Enabling Tractable Exploration of the Performance of Adaptive Mesh Refinement

Courtenay T. Vaughan and Richard F. Barrett
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
Email: ctvaugh@sandia.gov, rfbarre@sandia.gov

***Abstract*—A broad range of physical phenomena in science and engineering can be explored using finite difference and volume based application codes. Incorporating Adaptive Mesh Refinement (AMR) into these codes focuses attention on the most critical parts of a simulation, enabling increased numerical accuracy of the solution while limiting memory consumption. However, adaptivity comes at the cost of increased runtime complexity, which is particularly challenging on emerging and expected future architectures. In order to explore the design space offered by new computing environments, we have developed a proxy application called miniAMR. MiniAMR exposes a range of the important issues that will significantly impact the performance potential of full application codes. In this paper, we describe miniAMR, demonstrate what is designed to represent in a full application code, and illustrate how it can be used to exploit future high performance computing architectures. To ensure an accurate understanding of what miniAMR is intended to represent, we compare it with CTH, a shock hydrodynamics code in heavy use throughout several computational science and engineering communities.**

***Index Terms*—High performance computing; adaptive mesh refinement; scientific applications; parallel architectures; performance evaluation.**

I. Introduction

There is a long history of finite difference and finite volume algorithms being used to study various physical phenomena in a breadth of scientific and engineering domains. When the algorithm is applied to a static grid, the mesh resolution needs to be universally fine enough to capture the phenomena being studied anywhere in the domain. As illustrated in Figure 1, incorporating Adaptive Mesh Refinement (AMR [8], [9]) focuses attention on the most critical parts of a simulation, enabling increased numerical accuracy of the solution while reducing memory requirements. However, this adaptivity comes at the cost of increased complexity, and runtime behavior becomes strongly tied to the particular problem being examined. This is expected to be especially challenging given the increased architectural complexities of emerging and expected future architectures.

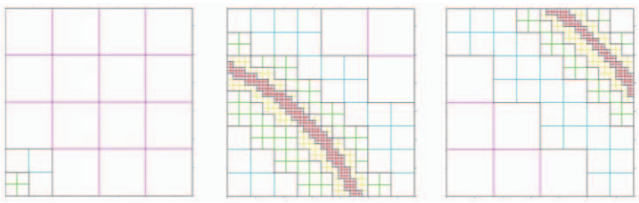


Fig. 1. Adapting mesh as wave front passes through a domain

In order to explore the performance design space of AMR-based applications, we have developed a proxy application called miniAMR. MiniAMR is a standalone code designed for the exploration of some important performance issues prevalent in finite difference or volume codes that use AMR. It is free of constraints of any specific physical behavior, with refinement driven by an object moving through the domain, and is therefore not intended to be representative of the computations found in a complex application. This provides a simple means of observing and experimenting with interprocess communication strategies, load balancing schemes, and refinement strategies. The computation is a simple difference stencil, and is therefore not intended to be representative of the computations found in a complex application. That said, it can provide a tractable means for exploring the computational syntax and semantics of different programming models, mechanisms, and languages.

In this paper, we describe miniAMR, demonstrate what it is designed to represent in a full application code, and illustrate how it can be used to exploit future high performance computing architectures. To ensure an accurate understanding of what miniAMR is intended to represent, we compare it with CTH, a shock hydrodynamics code in heavy use throughout several computational science and engineering communities.

A. Related work

AMR was first introduced by Berger et.al. [8] [9]. Several libraries, including BoxLib [6], Chombo [10], SAMRAI [15], and others, allow code developers to develop an AMR application.

MiniAMR is available in the Mantevo project suite of miniapps. Mantevo includes a related miniapp, called miniGhost [4]. Operating on a static grid, miniGhost is primarily designed for explorations of the halo exchange [1]. We are using miniGhost to explore areas such as the task parallel over decomposition programming model, where the complexity of miniAMR would be difficult, but is a natural extension that we will explore. CloverLeaf [13] is a Mantevo miniapp that includes meaningful physics, solves the compressible Euler

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.129

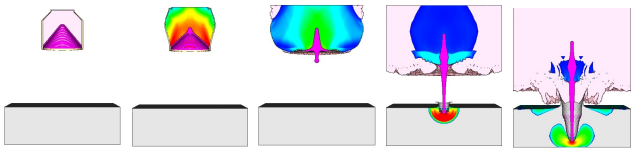


Fig. 2. CTH shaped charge simulation. Time progresses left to right.

equations on a two dimensional Cartesian grid, using an explicit, second-order accurate method. CleverLeaf [5] adds a patch based AMR scheme to CloverLeaf using the SAMRAI toolkit.

