Multi-criteria partitioning of multi-block structured grids

Hengjie Wang
University of California, Irvine
hengjiew@uci.edu

Aparna Chandramowlishwaran
University of California, Irvine
amowli@uci.edu

ABSTRACT

Partitioning of multi-block structured grids impacts the performance and scalability of numerical simulations. An optimal partitioner should achieve both load balance and minimize communication time. The state-of-art domain decomposition algorithms do a good job at balancing the load across processors. However, even if the work is well balanced, the communication cost might not be. The two main factors that impact communication cost are edge cuts and communication volume. The current partitioners primarily focus on reducing the total communication volume and rely on simple techniques such as cutting at the longest edge which does not capture the connectivity in the geometry. They also don't factor the effect of the network's latency and bandwidth for partitioning resulting in the same partition across all platforms. In addition, their performance tests mostly adopt a flat MPI model where the partition's effect on communication is hidden by the fast shared memory accesses between cores on the same node.

In this paper, we present new partitioning algorithms for multi-block structured grids that address the above limitations of current partitioners. The new algorithms include a cost function which not only accounts for both the communication volume and edge cuts but also takes into account the network's latency and bandwidth. We minimize the overall cost among all processors in an effort to create optimum partitions. To demonstrate the efficiency of the proposed algorithms, we test the partitioners with an MPI+OpenMP hybrid model where MPI routines handle inter-node communication and OpenMP threads take advantage of the shared memory within a node. On the Mira supercomputer, our partitioners coupled with a Jacobi solver demonstrate $5.5 - 15\times$ speedup in communication against a greedy algorithm on a synthetic multi-block structured grid and $1.5\times$ speedup on the Falcon Heavy Space-X grid consisting of 769 blocks.

KEYWORDS

Domain decomposition, multi-block structured grids, α-β model, MPI+OpenMP hybrid solver

ACM Reference Format:
Hengjie Wang and Aparna Chandramowlishwaran. 2019. Multi-criteria partitioning of multi-block structured grids. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330369

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330369

1 INTRODUCTION

Evolution of physics-based simulations and Computational Fluid Dynamics (CFD) in particular has fundamentally reshaped the design and engineering process in the last several decades. To simulate physical phenomena numerically, the domain is discretized with a grid. A PDE solver then computes the physical variables (such as density and velocities) either in the cells or on the vertices of the grid. When applying numerical methods to solve PDEs, the type of grid determines the type of solver and target optimizations.

There are two main types of grids namely, *structured and unstructured grids*. Structured grids are characterized by regular connectivity between its grid cells. They have the advantage that the physical grid maps ideally with the memory layout (cell i is adjacent to cell $i + 1$, both physically and in memory) and each cell's neighbor can be accessed by simply shifting the cell's index. Typically, structured grids for complex geometries such as an aircraft or turbo-machinery contain on the order of 100s of blocks. We call such grids *multi-block structured grids*. On the contrary, unstructured grids lack regular connectivity and can form arbitrarily connected shapes to capture complex geometries.

To run a CFD application in parallel, the multi-block structured or unstructured grid has to be further partitioned into sub-blocks to be distributed among the processors. A "good partitioner" should balance the workload i.e., the number of grid cells, among the processors to achieve load balance. The processors also communicate to exchange data (called *halo* cells) at the boundary of their respective computational domains (called *block2block* boundary) at regular intervals during the simulation. This results in inter-node communication when connected blocks reside on different processors. It is critical to minimize this communication. Therefore, the partitioner has to also take into account the cost of communication which is both application-specific (the number of halo cells is dependent on the numerical scheme) and architecture-specific (the communication cost is network dependent). There are two primary metrics that influence the cost of communication – the communication volume and the number of edge cuts. The former denotes the volume of data transferred through the network and the latter refers to the number of messages communicated between processors. As a result, there are numerous factors and trade-offs to consider when devising an optimal grid partitioner.

A grid can be represented as a graph. Several libraries exist such as METIS [12], CHACO [13] which can partition unstructured grids. However, a direct application of graph partitioners to structured grids is not feasible since it fails to preserve the regular connectivity. The work on partitioning structured grids can be categorized into two classes – *top-down* and *bottom-up* strategies. The *top-down* strategy either cuts off chunks of large grid blocks or groups small blocks to fill the capacity of available partitions. Greedy heuristics are typically suitable for this type of approach. A classical algorithm was proposed in [20] and later studies [1, 3, 5, 7, 18, 19] can all be

viewed as an improvement of a greedy framework. The key idea behind the *bottom-up* strategy is to treat a structured grid as a graph of blocks and then apply a graph partitioner. For large scale parallel simulations, the number of computation units, i.e., the number of partitions needed is typically larger than the number of grid blocks. To feed enough blocks to a graph partitioner, blocks are cut into smaller sub-blocks and then partitioned as a graph of sub-blocks [17]. Finally, the sub-blocks within the same partition are merged. A large number of small sub-blocks are desirable for graph partitioners but this is likely to result in excessive blocks and halo regions which can, in turn, impact the communication cost [14].

Prior works [1, 3, 7, 14, 17–20] mostly share the following drawbacks. First, they mainly focus on reducing the total communication volume by techniques such as cutting at the longest edge [3, 18, 20] and edge cuts are only implicitly factored by avoiding splitting block2block boundaries [19]. A cost function that estimates the communication cost introduced by cutting and assigning blocks which combines the communication volume and edge cuts is yet to be explored. Second, current domain decomposition schemes have only been studied in the context of a flat MPI or MPI-everywhere model where each MPI process is mapped to one core. The latency of data access within a node is significantly lower than accesses across nodes. Therefore, the partitioner should largely consider the cost of communication in the latter case. Finally, most partitioners produce the same partition regardless of the underlying network. Some researchers [5, 7] use the network bandwidth to estimate the communication cost of assigning a block but ignore the network's latency. We argue that the communication cost should take into account both the latency and bandwidth of the network. The goal of this paper is to make a concerted effort to design *portable multi-criteria partitioners for multi-block structured grids.*

To that end, this paper makes the following contributions.

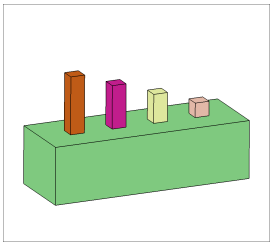
- We use the $\alpha - \beta$ model to construct a cost function for inter-node communication (Section 2). The cost function captures both the total communication volume and the number of edge cuts in the network. The latency, α, and bandwidth, β values are measured empirically on the target platform, which ensures the portability of our partitioner.
- We design new *top-down* partitioning algorithms to minimize the total communication cost estimated by the above cost function. The algorithms are composed of new heuristics to cut large blocks with a minimal increase in communication cost and to group small blocks to map block2block boundaries to shared memory accesses (Section 2).
- We evaluate the quality of the proposed algorithms on two 3D geometries namely, a synthetic 5-block grid and the SpaceX's Falcon Heavy rocket consisting of 769 blocks. A hybrid MPI-OpenMP Jacobi solver is used to evaluate different partition's effect on performance (Section 3). On the Mira supercomputer, the new algorithms outperform the top-down greedy heuristic by $1.5 - 3\times$ (Section 4).

