International Scholarly Research Network
ISRN Applied Mathematics
Volume 2012, Article ID 246491, 24 pages
doi:10.5402/2012/246491

Research Article

Parallel Adaptive Mesh Refinement Combined with Additive Multigrid for the Efficient Solution of the Poisson Equation

Hua Ji,[1] Fue-Sang Lien,[2] and Eugene Yee[3]

[1] *Waterloo CFD Engineering Consulting Inc., Waterloo, ON, Canada N2T 2N7*
[2] *Department of Mechanical and Mechatronics Engineering, University of Waterloo, Waterloo, ON, Canada N2L 3G1*
[3] *Hazard Protection Section, Defence R&D Canada, Suffield, P.O. Box 4000 Stn Main, Medicine Hat, AB, Canada T1A 8K6*

Correspondence should be addressed to Eugene Yee, eyee0309@gmail.com

Received 6 October 2011; Accepted 9 November 2011

Academic Editors: A.-C. Lee and A. Stathopoulos

Copyright © 2012 Her Majesty the Queen in Right of Canada. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Three different speed-up methods (viz., additive multigrid method, adaptive mesh refinement (AMR), and parallelization) have been combined in order to provide a highly efficient parallel solver for the Poisson equation. Rather than using an ordinary tree data structure to organize the information on the adaptive Cartesian mesh, a modified form of the fully threaded tree (FTT) data structure is used. The Hilbert space-filling curve (SFC) approach has been adopted for dynamic grid partitioning (resulting in a partitioning that is near optimal with respect to load balancing on a parallel computational platform). Finally, an additive multigrid method (BPX preconditioner), which itself is parallelizable to a certain extent, has been used to solve the linear equation system arising from the discretization. Our numerical experiments show that the proposed parallel AMR algorithm based on the FTT data structure, Hilbert SFC for grid partitioning, and additive multigrid method is highly efficient.

1. Introduction

Although an unstructured mesh is highly flexible for dealing with very complex geometries and boundaries [1], the discretization schemes used for these types of meshes (usually in conjunction with control-volume-based finite element methods) are generally much more cumbersome than those used for structured meshes. In particular, the data structure used by an unstructured mesh requires information on “forming points” that are needed to describe the (arbitrary) shape and location of the faces of control volumes where the conservation laws need to be satisfied and to define the adjacent neighbors for each of these control volumes through an appropriate connectivity matrix. These types of operations require generally

indirect memory referencing, and, as a consequence, efficient matrix solvers widely used for structured meshes cannot be easily adapted for unstructured meshes. In view of this, computational methods based on unstructured meshes are generally computationally less efficient than those based on structured meshes, particularly those that use a Cartesian grid. In order to utilize the computational efficiency afforded by a Cartesian structured grid while retaining the flexibility and accuracy in the discretization process required to address problems involving complex geometries and large ranges in spatial scales, adaptive mesh refinement (AMR) can be applied to focus the computational effort and memory usage where it is required for the accurate representation of the solution at minimal cost.

There are two main approaches that have been developed for structured grid AMR. The first approach utilizes a block-structured style of mesh refinement [2–5], in which a sequence of nested structured grids at different hierarchies or levels (cell sizes are the same for all grids at the same hierarchy or level) are overlapped with or patched onto each other. Although a tree-like (or directed acyclic graph) data structure is usually used to facilitate the communications between the regular Cartesian grids at the various resolutions, it should be noted that each node in this tree-like data structure represents an entire grid rather than a cell, with the result that efficient solvers for structured grids can be used to solve the system of algebraic equations resulting from the discretization on each node. However, this type of AMR framework is limited in its flexibility in the sense that the clustering methods used to define the subgrids in the hierarchy of Cartesian grids can result in portions of the computational domain covered by a highly refined mesh when it is not needed (e.g., solution is smooth in this region) resulting in a wasted computational effort. The hierarchical structured grid approach for AMR developed by Berger and Oliger [2] was originally implemented as a serial code. More recently, a number of researchers [6–8] have implemented the classical Berger and Oliger AMR approach on parallel machines, using an ordinary tree data structure to organize the blocks of grid cells corresponding to grids on different levels (resolutions). The disadvantage with this type of data structure is that an expensive search is required to find all the neighbors of a given computational cell in the ordinary tree. In particular, two levels of the tree need to be traversed on average before a neighbor can be found. More importantly, in the worst-case scenario, the entire tree may need to be traversed in order to find all the neighbors of a cell making it difficult to parallelize the operations since the tree traversal may need to be extended across a number of processors.

In the second approach for AMR, the hierarchy of grids at different levels is represented by a quadtree/octtree data structure in two/three dimensions (2D/3D), with each node representing an individual grid cell (rather than a block of grid cells as in the first approach). Because the mesh is only locally refined in the region where it is required, the second approach results in increased computational efficiency, increased storage savings, and better control of the grid resolution in comparison with the first approach. In consequence, it is easier to parallelize the second approach for AMR, and, to this purpose, the fully threaded tree (FTT) data structure [9] can be employed to enhance the parallel efficiency of the algorithm. The application of FTT is advantageous in that it guarantees that no more than one level of the tree is required to access all the neighbors of a particular cell. In consequence, a one-cell thick layer of ghost cells is all that is required for intragrid communication in the parallel computation because this is sufficient to guarantee that all the nearest neighbors of a given cell can be found.

Although a number of researchers [9–12] use the FTT data structure to manage the AMR, the implementation required to parallelize FTT is only addressed briefly by Popinet [12]. However, Popinet only considers a static approach for partitioning and mapping

the mesh onto the processors. In this approach, the grid is partitioned and mapped to the processors only once after the initial grid has been generated. The disadvantage of this approach is that the parallel efficiency will deteriorate as the parallel computation proceeds and as the resolution in the computational grid evolves when the mesh is refined and derefined. Furthermore, to simplify the implementation of the parallelization procedure, Popinet kept the size and level of the cells adjacent to the buffer or ghost region exactly the same as the cells comprising the ghost boundary. Unfortunately, this approach is wasteful in terms of the number of cells required to represent the solution at the boundary between two processors (and also wasteful of the computational resources).

In the present paper, the efficient parallelization of an AMR algorithm that uses the FTT data structure and dynamic partitioning of the mesh is investigated. In particular, the challenging problem associated with dynamic grid partitioning and optimal load balancing for conducting AMR in a massively parallel computational environment is addressed. The paper is organized as follows. In Section 2, the parallelization of AMR is described in detail, which includes (1) domain decomposition and grid partitioning within the context of the FTT data structure using the Hilbert space-filling curve approach; (2) the coarsening and refinement of grids in parallel; (3) construction of the ghost boundary cells for information transfer between processors; (4) application of the BPX (parallel) multigrid approach for solving the algebraic equations arising from the AMR discretization in the parallel environment. In Section 3, numerical experiments using the Poisson equation are conducted in order to demonstrate the parallel efficiency that can be achieved using our proposed approach. Finally, Section 4 contains our conclusions.

2. Numerical Methods

2.1. Modification of the FTT Data Structure

When a computational domain is spatially discretized using AMR, the individual cells in this discretization are usually organized hierarchically in the form of an octtree in 3D (quadtree in 2D). In a conventional oct-tree discretization and representation, such as that used by MacNeice et al. [6], the connectivity information between an individual cell and its neighbors needs to be stored explicitly with each cell requiring 17 words of computer memory to store the following information:

(i) level of the cell in the tree;

(ii) cell location (position);

(iii) identification (ID) of the parent cell;

(iv) IDs of the eight child cells;

(v) IDs of the six neighbor cells.

In addition to the large memory overhead required to maintain this oct-tree data structure, it is also very difficult to parallelize owing to the fact that the neighbors of each cell are connected (linked) to each other. Consequently, removing or adding one cell in the process of adaptive mesh refinement will affect all the neighboring cells of this one cell. If the IDs of the neighboring cells are not stored explicitly, the oct-tree must be traversed to find these neighboring cells. On average, it is expected that two levels of the oct-tree must be traversed before a neighbor can be found, but in the worst-case scenario it may be necessary

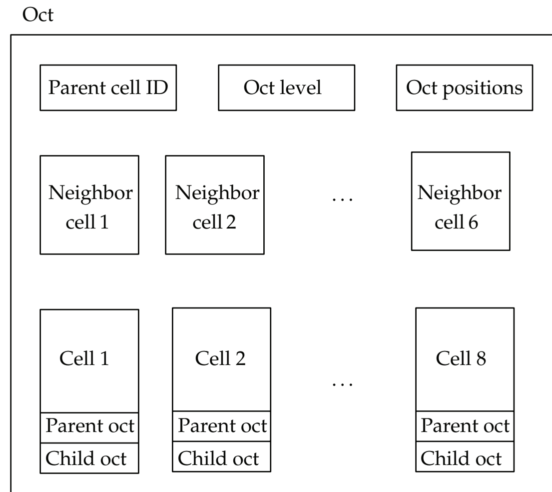


Figure 1: The structure of a modified oct in a fully threaded tree (FTT).

to traverse the entire tree to access the nearest neighbors. Tree traversals of this type can extend from one processor to another making it difficult to vectorize and parallelize these operations. The fully threaded tree [9] has been developed to reduce the memory overhead required to maintain the information embodied in the tree and to facilitate rapid and easy access to the information stored in the tree.