MiniAMR is simpler than benchmarks, in keeping with the Mantevo goal of providing a tractable means for modifying the code, including swapping in new communication, refinement, and load balancing strategies. It would also be relatively straightforward to add more meaningful computations by a researcher looking at more specific physics capabilities.

II. CTH

To ensure meaningful representation of the behavior of miniAMR, we compared it with a full, complex application program. CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories [14]. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials, using second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results.

CTH can be run in flat mesh mode, where each processor has one rectilinear block of the problem, which can have tens of thousands of cells. In this mode, each processor communicates with six neighbors, one in each direction along Cartesian axis. This is the communication pattern that is modeled by miniGhost, previously validated as described in [1]. For that, we use the shaped charge problem, illustrated in Figure 2, which involves four materials, inducing a boundary exchange of 40 variables. Several times each time step boundary information is aggregated and exchanged with up to six neighbors in the grid of processors. There are 89 collective operations (mostly `MPI_Allreduce` on 8-byte data) during each time step [17].

CTH can also be run in AMR mode, using an octree based AMR scheme, where each processor has a number of smaller blocks, each of which has a few hundreds of cells. As the calculation progresses, the number and placement of these blocks in the calculation can change. Each of these blocks has to communicate with its neighboring blocks in the mesh, so each processor ends up performing communication within the processor as well as to some number of neighboring processors, which can change as the simulation progresses. Following a description of miniAMR, comparisons are made with CTH.

III. MiniAMR

The general structure of the miniAMR code is shown in Figure 3. Each processor begins with one block per processor,

```
for some number of timesteps do
  for some number of stages do
    communicate ghost values between blocks
    perform stencil calculation on variables
    if stage for checksums then
      perform checksum calculations
      compare checksum values
    end if
  end for
  if time for refinement then
    refine mesh
  end if
end for
```

Fig. 3. miniAMR code flow

applying refinement to get to the desired resolution. (In a full application, the inner **for** statement would be replaced with the physics).

Computation is performed on a block by block basis, with each block containing the necessary ghost cells. The computation is a seven point stencil, where the result for a cell is the average of its value with the six values that are in the cells whose index differs by one in each of the three Cartesian directions.

Reflective boundary conditions are applied to the physical boundary of the global domain. Thus a constant sum across each variable is required for correctness, which is optionally periodically checked during execution.

CTH, as well as other application codes, uses several different stencils, including portions where results for a cell depend only on the variables present in that cell, portions where the result depends on the value in that cell and all of its 26 neighbors, and several cases in between. For miniAMR, this 27 point stencil will not work correctly with blocks of different levels of refinement since the values of the corner points in the halo can not be determined so that the averaging conserves the sum of the values in the cells. That said, miniAMR computation is similar to CTH in that it computes on all of the real cells in a block and therefore accesses all of the memory associated with the blocks, which is arguably more important, with regard to execution time, than the actual computation.

Refinement is driven by geometric shapes moving across the domain. As these shapes move through the mesh and change size, their boundary or volume can be used to define which blocks are refined. These can be used to model physical structures in the problem, such as modeling a shock wave with an expanding sphere.

The blocks are stored in an array of structures, in which each block is represented by a structure that has its number, level of refinement, information about its parents, information about its neighboring blocks, position, a pointer to the array of variables that are associated with that block, and other information about the block. Inactive blocks are given a negative number, and are free to be reused. When a block is refined, its information

is recorded into a parent block, which stores some of the information about the block as well as information about the block's children.

