A Pattern Language for Topology Aware Mapping

Abhinav Bhatelé, Laxmikant V. Kalé, Nicholas Chen and Ralph E. Johnson

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{bhatele, kale, nchen, johnson}@illinois.edu

Abstract

Obtaining the best performance from a parallel program involves four important steps: 1. Choice of the appropriate grainsize; 2. Balancing computational and communication load across processors; 3. Optimizing communication by minimizing inter-processor communication and overlap of communication with computation; and 4. Minimizing communication traffic on the network by topology aware mapping. In this paper, we will present a pattern language for the fourth step where we deploy topology aware mapping to minimize communication traffic on the network and optimize performance.

Bandwidth occupancy of network links by different messages at the same time leads to contention which increases message latencies. Topology aware mapping of communicating tasks on the physical processors can avoid this and improve application performance significantly.

1 Introduction

Arguably, writing parallel programs is hard. Optimizing parallel programs so that they obtain the best performance possible is even harder. There are some fundamental things which every programmer has to consider irrespective of whether the program will be run on a multi-core desktop or a large supercomputer.

Grainsize: The amount of computation a processor does before communicating with other processors should be chosen to allow maximum overlap of communication and computation while still minimizing the overhead of dividing the work.

Load Balancing: All processors should do similar amounts of work (inclusive of both computation and communication) since the time to execute a parallel program is bound by the processor with the most work.

Communication Optimization: For applications running on large machines, we should minimize inter-node communication and maximize computation-communication overlap wherever possible. This reduces the communication volume on the network i.e. the total amount of bytes transferred over the network, thereby minimizing the time for which processors are idle, waiting for messages.

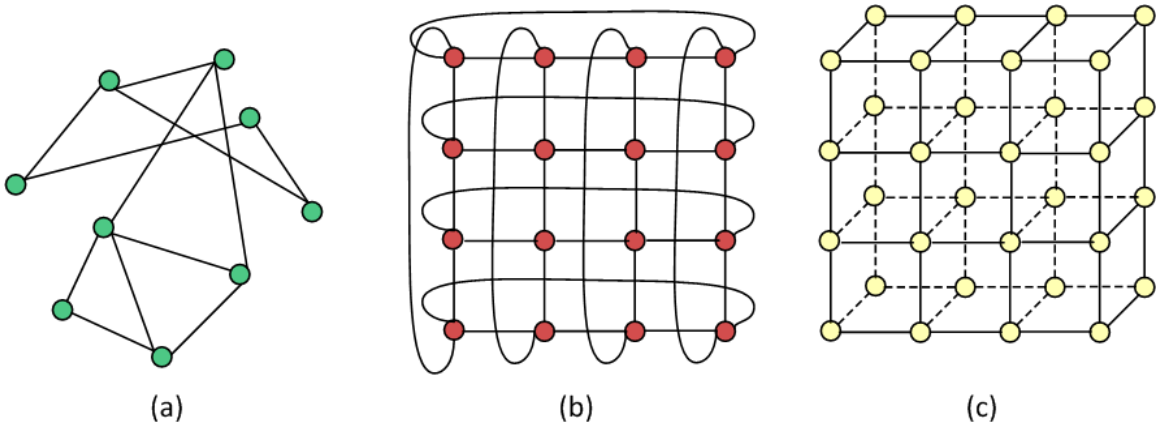

Figure 1: (a) An arbitrary object communication graph, (b) Object communication graph for a regular communication pattern, (c) A processor topology graph - 3D mesh

Minimizing Contention: Sometimes these three steps are not enough. A fourth step is necessary to minimize contention and communication traffic. Communication traffic is quantified by the *hop-bytes* metric, where hop-bytes for a message is the message size times the number of hops it travels. Hop-bytes for an application is the sum of the hop-bytes of all its messages. Contention is caused by different messages sharing a network link. Topology aware mapping is a technique for reducing contention and communication traffic by placing communicating tasks on physically nearby processors.

The problem of topology aware mapping can be stated as follows: Given a *object communication* graph and a *processor topology* graph, how can we map the first graph onto the second such that most messages travel a small number of hops (links) on the network. Figure 1(a) and (b) present an irregular and regular object communication graph and Figure 1(c) presents a 4 x 4 x 2 dimension mesh (processor graph). Many of the large parallel machines today have a three-dimensional mesh or torus interconnect with large network diameters. If most messages in a program travel a large number of hops, they share the bandwidth on each link with other messages and hence suffer delays. In other words, contention for the same links on the network leads to increase in message latencies [1, 2]. Topology aware mapping of tasks or objects helps us avoid this by co-locating communicating tasks on nearby physical processors [3, 4, 5]. Reducing the total hop-bytes for an application signifies a reduction in the communication traffic on the network.