2 DOMAIN DECOMPOSITION ALGORITHMS

In sections 2.1 and 2.2, we first describe one *top-down* and one *bottom-up* algorithm as the baseline for comparison. Among the *top-down* algorithms, the classical greedy heuristic [20] is most widely used in CFD software such as elsA [9]. Although researchers [1, 19] claim improvements over the greedy heuristic, their performance enhancement is observed in the context of a flat MPI model. As a result, the partitions' effect on inter-node communication is not entirely clear. In addition, their implementation details are not available to re-produce their partitioners. Therefore we choose the classical greedy heuristic as the baseline for *top-down* algorithms. As for the *bottom-up* algorithm, the creation of sufficient small sub-blocks is critical while the choice of the actual graph partitioner is not as important. Therefore, we choose METIS for its popularity and widespread use.

Given the number of partitions, P, the blocks are divided into large and small sub-blocks based on the average workload $\overline{W}$ i.e., the average number of grid cells in a partition. A *top-down* partitioner must handle two tasks – (a) cut large blocks into sub-blocks such that each sub-block fits in one partition, and (b) group small blocks to fill remaining partitions. A small block may also be cut if the partition's remaining capacity is not large enough. In section 2.3, we propose a cost function based on the well-known $\alpha - \beta$ model to decide which block to cut/group such that the communication cost is minimized. In the following sub-sections 2.4 - 2.7, we develop new algorithms for cutting large blocks and grouping small blocks based on the new cost function.

We explain the different partitioning strategies using an example synthetic grid called *Bump3D* which consists of 5 blocks as shown in Figure 1. Bump3D has one block that is significantly larger than the others which challenges the algorithms' ability to cut large blocks. Note that although Bump3D is synthetic, it resembles the flow through a pipe with outlets on the sides.



(a) Geometry

Block ID	Size
0	$224 \times 64 \times 80$
1	$16 \times 16 \times 16$
2	$16 \times 32 \times 16$
3	$16 \times 48 \times 16$
4	$16 \times 64 \times 16$

(b) Block Sizes

Figure 1: Illustration of the geometry of the Bump3D grid with 5 blocks and its corresponding block sizes.

2.1 Greedy Algorithm

The greedy algorithm [20] chooses the largest unassigned block (i.e. the maximum number of grid cells) and the most underload partition at any step. If the block exceeds the remaining capacity of the partition, a sub-block is cut off to fill the remaining capacity and the remainder is added to the list of unassigned blocks. Otherwise, the entire block is assigned to that partition. The algorithm terminates when all blocks have been assigned. To minimize communication volume, the greedy algorithm always cuts across the longest edge.

In the greedy strategy, the cut position is round up to an integer, which can result in load imbalance. For instance, if the largest block has a size of $8 \times 8 \times 32$ and the average workload $\overline{W} = 544$, splitting along the longest edge at z = 8 or 9 would introduce 6.75% load

imbalance. Even worse, the algorithm may fail if the smallest surface of a block has more cells than $\overline{W}$. To address this limitation, we propose the following solution. Given an imbalance tolerance ϵ, a block of size $N_x \times N_y \times N_z$, $N_x < N_y < N_z$, and remaining capacity R of the most underload partition, if a cut position c_z satisfying $|R - N_x N_y c_z| < \epsilon\overline{W}$ cannot be found along the longest edge, the algorithm traverses possible cut positions c_z and c_y in the longest and second longest direction to minimize the difference $|R - N_x c_y c_z|$. Once cut positions are located, the block is cut into four sub-blocks and a block of size $N_x c_y c_z$ is assigned to the partition.

We denote the greedy algorithm [20] with the above fix as *pure greedy (PG)*. This algorithm still has two drawbacks. First, when the prescribed load imbalance tolerance is small, it may create too many small blocks and increase both the communication volume and number of edge cuts. Second, the arrangement of small blocks does not respect the connectivity of blocks and results in increased communication volume. As seen from Figure 2a and Table 1, this algorithm creates small blocks at the end of the original large block and also result in large communication volume.

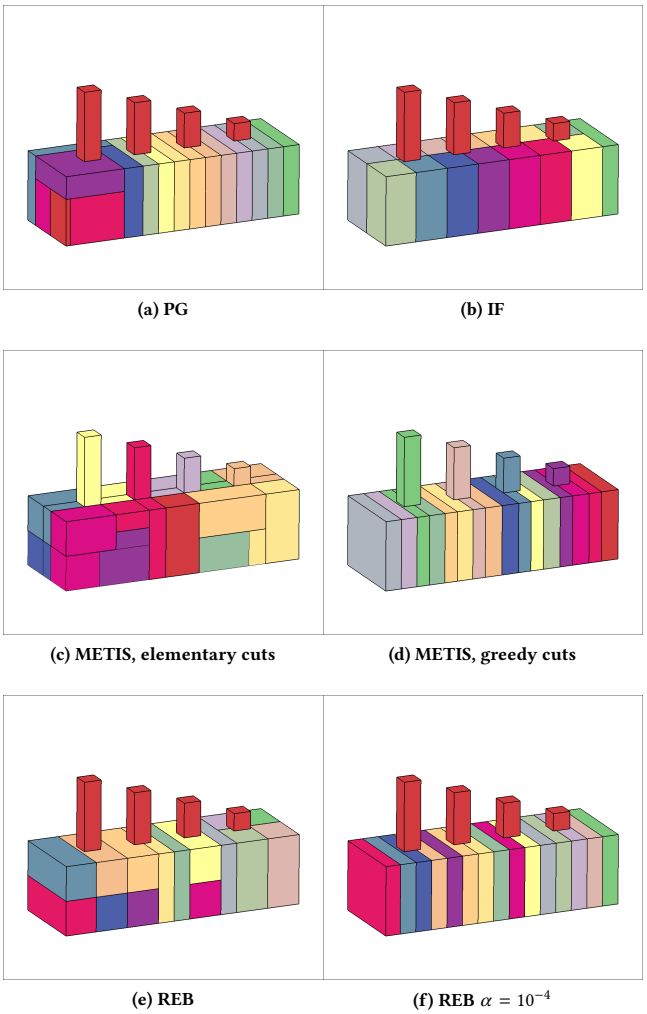
Table 1: Partition metrics of different algorithms for the Bump3D grid when $P = 16$. The latency and bandwidth are set to 10^{-5} (s) and 10^{9} (bytes/s).

Algorithm	Load imbalance ratio	Total volume	Total edge cuts	Total cost
PG	0.035	2.19E+06	60	2.79E-3
METIS 1	0.214	1.72E+07	138	3.35E-3
METIS 2	0.103	2.47E+06	38	2.85E-3
REB	0.047	1.57E+06	66	2.23E-3
IF	0.035	1.61E+06	66	2.27E-3

2.2 Bottom-up Algorithm

The bottom-up algorithm decomposes the original blocks into smaller sub-blocks and partitions them using a graph partitioner (the sub-blocks are treated as vertices of a graph). Sub-blocks within the same partition are merged after partitioning. Graph partitioners like METIS move vertices between partitions to achieve load balance and reduce communication cost. If there are too few vertices to move or the vertices have large differences in weight, the graph partitioner may produce imbalanced partitions. Therefore, it is desirable to have the number of small sub-blocks to be at least several times the number of partitions and to be of equal size.

In this paper, we examine two strategies. Both strategies try to create sub-blocks of size one-quarter of the average workload $\overline{W}$. The first method is to decompose the original blocks into elementary blocks [1] i.e., blocks with only one boundary condition on each surface. If the elementary block is still too large to fit in one partition, it is further cut by our IF algorithm proposed in section 2.5. The second method is to directly decompose the large blocks with IF. As shown in Figures 2c and 2d, different decomposition strategies can result in very different partitions. Clearly, the first method results in too many sub-blocks in this case. The second method generates simple connectivities in the graph of sub-blocks and therefore easier to partition. Note that in Figure 2d, the four original small blocks are grouped with their connected sub-blocks in one partition. This shows that the graph partitioner preserves the connectivity between blocks. However, it is also prone to load imbalance and large communication cost compared with the other algorithms as seen from Table 1.



