Heuristic-based techniques for mapping irregular communication graphs to mesh topologies

Abhinav Bhatele and Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {bhatele, kale}@illinois.edu

***Abstract*— Mapping of parallel applications on the network topology is becoming increasingly important on large supercomputers. Topology aware mapping can reduce the hops traveled by messages on the network and hence reduce contention, which can lead to improved performance. This paper discusses heuristic techniques for mapping applications with irregular communication graphs to mesh and torus topologies. Parallel codes with irregular communication constitute an important class of applications. Unstructured grid applications are a classic example of codes with irregular communication patterns. Since the mapping problem is NP-hard, this paper presents fast heuristic-based algorithms. These heuristics are part of a larger framework for automatic mapping of parallel applications. We evaluate the heuristics in this paper in terms of the reduction in average hops per byte. The heuristics discussed here are applicable to most parallel applications since irregular graphs constitute the most general category of communication patterns. Some heuristics can also be easily extended to other network topologies.**

***Keywords*-mapping; interconnect topology; communication optimization; performance; irregular patterns**

I. Introduction

The field of high performance computing has made tremendous progress in the last ten years. The size of the largest machines has increased by over an order of magnitude from 10,000 cores in 2001 to 500,000 in 2011. Interconnect topology of the supercomputers can play an important role in determining application performance at this scale. Interference both within and across jobs can affect performance which necessitates topology aware mapping of codes to processors.

N-dimensional mesh and torus interconnects are in widespread use today on the largest machines, in part due to their ease of design and installation. IBM Blue Gene and Cray XT/XE machines are relevant examples of such supercomputers. The machine at the top on the June 2011 Top500 list[†] has a 6D mesh interconnect [1]. Increasing sizes of such machines leads to networks with a large diameter. Messages travel farther with increasing network diameters leading to sharing and contention for links. Network contention can degrade application performance and hence, there is a need for topology aware mapping algorithms for parallel applications [2], [3].

[†]http://www.top500.org/lists/2011/06

Mapping of one graph on to another is a well-analyzed problem in mathematics, VLSI and parallel computing. It is also known to be NP-hard [4], [5]. Techniques from genetic algorithms, simulated annealing, graph partitioning and heuristics-based methods have been used to attack this general problem [5]–[12]. In this paper, however, we focus specifically on mapping of applications with irregular communication graphs to mesh topologies. Graph partitioning libraries such as SCOTCH provide support for mapping graphs to network topologies [13]. However, there are no published results using such libraries for mapping scientific applications running on real hardware. We also exploit domain specific knowledge about the application such as geometric coordinates associated with the physical space being simulated, to aid our mapping decisions. Application-specific information has not been used by most mapping algorithms and frameworks, to the best of our knowledge.

Heuristics presented in this paper are part of a larger mapping framework that handles both regular and irregular communication graphs. The automatic mapping framework is described in [14] which discusses various aspects of the mapping problem – obtaining the application graph and processor topology, pattern matching to identify regular patterns and heuristics for mapping of regular (structured) communication graphs.

The mapping heuristics presented in this paper are evaluated in terms of the success in reducing the average hops traveled per byte,

$$\textit{average hops per byte} = \left(\sum_{i=1}^{n} d_i \times b_i\right) \div \left(\sum_{i=1}^{n} b_i\right)$$

where d_i is the number of hops traversed by a message of b_i bytes and n is the total number of messages sent. The hop-bytes metric or hops per byte gives an approximate indication of the overall contention on the network [7], [15]. Although it does not capture hot-spots created on specific links, it is still an easily derivable metric that correlates well with actual application performance when communication to computation ratio is high [16].

II. The Mapping Problem

The mapping problem involves computing a mapping for each node/task/object in the application communication graph