Communication is done between all blocks which share a face or portion of a face. Blocks that are adjacent to each other will differ by at most one level of refinement. Adjacent blocks on the same processor share data using a memory copy. Blocks on different processors exchange data using MPI non blocking point-to-point communication. When two blocks with different levels of refinement are communicating, the value for a ghost cell in the block that has the lower level of refinement is averaged from the corresponding eight cells in the block with the higher level of refinement. Likewise, the value for a ghost cell in the block that has the higher level of refinement are copied from the corresponding cell in the block that has the lower level of refinement.

The data structure for each block contains information on the neighbors for each of its faces. On-processor communication is accomplished by sweeping across the blocks and exchanging ghost information with the blocks that are on the same processor. For interprocessor communication, each processor maintains a list for each communication direction, with a list of processors and list of faces associated with blocks to communicate. Lists are sorted so that each processor constructs a message of ghost values for block faces that it sends and the other processor can decipher since its communication list is sorted in the same order.

Communication is performed in three stages, one for each direction of the Cartesian coordinates. For the 27-point stencil, which runs correctly on a uniform grid, this removes the need for direct communication with adjacent diagonal processors, by bringing that data along in the coordinated three-way process. Each processor begins by going through its communication list in each direction, posting required receives, and then packing and sending its data, using non blocking sends, to neighboring processors. Then the communication between blocks on the same processor is performed, allowing messages to work their way through the machine, enabling communication cost hiding. During this phase, faces on the boundary apply reflective boundary conditions. Finally, each processor waits for outstanding messages, with faces unpacked and placed in the appropriate blocks are receives are completed.

When a block is refined, it is decomposed into 8 blocks which are half of the size of the original block in each dimension, but with the same number of cells as the original block. This results in new cells that are half of the size in each dimension than the original cells. Communication information for the parent block is modified and transferred to the 8 child blocks. On-processor communication updates child and neighbor blocks. Off-processor communication updates the communication list by deleting the parent's communication entry and adding ones for each of the child blocks which have a face that is a subset of the parent block's face. Before the blocks are refined, information is exchanged through the communication list of blocks being refined. Thus when a block is being refined, its neighbors on other processors can also update their communication list to account for the refinement. The process is similar when blocks are coarsened. When a block is refined, a parent block is created which holds the processor location of the child blocks, so when those children are coarsened, the block can be put back together. Once all child blocks belonging to a parent block have been coarsened, all of those blocks will be moved back to the processor on which the parent block resides. Once that is done, the child blocks are consolidated back into the parent block. When the child blocks are moved back, the communication lists are updated and consolidated.

Next, the workload is re-balanced using Recursive Coordinate Bisection (RCB) [7]. For each step, a direction and a number of divisions is chosen and the blocks are sorted in that direction and divided into equal sized sets. Divisions are based on the prime factorization of the total number of processors and the number of blocks in each direction of the original mesh. Each set corresponds to a group of processors such that a processor maintains its region throughout the calculation. The sorting is done by binning the centers of the blocks in the current set in the chosen direction so that each processor can determine the division points. The number of positions that the centers and corners of the blocks can be located at is

$$b * (2^{(r+1)}) + 1,$$

for b blocks in the original mesh in each direction and r maximum levels of refinement.

If there are multiple blocks at the binning position chosen for the cut, then those blocks are binned again in one of the other directions, and if there is a tie for the cut, then the blocks that tied are binned in the third direction, where there is guaranteed to be no ties and a cut can be made. Each block in this process is represented by a smaller data structure, a "dot" which contains the block number, the processor it started on, and the integer coordinates of the center of the block, which is then passed to a processor in the division that is chosen. This process is repeated with each of the sets of processors, until the dots eventually get to the processors that their corresponding blocks will be assigned to. The processor number where the dots end up is then communicated back to the processor where the corresponding block is located, and the blocks are be packed and sent to their new processors.

Once it is known which blocks are being moved, this information is sent to all of the neighboring blocks using the list for communicating ghost face values. This information is used to delete information from the communication list on each processor for the blocks being moved from that processor and to modify the communication list for those blocks that are being moved from one neighboring processor to another neighboring processor. Once the blocks are moved, the communication links with the blocks on its new processor are established, and its neighbors are added to the new processor's communication list.