(a) PG **(b) IF**

(c) METIS, elementary cuts **(d) METIS, greedy cuts**

(e) REB **(f) REB $\alpha = 10^{-4}$**

Figure 2: Partitions created by the different algorithms for the Bump3D grid when $P = 16$. Blocks of the same color belong to the same partition.

2.3 Measure of Communication

The commonly used metrics for communication in both graph partitioning and domain decomposition are the total communication volume and the total number of edge cuts. Top-down approaches try to minimize the total volume but not the total number of messages. However, in some cases, latency can be more important than volume. The time to send a message between two nodes consists of two components – startup time (or latency) and the time for sending or receiving data which is proportional to the length of the message. The cost of sending a message can be approximated by the $\alpha - \beta$ model as,

$$t_m = \alpha + S_{msg}/\beta, \tag{1}$$

where α denotes the latency, S_{msg} is the size of the message in bytes, and β is the bandwidth of the network. Summing this cost over all the messages results in the following total cost.

$$\text{Total cost} = \alpha \cdot \text{Total Cuts} + \frac{\text{Total Volume}}{\beta}$$

Therefore, the impact of communication volume and edge cuts on the total cost depends on the chosen network's latency and bandwidth. For instance, in Table 1, the second partition created by METIS has fewer edge cuts but a large communication volume. The total cost based on specific α and β enables us to evaluate the partition's quality against other algorithms.

For a block2block boundary with surface area A, the communication cost $t_{b2b}(A)$ is given by

$$t_{b2b}(A) = \alpha + \frac{A \cdot \#\text{halo} \cdot S_{\text{cell}}}{\beta}. \tag{2}$$

The number of halo layers (#halo) and the size of data per grid cell (S_{cell}) depend on the specific solver running on the partition. Using Equation 2, the communication cost of a block can be computed by summing the cost of all its block2block boundaries.

Note that we use the $\alpha - \beta$ model solely as a cost function to capture both the communication volume and edge cuts rather than as a prediction of the actual communication time. Such a prediction would not be realistic since this model's simple formulation is derived from several ideal assumptions about the network such as free of congestion, minimal queue lengths, etc [10]. To profile the communication time accurately, more realistic models such as logGPS [11] can be used and is beyond the scope of this work.

2.4 Find a Cut of Block

The elementary operation in partitioning a grid is to cut off a sub-block of a given workload from a block. The function find_min_cut shown in Algorithm 1 chooses the cut that adds the minimum communication cost δt_{cut} among all possible cutting positions allowed by the imbalance tolerance, ϵ.

Algorithm 1 Find the cut of a block to fit in a given workload

1: **function** find_min_cut(B, W_{cut}, ϵ, cut, p)
▷ B: block to be cut.
▷ W_{cut}: workload to be cut off
▷ ϵ: tolerance
▷ cut: data structure for cut info
▷ p: current partition (optional input, empty by default)
2: $\delta t_{min} = \infty$
3: **for** i = x, y, z **do**
4: Get area of i's norm face A_i
5: posFloor = floor($W_{cut}(1-\epsilon)/A_i$)
6: posCeil = ceiling($W_{cut}(1+\epsilon)/A_i$)
7: **for** pos∈[posFloor, posCeil] **do**
8: $\delta t_{cut} = \sum_{b2b \ni pos} \alpha + t_{b2b}(A_i) - \sum_{B_i \in p} t_{b2b}(\text{cut}, B_i)$
9: **if** $\delta t_{cut} < \delta t_{min}$ **then**
10: $\delta t_{min} = \delta t_{cut}$
11: cut.pos = pos

The communication cost of cutting a block, δt_{cut} is computed in line 8 of Algorithm 1. The first term includes the latency increase if the cut splits any block2block boundary on the orthogonal surfaces (each cutting plane is orthogonal to four surfaces of a block). Adding this term allows the algorithm to align the sub-blocks' boundary with block2block boundaries. The second term adds the communication cost of the new surface created by the cut. When the cut sub-block is assigned to partition p, the block2block boundaries in contact with the blocks in p become shared memory accesses, which is much faster than inter-node communication and therefore subtracted from δt_{cut} in the last term. The last term takes advantage of the blocks' connectivity to reduce overall communication volume.

2.5 Cut Large Blocks

Given a large block that fits evenly in multiple partitions, we present two approaches for cutting such blocks namely, Recursive Edge Bisection (*REB*) and Integer Factorization (*IF*).

2.5.1 ***Recursive Edge Bisection (REB).*** The classical REB recursively bisects the block at the longest edge until each resulting sub-block fits in a partition. Such a bisection ignores the block2block boundaries. As a result, when a bisection intersects a block2block boundary, it increases the edge cut. We improve REB by using Algorithm 1 to find the cut position such that it is aligned with block2block boundaries. Our modified REB is outlined in Algorithm 2. Note that the imbalance from the first several bisections may accumulate to the final sub-blocks and lead to overload partitions violating the imbalance tolerance. A fix is proposed in section 2.7.

Algorithm 2 Recursive Edge Bisection

1: **function** reb_block(B, n_p)
▷ Block B fits in n_p partitions.
2: **if** $n_p == 1$ **then**
3: return
4: $W = B$'s workload
5: $W_l = W \cdot \frac{\lfloor n_p/2 \rfloor}{n_p}$, $W_r = W - W_l$
6: find_min_cut(B, W_l, ϵ, cut)
7: cut B into B_l with workload W_l and B_r with workload W_r
8: reb_block(B_l, $\lfloor n_p/2 \rfloor$)
9: reb_block(B_r, $\lceil n_p/2 \rceil$)

Each bisection found by Algorithm 1 introduces minimum communication cost at that step. Therefore, REB bounds communication in a greedy fashion. The partition created by REB for Bump3D grid is shown in Figure 2e and the corresponding metrics in Table 1. REB produces the least communication volume compared to the other algorithms. Figure 2f shows the decomposition created by REB with a latency, $\alpha = 10^{-4}s$. Given the large latency, edge cuts become the dominating metric for communication. As a result, the algorithm now aims to reduce edge cuts at the cost of increased communication volume.

2.5.2 ***Integer Factorization (IF).*** Given the number of partitions, n_p for a large block, a factorization $n_p = n_x \cdot n_y \cdot n_z$ according to the block's length ratio, i.e., $n_x : n_y : n_z \approx l_x : l_y : l_z$, can be an

optimum decomposition for that block. However, there are two limitations to this decomposition. First, when n_p is prime, the unique factorization $1 \cdot 1 \cdot n_p$ may not be proportional to the length ratio. Second, it may split block2block boundaries and increase the communication cost. In [1], when n_p is prime, a sub-block is cut off to feed one partition and the algorithm searches for an optimum factorization of $n_p - 1$ to decompose the remaining sub-block. We propose a more generalized solution in Algorithm 3 to address both limitations.