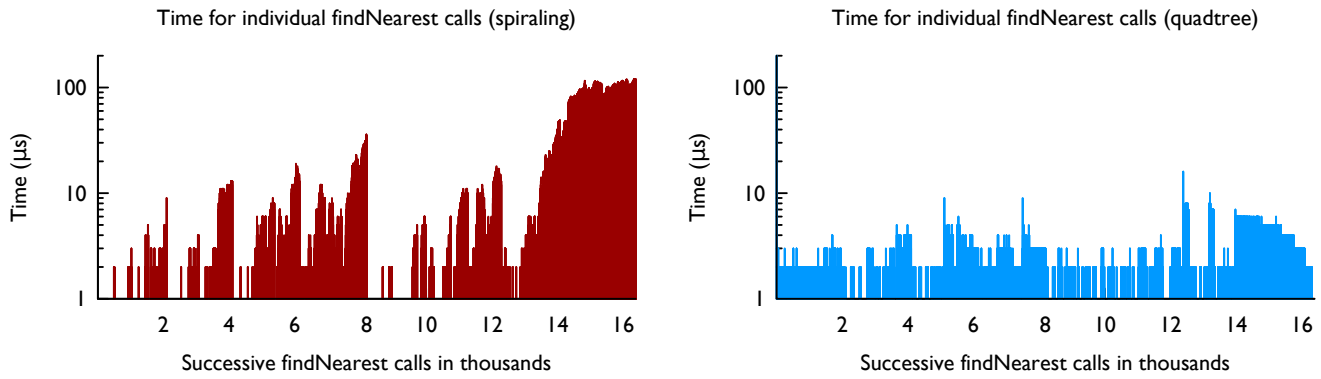

Fig. 1. Execution time (in microseconds) for 16, 384 consecutive calls to the spiraling and quadtree algorithms for `findNearest` from the `AFFN` mapping algorithm for irregular graphs

to a processor in the physical topology. Irregular communication graphs provide an instance of the mapping problem in its most general form. In an irregular communication graph, each node can have an arbitrary number of neighbors and the weights on the edges can also be different. We discuss two kinds of heuristics to handle two different cases –

Scenario 1: There is no information about the physics behind the application, from which the communication graphs were obtained. In this case, we use heuristics that exploit the neighbor relations between different nodes (Section IV). The heuristics make no assumptions about patterns in the communication graph.

Scenario 2: It is known that the application has a geometric structure even though the graph is irregular. Quite often, when simulating fractures in planes or solid objects using unstructured grids, the tasks in the parallel application have some geometric coordinate information associated with them, and the communication structure is related to the geometry (*i.e.* entities with nearby coordinates communicate more.) If we have this coordinate information, we can exploit it to do a better mapping (Section VI). Even if we do not have access to this information but the domain is known to have a geometric structure, we can try to infer the geometric arrangement of the tasks using graph layout algorithms (Section V).

The heuristics presented in this paper are applicable for mapping to two or three-dimensional mesh topologies but also extensible to any other topology. This is facilitated by the general idea on which all algorithms in this paper are based: At each step, they select a "suitable" object to map and find a "desirable" processor to place the object on. If the desired processor is not available (it is overloaded based on some criteria), another processor *close to this processor* is chosen. As long as we can define functions to choose a desirable processor and to find the nearest available processor for a specific topology, the heuristics are generally applicable.

We begin with providing an efficient implementation to find the nearest available processor in a two-dimensional (2D) mesh. We then discuss mapping heuristics for different scenarios and evaluate the heuristics based on the mapping of communication graphs obtained from an unstructured grid application.

III. Finding the Nearest Available Processor

We want to find the nearest available processor given 1) a "desirable" processor, and 2) a table indicating which processors are available. One possible implementation is to start at the desirable processor and spiral around it, first looking at processors at distance 1, then distance 2, 3 and so on. All processors at a certain distance are enumerated by choosing one coordinate (x) first and then calculating the other coordinate (y) based on the current value of distance being considered. The first available processor that we come across is returned as the answer. We refer to this as the spiraling (through enumeration) algorithm for finding the nearest available processor.

The spiraling implementation presented above has a worst case time complexity of $\mathcal{O}(p)$ where p is the number of processors. Hence, if `findNearest2D` is called for each node during mapping, it leads to a worst-case time complexity of $\mathcal{O}(p^2)$ for the mapping algorithm (number of nodes in the communication graph, $n = p$). Figure 1 (left) shows the running time for the algorithm when it is called from one of the mapping algorithms (`AFFN`, see Section VI) for irregular graphs. All timing runs for this paper were done on a 2.4 GHz Intel Core 2 Duo processor. Towards the end (for the last two thousand calls), the execution time for `findNearest2D` calls is quite significant. As more and more processors become unavailable, spiraling around the desirable processor continues for longer and longer distances before an available processor is found. This can be avoided, in practice, by keeping a list of the available processors when their number drops below a certain threshold.

