Implementing the 3D Alternating Direction Method on the Hypercube [*]

John Bruno and Peter R. Cappello
Department of Computer Science
University of California
Santa Barbara, CA 93106

August 16, 1989

Abstract

The paper considers computational domains structured as a 3D grid of cells. It presents a cell-to-hypercube map that is useful for implementing the *Alternating Direction Method* (ADM). The map is shown to be perfectly load-balanced, and to optimally preserve adjacencies between cells in the computational domain.

1 Introduction

Twizell [11] notes,

> The ADM was introduced by Peaceman and Rachford [6] for the numerical solution of elliptic and parabolic partial differential equations.

These methods are used for the solution of the Dirichlet problem, the Neumann problem, and the Robbins problem, among others [11]. The Cartesian space is taken to be the *physical problem domain* and the solution is sought over a rectangular box. Other geometries are accommodated by transforming the cartesian domain into other curvilinear coordinates. The ADM's *computational domain* $\mathbf{D}$ consists of a set of points which are uniformly spaced over the *problem domain*. These points are called *grid points* and are indexed by positive integers (i, j, k) such that $1 \leq i \leq I$, $1 \leq j \leq J$, and $1 \leq k \leq K$, where I, J, and K are positive constants.

Abstractly, each iteration of the procedure consists of 3 steps:

1. Perform $I \times J$ independent computations (e.g., solving tridiagonal systems) each of size K.
2. Perform $J \times K$ independent computations (e.g., solving tridiagonal systems) each of size I.
3. Perform $I \times K$ independent computations (e.g., solving tridiagonal systems) each of size J.

[*]This work supported in by the National Science Foundation under grant MIP89-20598, Intel Corporation, the University of California MICRO Program, and by a grant from the Lawrence Livermore National Laboratory.

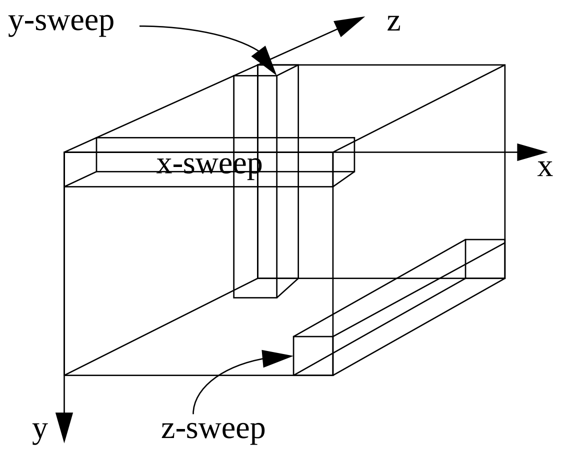

Figure 1: Preferred Cell Shapes

Considering step 1 above, the $I \times J$ independent computations are indexed by (i, j) and can be carried out in parallel. For a fixed value of (i, j), the computation typically requires values associated with grid points with indices (i, j, k), for $1 \leq k \leq K$, and values associated with neighboring grid points. For example, a typical computation involves solving a tridiagonal linear system of equations where the coefficients are associated with the grid points along the line (i, j, k), for $1 \leq k \leq K$. The computation "sweeps" along the line from $(i, j, 1)$ to (i, j, K) during the *elimination* phase, and then "sweeps" along the line from (i, j, K) to $(i, j, 1)$ during the *back substitution* phase. Similar comments apply to steps 2 and 3 where the "sweeps" traverse coordinate directions i and j, respectively.

When implementing the ADM, an important, if not overriding, consideration is the processor communication resulting from the assignment of grid points to processors [2, 5, 8, 4]. Considering only step 1 of an iteration, it is desirable to map the computational domain onto the hypercube so that all grid points with equal i and j coordinates are assigned to the same node. If this were so, the computation associated with a particular i and j would be completed using little or no communication with other nodes. Considering only step 2 of an iteration, it is desirable to map the computational domain onto the hypercube so that all grid points with identical j and k coordinates are assigned to the same node. Finally, considering only step 3 of an iteration, we want to map all grid points with identical i and k coordinates to the same node.

In Fig. 1, we show the 3 different partitions of the computational domain into "cells" which would accommodate the above computations. According to step 1, we prefer cells shaped like the one labeled z-sweep; according to step 2, we prefer cells shaped like the one labeled x-sweep; and lastly, for step 3, we prefer cells shaped like the one labeled y-sweep.

The 3 different partitions are not compatible, and it is costly to change the partition (and the assignment of cells to processors) between each step of an iteration. We thus are motivated to seek a cell size and shape and assignment of cells to processors that can be maintained throughout the computation and which accommodates an efficient implementation of each step of an iteration.