This paper presents a pattern language for topology aware mapping of parallel applications. Depending upon the communication graph of a particular application and the processor topology, the problem statement and correspondingly, the solution for mapping changes. This paper will discuss the different patterns which these problems present, set a context for them and discuss generic solutions in each case. It should be noted that solutions presented here are general techniques and specific implementations would differ depending on the nature of the specific problem.

We will mostly be discussing three-dimensional (3D) torus and mesh topologies for parallel machines for the patterns in this paper. Such topologies are commonly used in some

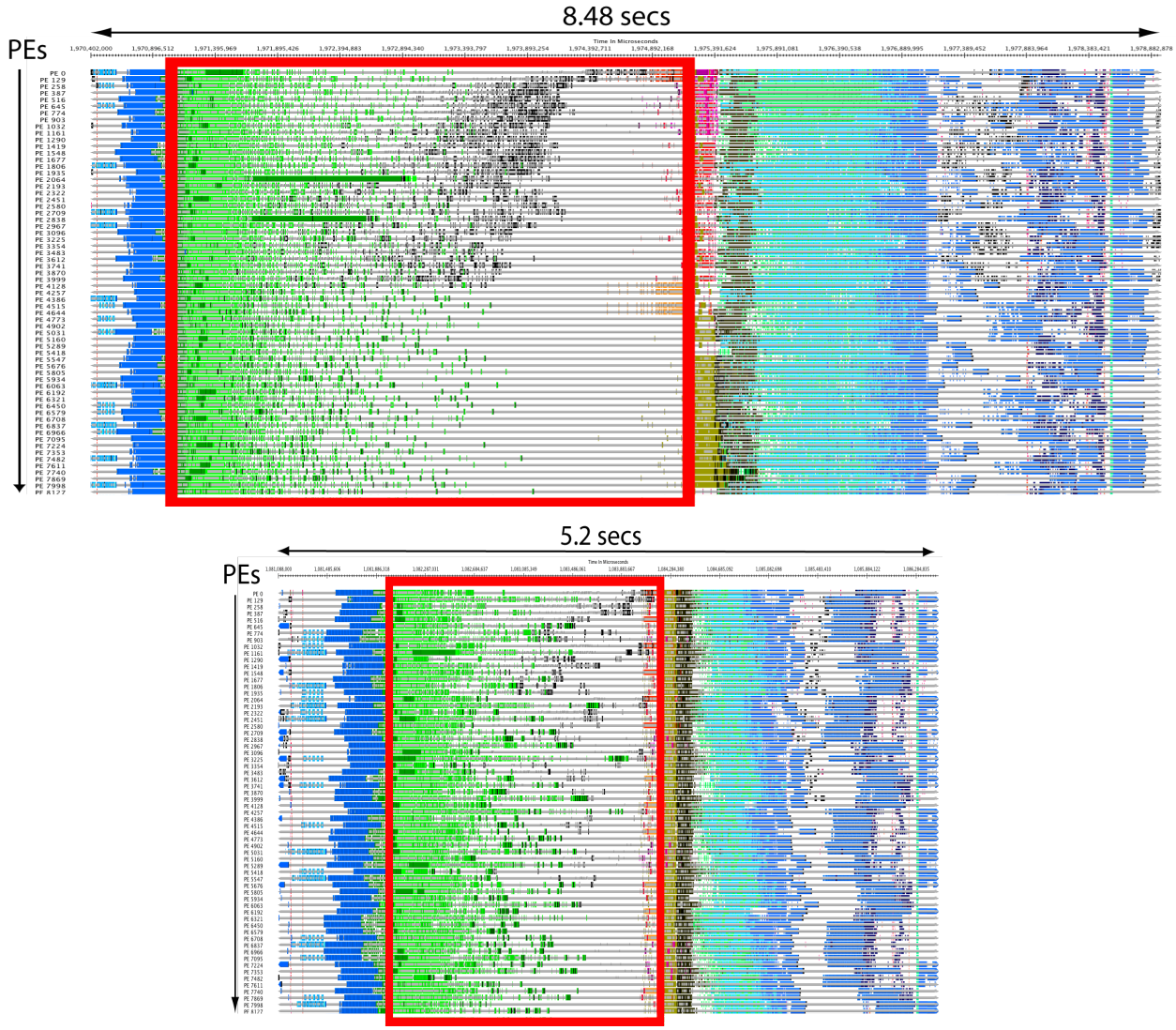

Figure 2: Projections overview snapshots showing a delay in phases

of the fastest supercomputers on the top500 list today[†]. IBM's Blue Gene/L, Blue Gene/P, Cray's XT3, XT4 and XT5 are some examples of highly scalable machines which use this topology. Topology aware mapping is an important technique for supercomputers, but is not needed for current multi-cores. However, as multi-cores get larger and more complicated, it will probably become important for them, too.

2 Motivation

OpenAtom is a fine-grained parallelization of the CPAIMD method to understand dynamics of atoms at a quantum scale [6]. Computation in OpenAtom is divided into a large number of objects, enabling scaling to tens of thousands of processors. Calculating the electrostatic

[†]http://top500.org/lists/2008/11

	WATER_32M_70Ry			WATER_256M_70Ry		
Cores	**Default**	**Topology**	**Improvement(%)**	**Default**	**Topology**	**Improvement(%)**
512	0.274	0.259	5	-	-	-
1024	0.189	0.150	20	19.10	16.4	14
2048	0.219	0.112	48	13.88	8.14	41
4096	0.167	0.082	50	9.13	4.83	47
8192	0.129	0.063	51	4.83	2.75	43
16384	-	-	-	3.40	1.71	50

Table 1: Execution time per step (in secs) of OpenAtom on Blue Gene/L (CO mode)

energy involves computing several terms. Hence, CPAIMD computations involve a large number of phases with high inter-processor communication. These phases are discretized into a large number of objects, which generate a lot of communication, but ensures efficient interleaving of work.

When running on 8192 processors of Blue Gene/L, it was noticed that the scaling and absolute performance are significantly lower than expected. Using the Projections performance analysis tool [7], we obtained a time profile of a particular run. Figure 2 shows the profiles for two different runs: the top one uses the default mapping and the bottom one uses a topology aware mapping. The x-axis is the timeline for one iteration and the y-axis has 63 randomly chosen processors (PE) out of the 8192. The top