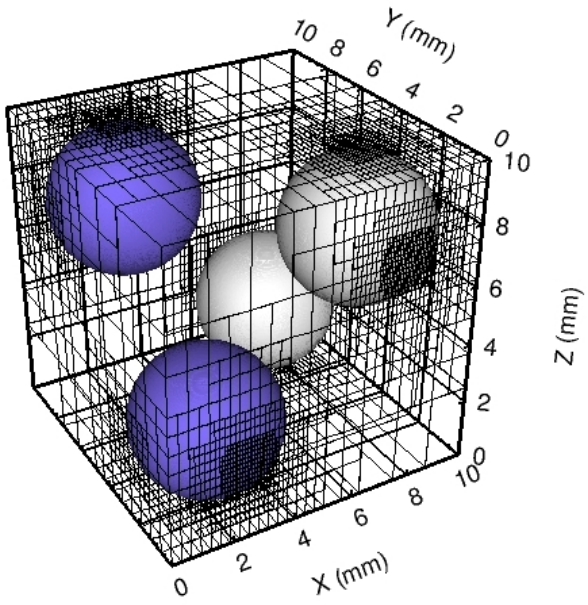


Fig. 4. CTH visualization of four spheres problem

IV. Experiments

We compared the performance of miniAMR with CTH using two distinct problem definitions. The first is a sphere hitting a block of material at an oblique angle producing a shock wave. The deforming sphere, the crater it leaves, and the shock wave in the block are all regions of the problem that are refined by AMR. The second problem, illustrated in Figure 4, consists of four spheres moving in the mesh such that they will not collide The dark blue spheres are moving in the positive X direction, while the light gray spheres are moving in the negative X direction. The boundary of the spheres is the portion of the mesh that is refined. Both of these problems have two materials and have 32 variables in the boundary exchange and 62 reductions per time step. The relationship between the number of materials, the number of variables communicated, and the number of reductions is not straightforward and depends on several factors.

MiniAMR has been tested on several computing platforms. Work presented here is performed on Cielo [12], an instantiation of a Cray XE6, which gives us a very large number of processors. Cielo is composed of 8,944 compute nodes, connected using a Cray custom interconnect named Gemini, and a light-weight kernel (LWK) operating system called Compute Node Linux. Each node consists of two oct-core AMD Opteron Magny-Cours processors, for a total of 143,104 cores. Each core has a dedicated 64 kByte L1 data cache, a 64 kByte L1 instruction cache, and a 512 kByte L2 data cache, and the cores within a NUMA node share a 6 MByte L3 cache (of which 5 MBytes are user available). Each Magny-Cours processor is divided into two memory regions, called NUMA nodes, each consisting of four processor cores and 8 GBytes of DDR3 1333 MHz memory. Thus each compute node consists of 16 processor cores, 32 GBytes of memory, evenly divided among four NUMA nodes, which are connected using HyperTransport version 3. The links between NUMA nodes run at 6.4 GigaTransfers per second (GT/s). For convenience of access, we also used a surrogate for Cielo called Muzia. Muzia serves as a small scale testbed for Cielo, and therefore its configuration and environment mirrors Cielo.

	Sphere/Block		Four Spheres	
	CTH	miniAMR	CTH	miniAMR
Calculation	27.3	35.4	29.5	30.1
Communication	61.5	64.0	67.3	69.6
Refinement	11.2	0.6	3.2	0.3