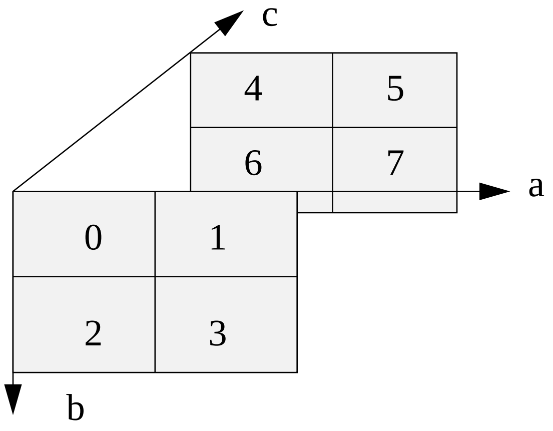

Figure 2: Adjacency Preserving 1-1 Mapping

2 Cell-To-Node Mappings

In this section we discuss the effect of "cell-to-node" mappings on the efficiency of our implementation. The efficiency depends on a number of factors [5]:

Load balancing: Assign an equal amount of computational work to each node.

Communication: Minimize the amount of communication and the distance messages must travel.

We partition the computational domain $\mathbf{D}$ into subsets called *cells*. Each cell is a "box" of grid points and is identified by its coordinates (a, b, c) where $0 \leq a < N_a$, $0 \leq b < N_b$, and $0 \leq c < N_c$. Assuming N_a divides I, N_b divides J, and N_c divides K, each cell is assigned $(I/N_a)(J/N_b)(K/N_c)$ grid points. Two cells are said to be *adjacent* if exactly one of their corresponding indices differs by one and the other corresponding indices are equal. For example, the cell with indices (a, b, c) is adjacent to the cell with indices $(a, b-1, c)$. Thus the cells form an $N_a \times N_b \times N_c$ grid.

A d-dimensional hypercube is a graph of 2^d nodes(processors), each numbered with a distinct non-negative integer less than 2^d, with an edge between a pair of nodes if the binary representations of their node numbers differ in exactly one bit position. Two nodes are said to be *adjacent* if there is an edge between them. The *distance* between two nodes is equal to the number of bit positions in which the representations of the node numbers differ. We are concerned about the distance between two nodes since it is equal to the minimum number of communication links that must be traversed by a message transmitted between the two nodes [1, 10].

A *cell-to-node mapping* assigns a node number to each cell in the computational domain. We say that a cell-to-node mapping is *adjacency preserving* if it maps adjacent cells into adjacent nodes.

Cell-to-node mappings are represented by labeling each cell with the hypercube node number assigned to the cell. That is, if θ is a cell-to-node mapping then the cell with indices (a, b, c) is labeled with node number $\theta(a, b, c)$. For example, suppose we have a 3-dimensional hypercube and $N_a = N_b = N_c = 2$. In Fig. 2, we show an adjacency preserving 1-1 mapping of the cells onto the nodes of the hypercube. The figure depicts a $2 \times 2 \times 2$ grid of cells in which each cell is labeled with the node number to which it maps.

Using the mapping shown in Fig. 2, the z-sweep of step 1 of the ADM sweeps along the c axis during the elimination phase from cell $(a, b, 0)$ to cell $(a, b, 1)$ and from cell $(a, b, 1)$ to $(a, b, 0)$ during the back-substitution phase (for each a, b such that $0 \leq a, b \leq 1$). Note that within each cell we may have up to $I/N_a \times J/N_b$ independent systems to solve. Since the cells $(a, b, 0)$ are labeled with only half of the

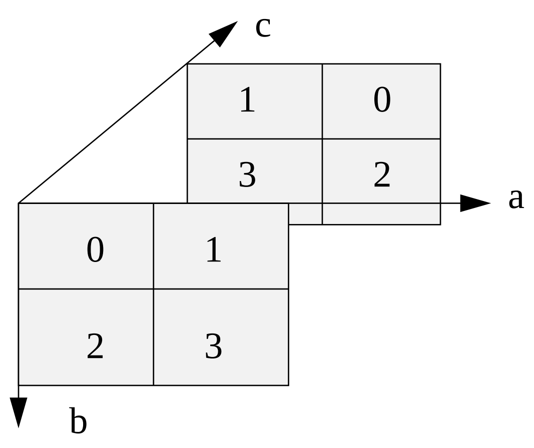

Figure 3: A Many-To-One Mapping

node numbers (0,1,2, and 3), the rest of the nodes are initially idle during the first part of the elimination phase. At the end of the first part of the elimination phase nodes 0,1,2, and 3 transmit intermediate results to nodes 4,5,6, and 7, respectively. When nodes 4, 5, 6, and 7 take over the elimination phase, the initially active nodes become idle. Indeed, it is easy to see that at least half of the nodes are idle at every point in time. This is also true for steps 2 and 3 of the ADM using this cell-to-node mapping.

