [7] also holds for single grid codes in BoxLib *despite* the non-overlap of communication and computation (Section VII-A).
- An assessment of the performance impact of utilizing non-cubic boxes in AMR techniques and to demonstrate that *a communication minimization scheme does not generally* yield the optimal execution time (Section VII-B).

III. Related Work

To the best of our knowledge, this is the first work to consider the impact of patch shape on the parallel performance of AMR-based solvers for Elliptic PDEs. Many software libraries, such as Paramesh [9] for example implicitly assume the use of cubic mesh patches in their parallel AMR implementation. Others such as BoxLib do permit non-cubic patches/boxes, however the box shape is determined purely based upon accuracy considerations rather than parallel performance. Indeed, the use of BoxLib library, by default gets only limited ability to control the box shapes. In this paper all of our numerical tests are undertaken with our own Finite Difference codes written to comply with BoxLib data structures.

Adaptive Mesh Refinement is a technique where a locally refined portion of the grid/mesh is solved at a higher resolution than the rest of the domain [1]. An application of such methods could be simulating a small area of interest with greater resolution as compared to the remaining region. For e.g., a tornado in a storm or air flow near the fuselage of an air plane can be areas of local interest in a global region [10]. AMR is extremely useful for applications involving a large gradient change, phase change, discontinuities, shocks and is implemented by adding new cells/grid points and deleting old cells/grid points [11]. The main goal of AMR is to obtain a desired accuracy of solution with the least possible mesh points, thus implying an optimal use of computational resources.

AMR can be used for both structured (SAMR) and unstructured meshes (UAMR) [1], [9]. SAMR uses logical rectangular grids refined spatially and temporally. The main advantage of SAMR is the ease with which the neighbours of a mesh point can be decoded/located. Refinement generally produces a hierarchy of grids at different resolutions. The level of a grid can be defined as the number of grids below it where the grid at level $l_1 > l_2$ is finer than the grid at l_2. *Mostly* the boundaries of finer grids coincide with the coarse grid cells to simplify inter-level communication and numerical approximations. In a true block SAMR approach, the entire block is refined even if a single mesh point belonging to the block is marked for refinement. Some notable software packages for parallel Structured AMR (SAMR) are : Chombo [12], BoxLib [4] (both from Lawrence Berkeley National Laboratory), Paramesh [6] (NASA) and SAMRAI [13] (Lawrence Livermore National Laboratory). A detailed survey of block-structured AMR can be found in [9].

BoxLib is a parallel SAMR software for building multiphysics multiscale codes that supports Hybrid parallelism at a massive scale [4]. The low level MPI communication calls are abstracted away from the user and functions are provided for same-level grids and fine-coarse grid interface data transfers. The major computational intensity in BoxLib lies in two types of computations : (i) Point-wise evaluation i.e. expressions of the form $\bar{\phi}_{i,j,k} = \phi_{i,j,k} + k(fx_{i,j,k} + fy_{i,j,k} + fz_{i,j,k})$ where a single point (i, j, k) in different arrays is used in a computation (ii) Stencil evaluations i.e. expressions of the form $\bar{\phi}_{i,j,k} = k\phi_{i,j,k} + m(\phi_{i\pm a,j,k} + \phi_{i,j\pm a,k} + \phi_{i,j,k\pm a})$ where a is some scalar offset [14]. In a recent work with Hybrid parallelism in BoxLib, the division of the entire index range of the set of boxes owned by a process to the set of threads (*Tiling*) notedly outperformed the strategy of dividing each box among the set of threads (*Striping*) [14]. Tiling exposes more parallelism and reduces the working set size of threads [15]. BoxLib has been used for creating several mature applications like MAESTRO (low Mach number code) [16], CASTRO (compressible Astrophysics) [17] and LMC (Combustion code) [18] which scale well but are limited by the high communication-intensive linear solves. The library can be downloaded for development at [19].

IV. BoxLib

The most basic constituent element/abstraction in BoxLib is the Fab (FArray Box) which represents a set of contiguous data on a Box. A Box is a data structure for representing a rectangular domain (in three dimensions, regular hexahedral) on an index space. Thus, a grid (a rectangular region in an index space) at any level is equivalent to a single Fab object [15]. The collection of all the Fab objects at a particular level is referred to as a MultiFab. There is no direct parent-child relationship between grids at different levels. It is the Fab objects that are distributed among cores and are acted upon independently by them. In AMR when the number of levels is greater than or equal to three, BoxLib requires and ensures proper nesting i.e. level $n + 1$ grids must be fully contained in level n grids (except at the physical boundaries).