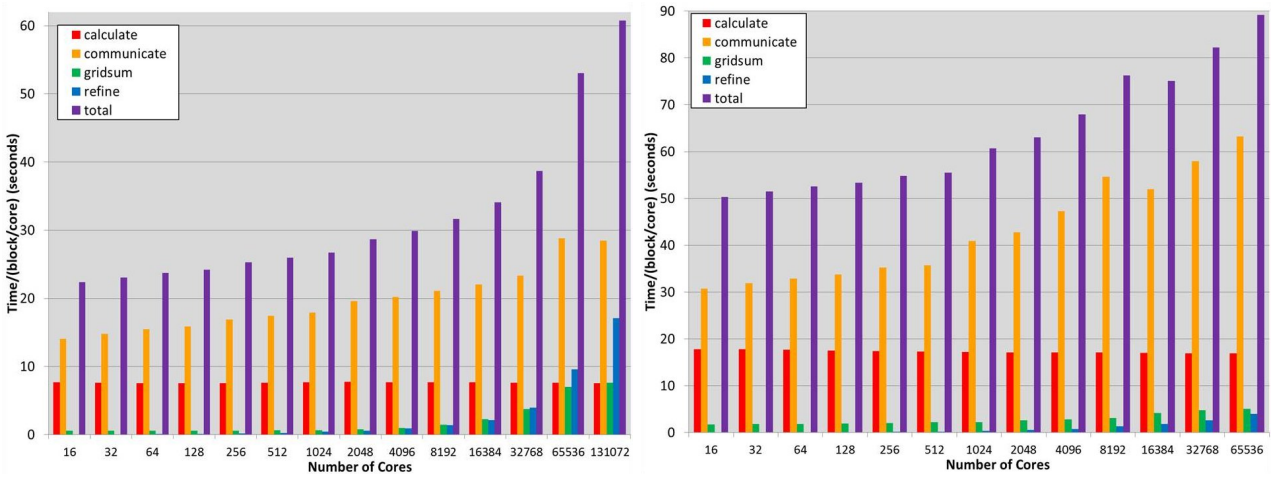
TABLE I
Time proportions on 128 cores. All values percentage (%)

AMR presents challenges with regard to weak scaling behavior due to the dynamic nature of problem execution. In particular, the workload per processor will change based on the number of processors employed, and therefore care must be taken in analyzing profiled performance. Scaling performance of the "sphere hitting a block" problem is shown in Figure 5(a). The same problem description was used for each of the runs and the number of starting blocks was changed to match the number of processors the problem was run on. This results in the average number of blocks ranging from 248 on 16 processors to 30 on 128K processors. The times are scaled by the average number of blocks per processor.

There are several things to note with this graph. The overall time grows moderately up to about 32K processors and then starts growing faster. The time for the calculation portion of the code is fairly constant over the range, which is expected since the time is scaled by the average number of blocks per processor and the same number of calculations are performed on each block. Communication time gradually rises as the number of processors increases, with the bigger increases attributable to refinement and summation across the grid (`gridsum`). This is also expected since the number of blocks per processor decreases as the number of processors rises.

To further illustrate, we ran miniAMR using an AMR version of the CTH shaped charge problem, which was visualized above in Figure 2. Scaling results are shown in Figure 5(b). In this simulation, the average number of blocks per processor ranges from 1009 on 16 processors to 415 on 64K processors. As a result, the problem scales better with the increase in time not being as dramatic as the run scales to larger numbers of processors. Also note that the time for the `gridsum` operation and refinement do not rise as much as with the sphere and block problem.

The sphere hitting a block problem that we used for the first scaling study matches a CTH problem. The time spent in communication dominates in miniAMR as well as for CTH. Table I shows the breakdown in time for both codes when run on 128 Muzia cores. Communication is simply the boundary



(a) Sphere hitting block

(b) Shaped charge

Fig. 5. miniAMR scaling

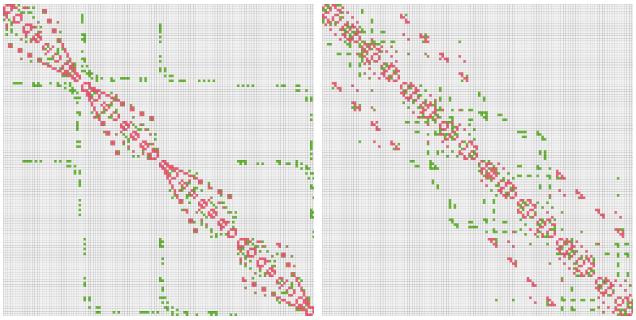
communication and the collective operations are included in the calculate portion, since they are interspersed in the code in CTH. For both problems, CTH and miniAMR spend about the same percentage of the time in the calculate and communicate phases of the codes. Using more time for communication while using less memory for the refined portion of the simulation is one of the tradeoffs of using AMR.

The other thing to note is that CTH spends significantly more time in the refinement stage. Part of this difference is that CTH performs several load balancing steps during this work; we continue to explore alternative reasons for this.

The graphs shown in Figure 5 show miniAMR scaling is similar for both problems, up to a large number of processors. We have also looked at the CTH percentages for a few other numbers of processors and found that those are also consistent.