Next consider a 2-dimensional hypercube, and a computational domain partitioned into 8 cells. A cell-to-node mapping for this case is shown in Fig. 3. Since each node is assigned a cell with $c = 0$, *all* nodes have work to do, and are initially active during the first part of the elimination phase of step 1. Furthermore, all the nodes are assigned a cell with $c = 1$, so that all the nodes remain active after they are finished with cells having $c = 0$. The situation is the same on the back-substitution phase. Step 2 is also favorable since all the nodes are assigned cells with $a = 0$ and $a = 1$. Step 3 is not as favorable since only half of the nodes are assigned cells with $b = 0$. Therefore, at least half of the nodes are idle throughout step 3.

These examples demonstrate the nature of the relationship between the cell-to-node mapping and the potential efficiency of our implementation.

> It follows that the best case would be to have every node assigned exactly one cell for each different value for a, b, and c and have adjacent cells map into adjacent nodes or the same node.

This would mean that in each step, every node would be busy except for the time during the transmission of intermediate results to neighboring cells. We have not achieved this condition in the previous example since nodes 0 and 1 are not assigned any cell with $b = 1$, and nodes 2 and 3 are not assigned any cell with $b = 0$. We prove in the next section that such a cell-to-node mapping does not exist.

We might relax the above conditions by requiring that each node be assigned *at least* one cell for each different value for a, b, and c and that adjacent cells map into adjacent nodes. We can find cell-to-node mappings which satisfy this condition and an example is shown in Fig. 4. Mappings such as these are obtained by increasing $N_a \cdot N_b \cdot N_c$. This is undesirable for the following reason. Normally, we associate a *process* with each cell, called a *cell process*, which is responsible for the computation associated with all the grid points within the cell. This means that if a cell-to-node mapping assigns r cells to a particular

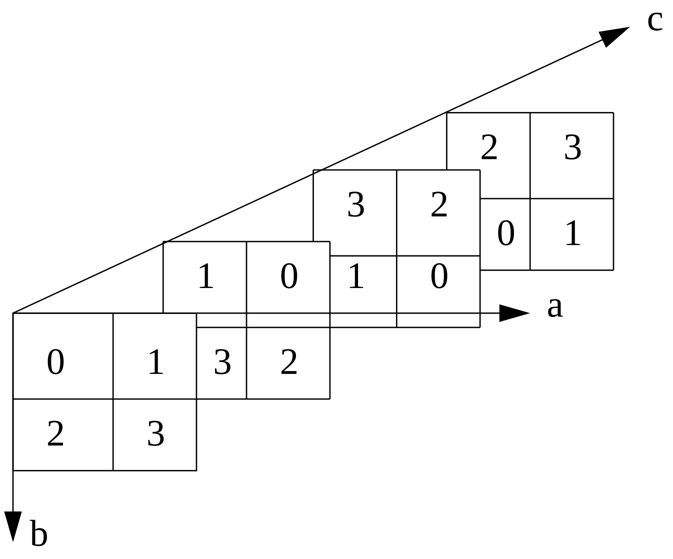

Figure 4: Easy Mappings

node, then the node will contain r cell processes, one corresponding to each cell, and the processor corresponding to the node will be multiplexed among the cell processes. The goal is to have the *fewest* number of cells consistent with keeping all the processors busy for the duration of the computation. By increasing $N_a \cdot N_b \cdot N_c$ we can often keep all processors busy (as we have done in this example) at the expense of additional overhead due to processor multiplexing among the cell processes and additional inter-processor communication [5].

Another possibility is to relax the constraint that adjacent cells map into adjacent nodes. Let $g_s(r)$ denote a function which is defined for all integers r and s where $0 \leq r < 2^s$, whose range is the set of all binary strings of length s, and has the property that $g_s(r)$ and $g_s(r + 1 \bmod 2^s)$ differ in exactly one bit position. There are many possible functions g_s and such functions are called Gray codes [7, 9]. We define a cell-to-node mapping $\theta(a, b, c)$, where $0 \leq a, b, c < 2^d$, as follows:

$$\theta(a, b, c) = g_d((b + c) \bmod 2^d) \odot g_d((a + c) \bmod 2^d),$$

where $\odot$ denotes concatenation. The cell-to-node function θ maps a $2^d \times 2^d \times 2^d$ domain of cells onto a hypercube with 2^{2d} nodes.