Boxes can be split up into multiple small boxes to be given to various cores according to a data distribution algorithm. Two data distribution schemes, namely, the Knapsack to equalize load distribution and Morton Space Filling curve to optimize communication are part of the software. Each process contains enough metadata to locate the index space region of each box on every level so that it knows which processor core contains which box. The scheme for numbering the Fabs is analogous to the column-major order. The MPI ranks of the processes to which these boxes must be given to create an MPI Cartesian Topology should be in the row-major order. BoxLib internally maintains a *one dimensional integer* array named the `prc` array which maintains a mapping from the box numbers to the ranks. As an example, if there are 16 boxes, the `prc` array will have a length of 16 and if `prc(7) = 10` then the 7^{th} box is given to the process having the MPI rank 10. Further, each process maintains a copy of this array and there exists a separate `prc` array for each level of AMR.

V. Implementing a MPI Cartesian Topology

To implement an MPI Cartesian Topology, we extend the `layout_set_mapping()` subroutine of BoxLib to contain the X, Y, and Z integer *process dimensions* in 3-D. The process dimensions are then captured into integer variables `D_x, D_y`, and `D_z` (see Listing 1). We use a rank array (`rank_array(1:no_of_processes)`) to fill the 1-D array `prc(1:no_of_boxes)` in column-major order that gives the mapping of boxes to MPI ranks as mentioned in the previous section. Listing 1 shows our relevant subroutine for creating a MPI Cartesian Topology. A separate similar subroutine to handle multiple boxes per core was also written (not shown here). The MPI ranks are filled in the rank array (`rank_array`) in the same order (row-major) as the coordinates used in the MPI process decomposition. Since the boxes are numbered in Fortran order, the order of the loops in Listing 1 is important.

```
subroutine layout_dd(prc)
 integer, intent(out), dimension(:) :: prc
 integer :: i,j,k,ctr

 allocate(rank_array(D_x,D_y,D_z))
 ! row-major order for MPI ranks
  ctr=0
  do i=1,D_x
   do j=1,D_y
    do k=1,D_z
     rank_array(i,j,k)=ctr
     ctr=ctr + 1
    end do
   end do
  end do

 ! Fill prc(:) - start bottom left-> up->next (
   column-major order)
 ! -> next 2-D slab in Z-dimension

 ctr=1
 do k=1,D_z
  do j=1,D_y
   do i=D_x,1,-1
    prc(ctr)=rank_array(i,j,k)
    ctr=ctr + 1
    end do
   end do
  end do
  deallocate(rank_array)
end subroutine layout_dd
```

Listing 1: 3-D MPI Cartesian Topology

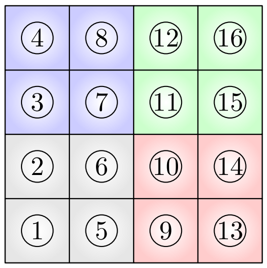


Fig. 1: 16 Fabs (or boxes) distributed on 4 processes decomposed as a 2x2 process topology. Each color shows a single MPI process and numbers inside circles show the Fab number

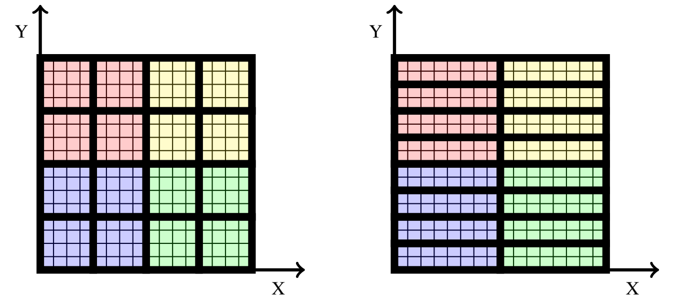
As an example, if the MPI Cartesian Topology is 2x3x4 corresponding to 24 boxes, then box 1 is allocated to rank 12, box 2 is given to rank 0, box 3 is given to rank 16 and so on.

A. Multiple boxes on a single core

It is possible to have multiple boxes per-core i.e. each sub-domain consists of multiple boxes. Assume a 2-D domain for which $n\_cells = 16$ i.e. the domain is 16x16, the number of processes is 4 and decomposed as $D_x \times D_y = 2 \times 2$, with the box size being 4x4. Thus, there are $\frac{16}{4} \times \frac{16}{4}$ boxes in all (boxes in X, Y direction are denoted by $N_x = 4$ and $N_y = 4$, respectively). The number of boxes for each process is given by $\frac{N_x}{D_x} \times \frac{N_y}{D_y}$ i.e. $\frac{4}{2} \times \frac{4}{2} = 2 \times 2 = 4$. This is shown in Figure 1. Then according to Fab numbering in BoxLib, boxes 3, 4, 7, 8 are assigned to rank 0, boxes 11, 12, 15, 16 are assigned to rank 1, boxes 1, 2, 5, 6 are assigned to rank 2 and boxes 9, 10, 13, 14 are assigned to rank 3, respectively. This is in accordance with the MPI process numbering in 2-D (or 3-D when appropriate).