In the four spheres problem, the spheres do not interact or deform, and therefore we expect (and see that) mesh dynamics are similar in CTH and miniAMR. For miniAMR, we refined on the boundaries of the spheres, while for CTH, we refined those cells which had some material, but were not full, which is effectively the same as refining on the boundary. For this problem, the number of blocks should remain close to constant over the length of the run. We ran both codes to a similar end point and found that CTH had an average of 685.8 blocks per processor over the coarse of the simulation, while miniAMR had an average of 669.3 blocks per processor, a 2.5% difference. This is close given that the process of determining which blocks to refine or coarsen differs between the codes. Further, with CTH, we averaged 18.4 messages per processor per communication stage with an average message size of 503 kilobytes (KBytes). With miniAMR, we averaged 17.3 messages per processor and had an average message size of 593 KBytes. Again, these numbers are in reasonable agreement.

Point-to-point communication patterns for CTH and miniAMR are somewhat different, as illustrated in Figure 6 by the communication matrices for the boundary exchange of the "four spheres" problem on 128 cores (we have data for these problems on other numbers of cores, but this size works well for illustrative purposes). The processor in row i communicates with the processor in row j, with green representing a small number of messages and red representing a larger number of messages. Empty positions in the matrix represent no communication between those processors. Since the communication pattern for both codes changes as the simulation progresses, we chose a time in each calculation such that both codes have progressed to a similar spot in the simulation. Point-to-point communication patterns for the "sphere hits block problem", also on 128 cores, are illustrated in Figure 7.

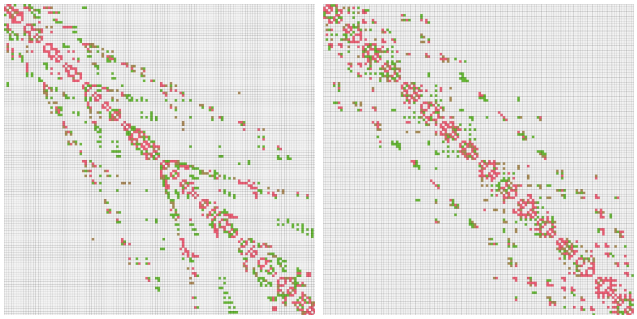


(a) CTH

(b) miniAMR

Fig. 6. Four spheres communication matrices

Further investigation showed that three factors contribute to the differences, each relating to the implementation of the RCB algorithm in the load balancing phase. In CTH, when there are several blocks that lie along the cut between groups of blocks that will be assigned to one processor set or another,

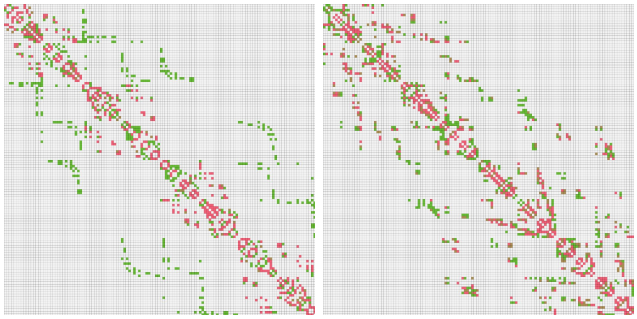


(a) CTH (b) miniAMR

Fig. 7. Sphere hits block communication matrices

the blocks are effectively assigned to one set or the other in a random fashion. In miniAMR, those blocks are assigned based on their position in the cutting plane so that blocks that are nearby are more likely to be assigned to the same set.