Algorithm 3 Integer Factorization

```
1:  function factorize_block(B, n_p)
    ▷ Block B fits in n_p partitions.
2:      if n_p == 1 then
3:          return
4:      t^0_min = ∞, t^1_min = ∞
5:      for factorization n_x · n_y · n_z = n_p do
6:          if t_b2b(B, n_x, n_y, n_z) < t^0_min then
7:              t^0_min = t_b2b(B, n_x, n_y, n_z)
8:              n^0_x = n_x, n^0_y = n_y, n^0_z = n_z
9:      find_min_cut(B, W̄, ϵ, cut)
10:     for factorization n_x · n_y · n_z = n_p − 1 do
11:         t^1 = max(t_b2b(B_cut), t_b2b(B_rem, n_x, n_y, n_z))
12:         if t^1 < t^1_min then
13:             t^1_min = t^1
14:     if t^0_min < t^1_min then
15:         cut B by n^0_x, n^0_y, n^0_z
16:     else
17:         factorize_block(B_rem, n_p − 1)
```

Algorithm 3 compares two cases. First, decompose the block B according to the factorization of n_p which introduces the minimum communication cost (lines 4-8). The cost of a factorization $t_{b2b}(B, n_x, n_y, n_z)$ is the maximum communication cost among $n_x \cdot n_y \cdot n_z$ sub-blocks. Second, cut off a sub-block, B_{cut} of average workload and decompose the remaining sub-block B_{rem} by the factorization of $n_p - 1$ which results in minimum overall communication cost (lines 9-13). If the first case costs less than the second, the factorization for block B is the optimum decomposition. Otherwise, the same comparison repeats on B_{rem}. Note that the imbalance problem mentioned in Section 2.1 also exists here. Therefore, the optimum decomposition of the blocks does not guarantee a load imbalance ratio below the given tolerance.

Figure 2b and Table 1 illustrate the partition created by IF. Each cut in IF cuts through the entire block. Therefore, compared with REB, IF is more apt to align block2block boundaries and reduces the edge cuts.

2.6 Group Small Blocks

The block2block connections between blocks require inter-node communication by default. By grouping several small blocks into one partition, we can convert some of this communication into shared memory accesses and reduce the overall communication cost. Therefore, the grouping algorithm should group blocks connected by large block2block boundaries in the same partition. Keeping this goal in mind, we propose the following two algorithms.

2.6.1 ***Cut-Combine-Greedy (CCG).*** After assigning a small block to an empty partition, there might still be room to fit additional blocks. As shown in Algorithm 4, CCG traverses all the unassigned blocks to find a block or sub-block that fits in the remaining capacity such that the communication cost is minimized. In line 6, $t_{b2b}(B, B_i)$ denotes the communication cost of all block2block connections between block B and B_i. The communication cost saved by adding a sub-block to a partition is computed using δt_{cut} in line 8 of Algorithm 1. This procedure is repeated until the given partition, p is full. This algorithm is a greedy heuristic since each *company* only minimizes the communication cost for that step but not the final partition.

Algorithm 4 Find company to fit in one partition

```
1:  function find_min_company(W_ub, cmpny, p)
    ▷ W_ub: upper bound of the company's work load
    ▷ cmpny: ID of the company block
    ▷ p: the partition to be filled
2:      δt_min = ∞
3:      for block B ∈ {unassigned blocks} do
4:          Get B's work load W
5:          if W < W_ub then
6:              δt = − Σ_{B_i ∈ p} t_b2b(B, B_i)
7:              if δt < δt_min then
8:                  δt_min = δt
9:                  cmpny = B.ID
10:         else
11:             find_min_cut(B, W_ub, ϵ, cut, p)
12:             if cut.δt < δt_min then
13:                 δt_min = δt
14:                 cmpny = cut.ID
```

2.6.2 ***Graph-Growth-Sweep (GGS).*** GGS first assigns each empty partition a small block in lines 4-5 of Algorithm 5. Using the small block as a seed, it starts the graph growing procedure for each partition. If moving a block can reduce the communication cost, i.e., $B.\delta t < 0$ in line 13, it will be saved for that partition. The saved blocks are sorted by the amount of communication reduced and then assigned to a partition until the partition is full. All the partitions are swept repeatedly until no more blocks can be assigned. After the sweep, the unassigned blocks, if any, are partitioned using the pure greedy algorithm.

2.7 Combined Algorithms and Partition Adjustment

We propose new partitioning algorithms by combining the methods for cutting large blocks and grouping small blocks. Following the streamline in [1], any block larger than the average workload is first truncated to a main sub-block which fits evenly in multiple partitions and a residual sub-block. The main sub-block is cut and assigned to partitions using either REB or IF. The residual sub-blocks together with the remaining small blocks are grouped using CCG or GGS.

Algorithm 5 Graph-Growth-Sweep

```
1:  Assign each partition a small block
2:  while blocks can be assigned do
3:      for all p ∈ P do
4:          if p.empty() then
5:              Assign a block to p
6:          else
7:              for all Block B connected to blocks in p do
8:                  if B fits in p's room then
9:                      B.δt = − Σ_{B_i ∈ p} t_b2b(B, B_i)
10:                     if B.assigned() then
11:                         p_B := B's partition
12:                         B.δt = B.δt + Σ_{B_i ∈ p_B} t_b2b(B, B_i)
13:                     if B.δt < 0 then
14:                         Save B in block array blks
15:             Sort blks in ascending order of blks[i].δt
16:             while !p.full() and i < blks.size() do
17:                 Assign blks[i] to p
18:                 i++
```

Both REB and IF may generate blocks that exceed the prescribed load imbalance tolerance at the trade-off of reducing communication cost. In case the computation becomes too imbalanced, we use a greedy heuristic to adjust the workload between overload and underload partitions. We denote a shift between two partitions as a sub-block cut from a block in one partition and moved to the other partition. The adjustment heuristic sorts all possible shifts from overload partitions to underload partitions according to its communication cost using an efficient *Bucket List* structure introduced in [8]. The adjustment starts from the shift that results in the minimum communication cost and updates the partition and *Bucket List* until all the overload partitions are within the given imbalance tolerance.

3 EXPERIMENTAL SETUP

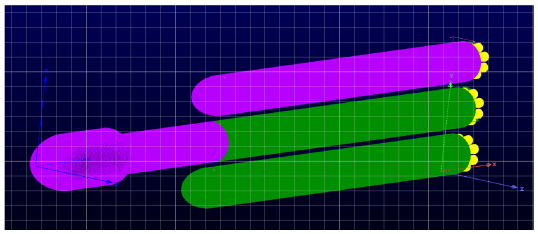
To examine the impact of the proposed algorithms, we apply our partitioners to two multi-block structured grids and simulate a benchmark solver with the resulting partitions. The experiments are conducted on the Mira supercomputer. We measure the latency and bandwidth of Mira and use that as the input for our domain decomposition algorithms. The load imbalance tolerance, ϵ is set to 5%. This section describes the geometries of the grids, benchmark solver, and Mira supercomputer.

3.1 Multi-block Structured Grids

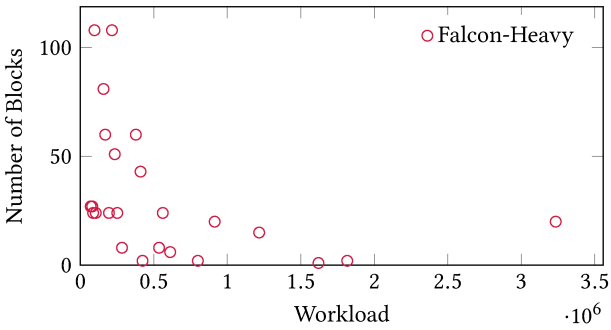
We refine the Bump3D grid 4 times in each direction. The sizes are shown in Table 2. This grid stresses the partitioner's ability to cut a large block. The second grid is based on the SpaceX Falcon Heavy rocket shown in Figure 3a. The complex geometry results in a large number of blocks with varying block workloads as shown by the block distribution in Figure 3b. This grid is chosen to test both the effect of cutting large blocks and the result of grouping small blocks.