B. Varying shape of box within sub-domain

When there is a single box per core, the sub-domain is the same as that box. Here the shape of the box (or sub-domain) is completely defined by the domain decomposition/-partition. When there are multiple boxes per core, the domain decomposition only determines the sub-domain shape (which in turn consists of multiple boxes). In the example shown in Figure 2a, the sub-domains have boxes of size 4x4 but it is possible to have boxes of size 2x8 or 8x2 etc. In BoxLib it is not possible to first divide the domain into sub-domains and then divide the sub-domain into boxes. Thus, we initially need to specify a box-size and then construct the sub-domain from these boxes. The process can be thought of as specifying the box-size first, then specifying the domain decomposition to create sub-domains of a specific box size i.e. implying a bottom-up approach as opposed to a top-down scheme. Figure



(a) box size=4x4 (b) box size=2x8

Fig. 2: Varying box sizes with Domain=16x16, 4 processes (2x2), and 4 boxes per sub-domain

2a shows a 16x16 domain divided among 4 cores arranged as 2x2 and each sub-domain having 4 boxes each of size 4x4. Figure 2b shows the same domain of 16x16 divided equally among 4 cores arranged as 2x2 but with each sub-domain having 4 boxes each having a size of 2x8.

VI. Test Platform and Test Problems

Our test platform is the ARC3 facility at the University of Leeds having 4056 cores of Intel Xeon Broadwell 2.2 GHz E5-2650v4 (12 cores per CPU or socket and each node consisting of 2 two such sockets) and 22 Tb of RAM in total. The network is a Mellanox FDR Infiniband interconnect with 56 Gbits/sec transfer rate and 2:1 blocking. The total flops delivered by the system is 152 Tflops/sec (35.2 Gflops/sec per core). The total memory per node is 128 GB arranged as 8 modules of 16 GB each ($\approx$ 5.3 GB per core). The Last Level Cache (LLC) memory is 30 MB shared between 12 cores (2.5 MB/core). Each core possesses an L1i/L1d cache of 32 KB and a unified L2 cache of 256 KB. The cache-line size is 64 bytes for all the caches. The associativity is 8 for L1i/L1d/L2 and 20 for the L3 cache. We use the Intel Compiler 17.0.1 and two implementations of MPI, namely, Intel MPI 2017.1.132 and OpenMPI 2.0.2.

We implement a cell-centered, Finite Difference discretization to solve the Laplace equation $\nabla^2\phi = 0$ on a unit cube with Dirichlet boundaries on uniform structured grids. The unweighted Jacobi iterative method with a 7-pt stencil is used to update the solution at mesh points. For AMR, we implement an Elliptic PDE $-\nabla^2 u = f$ on a unit cube with Dirichlet boundaries where f is chosen so that the solution $u = \tanh(k(x - 0.5))$. As before, the discretization uses the Finite Difference method with a cell-centered scheme. Since the Dirichlet boundaries do not coincide with the actual boundaries in the cell-centered scheme, they are updated as the average of the ghost cell representing the boundary and the next-to-boundary internal cell values after each iteration. Further, we use a 1-element deep ghost zone for both the problems. The refinement criterion for the first level in the AMR problem is that the y-coordinate should lie between 0.35 and 0.65 i.e. $0.35 < y < 0.65$. Whenever any cell is tagged for refinement in a block, the entire block is refined (block-structured AMR). The refinement criterion for the second level changes the range of the y-coordinate to $0.455 < y < 0.545$. Any box (or region) that is refined is marked as an *inactive* box. Any box that is not refined remains an *active box* and the solution must be updated at the mesh points constituting it. Figure 3 shows the nature of the solution and levels of the AMR problem described above when 2 levels (1 refined and 1 unrefined) are considered.

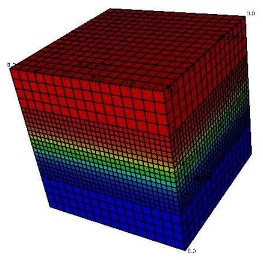


Fig. 3: Solution on a unit cube and two AMR levels for the AMR test problem with colors representing solution values from -1 (blue) to +1 (red) through 0 (green)

VII. Experimental Results

Our experiments for both single grids and AMR compare cubic partitions with non-cubic partitions to test the expandability of our hypothesis [7] that minimizing only the communication volume is insufficient for optimality. Thus, we generate several MPI Cartesian Topologies for single grids using our subroutine shown in Listing 1 and compare them against the default `mpi_dims_create()` (MDC) topology of MPI - a topology which minimizes the communication volume by producing cubic (or closest to cubic) sub-domain dimensions. For AMR, since the total boxes after refinement must remain equal for different sub-domain shapes to carry out a fair comparison, the sub-domain permutations are much more restrictive. Further, we ensure that all comparisons for all possible sub-domain shapes in an experiment are carried out on the same set of cores to remove ambiguities in execution timings due to process placement.