In the FTT structure employed by a serial code (its parallel implementation will be discussed later in Section 2.2.2), all cells are organized in groups referred to as octs. As shown in Figure 1 and as the name implies, each oct maintains pointers to eight (child) cells and a parent cell. Each oct incorporates information about its level which is equal to the level of the parent cell and maintains the information about its position in the computational domain. Furthermore, each oct maintains pointers to the parent cells of the six neighboring octs and each of the eight cells in the oct maintains a pointer to the “parent oct” (viz., the oct containing the eight child cells, which are cells 1 to 8 in Figure 1) and a pointer to the “child oct” if the cell is not a leaf cell (or to a null pointer if the cell has no children). In the FTT data structure proposed originally by Khokhlov [9], each cell of the oct only contained a pointer to its child oct (or to a null pointer if the cell has no children). As a result, finding an individual cell in the original FTT data structure was complicated by the fact that the parent oct of the cell must be accessed first.

In a parallelization of AMR that utilizes the original FTT structure, each oct becomes the basic unit in the grid partitioning, which greatly complicates the load balancing on each processor (because the latter operation is based on the coarse-grained oct, rather than the fine-grained cell as the basic unit). In light of this, an extra word of computer memory is added to maintain a pointer to the parent oct for each cell of the oct. With this modification to the FTT structure, it is now possible to traverse each cell instead of each oct as implied in the original FTT data structure, with the result that the neighboring cells of this cell (either parent cell or child cells) can be accessed much more efficiently using the present modified oct structure (see also [12]). This is because even for a leaf cell (whose pointer to its child oct is null) the neighboring information can still be accessed directly through its parent oct. However, this modification incurs a computational cost in that the memory requirement is now 3.125 words per cell, rather than the 2.125 words per cell required for the implementation of the original

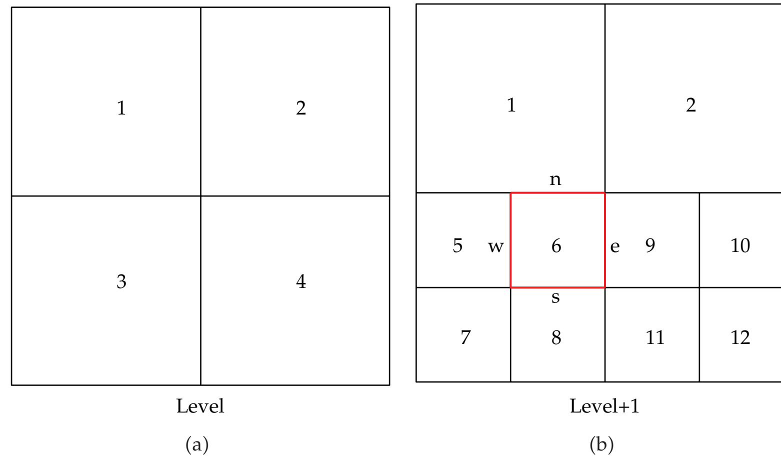


Figure 2: An example illustrating how to access the neighbors of a cell using a modified FTT quad structure without the need for a search operation.

FTT structure. Nevertheless, the memory requirement for our modified FTT oct data structure is still significantly less than that required for the conventional (ordinary) tree data structure (the latter of which requires 17 words of storage for each cell).

Using our modified FTT structure, it is now possible to access the neighbors of a cell without the need for a search operation. Figure 2 shows an example of how this can be done: suppose we need to find the four neighbors of cell 6 in the two-dimensional FTT quad structure exhibited here. To access the west and south neighbors of cell 6, cells 5 and 8 can be found directly using a simple add/subtract operation when provided with the parent oct ID for cell 6, which is stored explicitly in our modified quad structure. The north neighbor of cell 6 can be accessed directly from the north neighbor parent cell by using its parent quad structure. To access the east neighbor of cell 6, the east neighbor parent cell 4 is found first through its parent quad structure. Since cell 4 possesses a child quad, the desired east neighbor (or cell 9) can be found through a simple add/subtract operation using the child quad for cell 4.

2.2. Domain Decomposition for Adaptive Mesh Refinement

Implementing an AMR algorithm in a parallel computational environment requires a grid partitioning strategy. More specifically, the grid has to be decomposed into a number of partitions and each partition needs to be mapped to one processor in the parallel computing system. The grid partitioning has to be done at program execution time since there is usually no prior knowledge about where the grid needs to be refined in order to provide an acceptably accurate solution that reduces the local error below a tolerable threshold. The key issue in the implementation of a computationally efficient AMR algorithm in a parallel computing environment is the development of an appropriate strategy for grid partitioning. This will require the consideration of the following issues. Firstly, the computational load has to be equally distributed amongst the processors of the parallel computing system. Secondly, the volume of data that is required to be transferred between the processors during the partitioning and mapping operations and during the simulation itself needs to be minimized. Finally, the computational cost associated with the load balancing and mapping strategies needs to be (significantly) less than that used for the actual computation (or simulation).

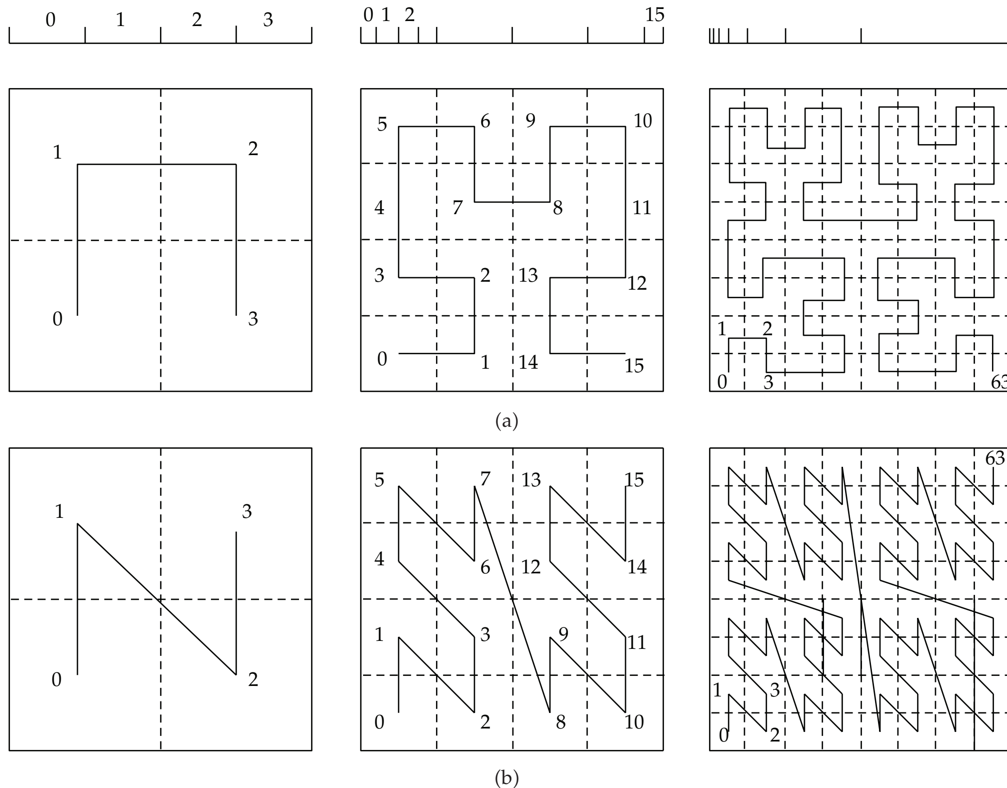


Figure 3: Space-filling curves in two dimensions constructed at three levels: (a) Hilbert SFC with the U-ordering generator and (b) Morton SFC with the N-ordering generator.

2.2.1. Parallel Load-Balancing Using Space-Filling Curves

Our proposed solution for the grid partitioning and subsequent mapping of partitions-to-processors problem is to utilize the space-filling curve (SFC) approach. There are three important properties associated with an SFC that makes it an attractive approach for grid partitioning in the context of AMR: namely, mapping, locality, and compactness [13]. Figure 3 shows two space-filling curves that have been used previously for grid partitioning in parallel computation: namely, the Morton (or N-ordering) and the Hilbert (or U-ordering) space-filling curves [14–16]. These space-filling curves involve the mapping of the points in a higher-dimensional space onto the points along a one-dimensional curve, thus reducing the topology to that of a one-dimensional coordinate along the curve. These curves can be constructed recursively from their underlying generators (e.g., U-shaped and N-shaped generators for the Hilbert and Morton SFCs, resp.) as shown in Figure 3. Here, three different levels of SFCs are shown, which connect 4, 16, and 64 points in the two-dimensional (square) domain at the first, second, and third levels, respectively.