Table 2: Refined blocks of the Bump3D grid.

Block ID	0	1	2	3	4
Workload	7.3E+7	2.6E+5	5.2E+5	7.8E+6	1.0E+6



(a) Geometry



(b) Block Distribution

Figure 3: SpaceX Falcon Heavy grid consisting of 769 blocks.

3.2 Case study: Jacobi solver

The experiments are conducted with an hybrid MPI+OpenMP Jacobi solver. The computation is a finite difference scheme using a 5-point stencil in each direction. Two halo layers extend outside the blocks' boundary. The communication consists of the exchange of one double precision variable i.e., 8 bytes per halo cell. The numerical experiment is done in the manner as outlined in Algorithm 6 for 512 iterations and the average running time per iteration is reported.

The function pack_halo_to_buffer packs the data to be sent into a 1D buffer. The function unpack_buffer_to_halo unpacks the buffer's data to the halo region. Both functions are executed by all the OpenMP threads in parallel. The master thread calls non-blocking MPI routines i.e., MPI_Isend, MPI_Irecv, and MPI_Waitall for inter-node communication. All the threads exchange the halos inside the same partition via shared memory copy. This step is often overlapped with inter-node communication. The computation is done by all threads in parallel where the blocks in one partition are evenly split among the threads. There are OpenMP barriers between each step for synchronization.

3.3 Platform and Architecture

We evaluate the proposed partitioning schemes on the Mira supercomputer, i.e. the petascale IBM BlueGene/Q cluster at the Argonne National Lab. Each node has a PowerPC A2 processor with 16 cores clocked at 1.6 GHz with 1GB DDR3 memory. The interconnect is a 5D torus. We measure the latency and bandwidth of Mira using

Algorithm 6 Numerical performance experiment

```
for i = 1 → NSTEP do
▹ #pragma omp for
    pack_halo_to_buffer
▹ #pragma omp barrier
▹ #pragma omp master
    update_halo
▹ #pragma omp for
    copy_halo_shared_mem
▹ #pragma omp barrier
▹ #pragma omp for
    unpack_buffer_to_halo
▹ #pragma omp barrier
▹ split blocks evenly among threads
    compute
▹ #pragma omp barrier
```

a *ping-pong* benchmark, i.e. timing two adjacent nodes exchanging messages with non-blocking MPI routines and fitting Equation 1 with the least squares method. The latency and bandwidth are measured as 1.73E-05 s and 1.77E+09 bytes/s respectively.

4 RESULTS AND DISCUSSION

In this section, we first evaluate the quality of the partitioners. Specifically, we compare the communication volume, the number of edge cuts, load imbalance ratio, number of sub-blocks created, and the total communication cost of the decomposition generated by the different algorithms, namely top-down pure greedy (PG), bottom-up graph algorithm (METIS), recursive edge bisection for cutting large blocks with cut combine greedy strategy for grouping small blocks (REB+CCG), REB with graph growth sweep for grouping small blocks (REG+GGS), integer factorization for cutting large blocks with CCG (IF+CCG), and IF with GGS (IF+GGS). Then, we compare their performance coupled with the OpenMP+MPI hybrid Jacobi solver. For large scale simulations, the applications are typically run on at least $O(10^3)$ nodes. Therefore, we only report the performance results from 1024 nodes.

4.1 Quality of multi-block structured mesh partitioners

4.1.1 ***Bump3D.*** Table 3 compares the quality of the different heuristics for the Bump3D grid on up to 4096 nodes of Mira. Since the decomposition of Bump3D is dominated by cutting a single large block, REB+CCG and REB+GGS have similar results. The same is true for IF+CCG and IF+GGS. Across the board, all the schemes result in significantly less total communication volume compared to PG at all processor counts. On the other hand, METIS has the highest number of total edge cuts. This is because METIS creates an excessive number of sub-blocks and consequently, both PG and METIS have a higher total communication cost compared to the proposed algorithms (REB and IF). The results also confirm the fact that REB is better at reducing communication volume while IF is better at reducing edge cuts.

The total communication cost captures the effect of both communication volume and edge cuts as given by Equation 1. For instance, on 64 nodes PG creates less cut edges but more communication volume than all the other algorithms. The total cost indicates that in this case communication volume plays a more important role than edge cuts and PG results in a higher cost than the other algorithms. On the other hand, on 4096 nodes, IF has the lowest communication cost. This indicates that at the highest processor count used for the experiments in this paper, reducing the number of total edge cuts is more critical than reducing the total communication volume. This further validates the need for a portable partitioner that is driven by a flexible cost model.

4.1.2 ***Falcon Heavy.*** Table 4 compares the quality of the different heuristics for the Falcon Heavy grid on up to 4096 nodes of Mira. For 64-256 nodes, METIS results in the lowest communication volume, edge cuts, and communication cost. Second to METIS are algorithms using CCG to group small blocks. To explain this, we introduce two parameters $\overline{n_{sm}}$ and $\%n_{psm}$. $\overline{n_{sm}}$ denotes the average number of small blocks that remain after cutting large blocks and $\%n_{psm}$ is the percentage of partitions filled with small blocks. As seen from Table 4, for 64-256 nodes, more than 60% of the partitions are made up of small blocks and such partitions have more than 4 blocks on average. Therefore, the partitioner's ability to group small blocks determines the partition's quality. As a graph partitioner, METIS is good at exploiting connectivity to reduce communication cost. As shown in Figure 4, although METIS creates more halos than other algorithms, it maps a large percentage of halo exchange to shared memory copy. A similar trend can also be observed for CCG, which uses connectivity in a greedy fashion to reduce communication. PG does not take into account the blocks' connectivity and introduces the highest communication cost. The small blocks' influence damps as the number of nodes increases. For 512 to 4096 nodes, REB+GGS produces the optimum partition because the sweeping process in Algorithm 5 effectively avoids cutting blocks or introducing communication. On the other hand, METIS loses its strength at large node count and leads to the most edge cuts and communication cost due to its creation of too many sub-blocks.

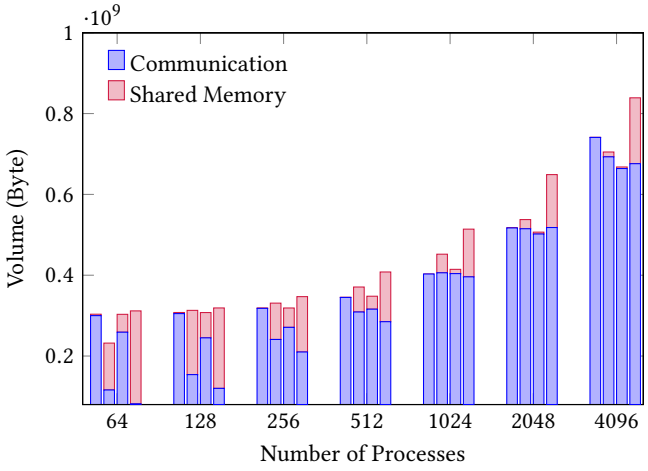


Figure 4: The total halo volume of PG, REB+CCG, REB+GGS, and METIS from left to right for the Falcon Heavy grid.