A. Single Uniform Grid

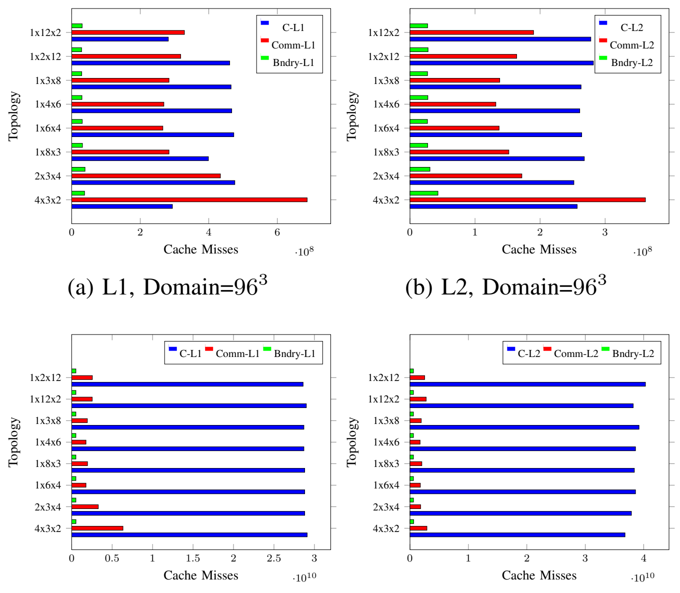
Table I compares the execution times per iteration of the topology which minimizes the communication volume, i.e. the topology returned by the default `mpi_dims_create()` (henceforth referred to as MDC) subroutine of MPI, and the best topology from 24 to 1536 MPI processes (running one process per core). It is also appropriate to compare the best timings with those from a partition using the reverse of the `mpi_dims_create()` output (referred to as Rev. MDC or Rev), as the code is written in Fortran, where the first dimension is the contiguous dimension. It can be seen from Table I that the Rev. MDC outperforms the MDC for all the domain sizes except for 768^3 for 24 cores. Further, in no

TABLE I: `mpi_dims_create()` (`MDC`) topology execution times per iteration as compared to best topology times and reverse MDC (Rev). Parenthesis indicate the number of topologies performing better than MDC and Rev, respectively. No Loop blocking/Tiling was used, Intel 17.0.1, OpenMPI 2.0.2:

Domain	Best	MDC (sec)	Best (sec)	Rev (sec)
	Cores=24	MDC=4x3x2	Rev=2x3x4	
48^3	1x12x2	3.98E-5 (18)	2.63E-5	3.25E-5 (7)
96^3	1x12x2	1.50E-4 (17)	9.14E-5	1.08E-4 (6)
192^3	2x6x2	1.95E-3 (11)	1.78E-3	1.80E-3 (1)
384^3	1x6x4	1.54E-2 (14)	1.38E-2	1.39E-2 (2)
768^3	3x8x1	1.18E-1 (8)	1.08E-1	1.45E-1 (17)
	Cores=48	MDC=4x4x3	Rev=3x4x4	
96^3	1x24x2	1.70E-4 (13)	8.77E-5	2.09E-4 (26)
192^3	2x12x2	7.07E-4 (1)	6.99E-4	8.46E-4 (12)
384^3	1x8x6	7.69E-3 (7)	7.22E-3	7.85E-3 (11)
768^3	2x12x2	5.73E-2 (6)	5.41E-2	6.00E-2 (13)
1536^3	3x16x1	6.25E-1 (23)	4.51E-1	6.25E-1 (23)
	Cores=96	MDC=6x4x4	Rev=4x4x6	
192^3	2x24x2	9.10E-4 (42)	2.80E-4	8.10E-4 (28)
384^3	4x6x4	4.98E-3 (22)	4.05E-3	4.86E-3 (18)
768^3	2x12x4	3.20E-2 (18)	2.78E-2	3.19E-2 (7)
1536^3	6x16x1	3.06E-1 (28)	2.23E-1	3.25E-1 (43)
	Cores=192	MDC=8x6x4	Rev=4x6x8	
384^3	4x12x4	2.96E-3 (12)	2.42E-3	2.68E-3 (8)
768^3	4x12x4	1.77E-2 (23)	1.49E-2	1.59E-2 (2)
1536^3	4x16x3	1.34E-1 (25)	1.14E-1	1.47E-1 (34)
	Cores=384	MDC=8x8x6	Rev=6x8x8	
768^3	4x24x4	1.01E-2 (15)	8.1E-3	1.01E-2 (15)
1536^3	4x24x4	6.20E-2 (12)	5.60E-2	6.31E-2 (12)
	Cores=768	MDC=12x8x8	Rev=8x8x12	
1536^3	4x48x4	3.45E-2 (17)	3.06E-2	3.51E-2 (17)
	Cores=1536	MDC=16x12x8	Rev=8x12x16	
3072^3	8x32x6	1.35E-1 (21)	1.20E-1	1.61E-1 (43)

case is the MDC the best topology. The number of topologies performing better than the communication minimizing topology (indicated in parenthesis) i.e. MDC or Rev. MDC is significant for most of the domain sizes and core counts. In BoxLib, by default, the *communication is not overlapped with computation*, yet the communication minimizing topology returned by the `mpi_dims_create()` is outperformed by several topologies.