However, using an alternate implementation based on a quadtree data structure (octree in case of 3D), we think that the average-case running time of this algorithm can be reduced further. It should be noted that both implementations give the

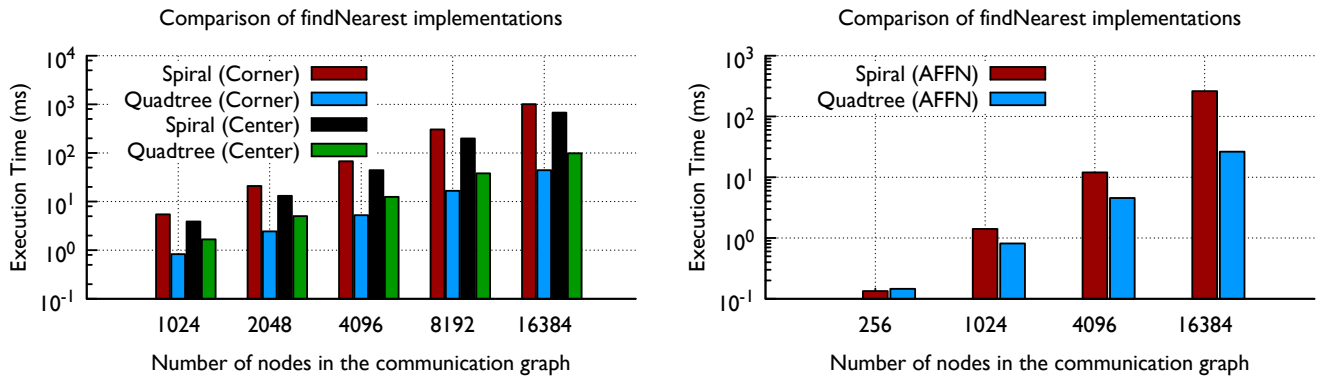

Fig. 2. Comparison of execution time (in milliseconds) for spiraling and quadtree implementations for synthetic cases (left) and when invoked from the AFFN algorithm (right)

same result for the next nearest processor given a desirable processor and a list of unavailable processors.

A. Quadtree: An Alternative to Spiraling

We build a quadtree representing the 2D mesh of processors. Each leaf in the tree holds one processor and each intermediate node in the tree represents a subdomain of the mesh (all processors in the subtree under it). The tree is obtained by recursive bisection of the mesh into approximately equal halves along both dimensions. The number of levels in the tree is $\log_4 p$. At each node, we maintain information about the number of available processors in its subtree and the extent of the subdomain of the mesh controlled by it.

To find the nearest available processor, we start at the leaf that holds the desirable processor. If it is available, we return immediately. If not we traverse up the tree to its parent and see if any of the parent's children have an available processor. This is done recursively until we reach the root of the tree. To avoid visiting each node in the tree, several pruning criteria are applied:

1) At each level, the intermediate nodes store the number of available processors in the respective subtrees. We go down a particular node only if it has at least one available processor.
2) At any point in the search, the best solution so far (in terms of the smallest hops to the desirable processor) is maintained. We do not visit those nodes for which all processors under their subtree are farther away from the desirable processor than the current best solution.

Traversals up and down the quadtree depend on the height of the tree which is $\log_4 p$. When looking for a nearest available processor we start from a leaf and traverse all the way to the root (which takes $\mathcal{O}(\log p)$). At each intermediate node encountered on the way, we might go down the tree depending on if we expect to find a processor in that sub-tree. We believe that the pruning criteria mentioned above can restrict the running time for this algorithm to $\mathcal{O}(\log^2 p)$ by avoiding unnecessary traversals (not proven yet). Figure 1 (right) shows the running time for the algorithm when it is called from one of the mapping algorithms (AFFN) for irregular graphs. We can see that most individual calls take no longer than 10 μs (compared to up to 100 μs in the case of spiraling). And, since both implementations return the nearest processor each time, they produce mapping results of the same quality in most cases. In the next section, we compare performance of mapping algorithms when using the two implementations of `findNearest`.

B. Comparison between Spiraling and Quadtree

On the average, calls to the quadtree implementation take significantly less time than the spiraling one (1.8 μs versus 16 μs, see Figure 1). Figure 2 (left) compares the execution time for the two implementations for a synthetic search scenario. We start with an empty 2D mesh (all processors available) and look for processors around a certain processor, making them unavailable as we find them. The total time for finding all the processors is recorded. The first two columns represent the case where we look for processors close to $(0, 0)$ and the remaining two refer to the case where we look for processors close to the processor at the center of the mesh. It is evident that the savings from using the quadtree implementation can be huge in some cases. For example, when looking for processors around $(0, 0)$, the speedup over the spiraling implementations is nearly 23 times for a graph of $16,384$ nodes (note the logarithmic scale on the y-axis).