Table 3: Partitioner quality for the Bump3D grid on Mira.

Number of processors, P	64	128	256	512	1024	2048	4096
Load imbalance PG	0.035	0.046	0.045	0.048	0.042	0.047	0.050
Load imbalance METIS	**0.256**	**0.517**	**0.491**	**0.516**	**0.226**	**0.253**	**0.257**
Load imbalance REB+CCG	0.035	0.050	0.044	**0.075**	**0.106**	**0.100**	**0.134**
Load imbalance REB+GGS	0.035	0.050	0.044	**0.075**	**0.106**	**0.100**	**0.122**
Load imbalance IF+CCG	0.035	0.019	0.035	0.035	0.043	**0.086**	**0.214**
Load imbalance IF+GGS	0.035	0.019	0.035	0.035	0.043	**0.086**	**0.214**
Number of sub-blocks PG	65	211	275	601	1090	2505	4855
Number of sub-blocks METIS	**89**	**166**	**421**	**866**	**1638**	**3532**	**7103**
Number of sub-blocks REB+CCG	**67**	131	259	514	**1025**	**2048**	**4096**
Number of sub-blocks REB+GGS	**67**	131	259	514	**1025**	**2048**	**4096**
Number of sub-blocks IF+CCG	**67**	**129**	**258**	**513**	**1025**	**2048**	**4096**
Number of sub-blocks IF+GGS	**67**	**129**	**258**	**513**	**1025**	**2048**	**4096**
Communication volume PG	**1.38E+08**	**2.94E+08**	**4.59E+08**	**9.32E+08**	**9.81E+08**	**8.21E+08**	**1.06E+09**
Communication volume METIS	5.28E+07	7.16E+07	9.93E+07	1.36E+08	1.75E+08	2.35E+08	**3.02E+08**
Communication volume REB+CCG	4.96E+07	6.89E+07	9.05E+07	**1.24E+08**	1.72E+08	**2.33E+08**	3.03E+08
Communication volume REB+GGS	**4.94E+07**	**6.86E+07**	**9.04E+07**	**1.24E+08**	1.71E+08	**2.33E+08**	**3.02E+08**
Communication volume IF+CCG	5.63E+07	1.24E+08	1.37E+08	1.47E+08	**1.67E+08**	3.25E+08	3.93E+08
Communication volume IF+GGS	5.61E+07	1.24E+08	1.37E+08	1.47E+08	**1.67E+08**	3.25E+08	3.93E+08
Edge cuts PG	**220**	**1108**	1592	3794	8340	18186	41188
Edge cuts METIS	**538**	1052	**2728**	**5790**	**11312**	**25548**	**54136**
Edge cuts REB+CCG	336	746	1616	3278	6744	14282	29086
Edge cuts REB+GGS	332	742	1614	3276	6778	14180	29104
Edge cuts IF+CCG	276	442	1012	2476	5388	**9938**	22352
Edge cuts IF+GGS	272	**438**	**1010**	**2474**	**5386**	**9938**	**22288**
Communication cost PG	**8.15E-02**	**1.85E-01**	**2.87E-01**	**5.91E-01**	**6.97E-01**	**7.77E-01**	**1.31E+00**
Communication cost METIS	3.91E-02	5.85E-02	1.03E-01	1.77E-01	2.94E-01	5.74E-01	1.10E+00
Communication cost REB+CCG	**3.38E-02**	5.17E-02	7.89E-02	1.27E-01	2.13E-01	3.78E-01	6.73E-01
Communication cost REB+GGS	3.36E-02	**5.15E-02**	**7.88E-02**	1.27E-01	2.14E-01	3.76E-01	6.73E-01
Communication cost IF+CCG	3.65E-02	7.77E-02	9.46E-02	**1.26E-01**	**1.87E-01**	**3.55E-01**	6.07E-01
Communication cost IF+GGS	3.63E-02	7.75E-02	9.45E-02	**1.26E-01**	**1.87E-01**	**3.55E-01**	**6.06E-01**

Note that PG results in comparable or even less cost compared with our algorithms at 1024 and 2048 nodes. Unlike Bump3D where the largest block occupies the majority of the partitions, the largest block of Falcon Heavy only occupies 12 and 25 partitions on 1024 and 2048 processors respectively. As a result, PG's greedy heuristic of cutting at the longest edge leads to a near optimum partition. However, at 4096 nodes, the largest block occupies 51 partitions and PG's disadvantage of creating excessive blocks re-appears and increases the communication cost.

4.2 Performance and Scalability

Figure 5 shows the running time of the different algorithms coupled with the Jacobi solver for the Bump3D and Falcon Heavy grids. The time consists of communication, computation, and others which is mainly the time for packing and unpacking the halos for communication (refer to Algorithm 6). As predicted by the communication cost in Table 3, METIS, REB and IF result in significantly better performance compared to PG for the Bump3D grid. The best performance is from the partition created by IF which achieves $5.5 - 15\times$ speedup for communication and $3\times$ overall speedup compared to PG. Note that in Table 3, only PG keeps the load imbalance within tolerance ($\epsilon = 5\%$) for all cases while METIS and IF cause more than 20% imbalance at 4096 nodes. Unlike PG, REB and IF don't pay the penalty of creating an excessive number of sub-blocks which in turn results in increased communication cost at the expense of increased load imbalance. Although METIS does create more blocks than PG, its lower communication volume indicates that a large amount of communication between blocks goes through the shared memory. Therefore, this trade-off between load balance and total communication cost is desirable in this case.

The trend for Falcon Heavy as seen from Figure 5 is likewise consistent with the communication cost estimation in Table 4 except for METIS at 4096 nodes. METIS achieves the worst performance at 1024 and 2048 nodes as predicted. However, the unexpected reduction of its communication time at 4096 nodes requires further investigation. PG leads to slightly better performance than CCG algorithms and IF+GGS at 1024 and 2048 nodes but has the worst runtime at 4096 nodes. The best performance comes from REB+GGS, which achieves $1.5\times$ overall speedup and $2.1\times$ better communication time compared to PG.

Note that the communication time stops scaling at 4096 nodes for the Jacobi solver while the communication cost estimated by the $\alpha - \beta$ model still continues to scale. As remarked earlier, the $\alpha - \beta$ model is only used as a cost function rather than as a prediction

Table 4: Partitioner quality for the SpaceX Falcon Heavy grid on Mira.