The map's adjacency properties affect its communication efficiency. We now turn our attention to these properties.

Theorem 2.1 *The cell-to-node mapping θ has the property that the pre-image of every node contains exactly one cell for each value for a, b, and c and adjacent cells map to nodes which are at most distance two apart.*

Proof. Notice that $\theta(a, b, c)$ and $\theta(a + 1 \bmod 2^d, b, c)$ differ in exactly one bit position. This is also true if we add one to the b coordinate. However, $\theta(a, b, c)$ and $\theta(a, b, c + 1 \bmod 2^d)$ differ in exactly two bit positions. Therefore adjacent cells map to nodes which are at most distance two apart.

Consider fixing the value of a. As c varies over its 2^d values the low-order bits of $\theta(a,b,c)$ range over 2^d distinct values. For any particular value of c, varying b over its 2^d values causes the high-order d bits of $\theta(a,b,c)$ to range over 2^d distinct values. Accordingly, with a fixed, as b and c vary over their 2^{2d} values (cells), $\theta(a,b,c,)$ ranges over 2^{2d} distinct values (nodes). Hence, we can conclude that the pre-image of every node contains exactly one cell for each value of a. The argument can be repeated for the b and c coordinates. Therefore, the pre-image of every node contains exactly one cell for each value for a, b, and c. ■

Theorem 2.2 *The cell-to-node mapping θ results in processor loading that is perfectly balanced: each processor is assigned 2^d cells.*

Proof. From the previous Theorem, each processor is assigned exactly 1 cell for each value of a: Each processor is assigned exactly 2^d cells. ■

Boundary cells are those where at least 1 of the a, b, or c values are either 0 or $2^d - 1$. Boundary processing occurs at boundary cells. The complexity of boundary processing may differ from interior processing. It therefore is of interest to consider the distribution of boundary cells among the processors. Using the Inclusion/Exclusion principle, we see that there are $6 \cdot 2^{2d} - 12 \cdot 2^d + 8$ boundary cells. Since the number of processors (2^{2d}) does not divide the number of boundary cells, it is mathematically impossible to assign every processor the same number of boundary cells. The following theorem establishes an asymptotic approximation to this mathematically impossible, but desirable assignment of boundary cells.

Theorem 2.3 *Let P denote the set of processors that are assigned 6 distinct boundary cells, one from each of the 6 boundaries. Under the cell-to-node mapping θ, $|P| = 2^{2d} - O(2^d)$.*

Proof. It follows from Theorem 2.1 that each node labels exactly one cell on each of the six boundary faces. We want to count all those nodes that label a cell that is part of two or more boundary faces. If we simply count all the cells that are part of two or more boundary faces, this provides an upper bound on the number of processors that label such cells. It is not difficult to count these cells and we get $12 \cdot 2^d - 16 = O(2^d)$. ■

Except for $O(2^d)$ processors, this map has the remarkable property of assigning each of the 2^{2d} processors the same number of boundary cells *of each type*: It is asymptotically balanced with respect to the assignment of boundary cells. Again, it is mathematically impossible to assign the same number of boundary cells to all processors.

An example of such a mapping is shown in Fig. 5. In this case, cells which are adjacent in the z-direction are mapped into nodes which are at a distance 2 from each other. All other adjacencies are preserved.

3 A Negative Result

In the previous section, we noted that the best map would have:

- adjacent cells map to adjacent nodes;
- every node contain exactly one cell for each different value for a, b, and c.