run has a huge white region where all processors were idle (enclosed in the red box). Using topology aware mapping of the communicating *chare arrays* in OpenAtom, we were able to reduce this white region. This improved the time per iteration of OpenAtom from 8.48 to 5.2 seconds. Time per iteration for an iterative algorithm determines the total time for the simulation to complete. Since we were able to nearly halve the time per iteration, a simulation which would normally take a year to complete can now be done in six months!

We studied the strong scaling (fixed problem size) performance of OpenAtom with and without topology aware mapping. Two benchmarks commonly used in the CPMD community: the minimization of WATER_32M_70Ry and WATER_256M_70Ry were used. As shown in Table 1, performance improvements from topology-aware mapping for Blue Gene/P (BG/P) can be quite significant. As the number of cores and likewise, the diameter of the torus grows, the performance impact increases until there is 51% improvement for WATER_32M_70Ry at 8192 and 50% for WATER_256M_70Ry at 16384 cores.

Improvements obtained for OpenAtom suggest that applications running on large scale parallel machines can benefit significantly from topology aware mapping if they create contention on the network. Hence application writers who intend to scale their codes to a very large number of processors would find this work quite useful.

3 Pattern Language

The problem of topology aware mapping can be stated as follows: Given a *object communication graph* and a *processor topology graph*, how can we map the first graph onto the second such that most messages travel a small number of *hops* (links) on the network. The aim is to

minimize the total *hop-bytes* for an application which signifies a reduction in communication traffic and hence contention on the network. The definitions of the terms in the previous problem statement are as follows:

Object Communication Graph Objects refer to the communicating entities in a parallel program. For example, in the case of MPI, these would be MPI ranks and in CHARM++, they would chare array elements. Objects in a parallel program communicate and define a directed graph with objects as the vertices and a edges between objects representing communication. Weights on the edges represent the number of bytes communicated.

Processor Topology Graph The processors in a parallel machine are connected in a certain topology. It can be a three-dimensional torus or mesh such as in IBM Blue Gene and Cray XT machines or a tree-based network such as in Infiniband clusters. The connections between processors define the processor topology graph.

Hops The number of network links a messages travels to reach from the source to destination.