We denote the MPI Cartesian topology process dimensions of the best topologies by D_{bx}, D_{by}, D_{bz} and that of the default/standard/communication minimizing `mpi_dims_create()` topology by D_{sx}, D_{sy}, D_{sz}. It can be seen from Table I that $D_{bx}D_{by} \geq D_{sx}D_{sy}$ holds with only two exceptions (Cores=24, Domain=384^3 and Cores=48, Domain=384^3. This implies that the three planes of the Compute kernel to be brought into the cache for updating a single plane of data for the best topologies are less in size than the ones which are brought into the cache with the communication minimizing topology (MDC). For all the



(a) L1, Domain=96^3 (b) L2, Domain=96^3

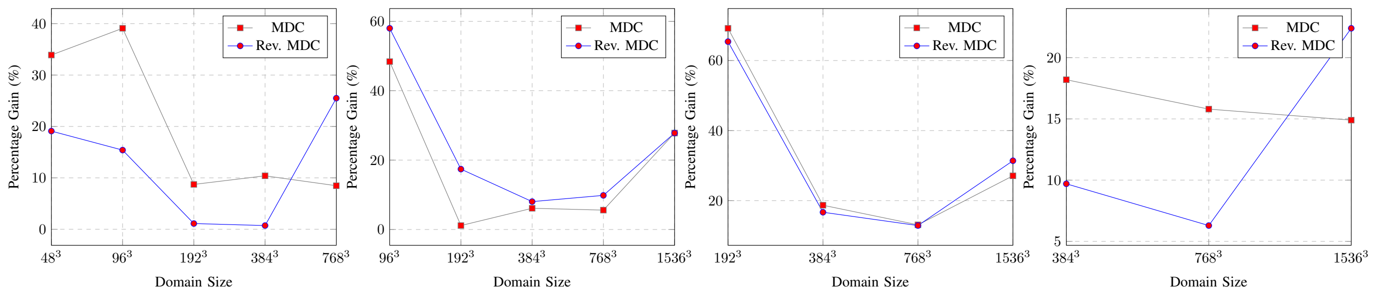
(c) L1, Domain=384^3 (d) L2, Domain=384^3

Fig. 5: L1 and L2 data cache-misses for domains 96^3 and 384^3 in the Compute (C), Communication (Comm) and Boundary update (Bndry) subroutines.

best performing topologies, $D_{by} \geq D_{bz}$ - a criterion that is in agreement with our discussion on optimal sub-domain dimensions in [20]. Further, for non-cubic sub-domains $D_{sx}D_{sy} > D_{rx}D_{ry}$, where D_{rx} and D_{ry} denote the Cartesian topology dimensions of reverse MDC (or Rev). At all processor cores and domain sizes, we were able to find topologies which performed better than the MDC or the Rev. MDC. Interestingly, even at a domain size of 3072^3, or 28 billion degrees of freedom, there exist 21 topologies which outperform the MDC and 43 topologies which performed better than the Rev. MDC. The percentage gains of the best topologies over the MDC and Rev. MDC for 24, 48, 96 and 192 cores is shown in Figures 4a, 4b, 4c and 4d, respectively. The percentage gain of the best topology over MDC ranged approximately from 1-70% and 1-66% for Rev. MDC at these core counts. The percentage gain of the best topology over the MDC for 384 cores at a domain of size 768^3 was 19.8%, and 9.67% at a domain size of 1536^3. For 768 cores the gain was 11.30%, with a similar improvement of 11.11% for a core count of 1536. Figure 5a and 5b show that the packing/unpacking cache-misses can be significant at smaller domain sizes. Figures 5c and 5d show that for large domains the compute cache-misses become more significant. In both the cases, however, the MDC topology has a higher number of cache-misses in packing/unpacking/communication due to a larger X-plane that has non-contiguous data.