Let us try to see the impact of using spiraling and quadtree in one of the mapping algorithms. We noticed that the AFFN heuristic (described in Section VI) for irregular graphs takes a significant amount of time. Figure 2 (right) shows the execution times for the mapping algorithm for different number of nodes. We can see that for more than $1,024$ nodes, quadtree is the correct choice for `findNearest`. At $16,384$ nodes, the run that uses a quadtree is nearly 10 times faster than the one that uses spiraling.

IV. Strategies for General Communication Graphs

In the most general mapping problem, we have a communication graph for an application and we do not have any

information about patterns, structure or geometry of the graph. In this scenario, we use heuristics that exploit the neighbor relations between different nodes. They do not assume that the communication graph has any spatial properties.

Heuristic 1 - Breadth First Traversal: A simple approach is to map nodes of a graph as we traverse it breadth-first. We start with a randomly chosen node (typically one with the id zero) and place it on processor zero. Then we map the neighbors of the mapped node near it and put neighbors of the mapped neighbors in a queue. Neighbors for a given node are mapped in an arbitrary order, and are mapped around the processor on which the given node is mapped. This algorithm is referred to as `BFT` in the following figures.

Since the algorithm visits each element in the graph once, it takes linear time (assuming that the search for a nearest available empty processor takes constant time). In the worst case, `findNearest2D` can take $\mathcal{O}(n)$ time at each call. However, if the mapping heuristic places objects on the processor mesh in an organized fashion (not leaving holes at random places all over the the mesh), each call to `findNearest2D` takes constant time.

Heuristic 2 - Max Heap Traversal: This is an optimization over Heuristic 1. Here, we start with the node that has the maximum number of neighbors and place it on the processor at the center of the 2D mesh. All unmapped neighbors of mapped nodes are put into a max heap. The nodes are stored in decreasing order of the number of neighbors that have already been mapped. Thus, by using a heap, we give preference to nodes that have the maximum number of neighbors that have already been placed.

Algorithm 1 Max Heap Traversal (`MHT`) Heuristic

procedure MHT($commMatrix, P_x, P_y$)
  // begin with the node with max neighbors ($start$) and a processor p and place $start$ on p
  Map[$start$] $= p$
  push all neighbors of $start$ into the $maxHeap$
  while !$maxHeap$.`empty()` **do**
    $start = maxHeap$.`pop()`
    $< c_x, c_y >=$ centroid of mapped neighbors of $start$
    Map[$start$] = `findNearest2D`(c_x, c_y, P_x, P_y)
    push neighbors of $start$ into the $maxHeap$ if they have not been mapped
  end while
end procedure

The node at the top of the heap is deleted and placed close to the centroid of the processors on which its neighbors have been placed. We use the `findNearest` function to find the nearest available processor to the centroid if the "desired" processor is unavailable. This algorithm is referred to as `MHT` in figures. Algorithm 1 shows the pseudo-code for the `MHT` heuristic.

V. Inferring the Spatial Structure

Sometimes we know that an application is simulating entities that are laid out in 2D/3D space but we do not have the spatial coordinates of the nodes in the communication graph. Even if we do not have coordinate information, we can still try to infer the geometric arrangement of the tasks. For example, graph layout algorithms assign coordinates to each node for a layout of planar graphs using force-directed graph algorithms [17], [18]. We observed that graph layout algorithms created graphs that matched the actual geometry of the meshes quite well.

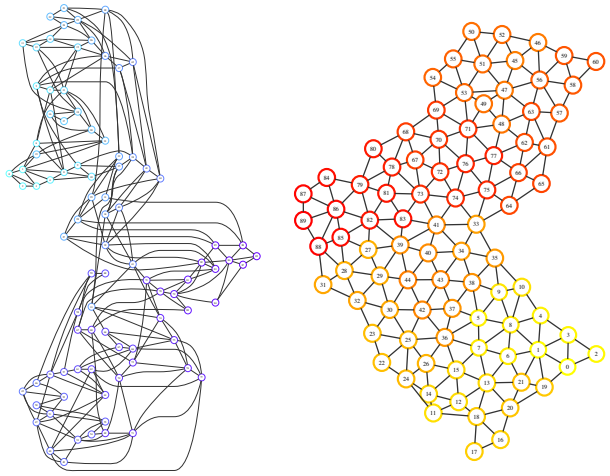

Fig. 3. Using the graphviz library to infer the spatial structure of an irregular graph – original graph is shown on the left and the force-based planar layout obtained using graphviz is shown on the right

To infer the coordinates of nodes in a graph, we use the graphviz library [19], specifically *neato*, one of the graph layout algorithms. The layout computed by neato is specified by a physical model where nodes are treated as objects being influenced by forces. The layout tries to find positions for nodes such that the forces or the total energy in the system is minimized. Neato implements the algorithms developed by Kamada and Kawai [18]. We can use coordinates output by the graphviz library as properties of the nodes for heuristics discussed in Section VI.