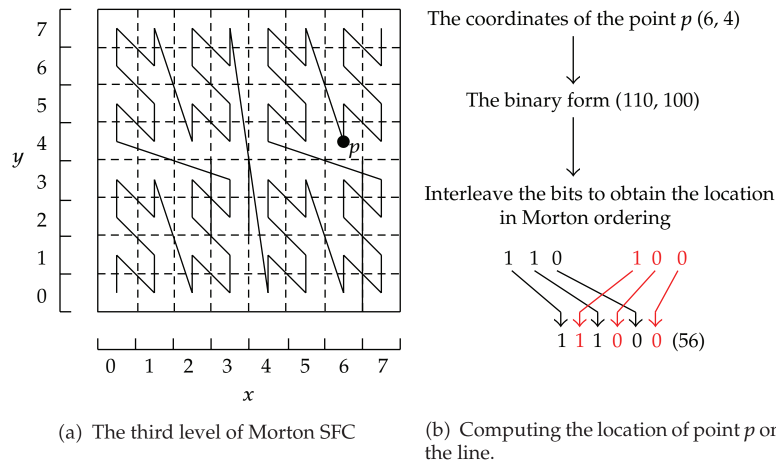
Any point in the computational domain will map to a unique point on a space-filling curve. In view of this, to partition the computational domain in N-dimensions (usually, $N = 2$ or 3) and to map each partition to a processor of the parallel computational system, all we need to do is fill the computational domain with an SFC, encode the grid points of this domain as one-dimensional coordinates along this curve, and divide the ordered points thus

encoded into equal (or approximately equal) parts (or workload) in order to achieve the best load balancing. If the workload for each Cartesian cell is the same, the partition problem is straightforward; namely, an equal number of points along the SFC should be assigned to each processor. However, it is possible for the workload of each cell to be different. For example, if a cut-cell approach [17] is used to handle the complex geometry on a Cartesian mesh, the workload associated with a cut-cell will generally be larger than that associated with a normal cell owing to the fact that more computational effort is required to determine the centroid, volume, and surface area of a cut-cell. In this case, different weights need to be assigned to the various cells to reflect the workload associated with them and these weights should be used in the partitioning and mapping with the SFC. In particular, the points along the SFC need to be partitioned so that the workload assigned to each processor is equal (approximately or better) as this choice will provide the best load balancing.

An important property to consider when using a space-filling curve for grid partitioning is the property of compactness which relates to whether the encoding of the coordinates of the curve requires only local information in the N-dimensional computational domain. This property is important because compactness implies that the SFC can be constructed in parallel on each local processor without the need to exchange information with other neighboring processors. This is because the compactness property implies that it is possible to compute the coordinates of a cell location in the SFC (encoding operation) using only the information embodied in the cell coordinates in the N-dimensional computational domain. In this regard, both the Morton and Hilbert SFCs satisfy the compactness property. However, we note that construction of an SFC with Morton ordering is generally easier and faster than that with Hilbert ordering owing to the fact that the N-shaped generator used for the Morton ordering does not need to be reflected and/or rotated in order to keep the ends of successive generators adjacent as is required for the U-shaped generator used for the Hilbert ordering (cf. the recursive construction of Hilbert and Morton curves in Figures 3(a) and 3(b), resp.).

Using the integer indexing for Cartesian grids described by Aftosmis et al. [13], the encoding of a Morton integer from a cell integer coordinates in an N-dimensional space is easily accomplished by interleaving the bitwise representation of these integer coordinates. For example, Figure 4 illustrates how this encoding is accomplished for a two-dimensional Cartesian mesh. To map the point p with integer coordinates $(6, 4)$ in the two-dimensional Cartesian grid shown here, we first construct the bitwise representation of these coordinates using m-bit ($m = 3$) integers to give $(110, 100)$. One then immediately obtains the location of p along the Morton curve by simply interleaving the bits of this representation to give a $2m$-bit (here, 6-bit) integer 111000 corresponding to the coordinates of point p along the Morton curve (cf. Figure 4(b)).

The encoding of a cell integer coordinates using the Morton ordering is simpler than that using the Hilbert ordering. However, the Hilbert curve preserves locality better than the Morton curve. Indeed, the Hilbert curve has the greatest degree of locality possible for any space-filling curve. Locality means that contiguity along the curve implies contiguity in the N-dimensional space of the Cartesian mesh. In consequence, in the U-ordering used for the Hilbert curve, face-neighboring cells of the Cartesian mesh will be mapped to face neighbors along the Hilbert curve using the U-ordering. This is advantageous when using the Hilbert curve as a grid partitioner as it results in reduced computational overhead arising from inter-processor communications owing to the fact that nearby points in the Cartesian grid are likely to be assigned to the same processor. For this reason, we have chosen to use the Hilbert SFC as the grid partitioning algorithm in conjunction with the modified FTT data structure



(a) The third level of Morton SFC

(b) Computing the location of point p on the line.

Figure 4: Mapping a point p with integer coordinates $(6, 4)$ in a two-dimensional Cartesian mesh to an associated point on a Morton curve.

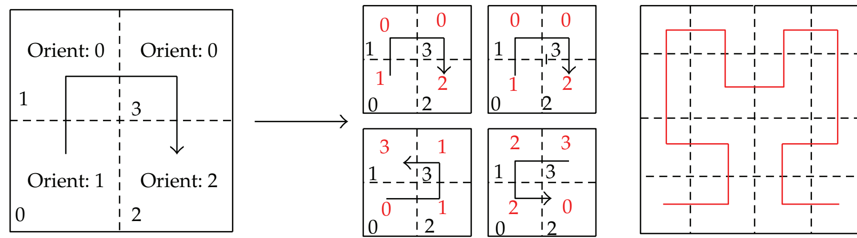
(described earlier). The penalty to be paid here is that encoding a Hilbert integer is more complex than encoding a Morton integer owing to the fact that rotations and inversions of the U-shaped generator (used for the recursive construction of the Hilbert curve) are required in order to assure that the ends of successive generators are kept adjacent (see Figure 3(a)). This complicates the encoding of a Hilbert integer, but there exist orientation and ordering tables [14, 15] that facilitate this encoding.

The algorithm that is used for encoding a Hilbert integer to represent the one-to-one mapping between points in an N-dimensional computational domain and points on the Hilbert curve can be described as follows. We focus the description on a two-dimensional computational domain, but the methodology can be applied to higher-dimensional spaces [14, 15]. For this case, the U-ordering of the Hilbert curve involving four orientations and orderings in two dimensions ($\mathbb{R}^2$) is summarized in Table 1. Here, the rows of the ordering and orientation tables determine the ordering and orientation of the offspring when a parent with orientation given by that row is refined. Figure 5 illustrates how to generate the first two levels of a two-dimensional Hilbert ordering using the ordering and orientation tables shown in Table 1. The root quadrant has orientation 00, so the offspring at the first level are ordered in accordance to the Morton index sequence provided by row 00 of the ordering Table 1: $\{00, 01, 11, 10\}$. Next these offspring are assigned orientations according to row 00 of the orientation Table 1: $\{01, 00, 00, 10\}$. The next (second) level uses this information to determine the order and orientation of their offspring. For the example shown in Figure 5(a), the orientation for quadrant 00 is 01, so we use row 01 of the ordering and orientation Table 1 to ascertain the offspring order (at the second level) and their offspring orientation (at the third level) as $\{00, 10, 11, 01\}$ and $\{00, 01, 01, 11\}$, respectively. In a similar manner, the offspring order and orientation at the second level for the other three quadrants can be determined and yield the results summarized in Figure 5(b). Proceeding in this fashion, all higher levels of the two-dimensional Hilbert ordering can be generated recursively using this simple and efficient procedure.

Using the ordering and orientation tables, the encoding of a Hilbert integer corresponding to a point p in an N-dimensional computational domain is now straightforward. The encoding is accomplished by simply taking the highest-order bit from each spatial coordinate and packing them together as 2 bits (for a two-dimensional computational

Table 1: Ordering and orientation tables used for mapping a Morton space-filling curve to a Hilbert space-filling curve. All numbers in the tables are expressed in binary notation.

	Ordering				Orientation			
00	00	01	11	10	01	00	00	10
01	00	10	11	01	00	01	01	11
10	11	01	00	10	11	10	10	00
11	11	10	00	01	10	11	11	01
Row/column	00	01	10	11	00	01	10	11



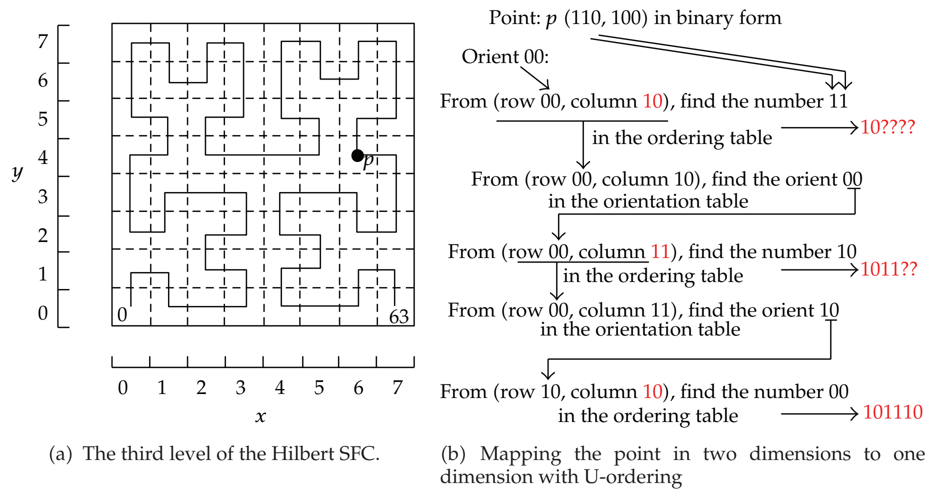
(a) The first level of U-ordering curve

(b) The second level of U-ordering curve

Figure 5: Generation of the first and second levels of a two-dimensional Hilbert ordering using the ordering and orientation tables given in Table 1.