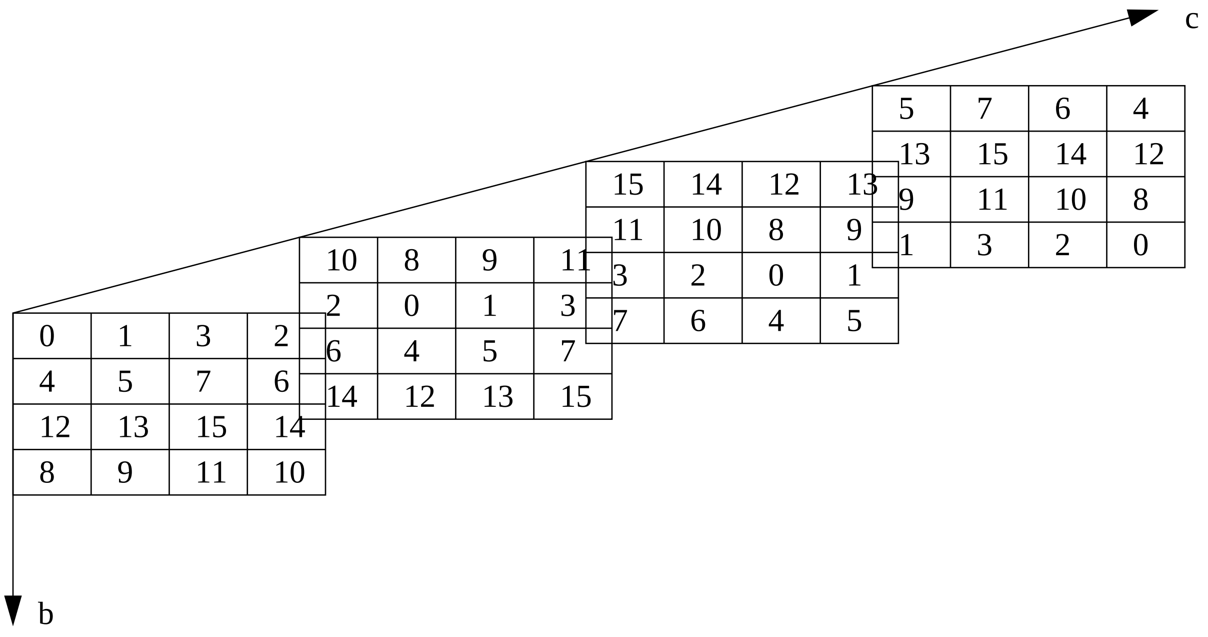

Figure 5: A Distance-2 Solution

In this section, we prove that such a map does not exist. If such a map did exist, then it would have the following properties:

- The map would be from a $2^n \times 2^n \times 2^n$ mesh of cells to a $2n$-dimensional hypercube of processors.
- Adjacent cells in the mesh would map to adjacent processors in the hypercube.
- A plane of the mesh has $2^n \times 2^n$ cells. For every mesh plane along the x, y, or z axis, each cell in the plane would map to a distinct processor in the hypercube.

This existence question is formalized as a mapping problem between graphs.

3.1 The Problem

Let $G_M = (V_M, E_M)$ be a $2^n \times 2^n \times 2^n$ mesh graph: $V_M = \{(i,j,k) | 1 \le i,j,k \le 2^n\}$, and $E_M = \{\{p,q\} | \forall p,q \in V_M, m(p,q) = 1\}$, where $m(p,q)$ denotes the Manhatten distance between $p, q \in V_M$.

Let $G_C = (V_C, E_C)$ be a $2n$-cube graph: $V_C = \{(i_1 i_2 \cdots i_{2n}) | i_j \in \{0,1\}, 1 \le j \le 2n\}$, and $E_C = \{\{p,q\} | \forall p,q \in V_C, h(p,q) = 1\}$, where $h(p,q)$ denotes the Hamming distance between $p, q \in V_C$.

Does there exist a surjection, $\eta : V_M \mapsto V_C$ satisfying the 2 conditions below?

1. $\forall p,q \in V_M, m(p,q) = 1 \Rightarrow h(\eta(p), \eta(q)) = 1$.
2. η is injective when restricted to any mesh plane along the x, y, or z axis.

3.2 Mesh Edge Labels

Let ζ be a map that satisfies condition 1 of η. If $\{p,q\} \in E_M$, then by condition 1, $h(\zeta(p), \zeta(q)) = 1$: $\zeta(p)$ differs from $\zeta(q)$ in exactly 1 bit position. This property is used to label the edges of the mesh.

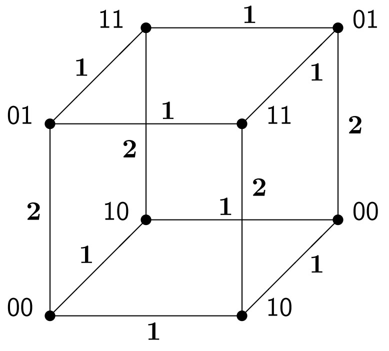

Figure 6: A ζ-labeling. Each mesh vertex is labeled with its image under ζ.

Definition *ζ-labeling*: A map $l_\zeta : E_M \mapsto \{1, 2, \dots, 2n\}$ such that $l_\zeta(\{p, q\}) = i$ when $\zeta(p)$ differs from $\zeta(q)$ in the i^{th} bit position (see, e.g., Fig. 6).