Figure 3 shows the geometry inferred by the graphviz library for an irregular graph of 90 nodes (original graph on the left and force-based layout on the right). This graph was obtained from a CHARM++ benchmark which does unstructured grid computations. Each node in the graph is a task or process in the program and contains a portion of the unstructured mesh. The mesh is distributed among the nodes by METIS, a graph partitioning library [20]. Each node might typically contain 100 to $10,000$ triangles. We will use a similar graph as shown in Figure 3 with varying number of nodes to evaluate the mapping algorithms in this paper.

VI. Strategies for Graphs with Coordinate Information

Quite often, when using unstructured grids to simulate fractures in planes or solids that are laid out in 2D/3D space, the tasks in the parallel application have some geometric coordinate information associated with them. This information can be used when mapping a 2D or 3D communication graph

to 2D/3D processor topologies. The heuristics below exploit coordinate information associated with the nodes to guide their decisions.

Heuristic 3 - Affine Mapping: Each node in the communication graph has X and Y coordinates which represent the centroid for all triangles of the underlying mesh in that particular node. An affine translation is done on the coordinates of the centroid to match the relative dimensions of the processor mesh. Based on the translation, if more than one node gets mapped to the same processor, all subsequent nodes after the first one are mapped by spiraling around their original mapping. For this we use the `findNearest` function discussed earlier. Affine mapping leads to a stretching and shrinking of the communication graph and may or may not give the best solutions depending on its aspect ratio. This algorithm (pseudo code in Algorithm 2) is referred to as `AFFN` in the figures.

Algorithm 2 Affine Mapping (`AFFN`) Heuristic

procedure AFFN($commMatrix$, $coordInfo$, P_x, P_y)
  // let min_x, max_x, min_y and max_y denote the minimum and maximum x and y coordinates
  // associated with any node
  for $i \in nodes$ **do**
    $af_x = \lfloor P_x \times \frac{x - min_x}{max_x - min_x} \rfloor$
    $af_y = \lfloor P_y \times \frac{y - min_y}{max_y - min_y} \rfloor$
    Map[i] = `findNearest2D`(af_x, af_y, P_x, P_y)
  end for
end procedure

Heuristic 4 - Corners to Center: A second heuristic that uses coordinate information starts from four corners of the communication graph simultaneously and maps progressively from those directions inward. The four corners for an irregular graph are obtained based on the coordinates associated with the nodes. Depending upon the shape of the graph, it might not be possible always to find four corners of a given graph. Hence, simplistically, we choose the four nodes with the minimum and maximum X and Y coordinates respectively.

After placing the four chosen nodes on four corners of the 2D mesh (in 3D, we would do the same with eight corners), we can use heuristics developed in Section IV to choose the mapping of the remaining nodes. We can either do a breadth-first traversal from each corner or we can do a max heap traversal and place nodes with the maximum mapped neighbors first. `COCE` refers to the algorithm that uses the `BFT` heuristic for mapping the remaining nodes after the corners have been placed. `COCE+MHT` refers to the algorithm that uses the `MHT` heuristic for the remaining nodes.

VII. Results: Mapping to a 2D Processor Mesh

We first compare the algorithms discussed above for mapping of irregular graphs to 2D processor meshes. The graphs used in this section and the next were obtained from a CHARM++ benchmark which does unstructured grid computations. Each node in the graph is a task or process in the program and contains a portion of the unstructured mesh. The coordinates for each node in the graph are the coordinates of the centroid for all triangles in a node.

A. Time Complexity

The time complexity of the mapping algorithms depends on the running time of the `findNearest` implementation, which is used by all of them. Let us assume that `findNearest` takes constant time at each call. The `BFT` heuristic visits each node in the graph once and hence takes linear time for doing the mapping. The max heap traversal (`MHT`) heuristic deletes the element at root of the heap which is $\Theta(\log n)$ and inserts new elements in the heap which takes $\Theta(\log n)$ for each insert (where n is the number of nodes in the communication graph). Every node in the graph is only inserted once and hence the total time complexity for the algorithm is $\mathcal{O}(n \log n)$. The `COCE` heuristic takes linear time since it visits each node only once. The `COCE+MHT` algorithm, which uses the max heap technique takes $\mathcal{O}(n \log n)$. The affine mapping heuristic also visits each node once and therefore has a linear running time.