Number of processors, P	64	128	256	512	1024	2048	4096
$\overline{n_{sm}}$	12	7.1	4.3	2.6	1.7	1.6	1.8
$\%n_{psm}$	100%	84.3%	69.5%	57.2%	41.5%	22%	9.8%
Load imbalance PG	0.005	0.028	0.022	0.040	0.047	0.049	0.049
Load imbalance METIS	**0.097**	**0.210**	**0.283**	**0.311**	**0.729**	**0.973**	**0.236**
Load imbalance REB+CCG	**0.100**	**0.100**	**0.100**	**0.097**	**0.100**	**0.089**	**0.098**
Load imbalance REB+GGS	0.049	0.049	0.034	0.048	0.047	**0.066**	**0.118**
Load imbalance IF+CCG	**0.100**	**0.100**	**0.100**	**0.059**	**0.097**	**0.078**	**0.099**
Load imbalance IF+GGS	0.049	0.049	0.034	0.048	**0.059**	**0.082**	**0.109**
Number of sub-blocks PG	**769**	**789**	**847**	995	**1466**	2445	4585
Number of sub-blocks METIS	**810**	**843**	**1005**	**1527**	**2652**	**4715**	**9006**
Number of sub-blocks REB+CCG	787	817	927	1147	1668	2620	4620
Number of sub-blocks REB+GGS	**769**	**789**	**847**	**993**	1477	2443	4443
Number of sub-blocks IF+CCG	787	817	927	1147	1668	2617	4614
Number of sub-blocks REG+CCG	**769**	**789**	**847**	**993**	1486	**2441**	**4429**
Communication volume PG	**3.00E+08**	**3.05E+08**	**3.18E+08**	**3.45E+08**	4.03E+08	5.17E+08	**7.41E+08**
Communication volume METIS	**8.17E+07**	**1.20E+08**	**2.10E+08**	**2.85E+08**	**3.96E+08**	5.18E+08	6.76E+08
Communication volume REB+CCG	1.16E+08	1.54E+08	2.41E+08	3.09E+08	4.06E+08	5.15E+08	6.93E+08
Communication volume REB+GGS	2.59E+08	2.45E+08	2.71E+08	3.16E+08	4.04E+08	**5.02E+08**	**6.64E+08**
Communication volume IF+CCG	1.16E+08	1.54E+08	2.41E+08	3.09E+08	**4.07E+08**	5.21E+08	6.92E+08
Communication volume IF+GGS	2.59E+08	2.45E+08	2.71E+08	3.16E+08	4.06E+08	**5.42E+08**	6.74E+08
Edge cuts PG	**3718**	**3826**	**4216**	5330	9268	17652	37598
Edge cuts METIS	**944**	**1466**	**3240**	**6890**	**14412**	**29418**	**57834**
Edge cuts REB+CCG	1368	1726	3398	5312	9504	16568	31428
Edge cuts REB+GGS	3266	3266	3626	**4622**	**8540**	**15392**	**30286**
Edge cuts IF+CCG	1368	1726	3410	5358	9394	16814	32746
Edge cuts IF+GGS	3266	3266	3626	4638	8822	16294	30508
Communication cost PG	**2.33E-01**	**2.38E-01**	**2.52E-01**	**2.87E-01**	3.87E-01	5.96E-01	1.07E+00
Communication cost METIS	**6.24E-02**	**9.27E-02**	**1.74E-01**	2.80E-01	**4.72E-01**	**8.00E-01**	**1.38E+00**
Communication cost REB+CCG	8.92E-02	1.17E-01	1.95E-01	2.66E-01	3.93E-01	5.76E-01	9.33E-01
Communication cost REB+GGS	2.03E-01	1.94E-01	2.15E-01	**2.58E-01**	**3.75E-01**	**5.49E-01**	**8.97E-01**
Communication cost IF+CCG	8.92E-02	1.17E-01	1.95E-01	2.67E-01	3.92E-01	5.84E-01	9.56E-01
Communication cost IF+GGS	2.03E-01	1.94E-01	2.15E-01	**2.58E-01**	3.81E-01	5.87E-01	9.07E-01

of the communication runtime. Nevertheless, it is still worthwhile to analyze the gap between the cost model and the actual measured time. Two factors may contribute to this gap. First, in our performance experiments, we ignore the topology of the network. Communication cost between any two partitions is estimated based on the latency and bandwidth values measured using a *ping-pong* benchmark between two adjacent nodes. However, two partitions may be mapped to two nodes that are physically separated by several hops. Second, some fraction of the total communication time is spent on waiting for other processes to issue their messages sends. This idle time may take up to 80% of the total communication time [16] for some applications. The waiting time, in turn, depends on several factors such as the overall time of communicating processes, load imbalance, and the congestion in the network. Nevertheless, as observed by the experimental data, the cost function is still a powerful and useful predictor for domain decomposition and results in better partitioning than the current state-of-the-art heuristics for multi-block structured grids.

5 RELATED WORK

Among the *top-down* strategies, the greedy heuristic [20] is the most widely adopted method. REB [4] is a good alternative to the former. In [2], a greedy heuristic is combined with REB. At any step, the largest block is assigned to the most underloaded partition. After assigning all blocks, if a certain number of partitions is overloaded, then the same number of large blocks is bisected in half at the longest edge. This process is repeated until all partitions are within the load imbalance tolerance. It is hard to say if this hybrid approach is better than the classical greedy heuristic [20] since no comparison has been made.

A decomposition according to the block's aspect ratio is optimum in the number of edge cuts. This idea is used in [1] for 2D grids. Compared to REB [4], this strategy results in less imbalance but more communication volume. No performance comparison is made. More recently, this algorithm has been extended to 3D problems [15]. Given the number of partitions, all their test grids are made of blocks larger than the average workload. Therefore, the grouping of small blocks is not clearly shown.

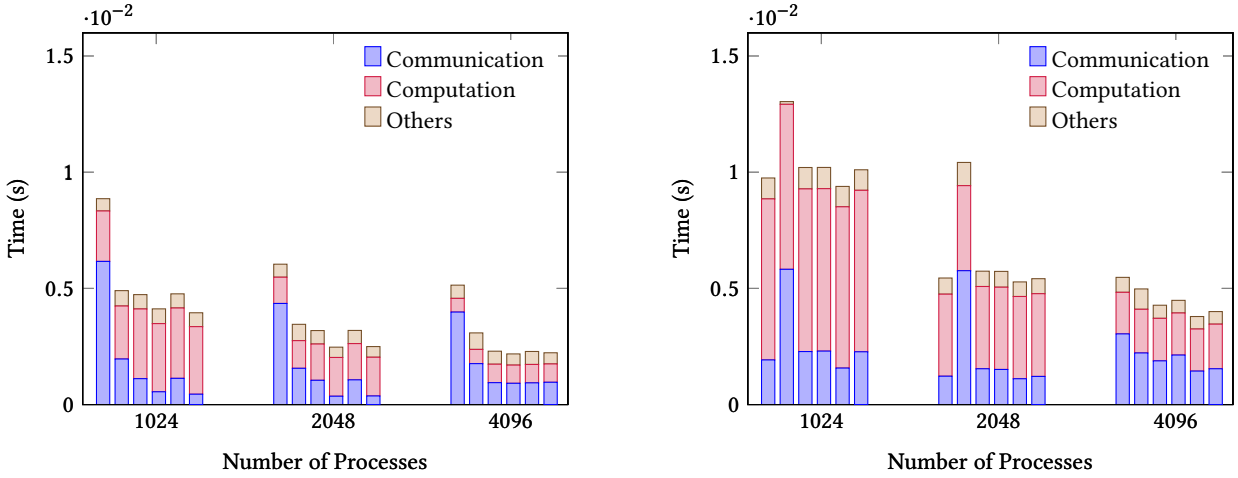


Figure 5: The average running time of 1 iteration of the Jacobi solver coupled with the different paritioners for the Bump3D (left) and Falcon Heavy (right) grids. Each bar from the left to right represent the performance of the solver coupled with PG, METIS, REB+CCG, IF+CCG, REB+GGS, and IF+GGS respectively.

In [19], large blocks are cut into cubic sub-blocks as much as possible because, for a fixed volume, the cubic shape has the minimum surface area. The residual blocks are assumed to have a minor effect on performance. Coupled with a CFD solver, they demonstrate $1.2 - 1.4\times$ speedup up to 800 processes against the greedy heuristic [20]. However, the communication time stops scaling between 300-400 processes.