B. Adaptive Mesh Refinement

We evaluate the behaviour of non-cubic blocks on domains of sizes 256^3 (see Figure 6a) and 512^3 (see Figure 6b) for



(a) 24 cores (b) 48 cores (c) 96 cores (d) 192 cores

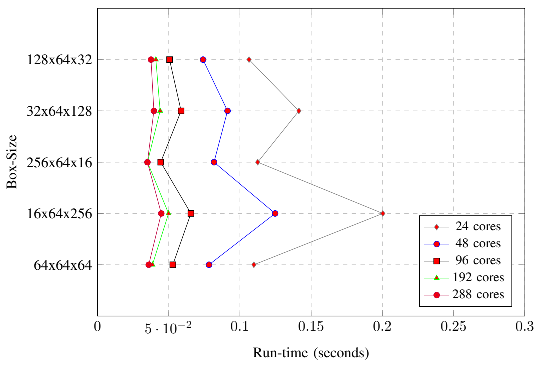
Fig. 4: Percentage gain of the best topology over MDC and Rev. MDC for varying domain sizes and cores

TABLE II: Gain percentage for the best performing topology over MDC for various core counts, MDC=Solve time/iteration in seconds, Best=Best solve time/iteration

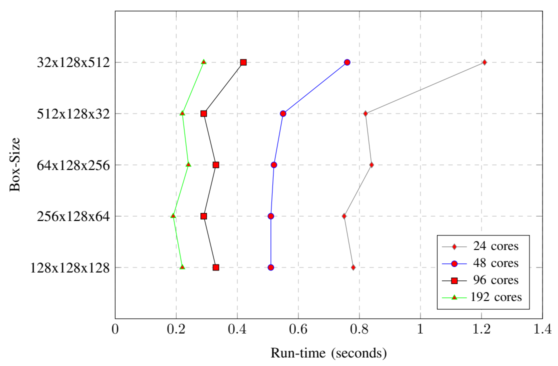
Domain=256^3, 2-levs, OpenMPI 2.0.2						
Cores	24	48	96	192	288	320
MDC (sec)	1.10E-01	7.83E-02	5.29E-02	3.86E-02	3.60E-02	3.43E-02
Best (sec)	1.06E-01	7.41E-02	4.44E-02	3.51E-02	3.51E-02	3.43E-02
Gain (%)	3.10	5.36	16.07	9.07	2.50	0.00
Domain=512^3, 2-levs, OpenMPI 2.0.2						
MDC (sec)	7.80E-01	5.10E-01	3.30E-01	2.20E-01	1.90E-01	2.00E-01
Best (sec)	7.50E-01	5.10E-01	2.90E-01	1.90E-01	1.90E-01	2.00E-01
Gain (%)	3.85	0.00	12.12	13.63	0.00	0.00
Domain=512^3, 3-levs, OpenMPI 2.0.2						
Cores	48	96	192	384	768	1176
MDC (sec)	1.73E+00	9.99E-01	7.38E-01	5.53E-01	4.28E-01	4.72E-01
Best (sec)	1.70E+00	9.99E-01	6.73E-01	5.28E-01	4.25E-01	4.36E-01
Gain (%)	1.99	0.00	8.75	4.61	0.58	7.53
Domain=512^3, 3-levs, Intel MPI 17.1.132						
MDC (sec)	1.73E+00	9.76E-01	5.90E-01	5.24E-01	3.40E-01	3.91E-01
Best (sec)	1.71E+00	9.76E-01	5.90E-01	4.70E-01	3.34E-01	3.91E-01
Gain (%)	1.19	0.00	0.00	10.31	1.76	0.00

one level of local refinement. For each of these cases, the total number of boxes at level 1 is 64, out of which 32 are refined (active at level 2) and 32 are unrefined (active at level 1 only). At level 2, there are a total of 256 boxes (as each of the 32 inactive blocks at level 1 have been divided into 8 boxes). Thus a total of 288 active boxes are updated for the solution. Considering a three level problem (see Figure 6d and Figure 6c), 128 boxes out of a total of 256 boxes at level 2 are refined again to give 128x8=1024 active boxes at level three. Thus, in the three level problem, we have a total of 32+128+1024=1184 active boxes which are updated. While varying the box-shape, the volume of the box is kept constant but the box-dimensions are changed.