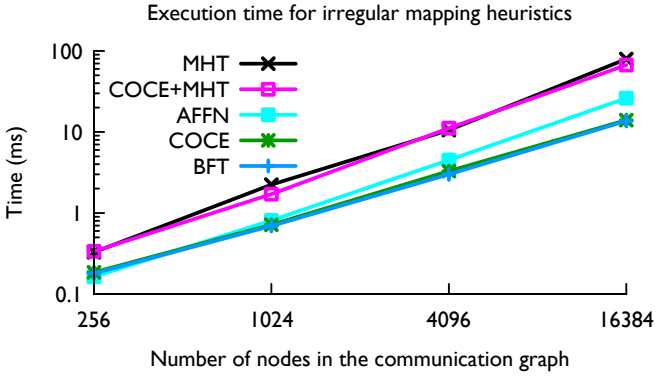

Fig. 4. Execution time for the five irregular mapping heuristics

If we assume that `findNearest` has an average case time complexity of $\mathcal{O}(\log^2 n)$, then an additional $\mathcal{O}(n \log^2 n)$ term is added to the all the algorithms. More powerful heuristics (than the ones implemented) are possible, but with scalability in mind, anything worse than linearithmic (for the average case) is not practical. Figure 4 shows the actual running times of the mapping algorithms (on a 2.4 GHz Intel Core 2 Duo processor). We can see that `BFT` and `COCE` take less than $10\ ms$ for a $16,384$-node graph.

B. Quality of Mapping Solutions: Hop-bytes

Next, we compare the mapping heuristics in terms of reduction in the average hops per byte, using irregular graphs of varying sizes (graph described in Section V). These are mapped on 2D processor meshes and the comparison is done by calculating the hops per byte for each mapping. The heuristics are also compared with the hops per byte obtained by the default mapping and using the pairwise exchanges technique [5]. The default mapping is a linearized mapping of the

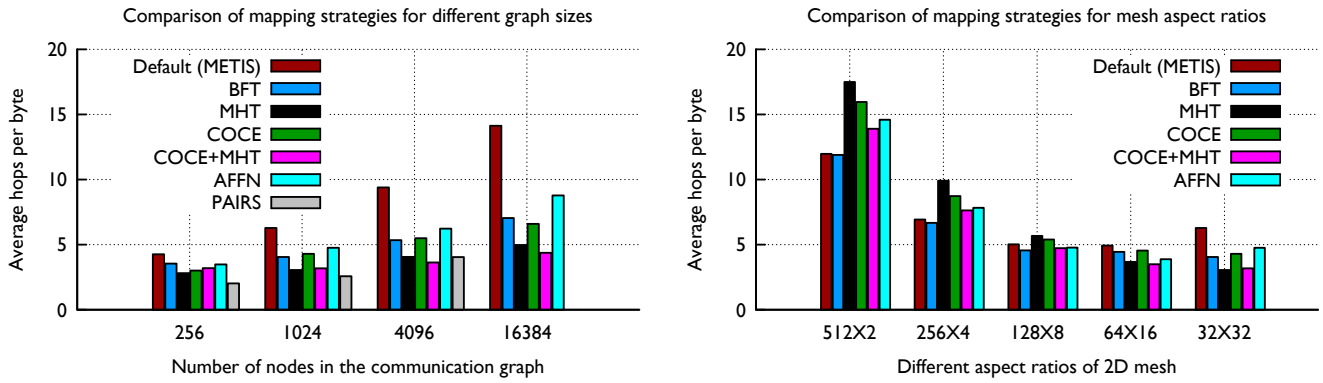

Fig. 5. Hops per byte for mapping of irregular graphs to 2D meshes of different sizes (left) and to $1,024$-node meshes of different aspect ratios (right)