Hop-bytes Communication traffic is quantified by the *hop-bytes* metric which is the weighted sum of the messages sizes where the weights are the number of hops traveled by the respective messages. Reducing the total hop-bytes for an application signifies a reduction in the communication traffic on the network.

The object communication graph **and** the corresponding processor topology graph determine the patterns we should use to solve the problem of topology aware mapping. Thus, the mapping algorithm needs to first obtain both graphs before the actual mapping can be done.

Most of the patterns described below assume that the processor topology is a 3D torus or mesh network. For such networks, it is relatively easy to obtain the processor topology graph through system calls.

Obtaining the object communication graph is harder – if the communication graph is static, we rely on the application writer's knowledge for the particular application. If the communication graph is dynamic, we must use runtime instrumentation to gather the communication graph. And if runtime instrumentation is not possible, we need to perform a test run of the program under typical conditions, collect information about its communication patterns and use this collected information to guide the mapping for subsequent executions.

We now describe a pattern language which helps a parallel programmer choose the solution for topology aware mapping depending on the configuration of the object communication graph and the processor topology graph. Our pattern language represents a progression from simpler patterns to more advanced ones. A simpler pattern is not only easier to understand and implement but also provides more predictability in improvements by taking advantage of the static nature of the problem. More advanced patterns are harder to implement but are useful if the nature of the problem is more dynamic and unpredictable.

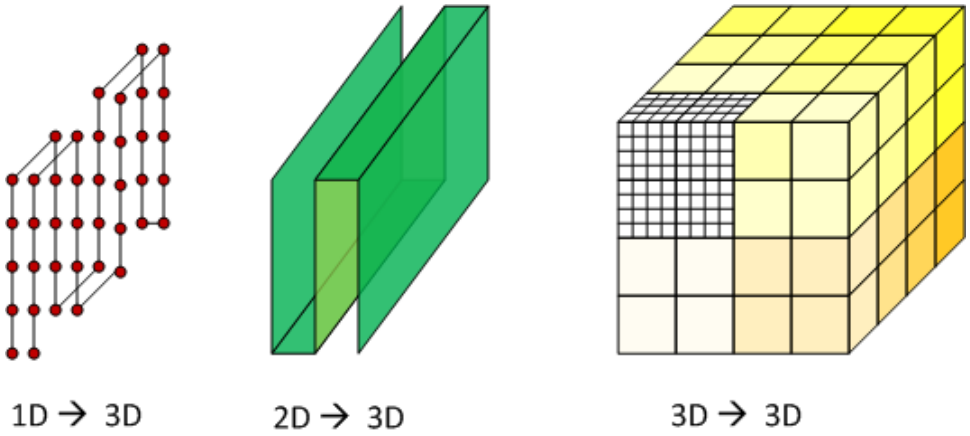

Figure 3: Embedding of one, two and three dimensional object graphs on to a three-dimensional processor graph

3.1 Static Regular Communication

Problem: How do we map a group of objects with regular/structured communication patterns?

Context: A wide range of parallel applications have fairly *regular* communication patterns [8, 9, 10]. In some cases, the communication is in a single dimension where processor 0 communicates with 1, processor 1 with 2 and so on. In other cases, there is a two-dimensional array of objects and each object communicates with its immediate neighbors in the four directions (up, down, left, right). This pattern is seen in a stencil computation where the value of each element in an array is updated to the average of the values of four elements surrounding it and itself.

For such applications, the communication is easily defined as a *function of the indices of the objects* and there are relatively few cross dependencies which makes mapping relatively easier.

Forces: The main constraint for this pattern is that the communication graph should be expressible as an n-dimensional array of objects with structured communication. There are a few factors which affect the choice of the mapping algorithm to be used. The first is the dimensionality of the object graph (if it is a line or a 2D or 3D grid). The second factor is the size of each dimension and if it matches the dimensions of the processor graph or not. The cardinality of the object and processor graph also affect the mapping decisions.

Solution: Given that the communication is structured and with neighbors whose indices are close to one's indices, the solution is to embed the object graph onto the processor graph so that objects with similar indices are on nearby processors.

First, check the cardinality of the object graph. If the number of objects is more than the number of processors, then we need to place multiple objects on each processor ensuring load balance. If the number of objects is not exactly divisible by the number of processors, it is always advisable to place less number of objects on a few processors than placing more

number of objects on a them. For example, if we have to place 100 objects on 8 processors, we could place 12 objects on 7 processors and 16 objects on one processor. Alternatively, we could place 13 objects on 7 processors and 9 objects on one. Since the speed of a parallel program is decided by the slowest processor i.e. the one with the most objects, the second choice is a better one.