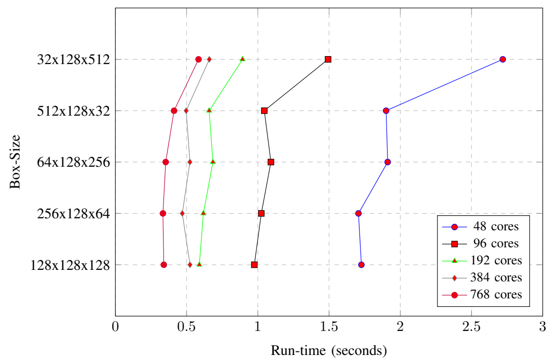
Figure 6b shows the performance of various box-shapes for a domain of size 512^3 and a two level problem (1 refined and 1 unrefined). It can be seen that a non-cubic box shape of 256x128x64 outperforms (or equalizes) the performance of the cubic block of 128x128x128 from 24 to 192 cores. Since in Fortran the first dimension is the contiguous data dimension, a box of shape 256x128x64 has twice the data points in the contiguous dimension as a box of shape 128x128x128. Thus, we can expect a better cache utilization when packing/unpacking the non-cubic block as the X-plane i.e. 128x64 is half the size of the cubic-block i.e. 128x128. The X-plane is the plane perpendicular to the direction of the unit stride dimension. Another topology that outperforms the cubic block is that with a box-shape of 512x128x32 at 96 and 192 cores. The total number of communication elements grow with an increasing size of a particular dimension in a non-cubic block, thus the unit-stride dimension cannot arbitrarily grow for large domains. There is always a trade-off between minimizing cache-misses and communication elements due to which we do not see a consistent performance at all process counts. Figure 6a also portrays the same picture in the sense that the cubic box-shape given by the communication minimizing topology is *not the optimal* choice at *all* core counts.



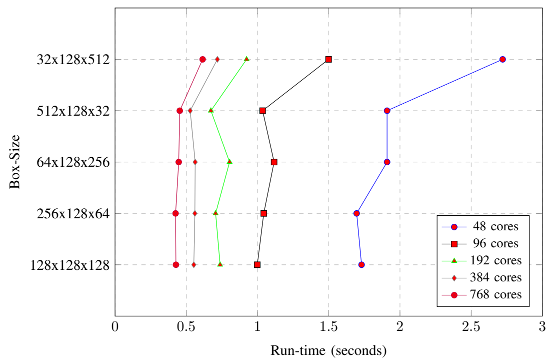
(a) 2 levels, domain 256^3, OpenMPI 2.0.2



(b) 2 levels, domain 512^3, OpenMPI 2.0.2



(c) 3 levels, domain 512^3, Intel MPI 2017.1.132



(d) 3 levels, domain 512^3, OpenMPI 2.0.2

Fig. 6: Strong Scaling (time/iteration) two and three level AMR problem with boxes of varying shapes but equal volume, optimization flags : `-O3 -xHost -ip -align array64byte`, Compiler used : Intel 17.0.1

Figure 6c and 6d show the performance of various block shapes for the AMR test problem when the number of levels is three (i.e. two levels of local refinement). It can bee seen from Figure 6c that the cubic block size is not optimal for 48, 384 and 768 cores and the performance difference between it and the optimal non-cubic block is approximately 1.18 - 10.30%. For Figure 6d with OpenMPI, the range of performance difference is 1.9 - 8.74%. The maximum and minimum ratio of the execution time per solve iteration when using OpenMPI, compared to when IntelMPI is used, is 1.27 and 0.99, respectively, for a domain of size 512^3 (see Figure 6c and Figure 6d). The ratio increases as the number of processes increase from 48 to 1176. It is not correct to say that one MPI implementation is faster than the other as the allocation of nodes changes between using the two MPI implementations. Table II shows the percentage gain of the best topology over the cubic sub-domain i.e. the sub-domain produced by the `mpi_dims_create()` topology for various core counts and varying levels of AMR for two domain sizes, namely, 256^3 and 512^3.

VIII. Discussion

In [7] and [20] we formulated a strategy for minimizing the cache-misses of a sub-domain and showed the superiority of such partitions by experimenting on single grids and Geometric Multigrid, respectively . Overlap of communication with computation formed a *significant* part of our analytical derivation for cache-minimizing topologies. The reason is that when communication is overlapped with computation, both while packing/unpacking and communicating data, the next-to-halo layers are accessed *separately* after the halo data arrives. This has the advantage of MPI advancing its communication progress engine while the serial computing thread updates the Independent Computation kernel but at the same time suffers from a disadvantage that the next-to-halo layers now cannot be updated along with the Independent Computational kernel, resulting in extra cache-misses.

Our model holds only partially when evaluating single uniform grids and AMR in BoxLib. Since the codes were in Fortran, we also took into account the reverse communication minimizing topology (Rev. MDC or Rev) but our experimental evaluation always found topologies which outperform *both* the MDC and Rev. MDC for *all* the cases considered. For AMR codes, there existed cases where the MDC was outperformed by specific non-cubic sub-domains, thus, establishing that the MDC is *not generally the optimal* choice at all domain sizes or core counts. We shed light on the plausible reasons for the partial correctness of our model in the BoxLib setting. In

BoxLib, communication of the halo zones is not overlapped with computation and further the packing and unpacking of data from the boxes does not use derived data types. This completely eliminates the cache-misses that we calculated separately for the Dependent Planes in [7]. It is also difficult to estimate the size and the consequent effect on the application performance because of the metadata that BoxLib maintains for both single grids and AMR. The user does not have any control over the distribution of boxes in AMR and this is completely controlled by BoxLib using the *Knapsack* or *Morton* ordering with a dynamic switching scheme implemented to choose the appropriate algorithm. Since boxes are distributed per-level, BoxLib does not distinguish between inactive or active boxes. Thus, there is a large probability that the active boxes may not be load-balanced. Furthermore, since the load balancing algorithm used by BoxLib takes account of the coordinates, the number of boxes per core can change when the shape of the box is changed, though the volume remains constant.