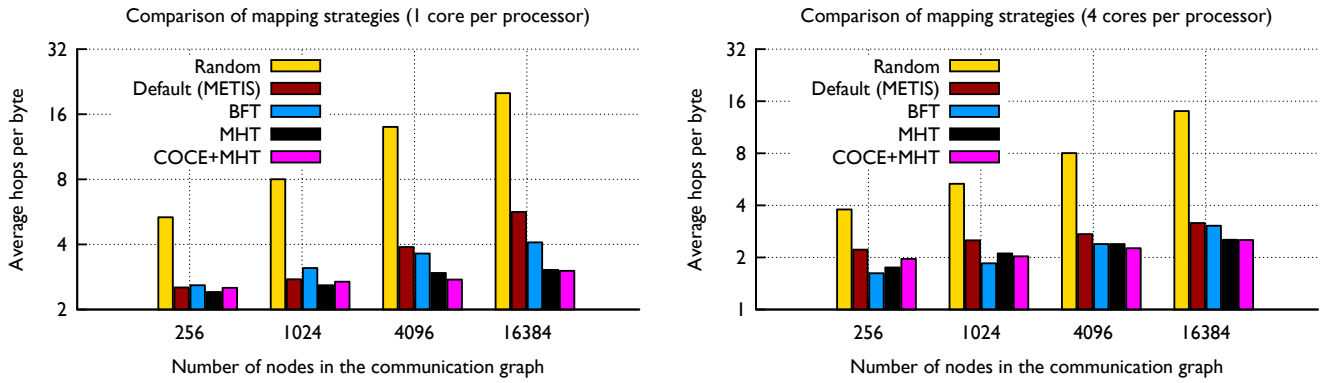

Fig. 6. Hops per byte for mapping of irregular communication graphs to 3D tori

nodes (numbered by METIS during mesh partitioning) by their IDs to the processors. For pairwise exchanges (PAIRS), we start with the solution obtained by MHT and do pairwise swaps until we have a "reasonable" value for hops per byte. This gives a near-optimal value for the hops per byte. Figure 5 (left) shows the average hops per byte for various algorithms when mapping graphs containing 256 to $16,384$ nodes respectively.

The MHT heuristic, which does a max heap traversal based on the maximum number of mapped neighbors, gives the best average hops per byte for lower graph sizes whereas COCE+MHT heuristic performs best for larger graphs (> 4096 nodes). The other three heuristics (BFT, COCE and AFFN) do not perform as well. For mapping to 2D meshes, the performance of mapping algorithms is much better than the default mapping especially for large node counts.

Next, we compare the effects of varying the aspect ratios of the processor mesh (keeping the total number of processors constant). Figure 5 (right) presents the hops per byte for mapping a $1,024$-node graph using various heuristics. The dimensions of the processor mesh are varied from 512×2 to 32×32. A general trend to be noticed is that mapping heuristics do not perform very well when the aspect ratio is very skewed (one dimension is much larger than the other dimension). For the first two mesh dimensions, the BFT heuristic alone does better than the default mapping. Even for the 128×8 mesh, BFT is the best.

The COCE+MHT heuristic performs well for the last three cases (and better than MHT in almost all cases.) This shows that it is useful to have the freedom to use the coordinate information associated with nodes when needed. Since different heuristics perform "best" depending on the communication graph and the processor mesh and the algorithms are very scalable, it may be worthwhile to try all of them, and choose the one that is best.

VIII. Results: Mapping to a 3D Torus

The previous section discussed strategies for mapping an irregular graph to a 2D processor topology. These strategies can be easily extended to map such graphs on to 3D topologies. This section discusses the mapping of graphs from an unstructured grid application to 3D torus partitions.

Figure 6 (left) presents the hops per byte obtained for mapping irregular graphs to 3D torus partitions ranging in size from 256 to $16,384$ processors. These experiments used a mesh similar to the one shown in Figure 3. As we increase the number of processors, the improvement in average hops per byte over the default mapping increases. For smaller number of processors, MHT performs the best, but gradually COCE+MHT improves and gives better average hops per byte on large number of processors.

An important observation is that the default mapping is much better than what a random mapping would obtain for the

hops per byte values. It is closer in performance to the mapping heuristics. Let us analyze a particular case to get a better idea. For the case when objects are mapped randomly, the average hops per byte would be half of the network diameter. For the $16,384$ processor case, the dimensions of the torus are $16 \times 32 \times 32$ and the diameter is $8 + 16 + 16 = 40$. So on the average, each message would travel 20 hops if the objects were mapped randomly. As we can see in Figure 6 (left), the hops per byte value for the random mapping is 20.03 and for the default mapping is 4.6 – significantly smaller than that for the random mapping.