3.3 The Surjection Does Not Exist

Lemma 3.1.1 *Let $p \in V_M$. If η exists then*

$$\forall q \neq r \in V_M, [m(p, q) = 1 \wedge m(p, r) = 1] \Rightarrow l_\eta(\{p, q\}) \neq l_\eta(\{p, r\}).$$

Proof. (By contradiction.) Assume that if η exists then

$$\exists q \neq r \in V_M, m(p, q) = 1 \wedge m(p, r) = 1 \wedge l_\eta(\{p, q\}) = l_\eta(\{p, r\}).$$

Since q and r are adjacent to p, they share a plane. By condition 2 of η, their images are distinct. But $l_\eta(\{p, q\}) = l_\eta(\{p, r\}) \Rightarrow \eta(q) = \eta(r)$. ■

Theorem 3.1 *Let $p = (x, y), q = (x + 1, y), r = (x, y + 1), s = (x + 1, y + 1)$ represent 4 points in the mesh that form a square*[1]*. If η exists then opposite sides of the square have the same η-label.*

Proof. This theorem asserts that a square in the mesh must have an η-labeling like that depicted in Fig. 7.

Since q and r are adjacent to p in the mesh, $l_\eta(\{p, q\}) \neq l_\eta(\{p, r\})$, by Lemma 3.1.1. Let $l_\eta(\{p, q\}) = i$ and $l_\eta(\{p, r\}) = j$.

Case $l_\eta(\{q, s\}) = k \notin \{i, j\}$: Then $\eta(r)$ differs from $\eta(s)$ in 3 bit positions: i, j, and k. Since r is adjacent to s in the mesh, condition 1 on η is violated.

Case $l_\eta(\{q, s\}) = i$: This case is precluded by the condition that Lemma 3.1.1 imposes on η. Therefore, $l_\eta(\{q, s\}) = j$.

By an analogous case analysis, $l_\eta(\{r, s\}) = i$. ■

Corollary 3.1.1 *Let $p = (x_p, y_p, z_p), q = (x_p + 1, y_p, z_p) \in V_M$. If η exists and $l_\eta(\{p, q\}) = i$ then*

$$\forall \{r, s\} \in E_M, [x_r = x_p \wedge x_s = x_p + 1] \Rightarrow l_\eta(\{r, s\}) = i$$

[1]Hence, 2 coordinates suffice.

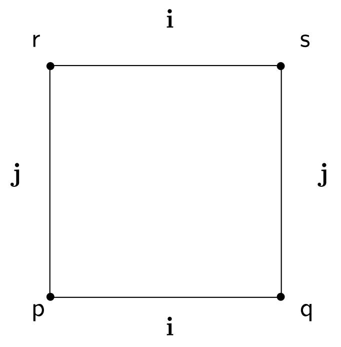

Figure 7: Mesh vertices p, q, r, and s whose edges form a square. $l_\eta(p,q) = i \Leftrightarrow l_\eta(r,s) = i$. $l_\eta(p,r) = j \neq i \Leftrightarrow l_\eta(q,s) = j$.

Proof. (By contradiction.) Assume that

$$\exists\{r,s\} \in E_M, x_r = x_p \wedge x_s = x_p + 1 \wedge l_\eta(\{r,s\}) \neq i$$

Let $S = \{\{t,u\} \in E_M | x_t = x_p \wedge x_u = x_p + 1\}$ and $T = \{e \in S | l_\eta(e) = i\}$. We are given that $\{p,q\} \in T$. Let $\{r,s\} \in S - T$ be an edge that is 'next to' an edge $\{t,u\} \in T$ (see Fig. 8). Then $l_\eta(\{t,u\}) = i$ and $l_\eta(\{r,s\}) \neq i$. The edges connecting mesh vertices r, s, t, and u form a square. The η-labeling of these edges however is precluded by Thm. 3.1. ■

From Corollary 3.1.1, one sees that if η exists, then all edges in a x–y 'plane' have the same η-label. This also is true for edges in a x–z, or y–z 'plane.' The definition of A_x, A_y, and A_z are illustrated by Fig. 9: A_x is the set of edges along the x 'axis;' A_y and A_z are defined similarly: $A_y = \{e_4, e_5, e_6\}$, and $A_z = \{e_7, e_8, e_9\}$. Let $P_x = \{i | \exists e \in A_x, l_\eta(e) = i\}$. P_y and P_z are defined similarly.

The following lemma shows that since the extent of the mesh is 2^n in each dimension, $|P_{x,y,z}| \geq n$.