In the above works [1, 2, 4, 19, 20], the communication volume can be viewed as the cost function. In [6, 7], the total running time is chosen as the cost function and balanced by grouping small sub-blocks with a greedy heuristic similar to [20]. The communication time is estimated using the bandwidth of the network. As discussed in Section 4.2, the actual communication time can be larger than the cost estimated using bandwidth and latency alone. Therefore, the estimated total running time may not provide a good insight for designing partitioners.

Compared to top-down strategies, unfortunately, there is considerably limited literature on *bottom-up* strategies for multi-block structured grids. The idea is first proposed in [17] where the original blocks are split into small sub-blocks, the number of which needs to be three times more than the partitions. Then, a graph partitioner is used to partition the small blocks. This algorithm is compared with *top-down* algorithms on a 2D multi-element and demonstrates improved performance. More recently, a new method for decomposing the original blocks is proposed in [14] which results in fewer blocks and communication volume than decomposing the blocks with REB. The effect on the performance of a solver is not yet assessed.

To the best of the author's knowledge, this paper is the first of its kind to design portable multi-criteria partitioners for multi-block structured grids using a cost function which not only accounts for both the communication volume and edge cuts but also takes into account the network's latency and bandwidth.

6 CONCLUSIONS

We use the $\alpha - \beta$ model to realize a new cost function to partition structured multi-block grids. Based on the cost function, we proposed two new methods to cut large blocks, namely the Recursive Edge Bisection (REB) and Integer Factorization (IF). REB recursively bisects a block at the position that introduces the minimum communication cost. IF decomposes a block according to the factorization of the number of partitions assigned which minimizes the communication cost introduced by cutting blocks. We also propose two methods to group small blocks, Cut-Combine-Greedy (CCG) and Graph-Growth-Sweep (GGS). CCG fills a partition by searching for a block or a cut-off sub-block which converts inter-node communication into shared memory accesses. This method works very well when the small blocks occupy a large percentage of the partitions. GGS first assigns one small block to each empty partition and then repeatedly performs graph growing until no more blocks can be assigned. It avoids cutting blocks as much as possible and results in less communication cost than CCG at large node counts. New domain decomposition algorithms are derived by combining REB/IF with CCG/GGS. We apply our algorithms to partition a synthetic grid, Bump3D and a grid based on SpaceX's Falcon Heavy rocket with 769 blocks with varying block distribution. The partitions are tested with a hybrid MPI+OpenMP Jacobi benchmark solver on the Mira supercomputer. Compared with the popular greedy heuristic, our algorithms result in $5.5 - 15\times$ speedup in communication for Bump3D and $1.5\times$ speedup for Falcon Heavy at 4096 nodes.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (NSF) under the award number 1750549. Any opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect those of NSF. We also wish to thank Pavan Balaji at Argonne National Lab for his insights on MPI and Ferran Marti, a former postdoctoral researcher in our group for generating the Falcon Heavy grid used in the experiments.

REFERENCES

[1] E. Ahusborde and S. Glockner. 2011. A 2D block-structured mesh partitioner for accurate flow simulations on non-rectangular geometries. *Computers and Fluids* 43, 1 (2011), 2–13. https://doi.org/10.1016/j.compfluid.2010.07.009

[2] Kwesi Parry Apponsah. 2012. *Multi-block CFD Applications Applied To A Parallel Newton-Krylov Algorithm.* Ph.D. Dissertation. University of Toronto.

[3] K P Apponsah and D W Zingg. 2012. A Load Balancing Tool for Structured Multi-Block Grid CFD Applications. In *20th Annual Conference of the CFD Society of Canada.*

[4] Shahid H. Bokhari. 1987. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.* C-36, 5 (1987), 570–580. https://doi.org/10.1109/TC.1987.1676942

[5] Y. P. Chien, F. Carpenter, A. Ecer, and H. U. Akay. 1995. Load-balancing for parallel computation of fluid dynamics problems. *Computer Methods in Applied Mechanics and Engineering* 120, 1-2 (1995), 119–130. https://doi.org/10.1016/0045-7825(94)00048-R

[6] M. Jahed Djomehri and Rupak Biswas. 2003. Performance enhancement strategies for multi-block overset grid CFD applications. *Parallel Comput.* 29, 11-12 SPEC.ISS. (2003), 1791–1810. https://doi.org/10.1016/j.parco.2003.05.019

[7] M Jahed Djomehri, Rupak Biswas, Noe Lopez-Benitez, and Bryan Biegel. 2002. Load balancing Strategies for Multi-Block Overset Grid Applications. (2002).

[8] C. M. Fiduccia and R. M. Mattheyses. 1988. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation - 25 years of DAC*. 241–247. https://doi.org/10.1145/62882.62910

[9] Laurent Y.M. Gicquel, N. Gourdain, J. F. Boussuge, H. Deniau, G. Staffelbach, P. Wolf, and Thierry Poinsot. 2011. High performance parallel computing of flows in complex geometries. *Comptes Rendus - Mecanique* 339, 2-3 (2011), 104–124. https://doi.org/10.1016/j.crme.2010.11.006

[10] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. 2010. Toward performance models of MPI implementations for understanding application scaling issues. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6305 LNCS (2010), 21–30. https://doi.org/10.1007/978-3-642-15646-5_3

[11] F Ino, N Fujimoto, and K Hagihara. 2001. LogGPS: A parallel computational model for synchronization analysis. *ACM SIGPLAN Notices* 36, 7 (2001), 133–142. https://doi.org/10.1145/568014.379592

[12] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[13] R Leland. 1995. *The Chaco User's Guide Version 2.0.* Technical Report. Technical Report SAND95-2344, Sandia National Laboratories, Albaquerque, NM 87185-1110.

[14] Hongkang Liu, Chao Yan, Yatian Zhao, and Boxi Lin. 2016. An improved partitioning strategy for structured multiblock grids. *Proceedings of 2016 7th International Conference on Mechanical and Aerospace Engineering, ICMAE 2016* (2016), 322–326. https://doi.org/10.1109/ICMAE.2016.7549559

[15] P. Lubin and S. Glockner. 2015. Numerical simulations of three-dimensional plunging breaking waves: Generation and evolution of aerated vortex filaments. *Journal of Fluid Mechanics* 767 (2015), 364–393. https://doi.org/10.1017/jfm.2015.62

[16] Qingyu Meng, Justin Luitjens, and Martin Berzins. 2010. *Dynamic task scheduling for scalable parallel AMR in the Uintah framework.* Technical Report. Citeseer.

[17] J Rantakokko. 2000. Partitioning strategies for structured multiblock grids. *Parallel Comput.* 26, 12 (2000), 1661–1680. https://doi.org/10.1016/S0167-8191(00)00044-2

[18] Kurt Sermeus and Eric Laurendeau. 2007. Parallelization and Performance Optimization of Bombardier Multiblock Structured Navier-Stokes Solver on IBM eserver Cluster 1600. *Aerospace* January (2007), 1–24. https://doi.org/10.2514/6.2007-1109

[19] Min Xiong, Chuanfu Xu, Xiang Gao, Dali Li, Dandan Qu, Zhenghua Wang, and Xiaogang Deng. 2018. Improved grid partitioning algorithms for load-balancing high-order structured aerodynamics simulations. *Computers and Electrical Engineering* 67 (2018), 70–84. https://doi.org/10.1016/j.compeleceng.2018.03.016

[20] Anders Ytterström. 1997. A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics. *The International Journal of Supercomputer Applications and High Performance Computing* 11, 4 (1997), 336–343. https://doi.org/10.1177/109434209701100407