Once the number of objects on each processor is decided and objects have been divided into blocks, we need to decide how to place these blocks on the processors. If we have a 1D or 2D array of objects, we fold one or two of the dimensions of the object graph and map it on the 3D torus. In the case of a 3D object graph, the embedding is simpler. This is simple if the dimensions of the object graph match the dimensions of the processor graph. If not, then a careful matching of similar sized dimensions between the object and the processor graph is required.

Since we *know* the object communication graph and processor topology graph, the mapping can be done once and will not change as the application executes.

Examples: Matrix algorithms such as multiplication, stencil computation in 1D, 2D or 3D, structured mesh computation are examples where this pattern works well.

The three-dimensional matrix multiplication algorithm [11] creates a 3D virtual topology of objects. The communication in this algorithm is perpendicular to the planes of the 3D array of objects. Using the techniques described above, we can map its object graph to a machine with a 3D torus or a mesh.

In a structured mesh computation application (MILC is one such example [8]), we generally have a 2D or 3D array of objects which need to be mapped on to the machine. Communication is near-neighbor and hence, the only consideration is to keep objects which are near each other in the array on processors which are nearby. In the case of MILC, we use the embedding of a 4D graph on to a 3D graph.

3.2 Static Irregular Communication

Problem: How do we map an object communication graph which is static but has irregular communication patterns?

Context: For some parallel applications, the object communication graph is irregular – it does not match any identifiable patterns. Objects which communicate with one another do not have any relationship with respect to their place in the data structures used to represent them.

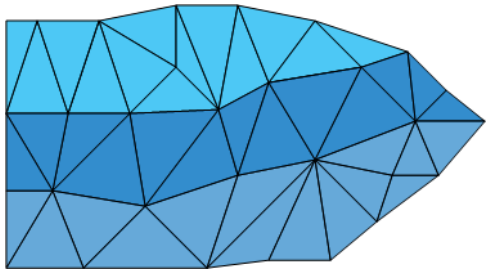

Figure 4: Example of an Unstructured Mesh

For example, consider an unstructured mesh used in representing the surface of a solid (Figure 4). The mesh is represented as a collection of triangle elements and each triangle element stores a list of its neighbors. Similarly each node in the mesh stores a list of all the triangle elements which share this node. The number of neighbors each triangle or node has is unbounded and each can potentially communicate with a *lot* of

neighbors and **not** just with 4 neighbors like the case of structured grid.

Forces: Any given object might communicate with any number of objects in the communication graph. This would create opposing forces wherein mapping in a certain fashion favors some communication arcs but penalizes others. In essence, finding the optimal solution for mapping is hard. It has been proven that the general mapping problem can be reduced to the graph isomorphism problem which is NP-hard [12, 13]. Heuristic techniques can be used for the regular pattern also but they might not perform as good as embedding techniques.

Solution: In the case of irregular communication, use heuristic techniques to generate a solution which is close to optimal. For example in the case of a mesh application, we can apply a simple heuristic of placing an element in the mesh and then attempting to place all its neighbors on processors around it, ensuring load balance. Once all its neighbors are placed, we pick one of the neighbors and try to place its neighbors around it. The choice of which node to start with and which neighbor to pick when mapping the second level and so on depends on the exact problem. Depending on the application, we might have to try several heuristics before we find a good solution.

Since we *know* the object communication graph and processor topology graph the mapping can be done once and will not change as the application executes.

Example: A variety of scientific applications which use unstructured meshes for their computation fall in this category. The photo transport code UMT [14] and Explicit Finite Element codes such as [15] and [16] are good examples. For unstructured mesh codes, one possible heuristic is described above where we start with one element in the mesh, place its neighbors around it and continue this with the neighbors. Another possible solution would be to create a space filling curve [17] from the mesh elements which will linearize the objects. We would then place this linear curve by folding in onto the 3D processor graph. The expectation is that mesh elements which are spatially close to one another will also be close on the space filling curve.

3.3 Dynamic Communication

Problem: How do we map an object communication graph which changes at runtime as the application progresses?

Context: For some applications such as molecular dynamics, as the simulation proceeds, particles move from one processor to another to support dynamic load balancing. Particles that used to communicate with one another now need to communicate through a proxy and the actual object that the proxy points to might have been moved to a different, further location. In other words, the communication graph changes significantly.