Lemma 3.2.1 *If η exists then $|P_{x,y,z}| \geq n$.*

Proof. Let $S = \{\eta(v) | \exists u, \{u,v\} \in A_x\}$, the set of vertex labels of vertices on the x 'axis.'. By condition 2 of η, $|S| = 2^n$. In order to achieve 2^n distinct vertex labels, at least n different bit positions must vary. ■

Lemma 3.2.2 *If η exists then $P_x \cap P_y = \emptyset$, for $x \neq y$.*

Proof. (By contradiction.) Assume that $\exists i \in P_x \cap P_y$. One sees (Fig. 10) that $i \in P_x \cap P_y$ produces squares whose edges all have the same label. This labeling, however, is precluded by Lemma 3.1.1. ■

Theorem 3.2 *Let $G_M = (V_M, E_M)$ be a $2^n \times 2^n \times 2^n$ mesh graph, and $G_C = (V_C, E_C)$ be a $2n$-cube graph. There does not exist a surjection, $\eta : V_M \mapsto V_C$ satisfying:*

1. *$\forall p, q \in V_M, m(p,q) = 1 \Rightarrow h(\eta(p), \eta(q)) = 1$,*

2. *η is injective when restricted to any plane of the mesh.*

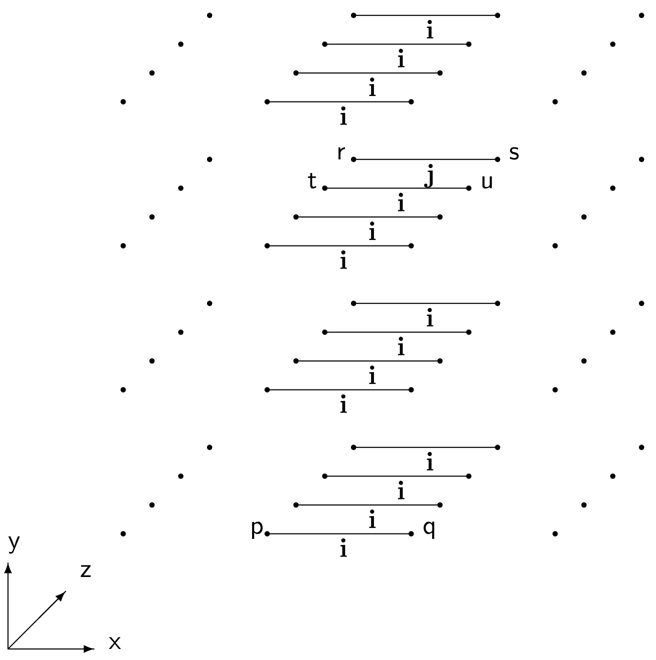

Figure 8: The vertices of a $4 \times 4 \times 4$ mesh are displayed. Only edges in S are shown.

Proof. Assume that η exists. By Lemma 3.2.2, $|P_x \cup P_y \cup P_z| = |P_x| + |P_y| + |P_z|$. By Lemma 3.2.1, $|P_x| + |P_y| + |P_z| \geq 3n$. But vertex names consist of only $2n$ bits. ∎

4 Conclusion

In this paper, we have considered implementing the 3D ADM on a hypercube concurrent processor system. Such an implementation entails mapping the grid points of the computational domain onto the nodes of a hypercube concurrent processor. The best map would be to have adjacent cells map to adjacent nodes (or the same node), and to have every hypercube node contain exactly one mesh cell for each different x coordinate, one mesh cell for each different y coordinate, and one mesh cell for each different z coordinate. We prove that such a map does not exist, but show how to construct the next best thing: a map in which cells that are adjacent in the z axis map to hypercubes nodes that are of distance exactly two apart, and in which x and y axis adjacencies are preserved. Moreover, this optimally adjacent map is shown to be perfectly load balanced.

Although this modulo-based map has been analyzed with respect to the hypercube concurrent processor, it is useful on any architecture that is capable of being configured as an m by n grid of processors with toroidal edge connections. If we assume N_a is a multiple of n and N_b is a multiple of m, then the mapping to an m by n torus is easy to specify and does not require the use of the Gray code transformation, namely, $\theta(a,b,c) = ((a+c) \bmod m, (b+c) \bmod n)$ where (r,s) denotes the processor located in the rth row and the sth column of the grid of processors. In order for each processor to be assigned an