domain) in the order xy where x refers to the highest-order bit of the word. The initial orientation is 0. Row 0 of the ordering table provides the value for xy (corresponding as such to the initial orientation) and the column index data. The latter data constitute the first two bits for the encoding of the location along the Hilbert curve. The next orientation value is found by looking up the corresponding element of the orientation table (column data and row 0). This recursive procedure continues until the last two bits of the spatial coordinate are processed. At the recursion level i, the ith bits from each spatial dimension of point p are packed together as the data to be looked up in the ordering and orientation tables. The ordering table lookup finds the packed two bits data at the column index value and the row determined by the previous orientation table value. The column index value is appended to form the next portion of the Hilbert integer (or coordinate along the Hilbert curve corresponding to the point p). Next, the orientation table is used to find the successive orientation value at the element corresponding to the column index and row of the orientation value determined in the previous step of the recursion. Figure 6 summarizes this procedure for a simple example involving the encoding of a point p in a two-dimensional Cartesian grid with integer coordinates $(6,4)$ as the Hilbert (binary) integer 101110 (or 46 in decimal notation).

The spatial coordinates of points in an N-dimensional computational domain are real numbers (rather than integers). Even so, it should be noted that, within the computer, these spatial coordinates will come from a finite set of numbers determined by the finite precision of the computer hardware. This observation suggests a digital representation for the spatial coordinates which is convenient since to compute the Hilbert integer corresponding to a given point of the N-dimensional computational domain using the ordering and orientation tables, we need to provide a representation of the various spatial coordinates of the point in terms of an unsigned integer. To this end, taking the computer word length to be



(a) The third level of the Hilbert SFC.

(b) Mapping the point in two dimensions to one dimension with U-ordering

Figure 6: The procedure used to encode the point p with integer coordinates $(6,4)$ in a two-dimensional Cartesian mesh to the Hilbert integer 101110 using the ordering and orientation tables given in Table 1.

W bits (usually $W = 32$) implies that an unsigned integer can be used to encode up to W levels of refinement of a Hilbert SFC. In view of this, it is convenient to map the N-dimensional computational domain into the unit hypercube $[0,1]^N$ and let the available spatial coordinates in this hypercube be odd multiples of $2^{-(W+1)}$, labelled by unsigned integers k from 0 to $2^W - 1$: $x = 2^{-W}(k + (1/2))$.

Figure 7 illustrates the application of the Hilbert SFC for grid partitioning. Here, we show an example of AMR applied to a two-dimensional Cartesian mesh, which is partitioned on four processors using a Hilbert SFC. In this example, all the cells are assumed to have equal weight in terms of their workload. Only the leaf cells have been shown in Figure 7.

2.2.2. Data Distribution

A shared-memory computer uses a global address space, whereas, in a distributed-memory computer, each processor has its own local address space. In view of this, parallelization of AMR on a distributed-memory computer is more difficult than on a shared-memory computer. This problem arises because if a pointer is used to refer to a cell in the adaptive Cartesian mesh, this pointer will need to be changed when the cell is migrated from one processor to another (during the dynamic adaptive mesh refinement) for otherwise the pointer of this cell will refer to an incorrect memory address on the processor to which it is migrated. To circumvent this problem, we propose that a unique global ID be assigned to each cell in the adaptive Cartesian mesh and used in the parallel code. Because each cell now has a unique identifier, this global ID does not need to be changed during the course of the simulation.

In a tree-based serial code for AMR, it is possible (and easy) to access any cell in the Cartesian mesh as the tree is traversed. However, it is very difficult to access an arbitrary cell in the tree using this technique in a parallel code owing to the fact that the parent of a particular cell may not even reside in the same processor. To address this problem, we use a hash table to store the information for all cells that reside in a particular processor. This technique is better than that based on a linked list owing to the fact that any destination

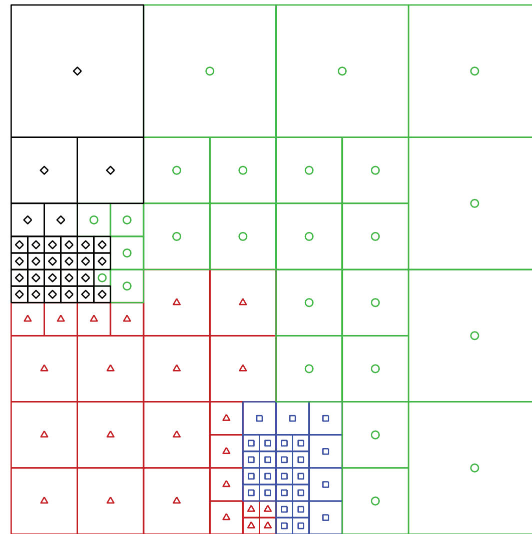


Figure 7: Grid partitioning of a two-dimensional adaptive Cartesian mesh on four processors using the Hilbert SFC.

(target) cell can be located immediately once the hash key (unique entity) for the hash table is given. Here, the global identifier for each cell is used as the hash key because it is unique to the cell. Furthermore, the FTT data structure is maintained to provide the full connectivity information for each cell, such as its parent cell, its child cells, and its neighbor cells. This connectivity information is critical for the discretization of the governing equations. To access a particular cell in the adaptive Cartesian mesh, the processor ID to which the cell belongs needs to be known. To this purpose, it does not make too much sense to explicitly store the processor ID for each cell because this ID will change dynamically at run time as the adaptive mesh is refined and derefined. Rather, it is more efficient (and very easy) to calculate the processor ID associated with a particular cell from its spatial coordinates using the Hilbert integer encoding. This only requires that the range of the coordinates of the Hilbert curve handled by each processor be stored.

If a cell is marked to be migrated to another processor as a result of grid partitioning using the Hilbert space-filling curve, all connectivity information (e.g., parent oct data, child oct data, etc.) associated with this cell will also need to be migrated and the FTT data structure for each processor will need to be regenerated. To facilitate this process, the oct data structure is stored in another hash table. We do not need to assign a global ID for each oct to use as the hash key. Instead, we use the global ID of the parent cell of the oct as the hash key.

2.2.3. Adaptive Mesh Refinement and Coarsening

The refinement mesh is generated level by level in parallel starting from a base-level mesh using some form of a user-supplied criterion (which is usually based on an estimate of the local truncation error in the solution). However, we impose a constraint on the refinement/derefinement process in order to maintain the FTT data structure. This constraint is that no neighboring cells with a difference in level greater than one are allowed in the Cartesian mesh. This rule guarantees that no sharp gradients in the solution will exist at the boundaries for each level of the refinement. Figure 8(a) shows an example of the application

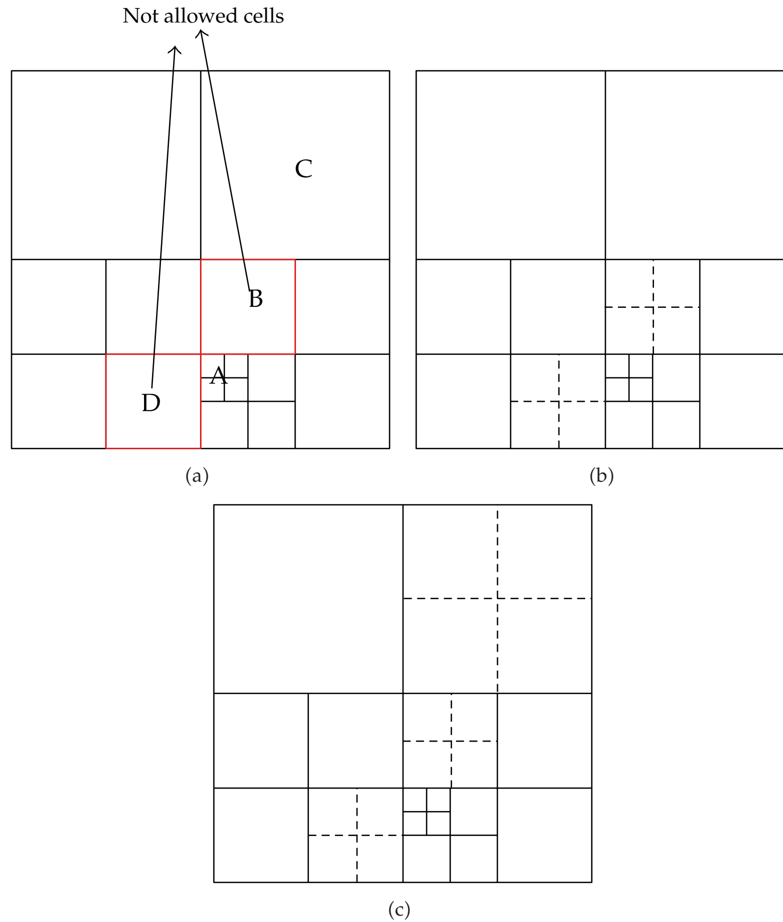
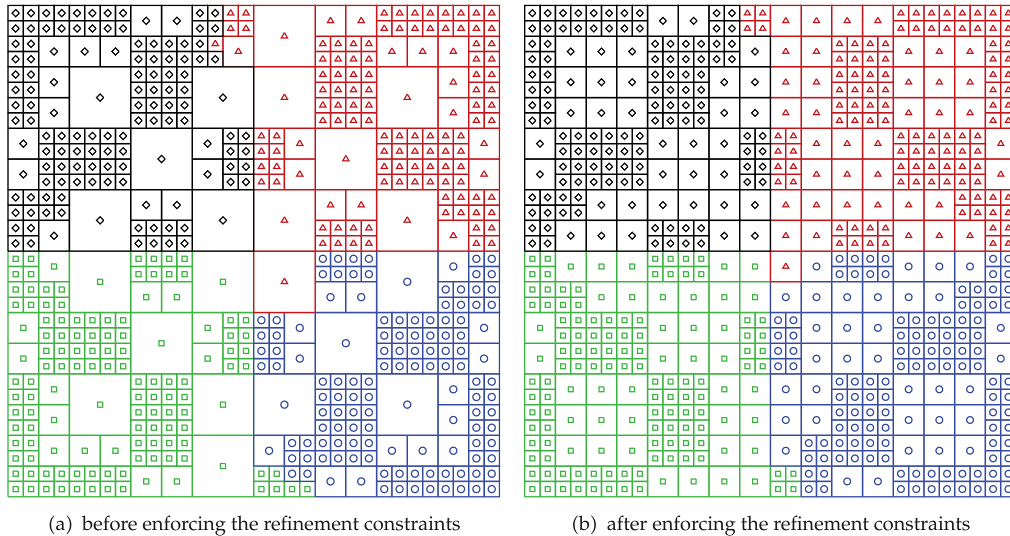