The second difference is that CTH only allows a certain percentage of blocks to be moved during any refinement step in order to limit the size of the messages that are being sent. This results in random blocks not being moved to the proper processor and has the effect of a processor communicating with more other processors. The simulation time in the above figure was chosen so that this effect was minimized, or else the matrix would have more random entries in it. The third difference is that CTH allows the cut direction for each group of blocks to be determined when the cut is being made, while miniAMR determines the order of cuts once at the beginning. This is what causes the miniAMR matrix to have more structure with the diagonals that are offset from the main diagonal. We implemented those changes into miniAMR and the results are shown in Figure 8. These

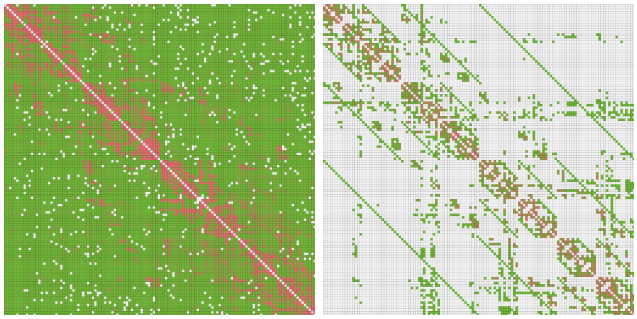


(a) Four spheres (b) Sphere hits block

Fig. 8. Modified communication matrices miniAMR

communication matrices have closer structure to those from CTH. The differences can be explained in that the two codes are not making all of the same choices for each of the cuts or for which block is assigned to each processor in the simulation. For the four spheres problem, these changes cause the refine time in miniAMR to nearly triple since the number of blocks that are moved between processors is multiplied by a factor of over 8. The number of communication partners increases to 20.4 messages per processor per communication stage and the average message size increases to 622 KBytes. As a result, the communication time also increases by 14%.

The communication matrices for the refinement portion of the codes for the sphere hits block problem, as illustrated in Figure 9, show a large difference between the two codes. Each matrix in the figure has the communication matrix for



(a) CTH (b) miniAMR

Fig. 9. Communication matrices for refinement

the boundary exchange, shown above, embedded into it since those neighbors have to be communicated with in order to pass information about which blocks are being refined. This information is necessary to ensure that when those blocks are refined, the neighbor lists on neighboring blocks can be updated correctly. If this is not done, then the mesh can become misconnected and the calculation will not be able to proceed. This portion of the communication matrix will be different since it is done with the default settings in miniAMR. The rest of the structure in the matrices include messages to perform the load balancing, messages to exchange blocks between processors, and message to and from parent blocks. The diagonal lines in the miniAMR communication matrix is communication for load balancing using RCB where the blocks are being divided into two groups of processors. One processor in a group of processors is communicating with one processor in the other group of processors. In contrast, CTH uses a library to do the load balancing and that library uses collective operations instead of point to pooint messages. The messages to and from parent blocks are necessary to allow blocks to be recombined when the region no longer needs to be refined. Since CTH load balances the parent blocks, these blocks are evenly distributed throughout the mesh, and this explains the nearly all to all communication pattern in the CTH matrix. There are not a lot of messages to and from the parent block, so the all to all pattern is in green with a fewer number of messages. In contrast, in miniAMR, the parent blocks stay on the processor where they are created and that tends to be closer to the processors where the child blocks are distributed to. Another difference is that CTH makes more passes over the blocks to do load balancing since it limits the number of blocks that can be exchanged in each pass, while miniAMR has no

limit to the number of blocks exchanged. For this simulation, CTH uses 34 times as many messages and communicates 56 times as much information on average for the refinement steps than miniAMR. This partially explains the above statistics that show that CTH spends more time performing refinement.

In order to compare the calculations in the two codes, we profiled portions of both codes using PAPI counters. This was run with the sphere hits block problem. For CTH, we profiled one of the convection routines. There are three convection routines and together they use between 20% and 25% of the compute time for the problems that we are running. These routines are fairly representative of the computations in CTH and all three of them have similar profiles. For miniAMR, we used the stencil calculation. What we find is that although CTH has better cache utilization, with a 99.1% L1 and 57.6% L2 cache hit rate, than miniAMR, with a 98.8% L1 and 9.8% L2 cache hit rate, CTH achieves a lower floating point operation rate, 22.4 MFLOPs, than miniAMR, 475.4 MFLOPs. Another difference is that CTH routines do several conditional calculations in the innermost loop resulting in several branches per cell, while miniAMR is a simple stencil calculation with no conditionals.