In this case, a static mapping done once at the beginning of the application is *not* sufficient.

Forces: In the first two patterns, since the communication graph was static, we relied on the application writer's knowledge of the graph. If the graph changes at runtime, we need to find an automatic solution to obtain the communication graph. CHARM++ provides a

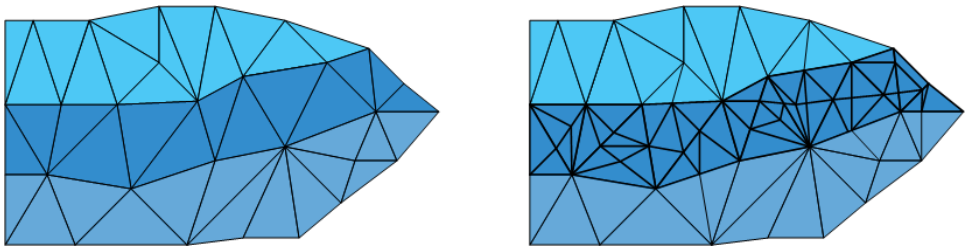

Figure 5: Structured Mesh Computation with Adaptive Mesh Refinement

load balancing framework to instrument the code at runtime which can be used to provide this information [18].

Unfortunately this dynamic instrumentation is not supported on all platforms. For instance, MPI does not provide this facility. Thus, the only solution requires obtaining this data in a test run and then use that data to do the mapping.

Solution: When the object communication graph is dynamic, use the technique of dynamic load balancing at runtime instead of initial static mapping.

The dynamic load balancing has to consider the changing topology of the problem, capture a snapshot of that change, and use the information in the snapshot to reassign objects to processors.

Different load balancing techniques should be used depending on the problem:

- If the communication graph does not vary significantly as the application progresses, a simple refinement scheme might be good where we move a few objects around to account for the change in computational and communicational loads.

- If the changes in load are drastic, it might take more work to refine the topology. In that case it is better to reassign all the objects and hence do a fresh mapping of all objects all over again.

A combination of refinement and re-mapping from scratch can be done based on the degree of change for each snapshot.

The techniques for performing the actual mapping algorithm for a particular snapshot are similar to the ones used in Section 3.1 and Section 3.2. Depending on whether the object graph is regular or irregular we use folding/embedding or heuristic techniques.

Example: Various parallel applications have dynamic communication patterns: structured mesh computation codes with adaptive mesh refinement (AMR) such as FLASH [19], Molecular Dynamics codes such as NAMD [20] and Barnes-Hut [21] codes such as ChaNGa [22]. We discussed some techniques for mapping mesh codes in the previous section. We can use similar techniques in every load balancing phase after refinement occurs (Figure 5).

As a second example of this pattern, we will discuss the mapping in a molecular dynamics code. The communication pattern in NAMD involves multicasts from one set of objects called *patches* to another set called *computes*. Every patch does a multicast to some computes and each compute receives a message from two patches. The patches are statically assigned to

the processor graph initially. Since each compute communicates with two patches, we want to place it close to its communicators. A simple heuristic is used for this: a compute is placed within a 3D box defined by the two processors which host the compute's communicators. The number of hops from the two processors to any processor within the box is the same (Figure 6). For further details, refer to [23].

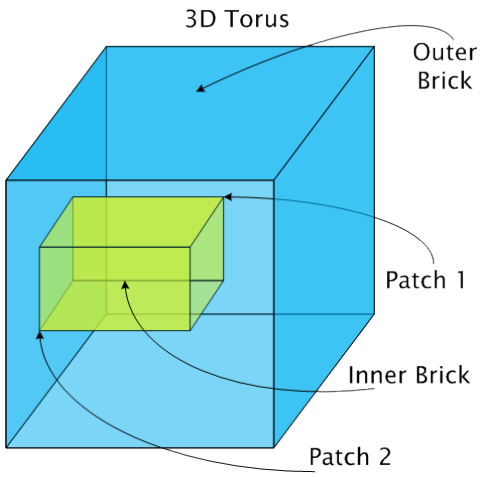

Figure 6: Topology aware mapping in NAMD

3.4 Applications with Multiple Object Graphs

Problem/Context: How do we map an application which has multiple arrays of objects communicating within themselves and with one another?

Context: The three patterns described in the previous sections cover the communication graphs for many common parallel applications and should be sufficient for most programmers' needs. This particular pattern addresses a more complex problem.