Figure 8: An example illustrating the method used to enforce the refinement constraint: namely, that no neighbor cells with a level difference greater than one are allowed in the Cartesian mesh.

of this rule. Here, it is seen that once cell A has been refined, cell B cannot remain a leaf cell (cell with no children) and must necessarily be refined if the constraint stated above is not violated. Also, the refinement of a cell in conjunction with the satisfaction of the user-supplied criterion for refinement/derefinement can provoke refinement of cells not only in the immediate neighborhood of the refined cell but also in the neighborhood of these immediate neighbors leading to a possible propagation of refinement throughout the entire computational domain. As shown in Figures 8(b) and 8(c), after cell B has been marked for refinement, cell C, which is not immediately adjacent to cell A, may need to be refined as well and this refinement can potentially propagate to the entire domain. Fortunately, the enforcement of the refinement constraint stated above on a level-by-level basis, in conjunction with the user-supplied criterion for mesh refinement/derefinement, alleviates the potential problem associated with a "runaway" refinement process.

We use the following strategy to enforce the refinement constraint which can be illustrated using Figure 8. Firstly, for each processor, all the cells flagged to be refined are added to a local linked list. Suppose cell A in Figure 8 has been tagged for refinement and



(a) before enforcing the refinement constraints

(b) after enforcing the refinement constraints

Figure 9: The states of an adaptive Cartesian mesh before and after one enforcement of the refinement constraint on four processors.

added to the local linked list. At this stage, we now need to check the state of the two neighbors of the marked cell parent, which from Figure 8(a) happen to be cells B and D. This process is repeated or iterated until the lengths of all the local linked lists (one for each processor) are the same. At this point, interprocessor message passing is required. Let us assume that cells B and C belong to different processors. It is necessary to send the global ID of the parent cell of cell B (and its associated oct) to the processor where cell C resides. Since C is a leaf cell and cell B (which is one level higher than the level of cell C) is flagged to be refined, cell C is also flagged to be refined in order to maintain the FTT data structure constraint mentioned previously and appended to the linked list for the processor where cell C resides. Generally, this iterative process converges after about one or two iterations. Figure 9 shows an example of an adaptive Cartesian mesh partitioned on four processors before and after one enforcement of the refinement constraint.

Even if the user-supplied criterion is satisfied for cell coarsening (derefinement), the cell cannot be coarsened immediately. Other conditions need to be satisfied before the coarsening of the cell can take place. These auxiliary conditions for coarsening can be understood with reference to Figure 10. Firstly, all the children of a proposed cell for coarsening (cell A in Figure 10) should be leaf cells. Secondly, for each neighbor of cell A that is not a leaf, it is necessary to examine the two child cells of that neighbor which border on cell A. In Figure 10, these cells are marked by a circle. If any of these cells is not a leaf, and their child cells are not marked to be coarsened, cell A cannot be coarsened any further. In this coarsening procedure, only the nearest neighbor cells are involved. Owing to the use of the FTT data structure, when a cell is marked to be coarsened it is only necessary to remove the child cells of the coarsened cell and the corresponding oct consisting of these child cells from the hash table. In consequence, it is not necessary to regenerate global IDs for all the neighbors of the coarsened cell as would be required in a parallel AMR code that uses an ordinary tree data structure [6]. As a consequence, the coarsening procedure described here is greatly simplified compared to the procedures described by MacNeice et al. [6].

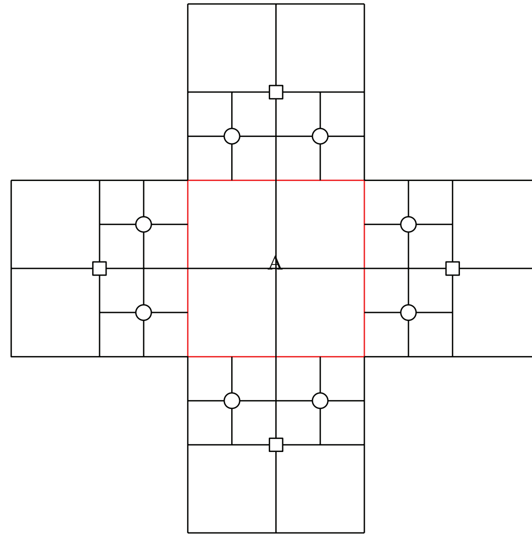


Figure 10: The neighbors of a marked cell A for coarsening need to be checked before cell A can be coarsened.

2.2.4. Construction of the Ghost Boundary Cells for Each Processor

The discretization of the governing equations for various conservation laws for parallel AMR will require the consideration of ghost regions needed to facilitate interprocessor communications. To this end, we will focus on the Poisson equation and use it to illustrate the process that is involved in the construction of ghost boundary cells for parallel AMR. If a second-order accurate scheme is used to discretize the Poisson equation, the ghost boundary needs to be only one cell thick. In the FTT data structure, an oct is the basic structural unit used to organize cells in such a manner that they are able to efficiently locate their neighbors and ascertain their level and position in the adaptive Cartesian mesh. In view of this, before the ghost boundary cells can be constructed, it is necessary to generate the corresponding oct data structure first.

To see how this is done, refer to Figure 11. Here, there are seven cells (marked by the 4 circles, 2 squares, and 1 triangle) in each coordinate direction that are directly related to oct A in the FTT data structure, depending on the level of the neighbor. If the neighboring cells (marked by triangles) and the central cell (marked by the solid circle) are located on the same processor, we can determine directly the level of all the neighbor cells and ascertain which cells depend on the oct A structure. If this is not the case, message passing between two or more processors will be required and this can involve multiple exchanges of data between these processors in order to determine the levels associated with all the neighbor cells. An example of this is shown in Figure 11. For this example, let us assume that octs A and B reside on different processors. To find the southern neighbor cell to A (cell B), interprocessor message passing will be needed to ascertain whether cell B is a leaf cell. If cell B has children as in the example shown here, then additional message passing will be required to determine if cells C and D (two child cells of cell B) are leaf cells. This type of communications overhead involving multiple exchanges of data between processors can be potentially expensive,

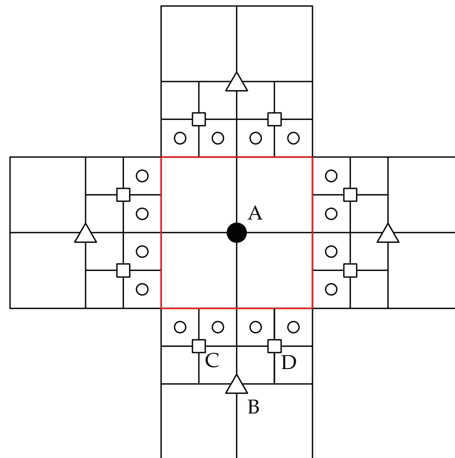


Figure 11: Two-dimensional example indicating the neighbor cells that are related to oct A in the FTT data structure.