V. Summary and Future Plans

MiniAMR, a new miniapp in the Mantevo suite, enables tractable exploration of some important performance issues in Adaptive Mesh Refinement finite difference and volume codes. We have compared miniAMR to CTH in several regards and found that it models the communication portion of CTH in representative ways. We have also shown places where the two codes differ, such as the communication patterns for the refinement operation and differences in the profiling of the calculation portion of the simulations. The latter is not unexpected since we were not trying to simulate any of the physics from CTH in the calculation in miniAMR.

Additionally, some design choices in miniAMR are intended to illustrate the value of alternative strategies, for consideration by CTH developers and others. Often application design choices are out-dated or were made for convenience, and miniAMR can be used to drive modifications to the application.

That said, miniapps are not intended to perfectly represent a particular application, but are instead designed to enable tractable exploration of relevant performance behavior [2]. It is imperative that the user understand what those differences are, and how the miniapp may be used to effectively manage performance in their code. Toward that end, we are encouraged that differences between CTH and miniAMR were relatively easy to understand and attribute. Further, as with any well designed miniapp, the modular nature of miniAMR allows a user to easily modify and extend its capabilities.

Future plans include implementation of an MPI-OpenMP hybrid scheme (in progress), a task parallel implementation (the basis of which is described using miniGhost in [3][16]), and incorporation of other load balancing schemes, such as those found in Zoltan [11].

Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] R.F. Barrett, P.S. Crozier, M.A. Heroux, P.T. Lin, T.G. Trucano, and C.T. Vaughan. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. *Journal of Parallel and Distributed Computing*, 75, 2015.

[2] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating An Evolutionary Fast Path to Exascale. In *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS12)*, 2012.

[3] R.F. Barrett, D.T. Stark, C.T. Vaughan, R.E. Grant, S.L. Olivier, and K.T. Pedretti. Toward an Evolutionary Task Parallel Integrated MPI+X Programming Model. In *Proceedings of the Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, 2015.

[4] R.F. Barrett, C.T. Vaughan, and M.A. Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing; Version 1.0. Technical Report 2012-10431, Sandia National Laboratories, 2012.

[5] D.A. Beckingsale, O.F.J. Perks, W.P. Gaudin, J.A. Herdman, and S.A. Jarvis. Optimisation of Patch Distribution Strategies for AMR Applications. *Lecture Notes in Computer Science*, 7587:210–223, 2013.

[6] J. Bell et al. BoxLib Users Guide. Technical report, Lawrence Berkeley National Laboratory, 2012.

[7] M.J. Berger and S.H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, 36:570–580, May 1987.

[8] M.J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989.

[9] M.J. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations . *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[10] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document. Technical report, 2003.

[11] K. Devine, B. Hendrickson, E. Boman, M. St.John, and C. Vaughan. Zoltan: A Dynamic Load-Balancing Library for Parallel Applications; User's Guide. Technical Report SAND99-1377, Sandia National Laboratories, 1999.

[12] D.W. Doerfler, M. Rajan, C. Nuss, C. Wright, and T. Spelce. Application-Driven Acceptance of Cielo, an XE6 Petascale Capability Platform. In *Proc. 53rd Cray User Group Meeting*, 2011.

[13] W.P. Gaudin, A. Mallinson, O.F.J. Perks, J.A. Herdman, D.A. Beckingsale, J. Levesque, M. Boulton, S. McIntosh-Smith, and S.A. Jarvis. Optimising Hydrodynamics Applications for the Cray XC30 with the Application Tool Suite. In *Proc. 56th Cray User Group Meeting*, 2014.

[14] E.S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. Mcglaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings, 19th International Symposium on Shock Waves*, pages 377–382, 1993.

[15] R.D Hornung and S.R. Kohn. Managing Application Complexity in the SAMRAI Object-Oriented Framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[16] D.T. Stark, R.F. Barrett, R.E. Grant, S.L. Olivier, K.T. Pedretti, and C.T. Vaughan. Early Experiences Co-scheduling Work and Communication Tasks for Hybrid MPI+X Applications. In *Proceedings of the 2014 Workshop on Exascale MPI*, ExaMPI '14, pages 9–19, Piscataway, NJ, USA, 2014. IEEE Press.

[17] C.T. Vaughan and S.P. Goudy. Analysis of an Application on Red Storm. In *Proc. 48th Cray User Group Meeting*, 2006.