The default mapping is a linearized mapping of the objects based on their IDs to the processors. This suggests that the object IDs correspond well with the communication properties of the objects (*i.e.* objects with nearby IDs communicate with each other.) We attribute this to the METIS algorithm [20] which is used for partitioning the unstructured mesh into the parallel tasks (nodes of the communication graph). METIS tries to preserve communication properties of the graph by numbering communicating objects with nearby IDs.

Finally, Figure 6 (right) presents the hops per byte when processors on the 3D network have multiple cores per node. This is fairly common on today's supercomputers and we assume four cores per node (as is the case for IBM Blue Gene/P). The hops in this case are calculated by counting only those edges in the graph that go across nodes. In general, the average hops per byte for each case are smaller for the right plot when compared with the left plot. This is because several edges are now within the node and hence do not count towards messages going on the network. As a consequence, the improvement in hops per byte when there are multiple cores is less than when there is one core per processor.

IX. Summary

In this paper, we presented fast, heuristic-based algorithms for the mapping of irregular communication graphs to 2D and 3D mesh topologies. We demonstrated that domain-specific knowledge of parallel applications such as the geometric coordinates can be used to aid the mapping process and obtain better solutions. We also presented an efficient implementation for finding the nearest available processor on 2D meshes based on a quadtree which leads to performance improvements of up to ten times!

The mapping algorithms were compared with the default METIS-based mapping and with the technique of pairwise exchanges and evaluated based on the metric of average hops per byte. As described earlier, some of the heuristics can be extended to map on arbitrary network topologies. The heuristics presented in this paper are part of an automatic mapping framework that is being developed to relieve the application developer of the mapping burden.

Acknowledgments

This work was supported in part by the DOE Grant DE-SC0001845 for HPC Colony II. This document was released by Lawrence Livermore National Laboratory for an external audience as LLNL-CONF-491311.

References

[1] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6d mesh/torus interconnect for exascale computers," *Computer*, vol. 42, pp. 36–40, 2009.

[2] A. Bhatelé, E. Bohm, and L. V. Kalé, "Optimizing communication for Charm++ applications by reducing network contention," *Concurrency and Computation: Practice & Experience*, vol. 23, no. 2, pp. 211–222, February 2011.

[3] F. Gygi, E. W. Draeger, M. Schulz, B. R. D. Supinski, J. A. Gunnels, V. Austel, J. C. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz, "Large-Scale Electronic Structure Calculations of High-Z Metals on the Blue Gene/L Platform," in *Proceedings of the International Conference in Supercomputing*. ACM Press, 2006.

[4] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. 33, pp. 1023–1029, November 1984.

[5] Shahid H. Bokhari, "On the Mapping Problem," *IEEE Trans. Computers*, vol. 30, no. 3, pp. 207–214, 1981.

[6] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, and R. Walkup, "Optimizing task layout on the Blue Gene/L supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 489–500, 2005.

[7] F. Ercal and J. Ramanujam and P. Sadayappan, "Task allocation onto a hypercube by recursive mincut bipartitioning," in *Proceedings of the 3rd conference on Hypercube concurrent computers and applications*. ACM Press, 1988, pp. 210–221.

[8] S. Arunkumar and T. Chockalingam, "Randomized Heuristics for the Mapping Problem," *International Journal of High Speed Computing (IJHSC)*, vol. 4, no. 4, pp. 289–300, December 1992.

[9] S. Wayne Bollinger and Scott F. Midkiff, "Processor and Link Assignment in Multicomputers Using Simulated Annealing," in *ICPP (1)*, 1988, pp. 1–7.

[10] Soo-Young Lee and J. K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 433–442, 1987.

[11] H. Yu, I.-H. Chung, and J. Moreira, "Topology mapping for Blue Gene/L supercomputer," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 116.

[12] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 75–84.

[13] C. Chevalier, F. Pellegrini, I. Futurs, and U. B. I, "Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework," in *In Proceedings of Euro-Par 2006, LNCS 4128:243252*, 2006, pp. 243–252.

[14] A. Bhatele, G. Gupta, L. V. Kale, and I.-H. Chung, "Automated Mapping of Regular Communication Graphs on Mesh Interconnects," in *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[15] T. Agarwal, A. Sharma, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[16] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, http://hdl.handle.net/2142/16578.

[17] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, pp. 1129–1164, March 1991.

[18] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Inf. Process. Lett.*, vol. 31, pp. 7–15, April 1989.

[19] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software - Practice and Experience*, vol. 30, pp. 1203–1233, 1999.

[20] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, 1996, p. 35.