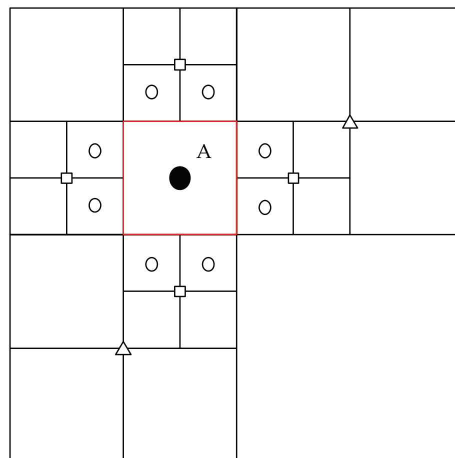


Figure 12: Cell A is a cell that lies on an interprocessor boundary. All cells needed to construct the ghost boundary cells for A are marked in this diagram.

so to reduce this overhead and hence improve the parallel efficiency, we compute the Hilbert coordinates of all the neighbor cells of A in order to ascertain their processor IDs. After this computation, we simply broadcast (send) the oct data for cell A to these processors.

Once the necessary oct structures have been constructed, the ghost boundary cells required for each processor can be determined from these oct structures. As an example, let us assume that one of the faces of cell A in Figure 12 coincides with an interprocessor boundary. Figure 12 displays all possible cells (marked by the 8 circles, 4 squares, and 2 triangles) potentially associated with cell A that will be required to construct the ghost boundary cells for this cell, depending on the level of the neighbor. The latter information is contained in the oct structure for the ghost boundary cells. An example of ghost boundary cells is shown in Figure 13, which displays the local leaf cells along with their corresponding ghost leaf boundary cells on two processors.

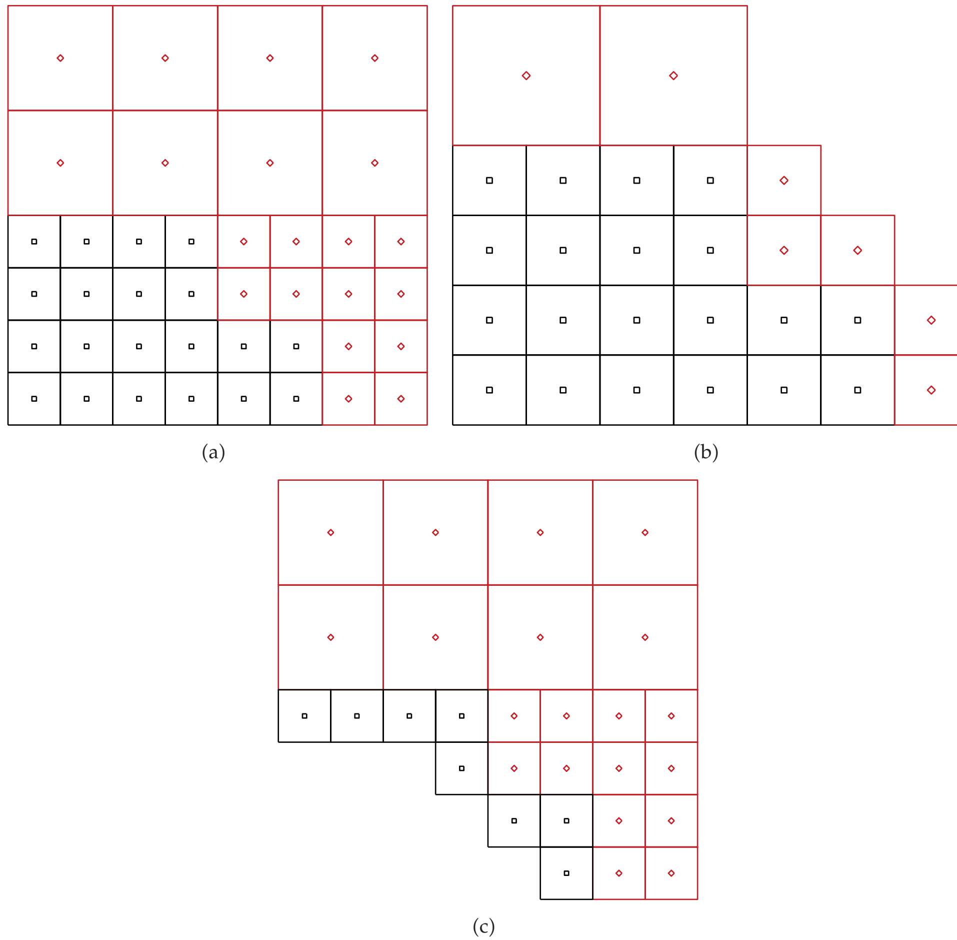


Figure 13: The local leaf cells with their corresponding ghost leaf boundary cells on (a) two processors: (b) on processor 1 and (c) on processor 2.

2.3. Discretization of the Poisson Equation

We will use the Poisson equation to illustrate aspects of the parallel AMR algorithm proposed in this paper. The Poisson equation has the general form

$$\nabla^2 \phi = f. \tag{2.1}$$

Using Stokes' theorem, the integral form of the Poisson equation can be expressed as

$$\oint_S \nabla\phi \cdot \vec{n} ds = \int_V f dV, \tag{2.2}$$

where V is the control volume, S is the control surface, and $\vec{n}$ is the unit normal direction to the control surface S.

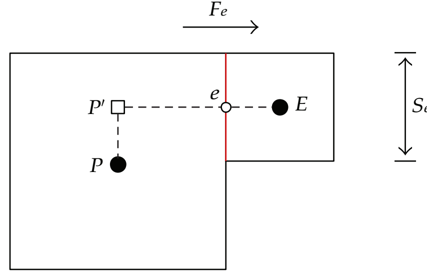


Figure 14: Approximation of the gradient flux F_e using the value of ϕ at an auxiliary node P'.

Given a control volume (cell), the discrete form of (2.2) can be written as

$$\sum_{d} S_d \nabla_d \phi = a \cdot f, \tag{2.3}$$

where d refers to a coordinate direction, S_d is the surface area of a face of the control volume normal to the direction d, and a is the volume of the cell. The gradient ∇_d is evaluated at the center of a control volume face in the direction d.

To achieve second-order accuracy in the discretization, the gradient flux of ϕ at the interface between cells at different levels will be approximated using the value of ϕ evaluated at an auxiliary node as illustrated in Figure 14. Here, using the auxiliary node P', the gradient flux through the east face F_e can be approximated as follows:

$$F_e = S_e \nabla_e \phi = \frac{\phi_E - \phi_{P'}}{x_E - x_P} S_e. \tag{2.4}$$

Because $\phi_{P'}$ can be approximated as $\phi_{P'} \approx \phi_P + (\nabla\phi)_P \cdot (\vec{r_{P'}} - \vec{r_P})$ (where $\vec{r_{P'}} - \vec{r_P}$ is the position vector from P to P'), the gradient flux on the east face of cell P can be written as

$$F_e = \frac{\phi_E - \phi_P}{x_E - x_P} S_e + \frac{S_e}{x_E - x_P} \left[\left(\frac{\partial \phi}{\partial y} \right)_E (y_e - y_E) - \left(\frac{\partial \phi}{\partial y} \right)_P (y_e - y_P) \right]. \tag{2.5}$$

In (2.5), the gradient flux has been decomposed into two parts: the first involving the cell-centered unknowns (ϕ_P and ϕ_E) and the second involving the cell-centered gradients of ϕ $[(\partial\phi/\partial y)_P$ and $(\partial\phi/\partial y)_E]$. We will refer to the first part as the active part and the second part as the lagged part, so

$$F_e(\phi) = F_e^{\text{active}}(\phi) + F_e^{\text{lagged}}(\phi). \tag{2.6}$$

Note that, in our case, at least one of the gradient terms in the lagged part will be zero (because y_e and either y_P or y_E will have the same value).

To approximate the contribution of the lagged part in (2.5), we have to evaluate the cell-centered gradient of ϕ at each cell. To this end, we use the least-squares approach [18, 19] for the reason that it is exact for a linear profile and can be easily parallelized since only

values of ϕ at the neighbor cells are needed to construct the cell-centered gradient of ϕ. For an arbitrary cell P and its set of immediate neighbors η_P, the change in value of ϕ between a central cell P and an immediate neighbor cell j ($j \in \eta_P$) is given by the difference $\phi_j - \phi_P$. If ϕ is linear, then this difference is given exactly by

$$\phi_j - \phi_P = (\nabla\phi)_P \cdot (\vec{r_j} - \vec{r_P}). \tag{2.7}$$

However, unless the solution ϕ is linear, the gradient here is not exact owing to the fact that cell P has more neighbors than the gradient has components. The least-squares approximation to the gradient is that which minimizes the cost function given by

$$E = \sum_{j\in\eta_P} w_j \left[(\nabla\phi)_P \cdot (\vec{r_j} - \vec{r_P}) - (\phi_j - \phi_P) \right]^2, \tag{2.8}$$

where w_j are weights that we choose as $w_j = 1/|\vec{r_j} - \vec{r_P}|^2$.