All patterns we discussed above have a single object communication graph. For some applications, especially in those written in Charm++, it is possible to have multiple object graphs at different levels. These graphs have communication among their own internal nodes and with one another. This increases the complexity of the mapping problem manifold because there are now many more factors to consider for doing the mapping.

Unfortunately, simply considering the top level object communication graph and ignoring the internal object communication graphs does **not** always improve performance.

Forces: This pattern should be used when there are multiple object graphs with strong dependencies between them. As mentioned in the previous section, NAMD has two object graphs (patches and computes) but the communication between patches is limited. So, in that case we simplify the situation by mapping the patches first statically and then mapping the computes around them. Hence, in some cases it might be possible to simplify the situation by mapping one object graph statically and placing the other object graph based on the first.

Solution: When there are multiple object graphs, the first step is to understand the various communication graphs for the application and dependencies across them. Graphs which have weaker dependencies can be mapped independently. For graphs with strong communication dependencies, they need to be considered **together** when mapping each of those graphs. For these graphs, it is important to identify which graph depends on which other graph and then map them in order.

Example: OpenAtom [6], which was introduced in Section 2 has a fairly involved communication. There are fifteen *chare arrays* each of which is an object graph (eight of them are shown in Figure 7). We first decide which object graphs create maximum contention

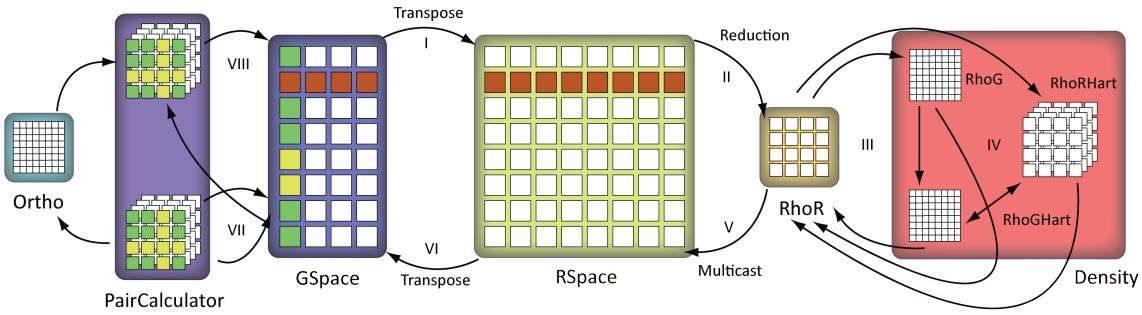

Figure 7: Communication Phases in OpenAtom

and which of those can be mapped independently. The GSpace, RSpace and PairCalculator graphs have closely-tied communication and the RhoR and other density objects communicate closely with RSpace. Hence we first map GSpace and then map RSpace and PairCalculators based on the GSpace mapping. Once this group has been mapped, we map the RhoR and density objects according to the RSpace mapping. For details on the mapping algorithm, please refer to [24].

3.5 Arbitrary Object and Processor Graphs

Problem: How do we map an object graph onto a processor graph when both graphs are irregular?

Context: The four patterns discussed above assumed a three-dimensional torus or mesh topology which is found in some of the prevalent supercomputers today. But there are other supercomputers with topologies such as a fat-tree or Kautz graph or something else. When we consider this general case where the processor topology can be one of the many possible options, the mapping problem becomes even harder. Most research on mapping in the past has focused on specific processor topologies such as mesh networks [4, 5, 25] or hypercubes [13, 26, 27] or other topologies.

Forces: When both the communication graph and the processor graph are arbitrary and we do not have any specific information to help our mapping decisions, it is hard to find optimal solutions. Also, it is hard to find solutions which will do a good mapping when the application is run on different topologies.

Solution: The solution in this case is to use heuristics to arrive at a solution which is close to optimal. Various techniques have been discussed in the literature for this most general case ranging from physical optimization techniques to heuristic approaches. Physical optimization techniques involve simulated annealing [26, 28], graph contraction [29, 30] and genetic algorithms [31]. Physical optimization techniques take a long time to arrive at the solution and hence cannot be used for a time-efficient mapping during runtime. They are almost never used in practice. Heuristic techniques such as pairwise exchanges [13] and recursive mincut bipartitioning [27] were theoretical studies with no real results on machines. Most of the recent work has been focused on specific applications and use of one

of the patterns in the previous sections for the solution.