IX. Conclusion

In this work we tested our high level model for predicting optimal domain partitions on uniform structured 3-D grids developed in [7] and [20] for the more general cases of an AMR solver. The model in [7] demonstrated that communication minimization is *not the sole criterion* upon which mesh partitioning should be based and that it is essential to take into account the cache-misses for optimality. Due to the combinatorial explosion of the possible topology space, optimality in the current work implies the best topology of those considered. We undertook our assessment based upon the model in [7], [20] and the use of the open source BoxLib library which we described briefly in the current work.

We have been able to demonstrate that the cache-misses minimizing topologies outperform the default communication minimization topology in *all* the cases that we tested for uniform single grids. To this effect we implemented an MPI Cartesian topology of processes and replaced the default box distribution policy of BoxLib with it. The best topologies on uniform grids with only two exceptions demonstrated that $D_{bx}D_{by} \geq D_{sx}D_{sy}$, $D_{by} \geq D_{bz}$ and $D_{bx} \leq D_{sx}$ is needed to outperform the MDC and Rev. MDC. The performance gain range of 1-70% and the significant number of topologies outperforming the default topology showed that for single grid applications, using non-cubic blocks/boxes is the optimal choice. Further, it is *possible* to obtain increasing gains from the cache-minimizing topologies while performing Strong Scaling, both for uniform single grids and AMR applications (as illustrated in Table I and Table II). Thus, even in the absence of overlap of communication with computation, our hypothesis remains true. When the use of non-cubic boxes is extended to the AMR test code, the performance gains are still observed, however they typically fall down to less than 10%. This can be attributed to the change in communication pattern, non-overlap of communication with computation, the load-balancing criterion, the increase in metadata and automatic box-distribution strategy in BoxLib. We thus conclude that the communication minimization topology/cubic-partitions are *not always* optimal and the emphasis/efforts should shift on obtaining a balance between minimizing *both* cache-misses and communication volume for optimality.

Acknowledgement

One author would like to thank Ann S. Almgren and Weiqun Zhang for their time and guidance regarding BoxLib.

References

[1] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[2] J. Rantakokko and M. Thuné, "Parallel structured adaptive mesh refinement," in *Parallel computing*, pp. 147–173, Springer, 2009.

[3] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[4] J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang, "Boxlib users guide," *github. com/BoxLib-Codes/BoxLib*, 2012.

[5] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[6] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer, "Paramesh: A parallel adaptive mesh refinement community toolkit," *Computer physics communications*, vol. 126, no. 3, pp. 330–354, 2000.

[7] G. Saxena, P. K. Jimack, and M. A. Walkley, "A Cache-aware approach to Domain Decomposition for Stencil-based Codes," in *International Conference on High Performance Computing and Simulation (HPCS 2016)*, pp. 875–885, 2016.

[8] Message Passing Interface Forum., "MPI: A Message-Passing Interface Standard, Version 3.0.," September 2012.

[9] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, *et al.*, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014.

[10] W. L. Briggs, S. F. McCormick, *et al.*, *A multigrid tutorial*. Siam, 2000.

[11] L. F. Diachin, R. Hornung, P. Plassmann, and A. Wissink, "Parallel adaptive mesh refinement," *Parallel processing for scientific computing*, vol. 20, p. 143, 2006.

[12] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for amr applications-design document," 2000.

[13] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured amr calculations using the samrai framework," in *Supercomputing, ACM/IEEE 2001 Conference*, pp. 22–22, IEEE, 2001.

[14] "BoxLib Case Study." http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/application-case-studies/boxlib-case-study/. Accessed: 2017-04-31.

[15] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S156–S172, 2016.

[16] "MAESTRO." https://ccse.lbl.gov/Research/MAESTRO/. Accessed: 2017-04-16.

[17] "CASTRO." https://ccse.lbl.gov/Research/CASTRO/. Accessed: 2017-04-10.

[18] "LMC." https://ccse.lbl.gov/Research/Combustion/. Accessed: 2017-04-11.

[19] "GitHub - BoxLib-Codes/BoxLib: Block-Structured AMR Framework." https://github.com/BoxLib-Codes/BoxLib. Accessed: 2017-04-22.

[20] G. Saxena, P. K. Jimack, and M. A. Walkley, "A quasi-cache-aware model for optimal domain partitioning in parallel geometric multigrid," *Concurrency and Computation: Practice and Experience*, 2017. In Press.