For the two-dimensional case, the solution to this least-squares problem reduces to the solution of the following linear system of algebraic equations:

$$\begin{bmatrix} \sum w_j \Delta x_j \Delta x_j & \sum w_j \Delta x_j \Delta y_j \\ \sum w_j \Delta x_j \Delta y_j & \sum w_j \Delta y_j \Delta y_j \end{bmatrix} \begin{bmatrix} \phi_x \\ \phi_y \end{bmatrix} = \begin{bmatrix} \sum w_j \Delta x_j \Delta \phi_j \\ \sum w_j \Delta y_j \Delta \phi_j \end{bmatrix}, \tag{2.9}$$

where

$$\begin{aligned} \Delta x_j &\equiv x_j - x_P, \\ \Delta y_j &\equiv y_j - y_P, \\ \Delta \phi_j &\equiv \phi_j - \phi_P, \end{aligned} \tag{2.10}$$

and ϕ_x and ϕ_y are the two components of the cell-centered gradient $(\nabla\phi)_P$ of ϕ at P.

2.4. Parallel Multigrid Preconditioner with Conjugate Gradient Method

The discretized Poisson equation can be written in the following form:

$$\mathcal{L}(\phi) = f, \tag{2.11}$$

where $\mathcal{L}$ is a linear operator. To solve this linear system efficiently, we will use a multigrid method that is advantageous because the convergence rate of these methods do not depend on the dimension of the system. There are two kinds of multigrid methods that can be considered, namely, multiplicative and additive multigrid methods [20]. The difference between these two types of multigrid methods can be seen as follows. Consider a "V-cycle" whose correction form for the linear system of (2.11) is

$$\mathcal{L}(\phi + \delta\phi) = f \Longleftrightarrow \mathrm{L}(\delta\phi) = R \quad \text{with } R = f - \mathcal{L}(\phi). \tag{2.12}$$

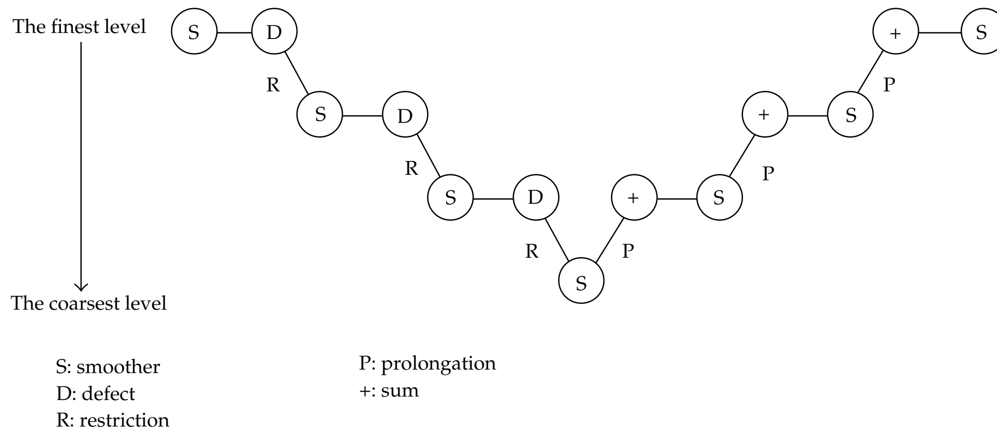


Figure 15: The sequence of operations required for the V-cycle multiplicative multigrid method.

Figure 15 summarizes the sequence of operations involved in the V-cycle multiplicative multigrid method. The values of ϕ in (2.11) at the finest grid level are first approximated by performing a few iterations with a smoother (resulting in a smooth error or residual at this grid level). The residual is then calculated on the finest grid and transferred to the next coarser level using a restriction operation. The correction $\delta\phi$ to the solution of (2.12) at the coarser level is then approximated by performing a few iterations with the smoother. This procedure is then applied recursively down to the coarsest level. The correction $\delta\phi$ of the solution on the coarsest grid level is then transferred back to the next finer grid level (prolongation operation). After performing a few iterations at this level, the correction of the solution on this level is transferred to the next finer level until the finest level is finally reached. The whole V-cycle is repeated until the residuals on the finest level have been reduced below some desirable (small) level. An examination of the sequence of operations involved here would reveal that the multiplicative multigrid method cannot be used to process the coarse- and fine-grid level information simultaneously (in parallel), because the processing on the coarser grids requires information from the finer-grid residuals and the finer grids require information from the coarser grid corrections. Hence, the multiplicative multigrid method is not amenable to parallelization.

In contrast, the additive multigrid method is amenable to some form of parallelization. More specifically, in this multigrid method the smoothing operations on the different grid levels can be performed simultaneously (or in parallel). Restriction and prolongation, however, still need to be conducted sequentially in the additive multigrid method. Figure 16 summarizes the operations involved in the V-cycle additive multigrid method. These operations involve three major stages: restriction (in sequence), smoothing (in parallel) and prolongation (in sequence). After the residuals (referred to as the defect in Figure 16) of a linear system such as (2.11) have been computed at the finest grid level, it is restricted down level by level from the finest to the coarsest levels. On all the grid levels, the correction form for the linear system of (2.12) can be solved in parallel. After this step, the correction $\delta\phi$ can be prolongated level by level from the coarsest to the finest level. Owing to the increased level of parallelization allowed by the additive multigrid (or BPX [21]) (The BPX designation here derives from the first letter of the last names of the three authors (Bramble, Pasciak, and Xu) who originally developed the method.) method, we choose to use this method for the solution of the discretized Poisson equation. It should be noted that the BPX method generally cannot

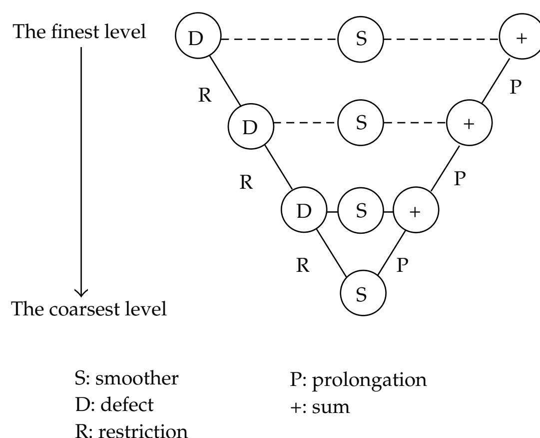


Figure 16: The sequence of operations required for the V-cycle additive multigrid method.

converge if simply used as a stand-alone smoother. For this reason, we use the BPX method in conjunction with a conjugate gradient method as the smoother.

A volume-weighted average is applied to restrict the residuals at a given grid level to a coarser grid level:

$$R_l = \frac{1}{4}\sum_{i=1}^{4}(R_{l+1})_i, \tag{2.13}$$

where the summation is over the four children of the cell considered. The correction form of (2.7) is used to prolongate the correction $\delta\phi$ from a coarser grid level to a finer grid level as follows:

$$(\delta\phi)_{fi} = (\delta\phi)_c + (\nabla(\delta\phi))_c \cdot (\vec{r_{fi}} - \vec{r_c}). \tag{2.14}$$

When a cell and all its child cells are located on the same processor, the restriction and prolongation operations can be done locally without having to incur any communications overhead. Otherwise, the information required between processors in the execution of the BPX method will need to be passed on a level-by-level basis. However, given the oct data structure and the Hilbert coordinates for the cells in the adaptive Cartesian mesh, the global and processor IDs of these cells can be easily determined and used to construct the appropriate communication map.

3. Numerical Results

A two-dimensional Poisson equation is considered in this study as our test problem:

$$\nabla^2\phi = -\pi^2\left(k^2 + l^2\right)\sin(\pi k x)\sin(\pi l y), \tag{3.1}$$

Table 2: The wall-clock time required to solve the Poisson equation on regular (uniform) grids for levels 5 to 10 using 1 to 64 processors on a Beowulf cluster.

Time (s)		Number of processors						
Level	Grid	1	2	4	8	16	32	64
5	1024	1.88	1.03	0.70	0.68	0.75	0.80	0.80
6	4096	7.52	3.75	2.23	1.75	1.38	1.53	1.23
7	16384	33.50	15.73	8.24	5.02	3.50	3.76	2.51
8	65536	158.11	73.32	34.21	18.23	10.02	7.01	5.23
9	262144		353.51	176.01	85.76	42.04	21.49	12.47
10	1048576				381.52	192.72	92.76	48.70

Table 3: The wall-clock time required to solve the Poisson equation on various adaptive Cartesian meshes using 1 to 64 processors on a Beowulf cluster.

Time (s)	Number of processors						
Grid	1	2	4	8	16	32	64
604	1.26	0.71	0.49	0.55	0.58	0.51	0.57
2320	4.89	2.54	1.44	1.31	1.20	1.23	1.31
10120	23.10	10.88	5.52	3.56	2.67	2.72	2.61
40048	109.98	53.42	24.53	12.53	7.41	5.22	3.87
160288		342.25	165.71	73.94	36.23	19.66	11.43
639376				388.14	187.12	87.58	45.62

with $k = l = 3$. The computational domain for the test problem is $[-0.5, 0.5]^2$, and the Neumann boundary conditions are used on the four boundaries of the domain. All wall-clock times shown in Tables 2–4 are measured on a Beowulf cluster on SHARCNET [22]. On this cluster, a high-speed wide-area network with a throughput of 1 Gigabit per second was used for the message passing between processors.

The Poisson equation was chosen for our test problem because the ratio of the computational work to the communications overhead is low compared to the computationally more demanding problem of solving the Navier-Stokes equation. However, the simple Poisson equation corresponds to a more difficult test for our algorithm with respect to the achievement of a high parallel efficiency.