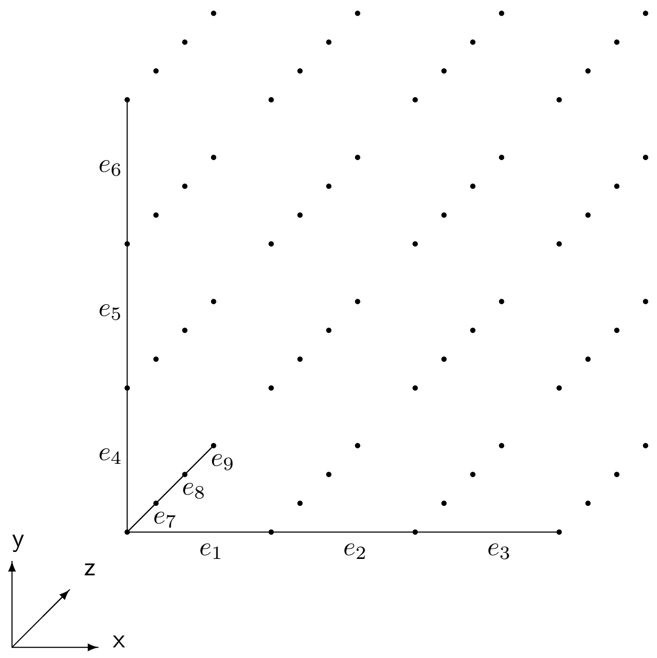

Figure 9: The vertices of a $4 \times 4 \times 4$ mesh are displayed. Edges along only the x, y, and z 'axes' are shown. $A_x = \{e_1, e_2, e_3\}$, the set of edges along the x 'axis.' A_y and A_z are defined similarly.

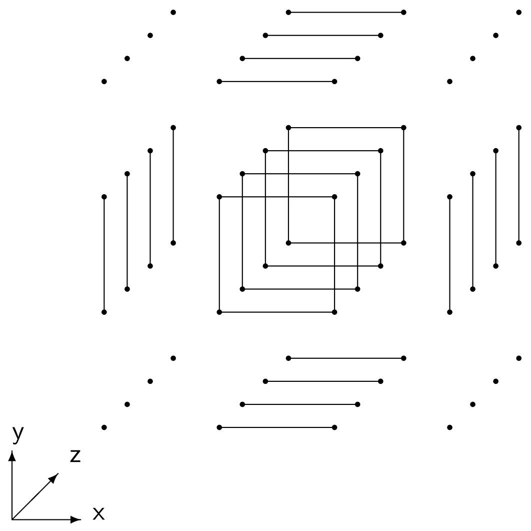

Figure 10: The figure displays the vertices of a $4 \times 4 \times 4$ mesh. Only those edges are shown whose label is i.

equal number of cells, it is necessary that N_c be a multiple of $\mathrm{lcm}(m, n)$. Normally, we would partition the computational domain so that $N_a = m$ and $N_b = n$ and $N_c = lcm(m, n)$. The reason is that creating more domain cells results in additional overhead in processor multiplexing within a node and additional inter-processor communication.

For the class of algorithms under consideration, one could use this mapping on SIMD machines which provide 2D toroidal or hypercube connections, such as the Goodyear MPP, the Thinking Machines Connection Machine, or the Active Memory Technologies DAP [3].

Acknowledgement

We benefited greatly from the criticism and suggestions of an anonymous referee.

References

[1] Geffrey C. Fox and et. al. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, 1988.

[2] Dennis B. Gannon and J. Van Rosendale. On the impact of communication complexity on the design of parallel numerical algorithms. *IEEE Trans. on Comput.*, C-33(12), December 1984.

[3] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, Bristol and Philadelphia, 1988.

[4] S. Lennart Johnsson, Youcef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. Dept. of Computer Sci. DCS/RR-382, Yale Univ., October 1985.

[5] R. Morison and Steve Otto. The scattered decomposition for finite element problems. *Journal of Scientific Computing*, 2, 1986.

[6] D. W. Peaceman and H. H. Rachford. Numerical solution of parabolic and elliptic differential equations. *SIAM J.*, 3:28–41, 1955.

[7] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[8] F. Saied, C-T Ho, A. Lennart Johnsson, and Martin Schultz. Solving schrodinger's equation on the intel ipsc by the alternating direction method. Dept. of Computer Sci. DCS/RR-502, Yale Univ., January 1987.

[9] J. Salmon. Binary gray codes and the mapping of a physical lattice into a hypercube. Caltech Concurrent Computation Group C^3P51, California Institute of Technology, January 1984.

[10] Charles Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[11] E. H. Twizell. *Computational Methods for Partial Differential Equations*. Mathematics and Its Applications. Ellis Horwood, Chichester, 1984.