Example: Much work was done in the 80s on embedding an arbitrary object graph on a processor graph [13, 26, 29, 31]. Most of this work is theoretical and has not been tried on science applications running on real machines. It is always beneficial to use application-specific information to assist the mapping algorithm in arriving at a better solution.

4 Conclusion

This paper presents a pattern language for optimizing communication in parallel applications running on large supercomputers using the technique of topology aware mapping. Which pattern to use depends on the object communication graph of the application and processor topology graph of the machine. We present our patterns in increasing order of difficulty for doing the mapping. Nonetheless, a harder pattern does not always guarantee better results; in fact, using a simpler pattern might actually lead to better results since the mapping is more predictable and easier to arrange statically. Use advanced patterns only when the dynamic nature of the communication and topology graphs precludes using the simpler patterns.

References

[1] Thierry Cornu and Michel Pahud. Contention in the Cray T3D Communication Network. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 689–696, London, UK, 1996. Springer-Verlag.

[2] Abhinav Bhatele and Laxmikant V. Kale. An Evaluation of the Effect of Interconnect Topologies on Message Latencies in Large Supercomputers. In *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS '09)*, May 2009.

[3] Brian E. Smith and Brett Bode. Performance Effects of Node Mappings on the IBM Blue Gene/L Machine. In *Euro-Par*, pages 1005–1013, 2005.

[4] 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*, 49(2/3):489–500, 2005.

[5] Abhinav Bhatelé and Laxmikant V. Kalé. Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, 18(4):549–566, 2008.

[6] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[7] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[8] MILC Collaboration. MIMD Lattice Computation (MILC) Collaboration Home Page. http://www.physics.indiana.edu/∼sg/milc.html.

[9] Young Yoon, James C. Browne, Mathew Crocker, Samit Jain, and Nasim Mahmood. Productivity and performance through components: the asci sweep3d application: Research articles. *Concurrency and Computation: Practice and Experience*, 19(5):721–742, 2007.

[10] Lattice QCD Collaboration. US Lattice Quantum Chromodynamics Collaboration Home Page. http://www.usqcd.org.

[11] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[12] Shahid H. Bokhari. On the Mapping Problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[13] Soo-Young Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Trans. Computers*, 36(4):433–442, 1987.

[14] ASC Sequoia Benchmark Codes. Lawrence Livermore National Laboratory Page. https://asc.llnl.gov/sequoia/benchmarks.

[15] Sandhya Mangala, Terry Wilmarth, Sayantan Chakravorty, Nilesh Choudhury, Laxmikant V. Kale, and Philippe H. Geubelle. Parallel adaptive simulations of dynamic fracture events. *Engineering with Computers*, 24(4):341–358.

[16] Isaac Dooley, Sandhya Mangala, Laxmikant Kale, and Philippe Geubelle. Parallel simulations of dynamic fracture using extrinsic cohesive elements. *Journal of Scientific Computing*, 39(1):144–165, April 2009.

[17] A. J. Fisher. A new algorithm for generating hilbert curves. *Software Practice and Experience*, 16:5–12, 1986.

[18] Sanjeev Krishnan and L. V. Kale. Automating Runtime Optimizations for Load Balancing in Irregular Problems. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, San Jose, California, August 1996.

[19] Greg Weirs, Vikram Dwarkadas, Tomek Plewa, Chris Tomkins, and Mark Marr-Lyon. Validating the Flash Code: Vortex-Dominated Flows. In *Astrophysics and Space Science*, volume 298, pages 341–346. 2005.

[20] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[21] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[22] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[23] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications. In *23rd ACM International Conference on Supercomputing*, 2009.

[24] Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé. A Case Study of Communication Optimizations on 3D Mesh Interconnects. In *Euro-Par 2009, LNCS 5704*, pages 1015–1028, 2009.

[25] P. Sadayappan and F. Ercal. Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Trans. Computers*, 36(12):1408–1424, 1987.

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

[27] 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*, pages 210–221. ACM Press, 1988.

[28] S. Wayne Bollinger and Scott F. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Trans. Comput.*, 40(3):325–333, 1991.

[29] Francine Berman and Lawrence Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, 1987.

[30] N. Mansour and R. Ponnusamy and A. Choudhary and G. C. Fox. Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 1–10. ACM, 1993.

[31] S. Arunkumar and T. Chockalingam. Randomized Heuristics for the Mapping Problem. *International Journal of High Speed Computing (IJHSC)*, 4(4):289–300, December 1992.