In the first test case, we consider only uniform grids with no adaptive mesh refinement and derefinement. Table 2 shows the wall clock times for the solution of the Poisson equation (3.1) on a sequence of six different regular grids using up to 64 processors. In these tests, the grid at level n has a size of $2^n \times 2^n$. From Table 2, it is observed that using more processors does not always reduce the wall-clock time monotonically. For example, for the cases corresponding to grids with levels less than 8, the wall-clock time required for the solution of the Poisson equation increases as the number of processors increases from 16 to 32. This is because the communications overhead becomes dominant as the size of the problem handled by each processor becomes smaller. As the overall size of the problem increases (e.g., for problems associated with levels 7 and above), the computational effort outweighs the communications overhead, so the parallel efficiency increases. For the last case in the Table 2, a parallel efficiency of $T_8/(64/8)T_{64} \times 100 = 98\%$ is achieved with 64 processors relative to the wall-clock time on 8 processors (here, T_n denotes the wall-clock time required to solve the problem using n processors).

Table 4: The wall-clock time required to solve the Poisson equation on an adaptive Cartesian mesh with the Hilbert SFC as the grid partitioner using 1 to 64 processors on a Beowulf cluster.

Adaptive grid				40048			
Processors	1	2	4	8	16	32	64
Time (s)	0	0.039	0.021	0.011	0.010	0.0092	0.0085
Percentage (%)	0	0.073	0.085	0.088	0.13	0.18	0.22

In the next test case, we consider various grids generated using AMR. The leaf cells in the grids are refined if $|\nabla\phi|$ is larger than the mean value of variable on all the leaf cells. Table 3 provides the wall-clock times for the solution of the Poisson equation (3.1) for 6 different levels of adaptive grids on different numbers of processors. It can be seen from Table 3 that, in conformance to what we observed in Table 2, the total time decreases monotonically as the number of processors increases for those problems with large grid sizes. For the last case in Table 3, a parallel efficiency of $T_8/(64/8)T_{64} \times 100 = 106\%$ is achieved. This is likely due to the efficient use of cache memory when the grid size handled by each processor is small enough so that the information in the grid can be stored in the cache memory.

Table 4 shows the wall-clock time for solving the Poisson equation (3.1) using the Hilbert SFC for grid partitioning and subsequent mapping of the data to each processor. The percentage of the total computational time required for the grid partitioning is shown in the same table for up to 64 processors. It is observed from Table 4 that this percentage increases slightly when a large number of processors is used. This is because a large volume of data needs to be migrated over a large number of processors. However, the method proposed here for parallel AMR is still very efficient. For example, in the case of 64 processors, the ratio of time spent in the grid partitioning to the total computational time is only 2.2×10^{-3}. Taken together, the results of Tables 3 and 4 demonstrate that our proposed parallel AMR algorithm when used in conjunction with the additive multigrid method and the Hilbert SFC for grid partitioning is highly efficient. Furthermore, comparing the results for run times summarized in Tables 2 and 4, it can be seen that combining the three different speed-up methods resulted in speed-up factors of $381.52/0.011 \approx 34700$ and $48.7/0.0085 \approx 5730$ using 8 and 64 processors, respectively, when comparing the parallel AMR solution against the solution obtained using a uniform grid with 10 levels (viz., a uniform 2D grid with $2^{10} \times 2^{10}$ cells).

4. Conclusions

From numerical point of view, it is important to develop an efficient Poisson solver since many numerical methods for more complex systems of equations (e.g., the Navier-Stokes equation) require the solution of the Poisson equation. In this paper, the three different speed-up methods (viz., additive multigrid, adaptive mesh refinement, and parallelization) are combined together to solve the Poisson equation efficiently. Instead of using an ordinary tree data structure to organize the information on the adaptive Cartesian mesh, a FTT data structure is used because this structure requires minimum computer memory storage and all operations on the FTT can be performed efficiently and are amenable to parallelization. The Hilbert space-filling curve approach has been used in conjunction with the FTT for dynamic grid partitioning (resulting in partitioning that is near optimal with respect to load balancing on a parallel computer system). Furthermore, the refinement and derefinement of the adaptive Cartesian mesh can be parallelized as well. Finally, an additive multigrid method

(BPX preconditioner) which itself is parallelizable to a certain extent has been used to solve the linear equation system arising from the discretization. Our numerical experiments show that the proposed parallel AMR algorithm based on the FTT data structure, Hilbert SFC for grid partitioning, and additive multigrid method is highly efficient.

Acknowledgments

This work has been supported by Chemical Biological Radiological-Nuclear and Explosives Research and Technology Initiative (CRTI) program under project no. CRTI-07-0196TD. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET).

References

[1] R. Lohner, "Adaptive remeshing for transient problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 75, p. 195, 1989.

[2] M. J. Berger and J. Oliger, "Adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 53, p. 484, 1984.

[3] M. J. Berger and P. Collela, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, p. 64, 1989.

[4] M. J. Berger and R. J. LeVeque, "An adaptive Cartesian mesh algorithm for the Euler equations in arbitrary geometries," AIAA Paper 89-1930-CP, 1989.

[5] J. J. Quirk, "An alternative to unstructured grids for computing gas dynamic flows around arbitrary complex two-dimensional bodies," *Computers and Fluids*, vol. 23, p. 125, 1994.

[6] P. MacNeice, K. M. Olson, C. Mobary, R. DeFainchtein, and C. Packer, "PARAMESH: a parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, p. 330, 2000.

[7] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowsky, "A computational infrastructure for parallel adaptive methods," in *Proceedings of the 4th U.S. Congress on Computational Mechanics: Symposium on Parallel Adaptive Method*, San Francisco, Calif, USA, 1997.

[8] C. A. Rendleman, V. E. Beckner, M. Lijewski, W. Y Crutchfield, and J. B. Bell, "Parallelization of structured, hierarchical adaptive mesh refinement algorithms," *Computing and Visualization in Science*, vol. 3, p. 147, 2000.

[9] A. M. Khokhlov, "Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations," *Journal of Computational Physics*, vol. 143, no. 2, pp. 519–543, 1998.

[10] P. Diener, N. Jansen, A. Khokhlov, and I. Novikov, "Adaptive mesh refinement approach to the construction of initial data for black hole collisions," *Classical and Quantum Gravity*, vol. 17, p. 435, 2000.

[11] T. Ogawa, T. Ohta, R. Matsumoto, K. Yamashita, and M. Den, "Hydrodynamical simulations using a fully threaded tree," *Progress of Theoretical Physics*, vol. 138, p. 654, 2000.

[12] S. Popinet, "Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries," *Journal of Computational Physics*, vol. 190, p. 572, 2003.

[13] M. J. Aftosmis, M. J. Berger, and G. Adomavicius, "A parallel Cartesian approach for external aerodynamics of vehicles with complex geometry," in *Proceedings of the Thermal and Fluids Analysis Workshop*, NASA Marshall Spaceflight Center, Huntsville, Ala, USA, 1999.

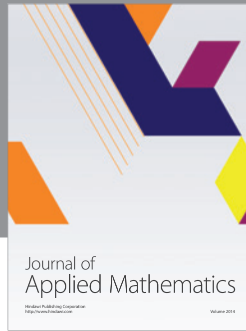
[14] P. C. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, "Dynamic octree load balancing using space-filling curves," Tech. Rep. CS-03-01, Williams College Department of Computer Science, 2003.

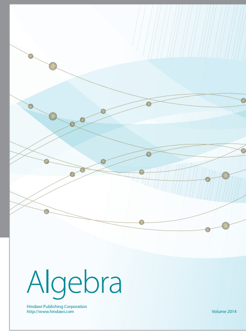
[15] J. K. Lawder and P. J. H. King, "Using state diagrams for Hilbert curve mappings," *International Journal of Computer Mathematics*, vol. 78, p. 327, 2001.

[16] J. R. Pilkington and S. B. Baden, "Partitioning with spacefilling curves," Tech. Rep. CS94-349, CSE, 1994.

[17] H. Ji, F.-S. Lien, and E. Yee, "An efficient second-order accurate cut-cell method for solving the variable coefficient Poisson equation with jump conditions on irregular domains," *International Journal of Numerical Methods in Fluids*, vol. 52, p. 723, 2006.

[18] T. J. Barth and D. C. Jersperson, “The design and application of upwind schemes on unstructured meshes,” AIAA Paper 89-0366, 1989.
[19] T. J. Barth, “A 3-D upwind Euler solver for unstructured meshes,” AIAA Paper 91-1548, 1991.
[20] J. E. Jones and S. F. McCormick, “Parallel multigrid methods,” in *Parallel Numerical Algorithms*, D. Keyes, A. Sameh, and V. Venkatakrishnan, Eds., vol. 4, pp. 203–224, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.
[21] J. H. Bramble, J. E. Pasciak, and J. Xu, “Parallel multilevel preconditioners,” *Mathematics of Computation*, vol. 55, p. 1, 1990.
[22] *The Shared Hierarchical Academic Research Computing Network*, http://www.sharcnet.ca/.





Hindawi

Submit your manuscripts at
http://www.hindawi.com